CMSC 216

Introduction to Computer Systems

Ekesh Kumar

Prof. Nelson Padua-Perez • Summer 2019, Section 0101 • University of Maryland

http://www.cs.umd.edu/class/summer2019/cmsc216/

Last Revision: November 15, 2019

Contents

1 Tuesday, May 28, 2019

Logistics

1. All lectures are recorded and posted online.
2. No pop quizzes, no collaboration on projects.
3. Website sign-in: cmsc216/sprcoredump.
4. Office hours are immediately after class in IRB 2210.
5. Everybody will get an Arduino to be used later in the course.
6. This class isn't curved.

Basic Unix Commands

Unix has lots of commands, but we want to first focus first on the ones that'll let us write and execute C programs.

- `pwd` → displays your current directory.
- `ls` → displays the files/directories in the current directory.
  - `ls -al` → lists all of the files and directories, including hidden ones (Here, the `a` flag functions to show hidden files, whereas the `l` flag functions to list all entries with detailed information, like last date accessed).
  - `ls -F` → identifies directories by listing them with a /.
- `cd` → change directory to the inputted parameter.

Introduction to C Programming

In CMSC131 and 132, we learned Java. Unlike Java, C is not object-oriented; it has no concept of classes, objects, polymorphism, or inheritance. However, C can be used to implement some object-oriented concepts, like polymorphism or encapsulation. Consider the following program:

Listing 1: A First Program

```c
#include <stdio.h>
int main() {
    printf("Fear the turtle\n");
    return 0;
}
```

How does this program work?

- The `#include` allows the compiler to check argument types. It can compile without declaration, though the compiler will warn you.
- Like Java, C provides a definition of the `main()` function, where all C programs begin.
- We return from `main()` to end the program. For standard practice, we return 0 to signal that everything worked out fine.

Now, let's say we want to run this program. How can we do this? C programs need to be compiled before they can be executed. With the `gcc` compiler, a very simple compilation command is `gcc file.c`, from which we can run the executable by just typing `./file`.

Some more compilation options are summarized below (these are called **flags**):

- `-g` enables debugging by generating and maintaining necessary symbols (e.g. line numbers) upon compilation.
- `-Wall` warns about common things that might be a problem.
- `-o filename` places an executable in the file name.

2 Wednesday, May 29, 2019

Last time, we analyzed a sample C program. It's important to know that returning 0 in the `main()` function is independent of the `void` that appears in the main's header. That is, even if our header is `int main(void)` instead of `int main()`, we will still return 0 at the end of the function. The `void` is just an explicit way of telling our compiler that we shouldn't be passing any parameters in.

More Unix Commands

Some more Unix:

- The `cp` command makes a copy of a file from a source to a destination. Some options are `-a`, which allows us to preserve attributes, like timestamp modified. Also, `-v` explains what's being done, while `-r` copies recursively.
- The `-rm` command removes a file.
- The `-mv` command renames a file or moves a file/directory to another directory. For example...
  - `-mv f1 f2` renames file `f1` to `f2`.
  - `-mv f1 d1` moves the file `f1` to the directory `d1`.
  - Finally, `-mv d1 d2` moves the directory `d1` to `d2`.
  - The `-cat` command displays the contents of a file.

In Unix, we can create **aliases**, which are shortcut commands to use a longer command. Users can use the alias name to run the longer command while typing less. Without any arguments, the `alias` command prints a list of defined aliases. A new alias is defined by assigning a string with the command to a name. We can add an alias by modifying the `.aliases` file in the home directory of Grace.

The general format for defining an alias is `alias [alias name] 'command'`. So adding the line `alias cookies 'ls'` would define the command `cookies` to do the same thing as `ls`

Compilation Stages of a C Program

C programs need to be compiled before they can be executed. What happens when we compile a C program? There are **three compilation stages**:

1. **Preprocessor Stage:** This stage is used to verify that program parts sees declarations that they need. Also, statements starting with a # are called **directives** (for example,
2. **Translation:** In this stage, an object (`.o`) file is created. In addition, the compiler checks to make sure that individual files are consistent with themselves.
3. **Linkage:** Finally, this stage brings together one or more object files. It makes sure that the caller/calee to functions are consistent. The result is an executable file (by default, it's named `a.out`),

Variables in C

There are a lot of data types in C, some of which include `char, short, int, long int, float, double`, etc. In Java, data types take up the same amount of space, independent of the system they're run on. This is not true in C; the minimum size of various data types are not necessarily the same size on grace. We do not need to memorize the sizes of various data types; however, it is important to know that a `char` data type is an exception to this rule: it always takes one byte.

Also unlike Java, there is no maximum size for a type; however, the following inequalities hold:

$$\texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

$$\texttt{sizeof(float)} \leq \texttt{sizeof(double)} \leq \texttt{sizeof(long double)}$$

Suffixes allow us to specify a number of a given type. For instance, `30000` is of type `int`, whereas `30000L` is of type `long`.

In C, there is no default `boolean` data types; anything with value 0 is considered false, whereas any other value is considered true. However, we can use integers to represent booleans with `true` mapping to 1 and `false` to 0.

Consider the following code example:

Listing 2: Conditional Example

```c
#include <stdio.h>
int main() {
    if (100) {
    printf("Fear the turtle\n");
    }
    return 0;
}
```

The print statement in conditional executes successfully for reasons described above.

3 Friday, May 31, 2019

printf() and scanf()

When we're using `printf()` to print something, we just print anything that's in the quotations. For instance, the line `printf("Hello");` would print "Hello," as we desire.

To print variables, we use **conversion specifications**, which begin with the `%`. These are just placeholders representing a value to be filled in during printed. More specifically, the `%` *specifies* how the value is *converted* from its internal binary form to characters. For instance, the conversion specification `%d` specifies that `printf` is to convert an `int` value from binary to a string of decimal digits. In summary,

- `%d` for integers,
- `%c` for chars,
- `%f` for floats,
- `%s` for strings (null-terminated char array)
- `%x` for hexadecimal form
- `%e` for exponential form
- `%u` for unsigned integer.

For example, the following code segment will print `i = 10`.

Listing 3: Printing a Variable

```
#include <stdio.h>
int main() {
    int i = 10;
    printf("i = %d\n", i);
    return 0;
}
```

`scanf()` is used for user input and it works similarly. The introduce the **address** operator, which is denoted by a `&`. The address operator is a unitary operator which, as its name specifies, returns the memory address of the variable on which it is acting on. When `scanf()` is called, it starts processing the information in the inputted string, from left to right. For each conversion specification in the format string, `scanf()` attempts to locate an item of the appropriate type in the input data, skipping any blank space if necessary.

Here's an example:

Listing 4: Reading Variables

```
#include <stdio.h>
int main() {
    int i, j;
    float x, y;
    scanf("%d%d%f%f, &i, &j, &x, &y");
}
```

Now suppose that the user enters the line

```
1 -20 .3 -4.0e3.
```

The code above will convert its characters to the numbers they represent and assign the values $1, -20, 0.3, -4000.0$ to the four variables.

Finally, one should keep in mind that `char` variables are actually just integers that map to an ASCII character. So something like `printf("%c", 65)` works completely fine; it prints the character `A`.

There are two important things that one should check when using `scanf()` and `printf()`:

1. Check that the conversion specifications match the number of input variables and that each conversion is appropriate for the corresponding variable (as should also be done with `printf()`. Since the compiler doesn't necessarily have to check for mismatches, there won't be any warning.
2. Another trap involves the `&` symbol, which should precede each variable in a `scanf` call. Forgetting to put it can lead to unpredictable results. It is wrong to use the address-of operator in a printf() statement.

A **segmentation fault** error occurs when the program attempts to access an area of memory that it should not be accessing. Why is it called a segmentation fault? Because the content of memory at the time of crash is stored into a **core file.**

We can get segmentation faults when using `scanf()` or `printf()` if we try to read into or print some variable that we don't have access to.

Control Statements

C has `if/else, for, do-while,` and `switch` statements, just like in Java. But due to the compiler flags in our submit server, we won't be allowed to declare variables in the `for loop` header.

There's also `break` and `continue`, but they are bad practice and shouldn't be used often.

Functions

C functions have the following format to create a function `returnType functionName(parameter list) { ... }` Just like in Java, to call a function, we just write `functionName(argument list);`.

However, if the function appears after the main, then we need to do something called **function prototyping**, which is just declaring the function before the main. This isn't necessary if we implement the function before the main, though. Function prototypes don't actually need the name of the variable, but it's easier to read with them.

In C, variables are passed in **by value**. This is the same as Java. Some other things that are similar/different from Java include:

- C supports recursion.
- C does **not** support function overloading. In particular, we can point out that `printf` and `scanf` are not overloaded functions; they refer to the same function!

4 Monday, June 3, 2019

The sizeof Operator

Before we talk about pointers, first we need to talk about the `sizeof` operator. The `sizeof` is a unitary operator tells us how many bytes are associated with a particular entity. This is an important operator when we're doing dynamic memory allocation. For instance, suppose we don't know how much memory to allocate when we're storing 10 integers. Then, we can do something like 10·`sizeof`

It is important to note that the `sizeof` operator does **not** evaluate the expression; for instance, doing something like `sizeof(x++)` will not increment `x`. It just looks at the type of what's inside.

Introduction to Pointers

A pointer is declared using the `*` symbol, right before the variable name. Consider the following code example:

Listing 5: Pointers Example

```
#include <stdio.h>
int main() {
    int y = 5;
    int *p;
}
```

Here, y is a standard integer variable, holding the value 5. By contrast, p is a pointer variable whose value is garbage. But each of these don't only have a value – they also have a **memory address**, which can also be represented by an integer. For example, the memory address of y might be 2000; p doesn't have a memory address yet. A program refers to a block of memory using the address of the first byte in the block.

Now let's say we add another line of code:

Listing 6: Pointers Example

```
#include <stdio.h>
int main() {
    int y = 5;
    int *p;
    p = &y;
}
```

Recall that the `&` symbol is the address-of operator. So at this point, p stores the memory address of y, namely, 2000. Now, we can do something like `printf("%d", *p)`, like we're used to. Also, whenever we change `*p`, we also change the value of y. **In summary, a pointer is a variable that stores a memory address.**

Why do we need the type when we declare a pointer variable? We need to know the number of bytes to grab. Since it's an integer here, we know to grab four bytes.

Now what if we want to read in the pointer using `scanf`? Then we don't need to use the `&` operator on the pointer – the pointer already refers to a memory address! To make this more clear, consider the following code:

Listing 7: Pointers Example

```
#include <stdio.h>
int main() {
    int age, values_read;
    int *age_ptr = &age;
    printf("Enter your age and salary");
    scanf("%d %f", age_ptr, &salary);
}
```

When we're taking in `salary`, we need to use the `&` operator since we want to retrieve the address. By contrast, we don't need the `&` operator for `age_ptr` since it already stores a memory address.

Pointers as Parameters

Recall that parameters in C are passed **by value**. To demonstrate this, consider the following code example:

Listing 8: Variables Passed by Value

```
#include <stdio.h>
int main() {
    int y = 7;
    f(y);
}

void f(int x) {
    x = 200;
}
```

When the code above is executed, the value of y doesn't change – we're passing a copy of y into the function. That is, the value of y is 7 even after Line 4 executes.

Now consider the following function `wrong_swap` below:

Listing 9: Variables Passed by Value

```
void wrong_swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int x = 2, y = 3;
    wrong_swap(x, y);
}
```

When the function terminates, the variables a and b are destroyed. The variables x and y **are not** swapped. The reason why is, again, because parameters are passed by value in C. So how can we swap variables, if we're only returning one value? This can be done with pointers, where the same idea of passing-by-value holds. Here's the correct way to swap —

Listing 10: Variables Passed by Value

```
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 2, y = 3;
    int p = &x;
    int q = &y;
    swap(p, q);
}
```

This is the same idea, but why does it work? Because we can dereference the pointer. We're note actually changing x and y – we're changing their memory addresses. So, this works.

5 Tuesday, June 4, 2019

The **comma operator** in C is used to separate expressions. It's a binary operator that evaluates its first operand and discards the result. It then evaluates the second operand and returns this value. For instance, `y = (3, 4);` is a valid expression, which assigns the value 4 to `y`.

Identifier Scopes

There are two main types of scopes in C:

- The **block scope** contains variables declared inside a block, and it is only visible within the block. They do not exist outside of the block.
- The **file scope** contains identifiers declared outside of any block; it is visible everywhere in the file **after** the declaration.

In the heap segment, text and data are constant from start to the end of the program. Execution follows the text segment of the memory. The data section contains global and static variables. Finally, the stack stores local variables and function parameters. There's some extra space in the heap which is used for dynamic memory allocation. The stack and heap grow in opposite directions, which is convenient to prevent overlapping. The heap goes up, and the stack goes down.

There are two types of storage types:

- **Automatic storage** occurs when the variable is transient. That is, after some time, it is no longer returned (e.g. when a function returns).
- **Static storage** occurs when the variable exists throughout the entire life of the program. Global variables have this kind of storage, and initialization to static variables only occur once.

You can make a block-scoped variable static, which would be important when you're counting the number of times a function executes.

A **linkage** is a property of an identifier that determines if multiple declarations of that identifier refer to the same object.

There are two main types of linkage that we should know about:

1. **Static linkage** is performed in the final step of compilation; it is fast, and it can be referenced from anywhere within the same file.
2. **Dynamic linkage** is performed during runtime at the cost of running slower.

6 Wednesday, June 5, 2019

Invalid Uses of Pointers

Consider the following code segment:

Listing 11: Incorrect Pointer Usage

```c
#include <stdio.h>
int main() {
    int *p;
    *p = 200; /* This is wrong! */
    printf("The value is %d\n", *p);
    return 0;
}
```

This is wrong, and it might generate a segmentation fault error. Why? We need p to be associated with an area of memory that is valid.

A quick fix is to initialize a variable, and assign p to the memory address of that variable. For example, the code segment below is correct, and it will print 200.

Listing 12: Correct Pointer Usage

```c
#include <stdio.h>
int main() {
    int *p;
    int x;
    p = &x; /* This is correct! */
    *p = 200;
    printf("The value is %d\n", *p);
    return 0;
}
```

The first code segment doesn't work correctly because the pointer is not initialized. Pretty much, we've created a pointer to "anywhere you want," which can be the address of some other variable, or some nonexistent memory.

When you have a program in C, there are four areas of memory: the **stack**, **heap**, **data**, and **code**. If some amount of memory is allocated for a function process, that memory becomes deallocated after the function is finished. So, we don't want to be messing with memory that no longer exists. For instance, the following code example is bad:

Listing 13: Incorrect Pointer Usage

```c
#include <stdio.h>
int* process() {
    int x = 10;
    int *p = &x;
    return p; /* This is bad - we're returning a
        pointer to some area that no longer exists! */
}
```

Even if the program seems to work, the local variable disappears – the space for it is gone, and we're not supposed to be messing with the memory that it used to be in.

Remark 6.1. We can print the memory address of a pointer using `printf` with the format specifier `%p`.

Null Pointers

The **null pointer** is a special pointer that points to the address 0, where nothing is allowed to be accessed. It's analogous to Java's null, except we use `NULL` rather than `null`.

You can assign null to any kind of pointer variable, and we also need to check if they're null prior to derefering them; using a simple `if (p != null)` conditional works.

Also, `null`'s numeric value is equal to zero, so a conditional statement with them will not execute.

Introduction to Arrays

Arrays are a bit different in C when compared to arrays in Java. In C, an **array** is just a chunk of bytes, one after another. We can declare an array of integers doing something like `int a[3]`, and indexing works the same as Java (starting at zero). Note that when we make the declaration `int a[3]`, the default elements are **not** zero (like in Java); instead, they are all garbage values. Also, you can't use a variable to declare the size of an array, but you can use it for indexing.

Note that an array is **not** an object, meaning that things like `a.length` don't exist. We need to keep track of the length ourselves. This can often be done with **constants**, which start with the `const` keyword. For now, we assume arrays are not dynamic in terms of their space.

If the array has three elements, then the size of the array is actually 12 bytes (four bytes per integer). We can use the operator `sizeof`, and something like `sizeof(a)` will return 12.

Some examples of array declarations are as follows:

- `int a[3] = {10, 20, 30};` will declare an array `a` of length three, with the three elements listed.
- `char b[] = {'A', 'B', 'C'}` will declare an array with size 3 with the provided elements. Note that we don't need to specify the length when we're initializing by list.
- `float c[4] = {1.5}` will declare an array of size 4 with first element equal to 1.5. The other elements will equal 0. This is really convenient because we can do something like `int a[3] = {0};` to initialize our array of length three to have all zero elements.

Arrays as Parameters

Recall that everything in C is passed by value.

7 Monday, June 10, 2019

The `pushd` and `popd` commands in Unix can be used to work with a directory stack. The command `pushd` pushes a directory on top of the directory stack, and the `popd` command returns to the path at the top of the stack.

The `history` command prints your most recent commands.

Pointing to a Local Variable

When working with pointers, you shouldn't return the address of a local variable. Consider the following code segment:

Listing 14: Incorrect Pointer Usage

```
   int* get_value-wrong() {
       int x = 20;
       return &x;
   }
5
   int add_value(int y) {
       int a = 99;
       return a + y;
   }
10
   int main(void) {
       int *a, b;

       a = get_value_wrong();
15     printf("First result %d\n", *a);

       b = add_value(7);
       printf("Second value %d\n", b);

20     printf("First result (changed?) %d\n", *a);
   }
```

The first print statement on Line 15 will work fine; it'll print out 20 as we'd want it to. This first part might seem counterintuitive because we usually think that the memory gets "thrown away" after the function finishes execution. In reality, this isn't what happens – the stack pointer just moves down, below the local variable. This tells our computer that the previously occupied area of memory is now available for reuse. In our case, the space occupied by the integer `x` will now be available for reuse.

But then, after we call the `add_value` function, the first result will have changed. Since we're declaring another local variable of the same type (integer), the same space that was previously being used will be filled for the second function call. The space that was previously holding the number 20 will now hold 99. Once again, after the function finishes execution, the stack pointer moves below the 99 again (but it does not disappear!).

And so, the print statement on Line 18 prints 106, and the print statement on Line 20 will print out 99.

String Comparison

To compare strings, we use the `strcmp` function, which is built in `string.h` library. The function header is as follows:

```
int strcmp(const char *s1, const char *s2);
```

Pretty much, it takes in two strings `s1` and `s2`, and it returns a negative number if `s1` is (lexicographically) less than `s2`, the integer 0 if they're (lexicographically) equal, and a positive number otherwise. Note that the `const` in the parameter list of `strcmp` indicates that the data of `s1` and `s2` can't be changed (this makes sense because changing the strings isn't necessary for comparison).

Remark 7.1. This functionality is pretty much the same as `compareTo` in Java

Here's one way to implement the `strcmp` function. Nelson says we should know this implementation for the exam.

Listing 15: String Comparison Implementation

```
int strcmp(const char *s1, const char *s2) {
    int i;
    for (i = 0; s1[i] && s2[i]; i++) {
        if (s1[i] != s2[i]) {
            break; /* Gets us out of the for loop */
        }
    }
    return s1[i] - s2[i];
}
```

- The for loop iterates until we reach the end of either string (the Boolean expression `s1[i] && s2[i]` is false only if we hit a null character in either string) or if we hit two characters that are different (this is what the conditional inside the loop does).
- Once we're at the differing character, we can just return their difference.

Note that the above implementation uses the fact that the null character has numeric value 0. This allows the code above to take care of the cases in which one string is shorter than the other.

Copying Strings

To copy a string, we use the `strcpy()` funcion, which is built into the `string.h` library. The header is as follows:

```
char* strcpy(char *dest, const char *src).
```

The function copies the string in `src` to the string in `dest`. After successful completion, the function returns a pointer to the destination string.

The danger with `strcpy()` is that it doesn't specify the size of the destination array, which can lead to a **buffer overflow error**. This type of error occurs when you put more data into a fixed-length buffer. The extra information has to go somewhere, and it can overflow into adjacent memory space, which corrupts other data.

Here's an implementation of the function `strcpy()` function:

Listing 16: String Copy Implementation

```
int strcpy(const *dest, const char *src) {
    int i = 0;
    while (src[i]) {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';
}
```

- We are copying the source into the destination.
- The while loop executes until we hit the null character in the source.
- We copy the first, second, third.... character into the destination.
- When the while loop executes, i is equal to the position where source has a null character, so we need to add that into the destination string.

Again, take note of the parameters. The string `dest` isn't constant because we're modifying it.

String Literals

The declarations `char name[] = "Mary"` and `char *name = "Mary"` are not the same. The first declaration is an array, which is what one should use if they're planning to change the value of the name from Mary to something else. On the other hand, the second declaration declares name as a pointer to a string literal. This should instead be declared as `const char* name = "Mary"`.

Void Pointers

A **void pointer** or **generic pointer** (they are the same) is a special pointer that's used to hold memory addresses of data when you don't know its type. They are used by functions (like C's built-in quicksort function) when you don't know the type of data you're dealing with. Void points are declared by simply replacing the type of a normal pointer with the void keyword. So, `void *ptr` would declare `ptr` to be a void pointer.

A void pointer can point to any type. So, you would be able to do `ptr = &someInt` or `ptr = &someChar`, etc. But integers, floats, and characters occupy different amount of space. So how does this work? The key is to note that pointer variables store the address of the first byte.

Note that **a void pointer cannot be dereference directly**. Dereferencing requires casting because the pointer needs to know how many bytes to grab. It's up to the user to make sure that the void pointer is casted right. So if you read in a float value into `ptr`, the statement `printf("V1: %f\n", * (float *) v_ptr)` would print the entity at the address stored in `v_ptr`. Note how this print statement has two asterick symbols: one is used in the cast, whereas the other is used for dereferencing.

It's also a good idea to use type casting when you're doing pointer arithmetic with void pointers. For example, consider the following code segment:

Listing 17: Bad Void Pointer Arithmetic

```
int one_d[5] = {12, 19, 25, 34, 46}, i;
void *vp = one_d;

printf("%d", one_d + 1); // bad
```

You might want to print the second element of the array with the above code. But this won't work. Adding a number to a memory address works by shifting logical units. However, these logical units are dependent on the type being worked with. Changing the fourth line to `printf("%d", (int *) one_d + 1)` would fix this.

The value of a void pointer can be assigned to integer/float/other pointer variables **without a cast**. For example, if we have a float pointer `f_ptr` and a void pointer `v_ptr`, the statement `f_ptr = v_ptr` works perfectly fine because it's just specifying how many bytes to grab.

Pointers to Pointers

You can have pointers to pointers (and even pointers to those pointers). The number of astericks indicates the degree-of-separation from the original variable. For instance, `int **p2` is a pointer to a pointer. Once `p2` has properly been initialized, we can do double dereferencing by typing `** p_{2}` to get the value of the original variable.

When do we use pointers to pointers? Consider a function we're writing that needs to modify a pointer. This would need to be implemented by taking in a pointer to a pointer;

8 Wednesday, June 12, 2019

The `grep` command in Unix looks for a pattern in a file. The general syntax is `grep [pattern] [file_name]`. So for example, if you wanted to find all instances of "cheese" in "homework.c," you could execute `grep cheese homework.c` to get this result (note how there aren't any quotes).

Command Line Parameters

So far, our main function's header has always been `int main(void)`, which indicates that the main method doesn't take in any parameters. However, it is possible to accept command line parameters into the main by instead using the header `int main(int argc, char **argv)`. This second form allows us to access command line arguments as well as the number of arguments specified (arguments will be separated by spaces).

In summary, the two arguments that the `main` function accepts in this second formulation are

- `int argc`, which represents the number of arguments passed into the program when it's run. This number needs to be at least 1.
- `char **argv`, which is a pointer to a character pointer. We can alternatively replace `char **argv` with `char *argv[]`, which is an array of character pointers.

For instance, consider the following program:

Listing 18: Command Line Parameters

```
int main(int argc, char *argv[]) {
    /* Processing */
    return 0;
}
```

We can pass in parameters through command line by typing, for example,

`./a.out hello my name is ekesh.`

The output is presented below:

```
argv[0]: ./a.out
argv[1]: hello
argv[2]: my
argv[3]: name
argv[4]: is
argv[5]: ekesh
```

Note how even `./a.out` counts as one of the strings processed. If we don't want this to happen, we can just treat the 0th index in the array as a sentinel. Also, keep note that !argv[i]! is a string. If we want to use a passed in value in, say, a loop, then we need to use the `atoi()` function, which converts a string argument into an integer.

Two-Dimensional Arrays

When you're passing in a two-dimensional array into a function, the first array dimension (i.e. the number of rows) does not have to be specified. The second (and any subsequent parameters, if we're working with more than two dimensions) need to be specified.

So, for example, a function with header `void print(int arr[][n], int m)` would be fine, whereas something like `void print(int arr[][], int m)` wouldn't work.

Obviously, this would only work if the second dimension is fixed and isn't user-specified; this is a clear drawback.

Two-Dimensional Character Arrays

Consider a two-dimensional array of characters declared as follows: `char friends[100][81]`.

Typically, we can view a two-dimensional character array as a one-dimensional array of strings. For example, `char friends[100][81]` would store 100 strings, each of which have a maximum length of 80. We can then access the i^{th} friend stored in the array by standard one-dimensional array indexing, like `friends[0]`. However, it's up to the programmer to verify that the null character is present at the end of each row in the array.

Since two-dimensional arrays are stored in row-major order, executing `strcpy(a[0], "12345")` to copy a five-character-long string into an array with column-length less than 5 will still work; however, the "trailing characters" will go into the next row. There is no compilation error here, though.

The typedef Keyword

The **typedef** keyword is used in C to create an alias for another data type. The general syntax for declaring a typedef is `typedef [data_type] [new_name];` By convention, the `new_name` of a data type usually starts with a capital letter.

The main reasons why we use typedefs are to improve code readability and maintainability. As per convention, it's good to start `new_name` with a capital letter so that we can distinguish it from other types.

It is important to note that the `typedef` and `#define` preprocessor are not the same: the `#define` preprocessor works by by blindly substituting what we're defining, whereas `typedef` actually defines a new type. In fact, a typedef is not a preprocessor directive; **typedef is a compiler token**, and the preprocessor doesn't care about it at all.

An Exception to Typedef

An exception to the standard `typedef [data_type] [new_name]` syntax for defining a typedef is when we're dealing with arrays. Pretty much, if we're typedef'ing something to become an array, from what we've learned, we would expect to write something like `typedef int[30] MyArray`. However, **this is wrong**. The correct way to do this would be to write `typedef int MyArray[30]`; the size of the array comes after the `new_name` identifier. This is an exception, and `MyArray` will now represent an array of 30 integer elements.

This exception also applies to multi-dimensional arrays.

Structures

Defined in terms of Java, a structure is a class without methods and without private fields.

More formally, a **structure** is a user-defined data type which allows one to group items of possibly differing types into one single type.

The basic syntax for declaring a structure type is `struct [struct_tag] { [member_list] };` (note how there is a semicolon at the end). Conventionally, structures are typically declared at the top of a program, before the main. Conventionally, structure tags begin with a lowercase letter.

Suppose we are writing a program that involves computer graphics. We might want to have a structure to represent a pixel. This structure should abstract the basic details about a pixel, like its x and y-coordinates and its color. This can be done with the following code:

Listing 19: Structure Example

```
struct pixel {
    int x, y;
    char color;
};
```

Here, we've declared a structure with the tag "pixel," which contains an integer `x`, an integer `y`, and a character `color`.

Fields in structures cannot be initialized (so, it would be invalid to set `x` and `y` to 0 by default in the above example). Why can't we initialize fields in structs? Basically, when the structure is declared, there isn't any memory allocated for it (there's no reason to allocate memory yet – we don't even know if the program will ever use the structure). Memory is allocated only when variables are created, so there isn't any space to actually declare a variable yet.

Once we've declared the `pixel` struct, we can declare a variable `p1` of its type by typing `struct pixel p1;`. The members of `p1` can be accessed by using the period: `p1.x = 50` would set the x variable associated with `p1` equal to 50.

C also supports using an initialization list to initialize a structure. For example, we could write `p1 = {1, 2, 'r'}` in order to declare `x, y,` and `color` to $1, 2,$ and 'r' respectively. The order in which the variables are provided is the same order in which these variables are assigned values. If we don't assign all of the values, their default values will be assigned.

There aren't any conversion specifiers that allow us to directly print out all of the variables associated with a structure (side-note: this is called a **reflection**). If we want to do this, we need a conversion specifier for every variable in the structure.

A couple of other things to remember:

1. Structures can be assigned to each other. For instance, $a = b$ will compile, and it will assign all of the field values of b to the corresponding fields in a. This performs a shallow copy.
2. Structures cannot be compared. The line $a == b$ will not even compile. Even structures with the same fields in the same order aren't compatible.

Combining Typedefs with Structs

Following the `typedef [data_type] [new_name]` syntax for declaring a new data type, we can typedef a structure in order to get rid of the `struct` that's usually necessary when declaring a structure.

For example, consider the pixel example from above. It was necessary to write `struct pixel p1;` to declare a pixel. However, if we modify the code to what follows, we can instead write `Pixel p1;`.

Listing 20: Typedef'ing a Structure

```
typedef struct pixel {
    int x, y;
    char color;
} Pixel;
```

We usually typedef a structure for brevity and readability.

Pointers to Structures

Since everything is pass-by-value in C, when we pass in a struct as a parameter to a function, we'll have a copy of the structure with every value equal to the original value's corresponding fields. Like we'd expect, this would mean that changing the structure inside the function doesn't change the original structure outside of the function (i.e. a shallow copy is performed). Like always, if we want to modify the actual structure, we need a pointer to the structure.

When we're dealing with pointers to structures, there's an **arrow operator**, which is used to dereference a pointer to a structure. Going back to our pixel example, for instance, if we have the pointer `p1` defined as `Pixel * p1`, we can set the structure's associated value of x equal to 50 by writing `p1->x = 50`.

Why do we need the arrow operator? There's nothing wrong with writing `(*p1).x = 50` – it does the same thing. But, something like `*p1.x = 50` **does not work** for precedence reasons. Hence, having the arrow operator improves readability instead of having a lot of parentheses and astericks.

9 Monday, June 17, 2019

Exit Codes

An **exit code** is a value that is returned to the **shell**, which is responsible for reading and executing your code. By convention, when everything goes well, we return 0 (as we have been doing in all of our programs).

The header file `stdlib.h` contains a lot of preprocessor directives, which represent exit codes. For example, `EXIT_SUCCESS` and `EXIT_FAILURE` can be used when the program successfully executes or unsuccessfully executes (these would be used instead of having a line that says `return 0`). It turns out that `EXIT_SUCCESS` is actually a preprocessor directive for 0.

In order to use an exit code, we use the built-in `void exit(int status)` function. So, for example, we could replace `return 0` in the main with `exit(EXIT_SUCCESS)`, and it would mean the same thing. On the other hand, if `exit()` is used *outside* of the main, the program will terminate once it reaches that statement, while a return statement would bring us back to the main.

How do exit codes help us? After executing a program, we can type `echo $?` to check the previous command's exit code. This can be used in shell programming, where we are telling the actual shell what to do.

In addition to exit codes and return values, there are a few important functions that are used to produce **error messages**:

1. The function `void perror(const char *str)` is used to describe the last error encountered during a library function or system call. If a string is provided, that string will be printed prior to the default error description. The default description is generated by a global variable called `errno`, which comes from the `errno.h` header file (i.e. it is an integer-to-string mapping).
2. The `char *strerror(int errnum)` function returns a pointer to the textual representation of the current errno value.

Note that neither of these functions kill the program.

Text and Binary Streams

In C, most input and output is provided in the sequence of bytes, which is more commonly known as a **stream**. There are two types of streams: **text streams** and **binary streams**.

- Text streams consist of lines of text, each of which are terminated by the `\n` character. They can be opened in text editors.
- Binary streams consist of raw data; they require a special editor to open.

What are the advantages of one type of stream over another? When we're using text streams, we can easily debug the program (it's human-readable and doesn't require additional tools, while binary files do). On the other hand, text streams might not be a great idea for when we're modifying files a lot: changing even a single character requires re-reading the entire file. If we were using a binary stream, however, we could (in most cases) just change the relevant bytes.

Standard Input/Output

Now, we'll discuss how to read and write to files.

For files you want to read or write, we need a **file pointer**, declared like `FILE *fp`. Realistically, it isn't really important what the type `FILE` actually is – we can just think of it as some abstract data structure which permits us to perform file I/O operations.

Performing file I/O operations has three key steps:

1. Open the file
2. Perform any processing
3. Close the file

To open the file, we use the `fopen` command, whose declaration is as follows: `FILE *fopen(const char *filename, const char *mode)` . Note that the function returns a file pointer, which we'll set our pointer equal to. If there's any error in opening the file, `fopen` will return `NULL`.

The `filename` parameter is a string, which holds the name of the file on the disk (including a path if necessary), and the `mode` is another string, which represents *how* we want to open the file. In this class, the file will be opened with mode equal to "r" (for reading) or "w" (for writing). Another mode is "a", which lets us append to a file, without losing the rest of its contents.

Once we've opened the file, we're ready for processing. If we're reading the file, we can use the `fgets()` function, whose declaration is specified as follows: `char *fgets(char *str, int n, FILE *stream)`. The parameter `str` in `fgets` stores the line read by the function, and it stops reading until either `n` characters have been read, or a `\n` character is encountered. Note that this `\n` character is also stored as a part of the out parameter. If there are any errors, the function returns `NULL`.

If we're writing the file, we can use the `fputs()` function, whose declaration is the following: `int fputs(const char *str, FILE *stream)`. It places the string `str` into the file `stream`. The function returns a non-negative integer upon success.

Finally, we need to close the file. This requires use of the `fclose()` function, whose declaration is as follows: `int fclose(FILE *stream)`. The function returns 0 upon success, and it signals that we are done processing the file.

The three key steps of file I/O operations are captured with the following code segment:

Listing 21: Processing a File

```c
#include <stdlib.h>

#define MAX_LEN 80

int main() {
   FILE *input; /* does not need to be named input */
   char line[MAX_LEN + 1], filename[MAX_LEN + 1];

   printf("Input file name (e.g., data.txt): ");
   scanf("%s", filename);
   if ((input = fopen(filename, "r")) == NULL) {
      perror("error opening file");
      exit(EXIT_FAILURE);
   } else {
      while (fgets(line, MAX_LEN + 1, input) != NULL) {
         printf("%s", line);
      }
      fclose(input);
      exit(EXIT_SUCCESS);
   }
}
```

On Lines 9 and 10, the program prompts a file name, which is subsequently stored. Line 11 attempts to open the file; upon success, each line is processed and printed. If the file cannot be opened, an error message is printed, and an exit code is returned. Line 18 closes the input stream, and Line 19 returns a successful exit code. Note that when we're printing on Line 16, there's no `\n` necessary. When we perform `fgets()`, we've already stored the new-line character, so adding an additional `\n` will put two spaces between lines.

If we want formatted input and output, we can similarly use `fprintf()` and `fscanf()`.

Nelson says that, at this point, we should be able to write a C program that copies one file to another using command line arguments.

Every program has three defined streams: **standard input**, **standard output**, and **standard error**. We can use the the keyword `stdin` in place of a file pointer to read from the user's keyboard.

Like standard input, standard error is also printed to the screen. It is denoted by the built-in file pointer `sterr`, and it is helpful since it allows us to sort out our print statements, depending on whether a program executed successfully or not.

So, standard input and standard error are different files; however, they both map to the screen. To direct standard error, we can use `> &` in Unix.

The end of a file is denoted by an invisible **end of file** (EOF) character. There's a function with the header `int feof(FILE *fstream)` that checks whether the EOF file has been reached, after the file has been attempted to been open. We can manually enter the end-of-file character with our keyboard by entering `CTRL + D`. Also, `EOF` is a preprocessor directive, so we can use that in our conditionals.

It is important to note that we need to first attempt to read the file before using `feof()`.

10 Wednesday, June 19, 2019

In addition, it's important to remember that `fgets()` returns `NULL` after we've reached the end of file (i.e. we shouldn't be using `EOF` with it. On the other hand, it's fine to use `EOF` with `scanf` statements.

The scanf() Family

There are three functions in the `scanf()` family: `scanf()`, `fscanf()`, and `sscanf()`.

First, `fscanf()` has the header `int fscanf (FILE * stream, const char * format, [address of variables] )`. Pretty much, we take in a file pointer along with some format, and we read it into some variables. When processing files, we want to keep reading until we hit `EOF`.

The loop conditional `while(fscanf(input_stream, "%s%d", students_name, &id) != EOF)` allows us to process the lines in a file one-by-one until we hit the end of the file. Note, however, that there's an assumption that the lines of the file are formatted in the same way. It's a good idea to use `fscanf()` when the lines are inputted in a uniform manner.

The `sscanf()` function has the header `int sscanf (const char * s, const char * format, ...)`, which returns the number of variables successfully read from the input string `s`. This can be helpful when we've already stored the string (e.g. a line) to be processed.

Sometimes, it's really helpful to combine the use of `fgets()` and `sscanf()`. The former allows us to store the entire line, and the latter allows us to make sure that everything is formatted properly (by using its return value).

The printf() Family

- `printf()`, we've already seen.
- `fprintf()` has the header `int fprintf ( FILE * stream, const char * format, ... );`. It takes a stream, and it's analogous to `fscanf()`.
- `int sprintf(char *str, const char *format, ...)` prints into the string variable `str`; it's the analogue of `sscanf()`.

Dynamic Memory Allocation

Dynamic memory allocation allows us to allocate storage space while the program is running. Once we're done using this allocated memory, it's important to call the `free()` function to make that space available again. There are a few other important functions that help us with dynamic memory allocation, the first of which is `malloc()` (which is short for memory allocation).

The malloc function has the header `void* malloc(size_t size)`. The function takes in a size parameter, specifying how much space to allocate. It returns a void pointer pointing to where that space begins. The function returns `NULL` if the memory allocation fails.

As an example, consider the following code segment:

Listing 22: Malloc Example 1

```
#include <stdlib.h>  /* For malloc, EXIT_FAILURE,
    EXIT_SUCCESS */

int main(){
   int *ip, i, array_length = 3;

   /* Allocating space for an integer */
   ip = malloc(sizeof(int)); /* notice casting is not
       necessary */
   if (ip == NULL) {
      exit(EXIT_FAILURE);
   }
   *ip = 104;
   printf("Value assigned is %d\n", *ip);
   free(ip); /* deallocating memory */
}
```

We start with an integer pointer `ip`, and we make it point to some space of memory using `malloc`. Note how we've specified that this memory has enough space to store one integer. Thus, we can dereference the pointer and assign it to an integer, and everything works fine. Further, observe that there is no need to cast the void pointer to an integer pointer.

Once we've called `free()` on a dynamically allocated memory address, it's important that we don't access that memory location again. When a memory location has been freed, any pointers that used to point to it become **dangling pointers**, which shouldn't be used. Doing so could lead to a segmentation fault, so it can be helpful to set the previously used pointer equal to null.

So, what happens internally when we call `free()`? Essentially, the heap manager marks the bytes in the memory specified as available for use. We don't actually care about what the `free()` returns – to us, it just means that the memory is free again. When we don't free the memory we've used, it's called a **memory leak**, which is bad.

We can also allocate memory for an entire array using the following code segment:

Listing 23: Malloc Example 2

```
int main() {
   /* Allocating space for array */
   int *ip = malloc(sizeof(int) * array_length);
   int array_length = 3;
   if (ip == NULL) {
      exit(EXIT_FAILURE);
   }
   for (i = 0; i < array_length; i++){
      ip[i] = i * 3;
   }
   for (i = 0; i < array_length; i++){
      printf("%d ",ip[i]); /* notice using array
          indexing */
   }
   printf("\n");
   free(ip); /* deallocating memory */
}
```

There aren't really any new concepts in this code segment. It should be carefully noted, however, that we check if the pointer returned from malloc and calloc is null after each call. The output of the program is `0 3`

`6`, and the memory allocated for the array is freed after this is printed. It's important to call `free()` on a pointer that points to the *start* of the array that we've allocated, otherwise freeing won't work.

When we assign `malloc()` to a pointer, the value at the assigned memory location is garbage.

There's an alternative way to allocate memory: with the `calloc()` function. Unlike the `malloc()` function, `calloc()` takes in two parameters: its header is `void *calloc(size_t count, size_t obj_size)`, and it allocates `count` objects of size `obj_size` each. The function returns a pointer to the beginning of the memory address created. Also unlike malloc, the calloc function initializes all spaces to zero, which can save time depending on what we're doing.

Having introduced calloc, there are a lot of shortcuts we can take. For example, consider the statement `int **q = calloc(4, sizeof(int *))`, which allocates space for an array of four integer pointers. Also, since calloc automatically initializes its spaces to zero, all of these pointers are automatically set to null for us.

11 Friday, June 21, 2019

Recap of Dynamic Memory Allocation

Today, we will continue discussing dynamically allocated memory (i.e. memory that isn't allocated until the program starts running). What's another reason we use it? Sometimes, the size of a data structure isn't known until runtime (for example, suppose we want to initialize an array of size N, where N is a positive integer provided by the user). Also, Linked Lists will use dynamic memory allocation everytime we make a new node.

To recap, there are two memory management are library functions that are used to allocate memory dynamically: `malloc()` and `calloc()`.

1. The `void *malloc(size_t amount);` function allocates `amount` bytes (if available) from the heap and returns a void pointer to the beginning of it. Note that there cannot be any initialization of this space.
2. The `void *calloc(size_t count, size_t obj_size);` function allocates `count` objects of size `obj_size` each (if memory is available), and it returns a void pointer to the beginning of it. By default, all the space is initialized to zero.

Both `malloc()` and `calloc()` return `NULL` if the allocation fails.

A third memory management function is `void free(void * ptr)` – after this function is called, the memory pointed to by `ptr` is now available for reuse by the memory allocator. Something to take note of is that `free()` has to be the same pointer that was returned from `malloc()` or `calloc()` – we can't call `free()` in the middle of the area that we allocated. Also, after the pointer is freed, the pointer becomes a dangling pointer, so we shouldn't dereference it.

Good programming practice should exhibit a one-to-one mapping between the number of calls to `malloc()` and `calloc()` and the number of calls to `free()`. It's also good to know that calling `free()` on null is harmless – you don't need any null checks for calling `free()`. Doing `free(NULL)` is completely harmless. Also, as one would expect, we can't free pointers whose data is constant (i.e. we can't free a pointer declared as `const char *p`).

We should only call `free()` on a pointer once. Why? When we call malloc or calloc, we're telling our computer that we want to reserve some memory just for that pointer. When we subsequently call free, we're telling the computer that we don't need that space anymore; however, the pointer still points to that memory address. If we invoke `free()` a second time, we're not freeing the previous data, but possibly some new data that resides at that memory address.

Dynamically Allocated Structures

Consider the following lecture example:

Listing 24: Dynamically Allocated Structure

```c
#include <stdio.h>
#include <stdlib.h>  /* For malloc, EXIT_FAILURE,
    EXIT_SUCCESS */

/* Notice tag and typedef identifier can be the same */
typedef struct Student {
   char *name;
   int age;
} Student;
```

```
10
   int main(){
       Student *student;
       int length;

15     /* Allocating space for a Student structure */
       student = malloc(sizeof(Student));
       if (student == NULL) { exit(EXIT_FAILURE); }

       /* Allocating space for name */
20     printf("Enter number of characters in your name: ");
       scanf("%d", &length);
       student->name = malloc(length + 1);

       /* Reading name and age */
25     printf("Enter your name: ");
       scanf("%s", student->name);
       printf("Enter your age: ");
       scanf("%d", &student->age);

30     /* Feedback */
       printf("Your name is %s and your age is %d\n",
           student->name, student->age);

       /* Freeing memory */
       /* We must free name first */
35     free(student->name);
       free(student);


       return EXIT_SUCCESS;
40 }
```

On Line 16, we dynamically allocate space for `student` to become a `Student` type. Immediately after, we check if this allocation was successful (i.e. check if `student == NULL` holds), and we continue if it was. Next, we allocate space for `name`, depending on how many characters the user requires (which is provided through stdin). Finally, we store the student's age (also from stdin), we print the student's information, and we free the space we allocated.

Some key things to note:

- Why did we allocate space for `student` and `name` but not for `age`? Because `age` isn't a pointer, so we already get space for `age` after we dynamically allocate space for `student`.
- Why do we free `name` before `student`? If we freed `student` first, then `student` becomes a dangling pointer. So, we shouldn't be accessing `student->name` afterwards (accessing the name depends on the existence of a student, but the student's existence doesn't depend on its name).

Pointer Aliases

We can have two pointers point to the same dynamically allocated memory area. For instance, consider the following code segment:

Listing 25: Pointer Aliases and Dynamic Memory Allocation

```
int *p, *q; /* Declare two integer pointers */
p = malloc(sizeof(int)); /* Dynamically allocate
    memory */
if (p != NULL) { /* Make sure allocation succeeded. */
```

```
    *p = 99; /* Dereference p; set its entity to 99. */
    q = p; /* q points to the same memory address as p
        */
    free(p) /* BOTH p and q are dangling pointers now
        */
    *q = 42; /* This is WRONG. */
}
```

In summary, if we free one pointer pointing to a memory address, *every* pointer pointing to that same memory address becomes a dangling pointer.

Common Errors

The common errors of dynamically allocating memory comes are summarized below:

1. Dereferencing pointers to freed space. We've already discussed this.
2. Forgetting to check if malloc or calloc returned null (i.e. the dynamic memory allocation was unsuccessful).
3. Forgetting to initialize the memory `malloc()` returns (`calloc()` automatically does this for us).
4. Attempting to free non-heap memory (i.e. we should *only* be calling `free()` on dynamically allocated memory).
5. Memory leaks: forgetting to call `free()` on dynamically allocated memory.
6. Calling `free()` twice on a pointer.

Using valgrind can help identify these errors.

12 Monday, June 24, 2019

Linked Lists

Like inner classes in Java, C structures can have pointers to structures of the same type. This allows us to define a Linked List's node as follows:

Listing 26: Linked List Node

```c
typedef struct node {
    int data;
    struct node *next;
} Node;
```

Note that the structure tag inside of the typedef is necessary here since we have a self-reference inside our definition of a node. Since pointers have the same size, the compiler won't have an issue in determining the size of `next`. However, this wouldn't be possible if `node` weren't a pointer.

In order to represent a Linked List, we declare a pointer to the head by typing something like `Node *head`. The pointer allows us to modify the Linked List inside of various functions. So, functions that modify the actual Linked List have function prototypes that take in a double pointer Node type.

For instance, a function which instantiates the Linked List might have header `void create_list(Node **head)`, and its body would simply be `*head = NULL`. In a similar manner, we'd need to pass in a double pointer to a node in order to add an element to the list (pretty much, we need a double pointer whenever we're modifying the list).

There are two noteworthy types of Linked Lists traversal:

- The "**print traversal**," which works by moving a current pointer forward after some processing:

  Listing 27: Print Traversal

  ```c
  /* The print traversal */
  Node curr = head;
  while (curr != NULL) {
      /* Processing */
      curr = curr->next;
  }
  ```

  If we need to visit every node in the list and do some processing, we perform the print traversal.

- The "**Tom and Jerry traversal**", which works with two adjacent pointers. The left-most pointer allows us to look back and access previous elements. The two pointers move in parallel.

  Listing 28: Tom and Jerry Traversal

  ```c
  /* The Tom and Jerry Traversal */
  prev = NULL;
  curr = head;
  while (curr != NULL) {
      /* Processing */
      prev = curr;
      curr = curr->next;
  }
  ```

This traversal allows us to add a node before or after any element.

Here's an example of an `insert()` function which maintains a sorted ordering in a Linked List:

Listing 29: Linked List Insertion

```c
int insert(Node **head, int new_value) {
   Node *current = *head, *prev = NULL, *new_item;

   while (current != NULL && new_value >
       current->data) {
      prev = current;
      current = current->next;
   }

   new_item = malloc(sizeof(Node));
   if (new_item == NULL) {
      return 0;
   }
   new_item->data = new_value;
   new_item->next = current;
   if (prev == NULL) { /* inserting at the beginning */
      *head = new_item;
   } else {
      prev->next = new_item;
   }

   return 1;
}
```

Note that it would be incorrect to pass just a single pointer to `head` since the Linked List won't get modified.

When we construct a Linked List, we will want to add nodes one by one. Creating a node requires three key steps:

1. Allocate memory for the node.
2. Store data in the node.
3. Insert the node into the list.

When we create a node, we write something like `struct node *new_node` followed by the command `new_node = malloc(sizeof(struct node)))`. Note that something like `new_node = malloc(sizeof(new_node))` would be incorrect as `new_node` is a pointer variable (this command would allocate 8 bytes rather than the actual amount of space needed). To change the data of a node, we use the arrow operator by doing something like `new_node->data = 5`.

Something important to keep in mind is that, in our model, `head` itself is not a node. It's simply a pointer to the first node in our Linked List.

Deletion from a Linked List involves calling `free()` on the deleted node. Here is an implementation of the `delete()` method:

Listing 30: Linked List Deletion

```c
int delete(Node **head, int value) {
   Node *prev = NULL, *current = *head;

   while (current != NULL && current->data != value) {
      prev = current;
```

```
        current = current->next;
    }
    if (current == NULL){
        return 0;  /* not found */
    }
    if (prev == NULL) {
        *head = current->next;  /* deleted first item */
    } else {
        prev->next = current->next;
    }
    free(current);

    return 1;
}
```

13 Wednesday, June 26, 2019

Midterm 1 will be graded by sometime next week.

Operating on Memory Blocks

Like `strcpy()` operates on strings, there is an analogous function that operates on entire blocks of memory. Namely, this function is memcpy, which has the following declaration: `void *memcpy(void *dst, void *src, size_t n)`. It works pretty much the same with `strncpy()` works: it takes a destination, source, and the number of bytes to copy. A prerequisite to using `memcpy()` is that `dst` and `src` cannot be overlapping pieces of memory (for instance, they shouldn't be pointing to different portions of the same array).

So, what do we do if we need to quickly copy overlapping memory areas? We can use the `memmove()` function, which has the declaration `void *memmove(void *dst, void *src, size_t n)`, but it's not as efficient.

A possible implementation of `memcpy()` is presented below:

Listing 31: Memset and Memcpy

```c
void *memcpy(void *dst, void *src, size_t n) {
    char *dp = dst;
    char *sp = src;

    while (dp - (char *) dst < n) {
        *dp++ = *sp++;
    }

    return dst;
}
```

Note how the function utilizes void pointers, which means that we can pass any type of pointer into the function. We cast to `char *` so that we can iterate byte-by-byte.

The function `memcpy()` is really helpful. So far, to make copies of arrays of structures, we've been iterating through every index with a for loop. But now, we can do all of this with a single statement. Nelson says that this is important to know for exams.

Function Pointers

In the same way that we can have pointers to variables, we can also have pointers to functions. **Function pointers** store the memory address of a function, which is possible as each function is located somewhere in memory. Similar to an array, the name of a function can be seen as a variable that stores the function's address.

We can use function pointers in pretty much the same way that we use normal pointers.

A procedure to write the declaration of a function pointer is to first just write the function prototype, add parentheses around the function name ("hug the function"), and add an asterisk to the start of the newly added parentheses ("kiss the function"). The following example illustrates the basics:

Listing 32: Introduction to Function Pointers

```
   void print_decimal(unsigned int i) {
      printf("%u\n", i);
   }

 5 void print_hex(unsigned int i) {
      printf("%x\n", i);
   }

   void print_octal(unsigned int i) {
10    printf("%o\n", i);
   }

   int main() {
      /* The following is the declaration of */
15    /* fp as a function pointer variable   */
      void (*fp)(unsigned int);

      /* Below both & and * are optional */
      /* due to the use of the function */
20    /* call operator */

      fp = print_hex;
      fp(16);  /* prints "10" */
      fp = &print_octal;
25    fp(16);  /* prints "20" */
      fp = print_decimal;
      (*fp)(16);  /* prints "16" */

      return 0;
30 }
```

On Line 16, we declare a function pointer `fp` using the exact procedure previously described. The key thing to note there is that the parentheses around `*fp` indicates that `fp` is a pointer to a function, not a function that returns a pointer.

Once we've got a function pointer, we can assign it to the name of a function (just as we could do with arrays), and everything works fine. In fact, the C compiler even allows us to assign a function pointer to the *address* of a function (like on Line 24), or even dereference while calling the function (like Line 27), and everything will still work fine. However, the statements on Line 22 and 23 illustrate standard way to do this.

14 Thursday, June 27, 2019

Memcpy and Memset

The `void *memcpy(void * destination, const void * source, size_t num)` function is really similar to the `strcpy()` function. The function `memcpy()` is used to copy a specified number of bytes from one memory to another, whereas `strcpy()` copies the contents of one string into another. Also, `strcpy()` works exclusively with strings, whereas `memcpy()` works with any type of data.

Another function is `void * memset (void * ptr, int value, size_t num)`, which sets the first `num` bytes in the block of memory pointed to by `ptr`, and sets them all to `value`. For instance, if you have a character array `arr`, the statement `memset(arr, 'a', 8)` would set `arr` to have eight characters, each of which are `a`.

A more comprehensive example is provided below:

Listing 33: Memset and Memcpy

```
   #include <stdio.h>
   #include <string.h>

   #define MAX_LEN 80
 5 #define ROSTER_MAX_LEN 2

   typedef struct student {
      int id;
      char name[MAX_LEN + 1];
10 } Student;

   void print_roster(Student *roster, int length) {
      int i;

15    for (i = 0; i < length; i++) {
         printf("%d - %s\n", roster[i].id,
            roster[i].name);
      }
   }

20 int main() {
      Student roster[ROSTER_MAX_LEN] = {{10, "Kelly"},
                                   {20, "John"}};
      Student copy[ROSTER_MAX_LEN];
      char name[MAX_LEN + 1];
25
      print_roster(roster, ROSTER_MAX_LEN);
      memcpy(copy, roster, ROSTER_MAX_LEN *
          sizeof(Student));
      print_roster(copy, ROSTER_MAX_LEN);

30    /* memset */
      memset(name, 'a', MAX_LEN / 2);
      name[MAX_LEN / 2] = '\0';
      printf("%s\n", name);

35    return 0;
   }
```

Upon running the code, Line 27 copies the contents of `roster` to the destination `copy`. It's important to note that this isn't the same as making the two array pointers point to the same block of memory. Using `memcpy()`, we've created two distinct blocks of memory with the same contents. Hence, the print statements on Lines 26 and 28 print out the exact same thing. Subsequently, the `memset` call on Line 31 sets the first half of the elements of `name` equal to `'a'`. Consequently, the print statement on Line 33 prints fourty `a`'s.

We cannot copy overlapping memory areas when using `memcpy()` or `memset()`.

Searching Files with Grep

`grep` is a command-line utility that can be used in Unix to search for patterns of characters in a file. The general format for `grep` is `grep [target_string] [file_name]` Consider the following text file, called `information.txt`:

Listing 34: Text Sample

```
This project is about hashing,
files, structures,
pointers
and dynamic memory allocation (and more pointers).
```

If we were to type `grep point information.txt`, the shell would return the lines in which the string `point` is found (namely, the third and fourth lines). A useful flag that we can add is the `-n` flag, which returns the line numbers alongside the lines that are found.

Now, what if we want to search multiple files? Like other Unix commands, we can use the `*` wildcard. For instance, `grep -n point *` would print the lines and line numbers of every file in the current directory that contains the string `point`. To search recursively (in subdirectories), we need the `-r` flag.

Data Representation

Character Representation

The two most common formats of representing characters are listed below:

1. **ASCII** is the most commonly used format; the capital letters are assigned numbers from 65-90, and the lowercase letters are assigned letters from 97-122.

2. **Unicode** is another common format. It stands for Unicode Transformation Format, and there are a few different versions. UTF-32 allows us to represent any character in any language (used by the Government), UTF-16 is the most popular, and UTF-8 provides backwards compatibility with ASCII.

Integers

When we're representing unsigned integers, the representation is more straightforward - all of the numbers are stored using binary. Now, this works pretty easily for positive integers, but what if we want to represent a negative number? The solution comes using a convention called **two's complement**.

Under two's complement, the positive value of a number is just its binary representation with its leftmost bit equal to 0. To obtain a negative value, we invert all of the bits of the corresponding positive value, and we add 1. The eight bits `00000101` typically correspond to the decimal number +5. Under the two's complement convention, the number -5 can be represented by `11111011`.

Why do we use two's complement instead of, say, just adding an extra bit at the start or end to indicate the sign? The reason why we use two's complement over just having an additional sign-indicating bit is mostly for math-simplifying reasons[1].

With two's complement, The range of values that we can store with n bits ranges from -2^{n-1} to $2^{n-1} - 1$, inclusive.

[1] https://stackoverflow.com/questions/1125304/why-prefer-twos-complement-over-sign-and-magnitude-for-signed-numbers

Floats and Doubles

We can store floats and doubles by writing the value we're storing in the form

$$(-1)^s \cdot m \cdot r^e, \tag{1}$$

where s is a bit (either 0 if we're representing a positive number or 1 if we're representing a negative number) representing the sign of the number, m is a **mantissa**, r is the **radix** (base) we're working in, and e is an exponent to be determined.

This might seem confusing at first, but we can break this process down into one simple step: writing the number in scientific notation.

All of the variables in Equation (1) comes from writing our number in scientific notation. For example, if we want to store the decimal number 51.432, we can write it as $5.1432 \cdot 10^1$, and we can examine this expression to see that $m = 5.1432$, $r = 10$, $e = 1$, and $s = 0$.

Similarly, if we're storing the binary number 1001.1101, we would be able to write this as $1.0011101 \cdot 2^3$ and find $m = 1001.1101$, $r = 2$, $e = 3$, and $s = 0$.

The number of bits allocated for the radix, exponent, and mantissa are specified by the IEEE 754 floating point standard. To store a negative exponent, we add an **exponent bias** to the exponent e, which normalizes the number zero to all zeroes.

Imprecision with Real Numbers

Some real numbers, like $1/3 \approx 0.33333$, have infinitely long decimal representations. This can create complications when we're storing these numbers because we are trying to store an infinite decimal with finite space. In summary, some of these bits get cutoff, which can cause some small imprecisions when dealing with real numbers.

15 Monday, July 1, 2019

A **nibble** is a term used to describe four bits. A very important thing to remember from last lecture is that each hexadecimal digit is represented by a nibble.

Unix File Permissions

Unix file permissions are based on the octal system. Every file is associated with three entities: the user entity, the group entity, and the "other" entity. The user entity describes the file permission of yourself. The group entity describes a set of users on the same system. The "other" section describes everyone else.

The `chmod` command can be used in Unix to change permissions of a file. The general format of the command is `chmod [settings] [file_name]`. What we enter for the `[settings]` portion of the command is a three-digit octal number, specifying the permissions for each entity. The first digit corresponds to the specifications for the user, the second digit corresponds to the settings of the group, and the last digit is for the "other" entity.

So, how does it work? We convert each octal digit in `[settings]` to a three-digit binary number, which specifies whether the entity has read, write, and/or executing permissions, in that order.

As an example, suppose we execute `chmod 400 a.out`. The corresponding binary representation of `4` is `100`, meaning that the user (the person executing the command) will be able to read the file, but they will not be able to write to the file or execute it. Since the other two octal digits are `0`, the permissions of the group and "other" categories aren't modified.

How do we determine what the `[settings]` number should be? This is easy - just work backwards. Suppose we want the user and the group to be able to read and execute the file but not write to it. This corresponds to the three-digit binary number `101`, which has octal representation `5`. So, `chmod 550 a.out` does exactly what we want.

We can call `chmod` recursively on a directory using the `-r` flag.

Introduction to Assembly Language

Assembly is a low-level, readable translation of machine language. It is a programming language that we work with when we want to see what the computer is doing. There are many assembly languages out there - in this class, we will use AVR Assembly. We can generate the `.S` file corresponding to the assembly instructions of a `.c` file by compiling with the `-S` flag when using `avr-gcc`. This is not allowed for projects/exercises.

A computer consists of some memory (RAM), and a CPU. Inside of the CPU, there is an **arithmetic logic unit**, which is responsible for performing computations. Additionally, inside of the CPU, there are **registers**, which are fast-access locations that instructions use instead of storing all values in memory. Registers are all one byte. In AVR, there are 32 registers, labeled r0, r1, ..., r31. A computer also has a **program counter**, which is a register that specifies the next instruction to be executed. Finally, the computer has an **immediate**, which consists of constants that are in the instruction itself.

A program written in AVR assembly has four components to it. Firstly, there are **instructions**, which specify what the processor will execute. These instructions typically consist of a name, a list of registers, and sometimes a constant value. In addition, there are **labels**, which represent an address. Labels are typically denoted by some text, followed by a colon. They are also used to define functions. Finally, there are **assembler directives**, which controls where code and data are placed, as well as **comments**.

How does our assembly code become machine language? We use an **assembler** (analogous to a compiler) to produce the zeroes and ones associated with our instructions.

An Illustrative Example

To illustrate how a basic assembly program works, we'll first consider some logically equivalent code written in C:

Listing 35: C Program for Assembly Example

```c
#include <stdio.h>

#define LETTER_A 65
#define NEWLINE 10

char x = 6; /* We are not using it */

int main() {

   putchar(LETTER_A);
   putchar(NEWLINE);

   return 0;
}
```

Note that we don't actually use the variable `x` - it's just there so that we can see how to create global variables in assembly. What does this code segment do? It prints the letter `A` (which has ASCII value 65), and it subsequently prints a newline character (which has ASCII value 10).

The corresponding assembly program is presented below:

Listing 36: Assembly Example

```
;;; Organization of typical avr program (; used for
    comments (don't use #))

;;; Symbolic constants
        .set LETTER_A, 65       ; Constant for A
        .set NEWLINE, 10        ; Constant for \n

;;; Global data
        .data                   ; Begins data section.
x:      .byte 6                 ; Label ''x" stores 6

;;; Program code
        .text                   ;Directive to start code

.global main                    ; Makes main label
    externally available
main:
        ;; main function. Task is defined at this
            point. This function prints letter A
            followed by newline character. We need the
             newline to force the flushing of
            character A.

        call init_serial_stdio ; To call a function we
            use the call instruction

        ldi r24, LETTER_A    ; The ldi instruction
            loads a value into a register.
        clr r25 ; Sets high byte of putchar's integer
            argument to 0.
        call putchar ; Calls putchar to print the
            character
```

```
ldi r24, NEWLINE ; Initializes r24 with '\n'
    character ascii value
    clr r25 ; Sets high byte of putchar's integer
        argument to 0
call putchar

cli                      ; We need to stop the
    program. Relying on cli and sleep.
    sleep

    ret ; Adding this ret to show functions end
        with ret but this ret is unreachable
        (program already stopped)
```

Before we start analyzing this code, it should first be noted that lines beginning with dots are directives, lines ending with colons are labels, lines beginning with a semicolon are comments, and everything else is an instruction.

What observations can we make?

- Comments begin with a single semicolon, but we sometimes prefer to use more semicolons for sylistic reasons.
- Symbolic constants are defined with `.set` directive followed by a target text, a comma, and a replacement text. The target text never actually makes it to the machine code; it is processed by the assembler.
- By writing the `.data` directive, we indicate that what follows is a data section. We create labels by having some text, followed by a colon. Here, our label is `x`, which stores the memory address of the value 6. What follows `.byte` indicates the entity being stored at the memory address. This can be written in decimal (as it is), hexadecimal, or even binary.
- The `.text` directive indicates that we're done with our initial setup, and everything that follows is actual code. This is a very important directive to have.
- The `.global main` directive allows `main` to be called outside of the current file. Functions, including `main`, begin with labels. Also, functions end with `ret`.
- To call a function, we use the call instruction `call init_serial_stdio`. We will always call this function to permit us to use input and output.
- To print the ASCII value of 'A,' we need to first load a value into a register using `ldi`. In our program, we move 65 to register 24.
- After we load into register 24, we clear register 25 with `clr`. Why do we clear register 25? Because `putchar()` assumes that the value it will display can be found in registers 24 and 25. This is a rule defined by gcc. So we clear register 25 to contain no data.
- To flush the buffer, we load in the new line character to register 24, and we re-clear register 25 (in case `putchar` messed it up). Finally, we call `putchar` again, and we get our desired output.
- The `cli` and `sleep` functions are necessary to stop the program.

Side-note: we can't have floating-point types in AVR Assembly, but we can have integers, characters, and strings.

16 Tuesday, July 2, 2019

No discussion today. Today is day two of Assembly.

Data Space Instructions

In order to transfer the contents of a register back into memory (like a variable), we can use the `sts` instruction, which is short for "store to data space." Using `sts` in assembly is analogous to assigning a variable some value. On the other hand, we can use stored memory to write to a register using the `lds` instruction, which is short for "load direct from data space." The general syntax for `sts` is `sts [data destination], [register source]`. The general syntax for `lds` is `lds [register destination], [data source]`.

Another useful instruction is `inc`, which simply increments the contents of a register. As we'd expect, the syntax for this instruction is just `inc [register name]`.

Consider the following code segment, which illustrates all three of these instructions:

Listing 37: Storing and Loading to Data Spaces

```
   ;;; Global data
       .data
   a:  .byte 0x2
   b:  .byte 0b00000100
 5 c:  .byte 0

   ;;; Program code
       .text

10 .global main
   main:

       call init_serial_stdio
       lds     r18, a ; stores contents of a into r18.
15     lds     r19, b ; stores contents of b into r19.
       push    r19 ; saves value of r19 on the stack.
       add     r19, r18 ; adds contents of registers.
       sts     c, r19 ; stores contents of r19 into c.

20     lds r18, c ; stores contents of c into r19.
       inc r18 ; increments r18
       pop r19 ; restores r19
       inc r19 ; increments r19

25     cli ; stops the program
```

What's happening here?

On Lines 3, 4, and 5, we're just declaring three global variables: `a`, `b`, and `c`. Like we saw yesterday, the `.data` directive indicates that we're starting our data section, and the `.byte` directive indicates that the value that follows should be stored in the specified variable. In this case, we store `0x2` (hexadecimal for 2) in `a`, 100 (binary for 4) in `b`, and 0 in `c`. The reason why we're using different base systems is just to clearly convey that it is allowed.

All of our variables have been initialized, so we can begin writing our code (officially, this is indicated by the `.text` directive on Line 8). In our main, we see an example of `lds` in action: the instruction takes a register and some data, and it loads the contents of the data into the register. In this case, `r18` stores `2`, and `r19` stores `4`. The contents of `r19` are pushed onto the stack for safekeeping.

The `add` instruction on Line 17 stores the contents of `r19` and `r18` and stores the sum in `r19` (it overwrites the previous value, which is precisely why we pushed the original value onto the stack). On Line 18, the `sts` instruction is used to store the sum into variable `c`. This value is incremented by one, the original contents of `r19` are restored, and the contents of `r19` are incremented by one. The final value stored in `r18` is 7.

Instructions List

The list below summarizes some important instructions that we should become familiar with (we've already seen some of these):

- `ldi` initializes a register with a constant value. Its general syntax is `ldi [register] [constant]`. For instance, `ldi r24, 5` would set the contents of `r24` to 5. We can only load constants to the registers `r16`, ..., `r31` (with the exception of `clr`, which loads in 0).
- `lds` loads data from memory into a register — we saw this in the previous example.
- `sts` stores data from a register into memory — we also saw this in the previous example.
- `clr` clears the contents of a register. Its general syntax is `clr [register]`. After this instruction is executed, the contents of the register it was performed on becomes 0.
- `add` is used to add the contents of two registers. Its general syntax is `add [register1], [register2]`. After this instruction is executed, the contents of `register1` are replaced with `register1 + register2`.
- `inc` is used to increment the value of a register by one. We saw this in the previous example.
- `push` is used to push a register value onto the stack. Its syntax is `push [register]`.
- `pop` is used to move data from the top of the stack into another register. Its syntax is `pop [register]`. Note that you don't have to pop to the same register that was pushed. There should be a one-to-one correspondence between `pop` and `push` instructions.
- `call` is used to call a function, as we saw with the putchar example yesterday.
- `ret` is used to return from a function.
- `nop` is short for "no operation," and it does nothing. Why does it exist? To make our processor wait and do nothing.
- `mov` has syntax `mov [destination register], [source register]`. It copies the contents of `source register` into `destination register`. Note that the name "move" is slightly misleading here - the contents of `source register` aren't moving anywhere - they are being copied, not moved.

One way to remember the syntax of some of these instructions is to note that, when we're writing to a register, the first register is always the target register where the result is being written to.

Caller/Callee Saving

The 32 registers that we use in Assembly are all global registers. What this means is that these registers are shared among every function, including the main. To illustrate why this matters, suppose we were to store 20 in register `r19` in the main. We then decide to call some function, which stores 100 in the register `r19` as a part of its processing. This change will also be reflected in the main (and everywhere else in the program).

Registers are shared across the entire program, which makes things more complicated. How can we avoid overwriting registers used by functions? The solution to this problem comes from two protocols that we use:

1. **Caller-saved protocol**: When writing a function, we assume the programmer who called the function has already saved the contents of registers from `r18` to `r27`, `r30`, and `r31`. So, the function writer should be operating on only these registers, and it's up to the programmer to have already saved anything of importance in these registers. We are expected to only operate on these registers when writing our functions as well.

2. **Callee-saved protocol**: If a function writer wants to use registers `r2` to `r17`, `r28`, or `r29` inside of their function, they need to be saving the registers prior to using them. Furthermore, the contents of these registers need to be restored back to their original values before leaving the function.

What do these protocols mean to us? It means that there is less work for both the function writer and the function caller. On one hand, the function writer can use several registers without worrying about overwriting important data. On the other hand, the function caller has a set of registers where they can store data and call a function with certainty that their data will not be overriden.

To better understand these two protocols, consider a blackboard that is shared among different professors. The blackboard follows the caller-saved protocol: a professor who enters the classroom to teach is allowed to erase whatever is on the blackboard. It is assumed that any important information on the blackboard has already been recorded by the previous user. On the other hand, if the blackboard were callee-saved, the most recent user would have to restore the blackboard to however it originally was, prior to their use.

We can use the fact that registers are shared across the entire program to pass and return values to functions.

Arguments and Return Values

Unlike C, Assembly doesn't have function headers: a function declaration is just a label. So, how do we take in arguments and/or return values? Both of these tasks are accomplished through registers. If we have a lot of arguments or return values and we run out of registers, we can use the stack (but we won't need to worry about that).

When we're passing arguments to a function, we can just load the argument into a register for the function to use. Arguments are aligned to start in even-numbered registers, beginning at `r24` and going downwards. Also, if we're passing an odd number of parameters, there will always be one free register above them. So, for instance, if we're passing in a single `char` to a function, we would load the value of that character into `r24`, and we would leave `r25` empty as there is an odd number of parameters.

The conventions for passing arguments to functions are illustrated below:

- If we're passing an argument that's just one byte, the argument will go in `r24` and `r25` will be cleared.
- If we're passing an argument that's two bytes, the arguments will go in `r24` and `r25` (note how there is no free register since there are two arguments).
- If we're passing four bytes of arguments, the arguments will go in `r25, r24, r23, r22`, and there won't be any empty registers.

Once these registers have been loaded with the arguments, we can use the `call` instruction to go inside of the function. The function will share the contents of these registers, and it will be able to perform any necessary processing.

How do we return values from functions? It's the same idea as passing arguments. The exact same convention for passing parameters. For instance, if the function returns one byte, the return value is loaded into register `r24`.

Note that the conventions for passing arguments and returning values align with the caller/callee-saved protocol. Particularly, we are passing parameters through caller-saved registers. The function caller is expected to have already saved any important data in these registers, as the function will be changing the contents of these registers after processing.

After introducing these concepts, let's look at an example that's very similar to what we saw yesterday:

Listing 38: Assembly: Arguments and Return Values

```
    .text

.global main
main:
    call init_serial_stdio

    ; Printing 'A'
    clr r25
    ldi r24, 65
    call putchar

    cli
    sleep
```

The function `putchar` from C takes in a single character as an argument. To pass in the argument 'A,' we clear register `r25` and load the ASCII value of 'A' into `r24`. This aligns with the convention we've previously discussed. Finally, we call `putchar`, which is now free to do whatever it desires with these parameters. It will no longer be guaranteed that register `r24` and `r25` are how they were prior to the function call.

Accessing Memory

In Assembly, `lo8` and `hi8` can be used to extract the lower and higher 8 bits of data (if we are passing in 2 bytes of data, there will be two 8 bit blocks). The instruction `ldi r24, lo8(x)` would load the lower byte of variable `x` into register `r24`.

We have already seen that `lds` and `sts` are instructions that allow us to read and write to memory. Assembly has three pairs of registers that allow us to access memory. These pairs of registers are called `X` (resides in registers `r26` and `r27`), `Y` (resides in registers `r28` and `r29`), and `Z` (resides in registers `r30` and `r31`).

`X`, `Y`, and `Z` represent addresses in memory (like pointers). Conventionally, the low byte is stored in the even-numbered register, and the high byte is stored in the odd-numbered register.

Consider the following example:

Listing 39: Assembly: Arguments and Return Values

```
    .data
pctd:
    .asciz  "%d "

values:
    .byte   15
    .byte   16
    .byte   17
    .byte   18

    .text

.global main
main:

    call init_serial_stdio

    lds r24, values
    clr r25
    call pint
    ldi r24, 10
    clr r25
    call putchar

    ldi r26, lo8(values)
```

```
   ldi r27, hi8(values)
   ld r24, X
   push r26 ; Caller-save
   push r27
30 clr r25
   ldi r24, 10
   clr r25
   call putchar

35 pop r27
   pop r26
   adiw r26, 1 ; Move the pointer

   ld r24, X
40 clr r25
   call pint
   ldi r24, 10
   clr r25
   call putchar
45
   cli ; Terminate program
   sleep
   ret
```

First off, we see a new directive: `.asciz`. This just indicates that we are declaring a string literal.

Next, we note the label `values` is defined differently than what we've seen so far: there are multiple `.byte` directives in its definition. The key thing to remember here is that a label is just an address in memory. Hence, one way we can interpret `values` is the value 15: if we were to load this value somewhere, the value 15 would be loaded. We can alternatively interpret `values` as the name of an array with contents $15, 16, 17$, and 18.

When we call `pint` (the function for printing), 15 is outputted. We then load the ASCII value for a new line, and we call `putchar` to print a new line.

On Lines 25 and 26, we load the low byte and high bytes of `values` into the registers `r26`, and `r27`. The registers `r26` and `r27` are special: they represent a register pointer, denoted by `X`. Hence, on the subsequent line, we can use `ld` to initialize `r24` to whatever `X` points to (note that we use `ld` for register pointers instead of `lds`). Finally, when we call `pint` again, 15 is printed again (as `r24` now points to `X`).

Okay, now what if we want to print the other values in the array? We use pointer arithmetic, just like in C. The `adiw` instruction is short for "add immediate to word'," and it has syntax `adiw [register], [number]`, and it increments the contents of `[register]` by `[number]`. This is exactly what's happening from Lines 37 to 44. These lines increment `X` and print `16` along with a new line character.

Tomorrow, we will pick up from this point and do some more Assembly.

17 Wednesday, July 3, 2019

More Assembly today. Before we start, here's some quick review from the past two classes:

- Assembly programs are written in a `.S` file.
- There is a `.data` directive, which denotes the portion of the code where we define data (like variables).
- A label represents a memory address (it can be viewed as a pointer to a variable or a function).
- A value in memory is defined using the `.byte` directive. Hexadecimal, decimal, or binary are all permitted.
- The `.global main` directive is like the `extern` keyword in C—it indicates that the function can be accessed from outside of the current file.
- The `lds [register], [data]` instruction stores `data` into the contents of `register`. Similarly, `ldi [register], [constant]` allows us to load a constant value to `register`.
- The `clr [register]` instruction clears the contents of `register` to zero.
- Assembly has a built-in pointer register, denoted `X`, which represents the combination of registers `r26` and `r27`. (Why do we need two registers? Memory addresses are 16 bits, or 2 bytes, so they need two bytes).
- To initialize a pointer register like `X`, we use `ldi` along with `lo8` and `hi8` on `r26` and `r27`, respectively. We use these directives on whatever value we want `X` to point to.
- To load the contents of a register pointer into another register, we can use the `ld` instruction in the form `ld [destination register], [register pointer]`. This is the C-equivalent of dereferencing a pointer. If we now increment `r26`, we can write `adiw r26, 1` to move the pointer `X` by one. Using `inc` would also work, but it's not great to use since with register pointers as it only operates on one register. Assembly also supports a `+` operator. The instruction `ld r24, X+` would load the contents of `X` to `r24` and move the pointer by one.
- To save a value in a register, we use `push` in the form `push [register]` to push the contents of the register onto the stack. You need to have one `pop` for every `push`.

More on Register Pointers

Consider the following code segment, which is an example of writing to memory:

Listing 40: Assembly: Register Pointers

```
;;; Example — writes the values 77 and 99 to memory
    using sts and st;
;;; Also using X, Y, Z register pointers

        .data
pctd:
        .asciz "%d "

values: ; represents data memory area where we will
    write
        ; note we are not using any .byte
          directive

        .text
```

```
.global main
main:
        call init_serial_stdio

        push r29                ; needs to save
            r29,r28 (callee-saved)
        push r28

        ldi r18, 77
        sts values, r18         ; assigning 77 to
            location values (using sts)

        ldi r28, lo8(values)    ; reading first value
            using Y (r29:r28)
        ldi r29, hi8(values)    ; r29:r28 = values
        ld  r24, Y+             ; using Y+ (increases
            pointer by one location)
        clr r25                 ; printing the value
        call pint

        ldi r18, 99             ; writing location
            after first entry
        st Y, r18               ; using st (NOT sts)
        ldi r30, lo8(values)    ; using Z pointer
            register to read value written
        ldi r31, hi8(values)
        adiw r30, 1             ; moving forward Z
            pointer one position
        ld r24, Z               ; reading value written

        clr r25                 ; printing value
        call pint

        clr r25                 ; newline
        ldi r24, 0xa
        call putchar

        pop r28
        pop r29

           cli ; stopping program
        sleep

        ret

pint:
        ;; prints an integer value, r22/r23 have the
            format string
        ldi r22, lo8(pctd)      ; lower byte of the
            string address
        ldi r23, hi8(pctd)      ; higher byte of the
            string address
        push r25
        push r24
        push r23
        push r22
        call printf
        pop r22
        pop r23
        pop r24
        pop r25

        ret
```

Everything up to the main is what we're used to. Note, however, that the `values` label doesn't have any

`.byte` directives—all this means is that the memory address corresponding to `values` hasn't been initialized.

In this example, we'll be using the `Y` register pointer (there's no particular reason why we're using `Y`—we could have used `X` or `Z` instead), which corresponds to the registers `r28` and `r29`. So, we save the contents of these registers, and we load the low and high byte of `r18` (which stores 77) into them. On Line 26, we load what `Y` is pointing to into `r24`, and increment `Y` by one (so `Y` is now pointing to a new uninitialized area of memory). Next, the value 77 is printed by calling `pint`.

Line 30 updates the contents of `r18` to 99, and line `31` stores 99 into `Y`. Note that we use `st`, which works with register pointers, instead of `sts`. Lines 32 to 33 initialize the register pointer `Z`, and Line 34 moves `Z` forward by one (`Z` is now pointing to 99). Thus, loading this value into `r24` and calling `pint` prints out 99.

Instruction Encoding and the Status Register

In Assembly, an instruction is represented by a set of bits which are assembled into a set of zeros and ones. But, not all of these bits are telling the assembler what operation to perform. The portion of the bits that encode what operation should be performed is known as the **opcode**.

So, what do the rest of the bits represent? The short answer is that it is dependent on the instruction we're considering. For instance, if we're dealing with `ldi`, we'd have the opcode, another portion to store the registers, and another portion to store the values of the registers.

The number of bits necessary to encode an instruction also varies. In AVR, however, we're guaranteed that instructions are either two bytes or four bytes. When memory is scarce, choosing the correct instruction is vital to saving memory.

In an Assembly program, there's a register—known as the **status register**—that keeps track of recent operations (particularly mathematical operations). We can view what's inside of the status register by typing `info r` in gdb. Why is the status register important? It allows us to perform conditional executions, known as **branching**.

Branch Instructions

We can use the status register to perform conditional execution of statements. This is done with the `cp` instruction, which has syntax `cp [register1] [register2]`. This is used to compare the contents of two registers.

Consider the following example:

Listing 41: Assembly: Register Pointers

```
;;; Example - if a == b prints 'Y' else prints 'N'
;;; Change a and b to see different outputs

    .set LETTER_N, 'N'
    .set LETTER_Y, 'Y'

;;; Global data
    .data

a:      .byte 0x6
b:      .byte 0x5

;;; Program code
        .text

.global main
main:
        call init_serial_stdio
```

```
20          lds r18, a              ; reading a from memory
            lds r19, b              ; reading b from memory
            cp r18, r19             ; comparing registers
            breq 1f                 ; 1 is the target: f
                represents forward
            ldi  r24, LETTER_N      ; store N
25          jmp 2f
    1:      ldi r24, LETTER_Y       ; store Y

    2:      clr r25                 ; printing result
            call putchar
30
            clr r25                 ; print newline
            ldi r24, 10
            call putchar

35          cli                     ; stopping program
            sleep

            ret
```

In this program, we read the value of `a` and `b` into registers `r18` and `r19`. These values are compared with `cp` on Line 22, which has syntax `cp [register1], [register2]`. The instruction `breq` is short for "branch if equal," and its syntax is `breq [label]`. Pretty much, this instruction utilizes the status register to check the value of the preceding comparison. If the comparison indicated that the two variables were equal, the program jumps to the target label (here, the `f` indicates that we're jumping forward).

Suppose $a \neq b$. In this case, we don't jump to label 1. Instead, we'll store the letter `N` in `r24`. The next instruction, `jmp`, indicates an unconditional jump (it always gets executed). Hence, once `N` is loaded into `r24`, we'll jump forward to label 2, and we'll print the result along with a new line.

Now suppose $a = b$ holds. In this case, we'll jump over the statement that loads `N` into `r24`. Instead, we'll load `Y` into `r24`, and we'll continue from there and print the character along with a new line.

The advantage of using `cp` is that we don't have to modify the registers.

Some other branch instructions are listed below:

- `cpi [register], [constant]` compares the contents of a register to a constant.
- `tst [register]` tests whether a register is non-positive.
- `breq [label]` branches to `label` if the previous comparison indicated an equality.
- `brne [label]` branches to `label` if the previous comparison did not indicate an equality.
- `brge [label]` branches to `label` if the previous comparison indicated `register1` was greater than or equal to `register2`. This should be used on signed integers.
- `brlt [label]` branches if `register1` is strictly less than `register2`. This instruction should also be used for signed integers.
- `brlo [label]` branches to `label` if the comparison is strictly less than. It is used with signed integers.
- Finally, `brsh [label]` branches to `label` if the comparison is greater than or equal to. It is used with unsigned integers.

It's important to remember that branching instructions always look at the last result with the status register. Thus, comparison instructions need to come immediately before the branch instructions.

Here's another example.

Listing 42: Assembly: Do-While Loop

```
;;; Example - prints values 1 to 5 (do while)

;;; Global data
    .data

pctd:
        .asciz "%d "

upper_limit: .byte 0x5

;;; Program code
        .text

.global main
main:
        call init_serial_stdio

        push r15                ; callee-save
        push r16

        lds r15, upper_limit    ; upper limit
        ldi r16,1               ; loop starts, r16 is
            the iteration variable

1:      mov r24, r16            ; printing value
        clr r25
        call pint

        inc r16                 ; increasing iteration
            variable
        cp r15, r16            ; checking whether we
            reach limit
        brge 1b                 ; go back as long as
            r15 >= r16

        clr r25                 ; newline
        ldi r24, 10
        call putchar

        pop r16                 ; restoring registers
        pop r15
    cli                        ; stopping program
        sleep

        ret

pint:
        ;; prints an integer value, r22/r23 have the
            format string
        ldi r22, lo8(pctd)       ; lower byte of the
            string address
        ldi r23, hi8(pctd)       ; higher byte of the
            string address
        push r25
        push r24
        push r23
        push r22
        call printf
        pop r22
        pop r23
        pop r24
        pop r25

        ret
```

There isn't anything confusing here. Register `r15` stores the upper limit of loop. After each iteration. we compare the iteration variable, stored in `r16`, to the upper limit. If we haven't hit the limit, we branch back to the start of the loop. Similarly, we could implement a while loop by performing an initial comparison.

18 Monday, July 8, 2019

When performing branching instructions, it's important to remember to use the one corresponding to the correct type (there are different instructions for unsigned vs signed integers). Also, keep in mind that `adiw` (or post-incrementation) should be used for moving a pointer rather than `inc` (or `dec` to decrement).

We've already seen the `add` instruction. There are also `sub` and `mul` instructions to subtract and multiply the contents of two registers; their syntaxes is exactly the same.

The `movw` has syntax `movw [register1] [register2]`, and it copies the register **pair** `register2` to `register1`. This instruction is useful when moving the results of multiplication.

The `lsl` and `lsr` perform left and right bit-shifts, respectively. They both require one register as input, and the result is that the contents of the register are either multiplied or divided by two (respectively). Unfortunately, there is no division operation; however, one could implement it themselves.

Large Addition and Unsigned Multiplication

Here, we'll discuss some constructs surrounding math in Assembly.

Consider the following code segment:

Listing 43: Unsigned Multiplication

```
;;; Example - Illustrates how to use mul (unsigned
    multiply)

;;; Global data

        .data

pctd:   .asciz "%d "            ; defines a string (nul
    terminated)
a:      .byte 200
b:      .byte 150

;;; Program code
        .text

.global main
main:
        call init_serial_stdio

        clr r25
        lds r18, a              ; reading value for a
        lds r24, b              ; reading value for b
        add r24, r18            ; just using add is
            wrong for 200 and 150
        adc r25, r25            ; we need adc
        push r24                ; caller save
        push r25
        call pint
        call prt_newline
        pop r25                 ; restoring caller save
        pop r24

        adiw r24, 5             ; adds five to previous
            result (r25:r24 is updated)
        call pint
        call prt_newline

        ldi r24, 8              ; multiplication
        ldi r25, 6
        mul r24, r25            ; result in r1:r0
        movw r24, r0            ; copies r1:r0 to
            r25:r24
        clr r1                  ; we should always make
            sure is back to 0

        call pint               ; printing
            multiplication result
        call prt_newline

        ldi r24, 32             ; multiplying by 2
        lsl r24
        call pint
        call prt_newline

        ; next example shows we need to use brlo with
            unsigned
        ldi r24, 2              ; comparison between
            r24 and 11
        cpi r24, 11             ; the smaller will be
            printed
        brlt 1f                 ; try r24 with 5, 199
            (fails)
        ldi r24, 11
1:      call pint
        call prt_newline

    cli                     ; stopping program
        sleep

        ret

prt_newline:
        ;; prints new line
        clr r25
        ldi r24, 10
        call putchar

        ret

pint:
        ;; prints an integer value, r22/r23 have the
            format string
        ldi r22, lo8(pctd)      ; lower byte of the
            string address
        ldi r23, hi8(pctd)      ; higher byte of the
            string address
        push r25
        push r24
        push r23
        push r22
        call printf
        pop r22
        pop r23
        pop r24
        pop r25

        ret
```

We can briefly recap what we already know to explain what's going on here.

At first, the variable `pctd` is declared as a string, and `a` and `b` are declared as integers. Subsequently, we compute the sum of `a` and `b` in `r24`. But since $a + b = 200 + 150 = 350 > 256$, we cannot store the contents of the sum in the eight bit register `r24` alone. So, how do we fix this issue? We can make use of the `r25` register.

It turns out that there's an `adc` instruction with format `adc [destination] [source]`, which helps us deal with overflows (it's short for "add with carry"). When we perform the `add r24, r18` instruction, the bits that don't overflow are added correctly. If there is a carry from an `add` instruction, a special bit in the status register is marked, and the value of the carry is kept for safekeeping. Basically, the `adc` instruction allows us to perform addition that exceeds 8 bits inside of a register pair. If we know that the numbers we're dealing with are small, we don't need to worry about a carry. We need at most an $(n+1)$-bit number to represent the sum of two n bit numbers.

After this addition is performed, we push `r24` and `r25` for safekeeping (note that we can't clear `r25` as we usually do—this would remove our carry value), and we call `pint` to print the sum. As we'd expect, the sum is 350. Moreover, now we need to perform the `adiw` instruction to add to this quantity (`adiw` acts on a register pair, whereas `add` doesn't). Now we move to unsigned multiplication.

On Lines $34, 35$, and 36, we load 8 into `r24`, 6 into `r24`, and subsequently multiply the two numbers. Since the product of two 8-bit numbers is at most a 16-bit number, we require two registers to hold the results of multiplication. By default, Assembly moves the results of the `mul` instruction to the `r0:r1` register pair. We should retrieve these values using `movw` immediately since the register pair `r0:r1` is temporary. Also, by convention, we clear `r1` back to zero (not `r0`!). Finally, we discuss bit-shifting. First, a simple example, which will be followed by a more intricate one.

On Lines 43 and 44, we load 32 into `32` and call the `lsl` instruction (which is short for "logical shift left") to perform a left bit shift. The resulting value in `r24` is 64. This was a very easy example.

Between Lines 49 and 54, we appear to be printing the value stored in `r24` if it's less than 11, and 11 otherwise. However, we're using `brlt` to compare, which is intended for signed integers (the unsigned analogue would be `brlo`). Although the example would work for the values provided (along with several other values), if we were to load 5 into `r24` and compare it against 199, we would unexpectedly print 199. Why? The two's complement signed representation of 199 is less than 5. This example emphasizes the importance of using the correct branch instruction.

Even More on Register Pointers

We've already seen that something like `ld [register] X+` loads the contents of `X` into `register` and subsequently moves `X` forward one. We can also pre-decrement our register pointer with, `ld [register] -X`. There is no post-decrementation or pre-incrementation.

It's important to remember that using register `Y` requires callee-saved registers, meaning that we need to push registers `r28` and `r29` prior to using them. As a result, it's usually a good idea to utilize the `X` and `Z` pointers prior to using the `Y` pointer (we'd lose points on exam if we needlessly used the `Y` pointer when it isn't necessary).

The increment and decrement operations don't affect the status register. So, if we were to perform a comparison, and increment our register pointer, we'd still be able to perform branching instructions as we'd want.

Another helpful instruction is `ldd`, which has syntax `ldd [register], [pointer] + [constant]`. This instruction **only works** on the `Y` and `Z` pointers, and it simply loads the location pointed to by `pointer + constant` into `register` without actually modifying `pointer`.

The Call Stack and Recursion

So far, we've been using the stack to save values using the `push` and `pop` instructions. The stack can also be used to support function calls, which is particularly useful when implementing recursive programs.

When you call a function, the address of the instruction that follows the call is placed on top of the stack. When the `ret` instruction is executed at the end of the function, whatever is on top of the stack (i.e. the instruction after the terminating function) is executed next. What does this mean to us? It emphasizes the importance of a one-to-one mapping between `push` and `pop` calls. If you're using the stack to preserve a value, and the correct number of `pop`s aren't called, the `ret` instruction of the function call will return to an invalid address.

The following program computes the factorial of a number through recursion:

Listing 44: Assembly: Factorial

```
.global factorial
factorial:
        ;; recursive computation of factorial
        tst r24             ; base case check (if value
            == 0)
        breq 1f
        push r24
        dec r24
        clr r25
        call factorial      ; recursive call
        pop r23             ; (original value of r24)
        mul r24, r23        ; factorial(n - 1) * n
        movw r24, r0        ; copies r1:r0 to r25:r24
                            ; movw is a register pair
                              copy
        clr r1              ; making sure is 0
        jmp 2f
1:
        clr r25             ; base case (value of 1)
        ldi r24, 1

2:
        ret
```

This is self-explanatory. We've got a base case and a recursive call. What's important to keep in mind when tracing this program is that `call factorial` will result in the factorial of $n - 1$, which will be stored in register `r24`. That's why we pop to `r23` instead of `r24` (so the result isn't overridden).

Starting on Wednesday, we'll discuss process control, which is the last big topic of our class.

19 Tuesday, July 9, 2019

Today, discussion is really short since we had a quiz. Just a couple of examples on encapsulation, abstraction, and some other miscellaneous things in C.

Encapsulation and Abstraction

In general, C has limited support for encapsulation. One of the primary features that C provides for encapsulation is the **incomplete type**. An example of an incomplete type would be a declaration of a structure without specifying its contents, like `struct my_type;`

This indicates to the compiler that `my_type` is a structure, but it does not provide any information about its members. This allows the user to complete the type elsewhere.

As a more illustrative example, suppose we want to hide the following structure:

Listing 45: Secret Structure

```
struct secret {
    char name[MAX_LEN + 1];
    int age;
};
```

We can then create a separate `.h` header file with all of our function prototypes. This file will contain the line `struct secret;` to declare the structure `secret` (even though the definition of `secret` is not in the `.h` file). If we then compile these files, along with a main with other functions, the object file will hide the implementation of the `secret` structure. We'd give our program users the `.o` and the `.h` file, which abstracts the function and structure implementations from the user (but, they can still use the functions since they have the function prototypes).

Miscellaneous

- Recall that dereferencing a null pointer results in a segmentation fault. Although this is true, we wouldn't get a segmentation fault if we dereference the pointer inside of a `sizeof()` call. For example, if we declare `int *p = NULL`, a subsequent `sizeof(*p)` expression doesn't result in a segmentation fault.
- Void pointers can be casted to any type of pointer – it's up to the programmer to make sure we're doing things right.

20 Wednesday, July 10, 2019

Process Control Terminology

Before we get started with process control, we need to introduce some new terminology.

The **kernel** is a component of the operating system that's responsible for maintaining security, performing file management, and managing processes. It's like the "manager" of the operating system—it enforces "policies."

When a program is running, there are various tasks that we don't consider to be sensitive (i.e. dangerous to the operating system). Hence, the program doesn't require too many permissions. We say that these programs are run in **user mode**. When we're executing a program in user mode, we can't perform any sensitive (dangerous) operations, and we don't have the privilege of accessing everything associated with the operating system.

By contrast, some instructions are restricted so that only the operating system itself can execute them. When a program has this privilege, we say that the program is running in **kernel mode**. Some examples of operations a program can perform that requires kernel mode include halting the CPU or performing I/O.

Context switching is a feature utilized by an operating system to store the state of a process so that it can be restored and its execution can be resumed from the same point at a later time. This happens really fast, so it seems almost as if we're performing multiple tasks at the same time. For example, if we've got a single CPU, and we're programming while listening to music, our CPU would be rapidly context switching the two processes. What dictates how the context switch chooses which processes to pause and resume? The kernel does.

System Calls

A **system call** is a special function that allows us to interact with the kernel. Functions that permit us to perform file I/O, create processes, and read the system clock all perform system calls.

System calls aren't the only way in which we can interact with the kernel. We can also use the shell, which allows for indirect interaction (when we're copying or moving files, we're interacting with the kernel). Are there different types of shells? Yes - so far, we've been using the **tcsh shell**; however, **bash** and **korn** are also shells (to change to either of these shells, we can execute the `bash` or `ksh` commands in Unix).

Processes vs. Threads

A **thread** is an execution path, almost like a program inside of another program. For example, suppose we've designed a clock that works in our own timezone. But now, we want to design four clocks, each of which represent a different timezone. We've already got one working clock, so we can spawn four threads, each of which represent a different timezone. The program will allow each thread to run for the correct amount of time.

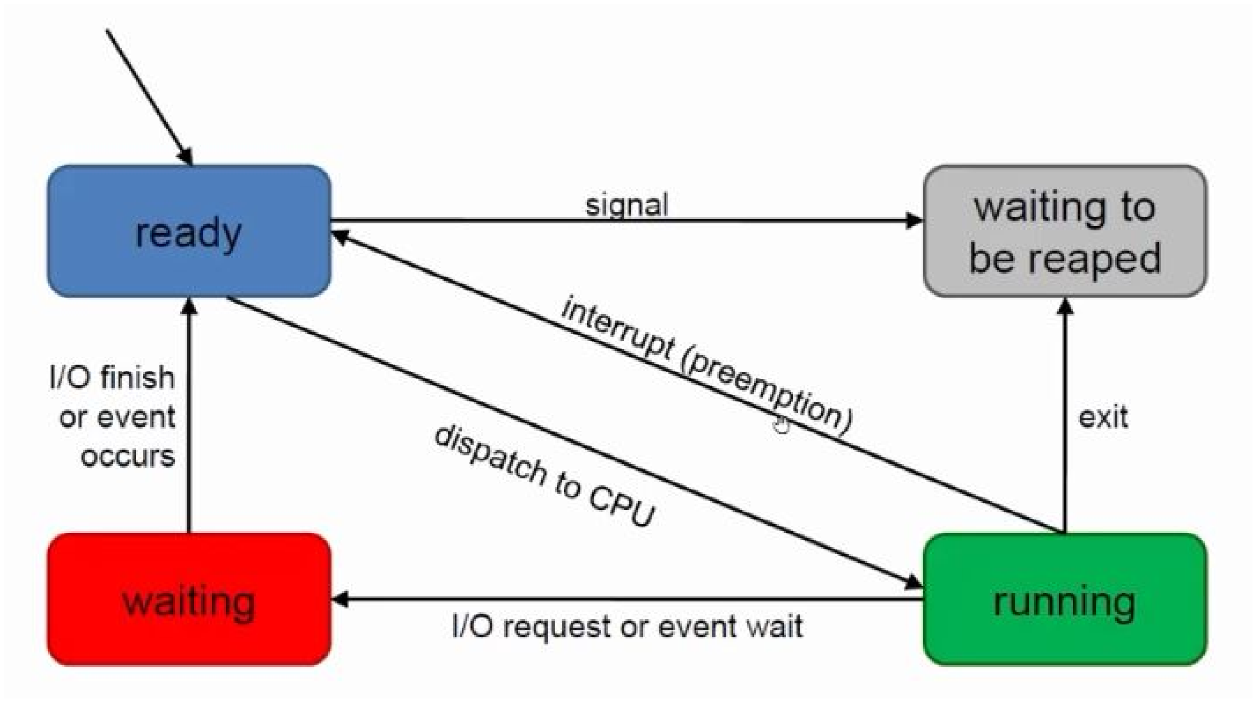
Here's another example. Suppose we're computing the sum of an array. We can use one thread to compute the sum of the first half, and a second thread to compute the sum of the second half. Depending on our hardware, this can save time.

What's the difference between a thread and a process? Threads lives inside of a process. The most minimalistic representation of a thread includes the stack (used to support function calls) and the program counter. We can have multiple threads helping us run a process with context switching.

Something key thing to note is that threads share the same resources. That is, if a process opens a file, all of the threads share the same file. Moreover, if a process dynamically allocates memory, all of the threads have access to the memory (the heap and global variables are shared by all of the threads).

A real life analogy is a household. A house can be viewed as a process, whereas the people inside of them are the threads. Everything inside of the house (process) is shared by the people (threads).

Every process goes through several states during its execution. Collectively, these states are referred to as the **process life cycle**. A pictorial representation of this life cycle is presented below:



What's happening here?

- When a process is in the **ready state**, it is not currently executing; however, it is ready to begin executing. It is waiting for the kernel to tell it to start running.
- Once the kernel tells the program to run, the process moves to the **running state**. What can happen from here? The program might move into the **waiting state**, where it awaits I/O. Alternatively, the program can finish executing and move to the **waiting to be reaped** stage. Finally, there's one last scenario: the program can be interrupted (by perhaps the programmer). If this happens, the process moves from the running stage back to the ready stage.
- Let's say our program takes in user input. Once it starts running, it'll move to the waiting state. Interestingly, once the I/O is completed, the program can't immediately go back to the running state again. It needs to go back in line to the ready state before it can go back into the running stage.

Signals

A **signal** is a method that allows two processes to communicate. A process is able to recognize when it receives a signal, and the process is able to react in response to that signal.

We've already been using signals: `CTRL + C` is a signal, and the program responds by terminating. Formally, the name of this signal is `SIGINT`.

Some other signals that are used by the kernel include the following:

- `SIGSEGV` is a signal indicating a segmentation violation (a.k.a. segmentation fault)

- `SIGFPE` is a signal used to indicate a floating point exception.
- `SIGCHILD` is a signal used to indicate that a child process has terminated.

A program doesn't necessarily terminate when it receives a signal — it completely depends on the signal being sent.

Creating Processes

In Unix, a new process (called a **child**) is created by an existing process (called a **parent**), making a parent-child relationship between the two processes. The new child process will then be able to execute the program we want.

How do we create a new process? The system call do create a new process is `fork()`. This call creates a copy of the parent process.

What gets copied when we fork a process? Just about everything:

- All variables (the entire address space) gets copied.
- The point of execution is copied (i.e. parent and child processes continue execution after the fork system call)
- The file descriptor table (files opened by the process) is copied.

The stack, heap, data, and code all get executed.

What are the benefits of forking if we're getting the same exact thing? Well, once we've forked a program, we can modify the child process and change it to a different program. This is done wit a system call that we'll see later on.

Let's look at an example in C:

Listing 46: Forking 1

```c
#include <stdio.h>
#include <sysexits.h>
#include <err.h>
#include <unistd.h>     /* Required by fork() */
#include <sys/types.h>  /* Required by pid_t */

int main() {
   pid_t result;

   printf("Hello\n");

   result = fork();
   if (result < 0) {
      err(EX_OSERR, "fork error");
   }

   printf("End: Value returned by fork: %d\n", result);

   return 0;
}
```

First off, the `pid_t` data type represents a signed integer used for process identification. (The `_t` portion of a data type means that the data type is internally mapped to an integer). Thus, we can print the variable `result` using the `%d` format specifier.

Next, we set `result` to `fork()`, which returns a signed integer. The `fork()` function returns a duplicate process of the program currently being executed. Once this `fork()` takes place, the processes will exist, and they will be executing after the function call. It doesn't concern us which process is executing first. All that we know is that we'll have two processes, both of which will be executing after Line 12.

Now, what's the return value of `fork()`? It's the Process ID (PID) associated with the child process. If we again were to call `fork()` on a child, the return value would be zero: the PID of a child process is always zero. So, the variable `result` has two different values: the PID of the child process in the parent process, and 0 in the child process.

Can forking fail? Yes. There is a limit to the number of processes we can fork since space is limited. If the fork fails, -1 is returned, which is why we have the conditional from Lines 13 to 15.

Finally, the program prints the PID of the newly created child process.

Here's another example:

Listing 47: Forking 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <sysexits.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
   pid_t result;
   int x = 20;
   char *p;

   printf("Hello\n");

   p = malloc(80);

   result = fork();
   if (result < 0) {
      err(EX_OSERR, "fork error");
   }

   /* By using the value returned by fork, we */
   /* we can tell which process is the parent */
   /* and which is the child. We can assign   */
   /* different tasks to each one.            */
   /* Notice the address values printed by    */
   /* the processes.                          */

   if (result == 0) {
      printf("I am the child (increases x) %d\n", ++x);
      printf("Value of address in child %p\n", (void
          *)p);
   } else {
      printf("I am the parent (decreases x) %d\n",
          --x);
      printf("Value of address in parent %p\n", (void
          *)p);
   }

   free(p);  /* Both must free */

   printf("Done\n");

   return 0;
}
```

This time, before we fork our process, we declare the variables `x` and `p`. We also dynamically allocate memory for `p`. Now since forking copies all components of the code, our new process will also have these variables. The only difference is that the variable `result` will be the PID of the child process for the parent process, and it will be 0 for the child process. We can use this fact to produce different outputs.

By comparing the PID to 0, we can obtain different outputs between the child and parent processes (for the child process, the Boolean expression `result == 0` will evaluate to true). In the program above, we'll increase the variable `x` to 21 for the child process; the variable `x` will remain 20 in the parent process.

Since we've dynamically allocated memory prior to forking, our child process has inherited this new memory area as well. So, we need to call `free()` in both the child and parent process (somewhere that's accessible by both processes).

One thing that is strange, however, is that when we print the address of the pointers, we'll obtain the same memory address. It'll appear as if the pointer `p` is pointing to the same place for both the child and parent process. However, this is not the case — the operating system will convert them to distinct areas in memory when it is needed.

Something else to note is that even though the `fork()` came after the variable declarations, the child process still inherits all of the variables that come before it. The only thing determined by *where* the `fork()` call comes is where the point of execution for the child process is set to.

The `getpid()` and `getppid()` functions return the PID of the process currently executing that function. What happens if we call `getppid()` on a process that doesn't have a parent? The PID of the the shell will be returned—the shell is the ancestor of all processes.

Recall that the newline character, `\n`, is used to flush the buffer. If we use `printf` without the new line character, whatever we're printing is placed in memory; it isn't printed until the buffer is flushed. So, if we print a statement without flushing the buffer prior to forking a process, the buffer also gets duplicated. Consequently, if we perform another `printf` statement (after the `fork` call), the message that came before the `printf` will be printed twice. Long story short, it's usually a good idea to make sure the buffer has been cleared prior to forking.

Now let's look at an application of forking:

Listing 48: Forking 3

```c
/**********************************************/
/* The program reads two integer values. The   */
/* parent will call even_odd on the first      */
/* value and the child on the second. Notice   */
/* we need the exit(0) in the process_values() */
/* function, otherwise you will be printing    */
/* "Done in Main" twice. In this example we    */
/* do not want the child to return to main().  */
/**********************************************/
#include <sysexits.h>
#include <stdlib.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

void process_values(int x, int y);

void even_odd(int a) {
    if (a % 2 == 0) {
        printf("%d is even\n", a);
    } else {
        printf("%d is odd\n", a);
    }
}
```

```c
int main() {
   int m, k;

   printf("Enter two integer values: ");
   scanf("%d%d", &m, &k);

   process_values(m, k);

   printf("Done in Main\n");

   return 0;
}

void process_values(int x, int y) {
   pid_t pid;

   if ((pid = fork()) < 0) {
      err(EX_OSERR, "fork error");
   }

   if (pid != 0) { /* parent code */
      even_odd(x);
   } else { /* child code */
      even_odd(y);
      exit(0);  /* WHY WE NEED IT? Remove it and run */
   }
}
```

This is pretty self-explanatory. We're reading in two integers, both of which are stored in the stack. Then, we call `process_values`. If we're the parent process, we'll check whether `x` is even or odd, and if we're the child process, we'll check whether `y` is even or odd. Note that we execute `exit(0)` at the end of the child's code in order to terminate the child process. We need this because, otherwise, the statement "Done in Main" will be printed twice. The key takeaway is that even though the child process was created locally in the function, it still inherits the main and other properties.

21 Friday, July 12, 2019

Last class, we started process control. Something important to keep in mind is that kernel can only hold a finite number of processes.

What happens if we try to exhaust the number of processes permitted? Consider the following code:

Listing 49: Fork Bomb

```
#include <unistd.h>
int main() {
    while (1) {
        fork();
    }
}
```

The above code segment tries to repeatedly spawn new processes. This is known as a **fork bomb**, and it exhausts all the possible space in a process table. An insecure system might crash, but most systems have something in-place to identify and stop these attacks[2].

Reaping Child Processes

After a child process finishes executing, we reap it in order to remove its details from the process table. Until the terminated process is reaped, we say that the process is a **zombie process**.

We can release zombie processes with the `wait()` or `waitpid()` system calls. What do they do? Once the program encounters a `wait()` call, the program will wait until the child finishes until completion. Thus, the parent process will be blocked until the child continues.

Consider the following example, which illustrates the `wait()` system call:

Listing 50: Wait Example

```
#include <sys/wait.h>
#include <sysexits.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
   pid_t pid, returned_value;


   if ((pid = fork()) < 0) {
      err(EX_OSERR, "fork error");
   }
   if (pid) { /* parent code, pid != 0 */
      printf("Parent waiting for child\n");
      returned_value = wait(NULL);  /* nothing happens
          until child exits; reaps the child */
      printf("Value returned by wait: %d\n",
          returned_value);
      printf("Parent pid = %d; my child had pid =
          %d\n", getpid(), pid);
   } else { /* child code */
      sleep(4); /* simulating child's processing,
          waiting 4 seconds */
```

[2] A fork bomb is a form of denial-of-service attack

```
        printf("Child pid = %d; my parent has pid =
            %d\n", getpid(), getppid());
    }

    return 0;
}
```

As we've seen before, we're forking the program at the start. Then, if we're the parent process, we'll set `returned_value` to `wait(NULL)`. What does this do? `wait()` does two things: (1) it reaps the child once it has finished executing, and (2) it blocks the parent from executing until the child has finished.

So now our program execute the child's code. The child's code calls `sleep(4)` to wait for four seconds. Finally, it prints information about its own process as well as its parent process.

After the child finishes executing, `wait(NULL)` will reap the child process, and the parent process will continue its own execution.

Some other things to note:

- We're passing `NULL` into our `wait()` call – why? `wait()` can be used to return information about what happened to the child process (i.e. a seg fault). If we just want to reap after the child process finishes, we pass in `NULL`.
- Will the order of the printed statements ever change? No – the child's code will execute first, followed by the parent's code.
- Why didn't we call `wait()` in our previous examples? We should have. Our previous examples didn't reap created child processes.
- What happens if a process doesn't reap a child? Formally, we call such a process an **orphan**, and the `init` process will take care of it (created by the shell). Note that `init` will only intervene when the process completes, so this isn't helpful in large programs.

Why is reaping important? Our program would eventually crash without it: the process table will get filled up if we create several processes without making space for more.

The system call `wait()` has function header `pid_t wait(int *status)`. It returns -1 once everything has been reaped. Otherwise, it returns the PID of the child that is being reaped. What if we don't know how many children there are? We can just perform a while loop, and wait until `wait()` returns -1.

The `status` parameter that `wait()` takes in acts as an out-parameter. We can then use various pre-defined macros to see what took place.

The following example demonstrates how the out-parameter can be used:

Listing 51: Wait Status

```
/*
 * 1. Do not confuse the exit status (value returned
    from
 *    the program using exit or return from main) with
 *    the value that is initialized by wait (e.g.,
    wait(&status)).
 *    The status integer has the exit status and
    additional
 *    information.  We use the macros
    WIFEXITED(status),
 *    WEXITSTATUS(status) and WTERMSIG(status) to
    retrieve that information.
 *
 * 2. WIFEXITED(status) - true if the program
    terminated
 *    normally via exit or return from main.  Two
    examples
 *    of when a program does not terminate normally:
    when
 *    a segmentation fault takes place or if the
    program is
 *    terminated via a signal (e.g., kill
    <process_id>).
 *
 * 3. Remember that in Unix a program indicates it
    completed
 *    the expected task by returning 0 (e.g., exit(0)).
 *
 * 4. You can list signals by using kill -l
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sysexits.h>
#include <err.h>
#include <unistd.h>    /* Required by fork(), getpid,
    getppid */
#include <sys/types.h> /* Required by pid_t */

int main() {
   pid_t process_id;

   if ((process_id = fork()) < 0) {
      err(EX_OSERR, "fork error");
   }

   if (process_id != 0) { /* Parent code */
      int status;

      wait(&status);
      if (WIFEXITED(status)) {
         printf("Child finished normally (via exit or
             return in main)\n");
     if (WEXITSTATUS(status) == 0) {
            printf("Child completed the task
                successfully\n");
     } else {
            printf("Child did NOT complete the task
                successfully\n");
     }
      } else {
         printf("Child did NOT finish normally (via
             exit or return in main); signal must have
             occured\n");
         printf("REPORT:\n");
     printf("WIFEXITED(status): %d\n",
         WIFEXITED(status));
     printf("WEXITSTATUS(status): %d\n",
         WEXITSTATUS(status));
     printf("WTERMSIG(status) - (signal caused child
         to terminate): %d\n", WTERMSIG(status));
      }

      exit(0); /* Parent exit */

   } else {  /* Child code */
      int value;
```

```c
60          printf("Enter case (1, 2, 3, 4): ");
            printf("Or instead of entering a number kill the
                child process (kill <process_id>) (see
                output)\n");
            scanf("%d", &value);
            if (value == 1) {
               char *p = NULL;
65             *p = 20;
               exit(100);
            } else if (value == 2){
               int x;

70             printf("Enter positive integer: ");
               scanf("%d", &x);
               printf("Squared value is %d\n", x * x);
               exit(0);
            } else if (value == 3) {
75             int x = 0;
           printf("%d\n", 1 / x);
               exit(30);
            } else {
               int x;
80
               printf("Enter positive integer > 0: ");
               scanf("%d", &x);
               if (x > 0) {
                  printf("The cube of %d is %d\n", x, x * x
                      * x);
85                exit(0);
               }
               exit(40);
            }
         }
90 }
```

Here, we're doing almost the same thing we did before. We fork the process, and the wait call on Line 39 waits for the child process to finish to execution, while using `status` as an out-parameter.

The child process requests an integer and does some sort of processing, some of which create errors (for example, entering 1 leads to a segmentation fault). Subsequently, we return to the parent code, and we use the `WIFEEXITED` macro, which tells us whether or not the child program has terminated. If it has terminated, we'll check whether the return value was 0 with the `WEXITSTATUS` macro. If the program doesn't finish to completion, we can check whether a segmentation fault occurred by using the `WTERMSIG` status.

If the out-parameter `status` equals zero by a `wait()` call, that means the program finished as expected.

Environmental Variables

Environmental variables and **shell variables** are are dynamically-named values that customize environments. For example, one thing that we can do is change our prompt. Typing something like, `set prompt = "Hi: "` would make Grace prompt `Hi:` prior to each command (which constrasts from the default directory prompt).

From a C program, we can find the value associated with an environment variable using the `getenv` function. This function has header `char *getenv(const char *name)`. For instance, `loc = getenv("HOME")` would store the path to our home directory in `loc`. The parameter `name` needs to be an environmental variable.

Why would we need the `getenv()` function? Say we're implementing the `cd` function in a shell. If we type in `cd` by itself, we're supposed to move to the home directory. How are we supposed to know where that is?

By using `getenv("HOME")`. Now, continuing with our implementation of `cd`, how would we change what the program considers to be the home directory? We use a new function: `chdir()`.

`chdir()`, short for "change directory," has header `int chdir(const char *path)`. This function returns -1 upon failure. So, if we wanted to perform the Unix command `cd ~/temp`, we could equivalently execute `chdir(~/temp)` in C.

Somewhat surprisingly, reproducing the `cd` command in a shell doesn't require any forking. The `exit` command doesn't require any forking either.

Nested Processes

So far, we've seen how to produce a child process of a parent process with `fork()`. However, we've only seen examples in which the child process is executing code from the same file as the parent process. We can do this with the `exelc()` or `exelcp()` function.

Suppose we have the following `evens.c` program:

Listing 52: Evens

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   int i, limit;

   /* Default of a 100; otherwise using command line
       arg */
   limit = (argc == 2 ? atoi(argv[1]) : 100);
   for (i = 1; i <= limit; i++) {
      if (i % 2 == 0) {
         printf("%d ", i);
      }
   }
   printf("\n");

   return 0;
}
```

This program prints all even numbers up to whatever integer the user provides as a command-line argument (or up to 100 if no argument is provided).

Now, consider the following driver program:

Listing 53: Exec Evens

```c
#include <stdio.h>
#include <sys/wait.h>
#include <sysexits.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
   pid_t child_pid;

   if ((child_pid = fork()) < 0) {
      err(EX_OSERR, "fork error");
   }
   if (child_pid != 0) { /* parent code */
       int status;
```

```
            wait(&status);  /* reaping and waiting for
                child */
            if (WIFEXITED(status) && WEXITSTATUS(status) ==
                0) {
                printf("Child has finished successfully\n");
20          }
        } else { /* child code */
            printf("PID %d (child) will now execute
                execlp\n", getpid());

            /* I want to become the evens program */
25          execlp("./evens", "evens", NULL);

            printf("Would this be ever printed?\n");
            err(EX_OSERR, "exec error");  /* why no if
                statement? */
        }
30
        return 0;
    }
```

Pretty much, the parent process just forks itself, and it waits for the child process to finish. Now, what's happening with the child process? We execute `execlp("./evens", "evens", NULL)`, which loads the program `evens.c`. Something important to know is that when we load in a program, the entire process "becomes" the program we loaded in. Thus, the print statement on Line 27 is never printed (as it is not in the `evens.c` program). When we execute `exelcp`, the stack and heap are all cleared. The only thing that is retained is the set of files that were already opened by the original process.

If the program doesn't exist, we'll exit with the error code `EX_OSERR`. This error code never gets executed if the `exelcp` is successful.

With fork, exec, and waiting, we've got everything we need to build our own shell. There are two types of commands we need to handle:

- Unix commands: These usually don't require forking.
- Shell commands: These are built-in to a shell; they typically involve forking. This includes `cd`, `set`, and `setenv`.

The "main loop" used to implement a simple shell is as follows:

1. Read in a command line.
2. Parse the command line.
3. If it's a shell command, process it directly.
4. If it's a Unix command, fork, make the parent wait, and make the child execute the command.

22 Monday, July 15, 2019

Midterm II next week is on dynamic memory allocation, Linked Lists, and Makefiles. The exam won't cover Assembly or Process Controls.

Recall from last week that we can categorize commands in shell programs into two categories: those that require forking and those that don't. Typically, when we aren't forking, we're executing Unix commands that have already been written for us (so we don't need to actually program what the command should do – that's already been done for us).

Hiding Processes

In Unix, we can execute the `ps` command (short for "process status"), which will display a list of our active processes. Two useful flags for this command are `-f`, which provides us with a "full-format listing" (it provides some additional details) as well as `-u`, which displays the processes associated with our user. Finally, the `-e` flag shows all active processes for every user (so if you run `ps -ef` on Grace, you can see what other users are doing). The processes we create in C programs end up getting listed in here.

But what if we want our process to hide what it's doing? We can create an alias for our process with the second argument of the `exelcp` function. In last week's example, we executed `exelcp("./evens", "evens", NULL)`, so this process would have been displayed with the alias "evens." This can be changed to whatever we want.

The waitpid() System Call

Nelson says `waitpid()` is important to know for the final exam.

Recall that last week, we executed `wait(NULL)` to wait until a parent's child finished execution. The `wait()` command suspends execution of the calling process until *any* one of its children terminates. This can be problematic. To see why this can be a problem, suppose we're executing a parent process with two children processes. Call the two children processes Process A and Process B. If we perform a simple `wait(NULL)` call, our parent process will continue executing as soon as either Process A or Process B finish execution. Now, what if the parent process is dependent on some task performed by Process B? We'd want to wait for Process B to finish executing, but a simple `wait(NULL)` call might cause the parent process to move on with only Process A completed.

The `waitpid()` function has header `pid_t waitpid(pid_t pid, int *status, int options)`, and it solves this problem. This function is used to suspend the execution of the calling process until a child specified by the `pid` argument has finished executing. In the previously mentioned problem, we'd be able to fix our issue by entering the PID of Process A into a `waitpid()` call. The `status` parameter acts as an out-parameter, and the `options` parameter is a special number (defined as a macro), or it can alternatively just be 0.

Instead of inserting the PID into the first parameter of `waitpid()` function, if we instead input -1, we'll instead reap any process that has finished. This is a special case, and it allows us to recreate the `wait()` function. More specifically, `waitpid(-1, &status, 0)` would perform the exact same thing as `wait(&status)`.

Here is an example which uses the `waitpid()` function:

Listing 54: Deterministic Reaping

```c
/**
 * The parent will wait for each child, in
 * the same order in which they were created.
 */
#include <stdio.h>
#include <sys/wait.h>
#include <sysexits.h>
#include <stdlib.h>
```

```
   #include <err.h>
10 #include <unistd.h>
   #include <sys/types.h>
   #include <errno.h>

   #define MAX_CHILDREN   12
15
   int main() {
      pid_t all_pids[MAX_CHILDREN], child_pid;
      int idx, status;

20    /* Creating children processes */
      printf("\n**** Forking processes ****\n");
      for (idx = 0; idx < MAX_CHILDREN; idx++) {
         if ((all_pids[idx] = fork()) < 0) {
            err(EX_OSERR, "fork error while creating
                children");
25       }
         if (all_pids[idx] == 0) { /* child code */
            printf("Child %d with pid %d created.\n",
                idx, getpid());
            sleep(rand() % 10); /* simulates task child
                is completing */
            exit(idx);
30       } else { /* parent code */
            printf("Parent (pid %d) created child %d.\n",
                getpid(), idx);
         }
      }

35    printf("\n**** Reaping processes ****\n");
      idx = 0;
      while ((child_pid = waitpid(all_pids[idx++],
          &status, 0)) > 0) {
         if (WIFEXITED(status)) {
            printf("Child (pid %d) finished (exit status
                %d).\n", child_pid, WEXITSTATUS(status));
40       } else {
            printf("Child (pid %d) terminated
                abnormally.\n", child_pid);
         }
      }
      printf("**** Done reaping processes ****\n\n");
45
      if (errno != ECHILD) {
         perror("waitpid error");
         exit(-1);
      } else { /* errno set to ECHILD when waitpid finds
          no child to reap */
50       printf("Reaping completed\n");
         exit(0);
      }
   }
```

On Line 17, we declare an array of PIDs so that we can keep track of the PIDs of every child we create. This array is filled up in the loop between Lines 21 and 25 by forking several times. From here, if we're a child, we'll print a statement, we'll sleep for a random amount of time, and we'll exit.

While these processes are executing, our parent process will be waiting on Line 37 to reap these processes. Note that our `waitpid` call allows us to specify the PID of the process we're reaping. In this case, we're reaping the processes in the order in which they are stored in the array. Since reaping a process acts like a "block," if every process except for the first one has finished, our program will be run inefficiently (we'll be stuck waiting for the first process to finish executing, which won't necessarily be first). Thus, there are pros and cons to this approach.

The `while` conditional on Line 37 stores the destroyed child process's PID into the variable `child_pid`. We then check whether this PID is positive (indicating success), and if it is, we'll check whether the child was reaped successfully or not. Finally, when there aren't any more processes left to reap, the macro `ECHILD` is returned by `waitpid`, and we make sure this is what was returned between Lines 46 and 51.

As a side-note, note that this code segment performs all of the forks followed by all of the reaps (this is in contrast to performing one fork, one reap, another fork, another reap, etc). In fact, it is more efficient to do it this way since it allows our computer to execute all of the processes in the most efficient manner.

In the above code segment, we reaped processes in the order in which they were created. That is, the reaping process was **deterministic** in the sense that we knew what to expect of the order of the processes being reaped. Next, we'll consider a reaping process that is **probabilistic** in the sense that there will be some randomization.

Here's the corresponding code segment:

Listing 55: Probablistic Reaping

```
/*
 * The parent will wait for each child, but
 * the order in which the output will appear
 * is non-deterministic.

 * Reaping children in no particular order.
 * pid = -1 parameter below defines wait set to
 * be all parent's processes. options = 0
 * makes waitpid suspend execution of calling
 * process until a child in wait set terminates
 * pid != -1 defines the wait set to be the child
 * process with the specified pid
 *
 * After all children have been reaped, next call to
 * waitpid returns -1 and sets errno to ECHILD
 *
 * Initially try 2 as MAX_CHILDREN.  To see output
 * different order in reaping try 12 as MAX_CHILDREN
 *
 */

#include <stdio.h>
#include <sys/wait.h>
#include <sysexits.h>
#include <stdlib.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>

#define MAX_CHILDREN  12

int main() {
   pid_t child_pid, pid;
   int idx, status;

   /* Creating children processes */
   printf("\n**** Forking processes ****\n");
   for (idx = 0; idx < MAX_CHILDREN; idx++) {
      if ((pid = fork()) < 0) {
         err(EX_OSERR, "fork error while creating
             children");
      }
      if (pid == 0) { /* child code */
         printf("Child %d with pid %d created.\n",
             idx, getpid());
45          sleep(rand() % 10); /* simulates task child
                is completing */
        exit(idx);
         } else { /* parent code */
            printf("Parent (pid %d) created child %d.\n",
                getpid(), idx);
         }
50    }

      sleep(2); /* Giving children a chance to finish
          printing created messages */
      printf("\n**** Reaping processes ****\n");
      while ((child_pid = waitpid(-1, &status, 0)) > 0) {
          /* notice -1 */
55       if (WIFEXITED(status)) {
            printf("Child (pid %d) finished (exit status
                %d).\n", child_pid, WEXITSTATUS(status));
         } else {
            printf("Child (pid %d) terminated
                abnormally.\n", child_pid);
         }
60    }
      printf("**** Done reaping processes ****\n\n");

      if (errno != ECHILD) {
         perror("waitpid error");
65       exit(-1);
      } else { /* errno set to ECHILD when waitpid finds
          no child to reap */
         printf("Reaping completed\n");
         exit(0);
      }
70 }
```

What's happening here? In the beginning of the code segment, we're just creating 12 processes, and we're printing the PIDs of the children. Meanwhile, on Line 54, we're reaping the children. However, note that we're now using -1 as the first parameter of our `waitpid` calls. So, we're reaping *any* process that finishes until we're out of processes. Unlike the previous example, this example doesn't enforce any ordering in the reaping process. If we run the program many times, the order in which our processes are reaped will change.

Unix I/O

So far, we've covered Standard I/O, but there's also Unix I/O. What's are the differences?

- Standard I/O is built using Unix I/O (so we're going backwards).
- Standard I/O is generally buffered, whereas Unix I/O doesn't feature buffers.
- A file in Unix is just a sequence of bytes, and all I/O devices (e.g. keyboards, screens, and disks) are modeled as files in Unix.

It seems like Unix I/O is older, worse, and harder to use than Standard I/O. So why are we using it? Because Unix I/O permits us permits us to not only read and write from files but also to send data between processes

Before we get started with Unix I/O, we need to learn about file management in Unix.

Each process has a **file descriptor table** which stores the set of files that the given process has open. For example, when we use `fopen()`, an entry will be added to the file descriptor table. By default, each process has three files listed in the file descriptor table: standard input, standard output, and standard error.

Essentially, we can treat the file descriptor table as an array, and each entry in the table is a file. A file can be opened more than once; if this is the case, there will be more than one entry with the same file.

So what does each entry in the file descriptor table store? First, there's some general information, like how far we are into the file (opening a file a second time would reset this entry). Information about a file related to permissions, size, and type are stored in a table called the **inode table**.

There's also an important piece of information, known as the **reference count**, which tells us how many processes are associated with an entry.

When we call the `fork()` command, the entire file descriptor table is copied over. If we then use the child process to execute a command, the address space gets modified; however, the file descriptor table does not get modified. What does this mean for us? When we perform `exec` calls, the file descriptor table is preserved.

File Operations

There are four primary system calls associated with Unix I/O. Their headers are listed below:

1. `int open(const char *filename, int flags)` or `int open(const char *filename, int flags, mode_t mode)`.
2. `ssize_t read(int fd, void *buffer_size, size_tn`.
3. `ssize_t write(int fd, const void *buffer_size, size_t n)`.
4. `int close(int fd)`.

A process can request access to a file using the `open()` system call. Upon success, the kernel returns a file descriptor (an index in the file descriptor table). We can then modify the file using the `read()` and `write()` functions. Once we're finished, we can use the `close()` function to close the file.

Here's an example:

Listing 56: Unix I/O Example 1

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <err.h>
#include <sysexits.h>
#include <string.h>

/* same as 0666, but a bit more symbolic (we could use
    #define DEF_MODE 0666 )*/
#define DEF_MODE (S_IRUSR | S_IWUSR | S_IRGRP |
    S_IWGRP | S_IROTH | S_IWOTH)

int main() {
   int fd;

   fd = open("message.txt", O_WRONLY|O_TRUNC|O_CREAT,
       DEF_MODE);
   if (fd == -1) {
      err(EX_OSERR, "can't open message.txt");
   } else {
      char msg[] = "Hi there!";
      write(fd, msg, strlen(msg));
   }

   if (close(fd) == -1) {
      err(EX_OSERR, "closing file failed\n");
   } else {
      printf("File message.txt has been created\n");
   }

   return 0;
}
```

First off, note that we use `open()` to open the file `message.txt` with various options that don't concern us right now. After the `open()` fuction is called on Line 16, the variable `fd` will be either -1 (upon failure) or 3 (upon success; this is the index directly after standard input, standard ouput, and standard error).

If opening the file succeeds, we'll set the string `msg` to "Hi there!," and we'll use `write()` to write to the file. Note the parameters here. The first parameter is the file descriptor, the second parameter is a pointer to the data, and the last parameter is the number of bytes we're requesting to write. Note that we don't actually need to have space for our null character—requiring space for a null character is a C construct, and Unix does not follow the same constructs.

Finally, we close the file with the `close()` call to indicate we're done processing our file. The `close()` function returns -1 upon failure, so we need to make sure that worked as well.

Okay, so this is an example of writing to files. What about reading to files? This is captured below:

Listing 57: Unix I/O Example 2

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>
#include <sysexits.h>

#define LENGTH 9

int main() {
    int fd;
    char buf[LENGTH];
    size_t bytes_read;

    fd = open("message.txt", O_RDONLY);
    if (fd == -1) {
        err(EX_OSERR,"Cannot open file");
    } else {
        int i;
        bytes_read = read(fd, buf, LENGTH);
        if (bytes_read != LENGTH) {
            err(EX_OSERR, "Problem reading data");
        }
        for (i = 0; i < LENGTH; i++) {
            printf("%c", buf[i]);
        }
    }

    if (close(fd) == -1) {
        err(EX_OSERR, "Closing file failed");
    }

    return 0;
}
```

Like before, we'll open the file and set the file descriptor to `fd`. The option specified in the last parameter of `open()` specifies that we're only allowed to read the file.

Upon success, we'll use the fact that `read()` returns the number of bytes read, and we'll store that return value into `bytes_read` (we just use this as a sanity check to make sure `bytes_read == LENGTH` holds). Again, taking note of the parameters of `read`(), we see that the first parameter is the file descriptor, the second parameter is a pointer to where we want to store the data, and the third parameter indicates the length of what we want to read.

Subsequently, we check to make sure that we've read in the desired number of bytes. If so, we'll print what we read (note that we use the `%c` format specifier rather than the `%s` format specifier since there's no null character), and we'll close the file.

We can use these Unix I/O commands to read standard input and standard output. How?

- Use 0 as our file descriptor for standard input.
- Use 1 as our file descriptor for standard output.
- Use 2 as our file descriptor for standard error.

This is demonstrated below:

Listing 58: Unix I/O Example 3

```c
/*
  Illustrates the file descriptors associated
  with standard input (0), standard output (1).
  Remember standard error is (2).  To run the
  example, enter a string with a length of at
  least 5 characters.
 */
#include <stdio.h>
#include <unistd.h>

#define LENGTH 5

int main() {
   char buffer[LENGTH];

   /* Use STDIN_FILENO instead of 0 */
   read(0, buffer, LENGTH);

   /* Use STDOUT_FILENO instead of 1 */
   write(1, buffer, LENGTH);

   write(STDOUT_FILENO, "Bye\n", 4);

   return 0;
}
```

What are we doing? We read a string of up to length 5 from standard input, we write whatever we inputted to standard output, and we also print "Bye."

Although using 0 and 1 works for our file descriptors, we conventionally use the macros `STDIN_FILENO` and `STDOUT_FILENO`.

23 Wednesday, July 17, 2019

Recall that each entry in the file descriptor table represents a file. Each entry has some general information, like how far we are in the file, as well as the reference count (represents the number of processes associated with the entry), and the inode, which contains some metadata about the file.

If we view the reference count of a single process, it'll just be one. If we fork the process, the address space and file descriptor table are duplicated; however, the reference count will be increased by one. Moreover, if the child were to perform an `exec` call, the address space "becomes" the address space of the new program (it gets overriden), but the file descriptor table will stay the same.

Unix I/O Redirection

The `dup2()` function allows us to perform input/output redirection.

Suppose we have opened a file `data.txt`, and we want to redirect its contents to standard input. We'd first open the file so that `data.txt` would be stored in the third index of the file descriptor table (after standard input, standard output, and standard error). Now, we can call the `dup2` function, which has function header `int dup2(int oldfd, int newfd)`. For instance, if we were to write `dup2(3, STDIN_FILENO)`, we'd be redirecting the contents of `data.txt` to standard input. Subsequently, if we call `read(STDIN_FILENO, buffer, 8)`, we'd be reading the first eight characters in standard input, which happens to be what we redirected from `data.txt`.

In a similar manner, if we perform output redirection on standard output to a file, printing to standard output would actually print to the file we being redirected to.

Here is an example:

Listing 59: Dup2 Example 1

```
/*
 * Input/output redirection example
 *
 * Try 1: dup2Ex1
 *        input a string with 8 characters
 *
 * Try 2: dup2Ex1 data.txt
 *        in data.txt you need to have 8 characters
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <err.h>
#include <sysexits.h>

#define NAME_LENGTH 8

int main(int argc, char **argv) {
    char buffer[NAME_LENGTH]; /* not including \0 */
    int fd;

    if (argc > 1) {
        if ((fd = open(argv[1], O_RDONLY)) < 0) {
            err(EX_OSERR, "File opening failed.\n");
        }
        if (dup2(fd, STDIN_FILENO) < 0) {
            err(EX_OSERR, "dup2 error");
```

```
      }
      close(fd); /* we need it otherwise resource leak
          */
   }
35
   read(STDIN_FILENO, buffer, NAME_LENGTH);
   write(STDOUT_FILENO, buffer, NAME_LENGTH);

   exit(0);
40 }
```

What is this program doing? If we run it with command line arguments, we'll open the specified file and redirect its contents to standard input. We'll then read from standard input and print its contents to standard output.

Suppose we execute the program without any command line arguments. If we type in "Maryland," the program will simply print "Maryland" to standard output (due to Lines 36 and 37 in the program above).

Now suppose we specify the file `data.txt`, which has contents "Hello." The `dup2` function on Line 30 will redirect standard input to map to the contents of our file. Thus, the read statement on Line 36 will read from `data.txt`, and we'll be writing the contents of `data.txt` to standard output (so the output is "Hello.").

Here's another example:

Listing 60: Dup2 Example 1

```
/*
 * Input/output redirection example
 *
 * Try 1: dup2Ex2
 *        input a string with 8 characters
 *
 * Try 2: dup2Ex2 data.txt
 *        in data.txt you need to have a character
   with 8 characters
 *
 * Try 3: dup2Ex2 data.txt output.txt
 *        output.txt will be overwritten with contents
   of data.txt
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <err.h>
#include <sysexits.h>

#define NAME_LENGTH 8

#define FILE_PERMISSIONS 0666

int main(int argc, char **argv) {
   char buffer[NAME_LENGTH]; /* not including \0 */
   int fd;

   if (argc > 1) {
      /* If we have a second argument that represents
          the input file */
      if ((fd = open(argv[1], O_RDONLY)) < 0) {
         err(EX_OSERR, "File opening (read) failed");
```

```
        }

        if (dup2(fd, STDIN_FILENO) < 0) {
           err(EX_OSERR, "dup2 (read) failed");
        }

        close(fd); /* Releasing resource */

        /* If we have a third argument that represents
            the output file */
        if (argc == 3) {
           if ((fd = open(argv[2], O_WRONLY | O_CREAT |
               O_TRUNC, FILE_PERMISSIONS)) < 0) {
              err(EX_OSERR, "File opening (write)
                  failed");
           }

           if (dup2(fd, STDOUT_FILENO) < 0) {
              err(EX_OSERR, "dup2 (write) failed");
           }

           close(fd); /* Releasing resource */
        }
     }

     /* At this point we are ready for reading/writing */

     read(STDIN_FILENO, buffer, NAME_LENGTH);
     write(STDOUT_FILENO, buffer, NAME_LENGTH);

     exit(0);
}
```

In this example, we can optionally provide command line arguments for input and output redirection. If we provide one file name, we'll direct its contents to standard input. If a second file is specified, we'll redirect its contents to standard output. Finally, we'll read whatever standard input is pointing to, and we'll write it to wherever standard output is pointing to.

Also, note that we check whether the return value of `dup2` is negative, which would indicate failure.

Here is one last `dup2` example:

Listing 61: Dup2 Example 3

```
#include <stdio.h>
#include <sys/wait.h>
#include <sysexits.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>

#define MAX_LEN 80

void print_powers(int limit) {
   int i = 0;

   for (i = 0; i <= limit; i++) {
      printf("%d\n", i * i);
   }
}

int main() {
   char filename[MAX_LEN + 1] = "results.txt";
   int fd;

   if ((fd = open(filename, O_CREAT | O_WRONLY, 0666))
       < 0) {
      err(EX_OSERR, "File opening failed\n");
   }
   printf("Results can be found at %s\n", filename);
   dup2(fd, STDOUT_FILENO);       /* redirecting */
   close(fd);                     /* releasing resource
       */

   print_powers(10);

   return 0;
}
```

First, we're opening a file called `results.txt`, and we're redirecting standard output to point to the contents of this file, and we release the file descriptor `fd` by closing it.

Finally, we call `print_powers`, which uses `printf` to print the first `limit` perfect squares. Since `printf` defaults to printing to standard output, we'll actually be printing to the file that we opened.

Introduction to Pipes

Most of our discussion on pipes will be done next class, but we'll briefly introduce them today.

A **pipe** is used to combine two or more commands and use the output of one command to act as the input to another command. Piping takes place in a left-to-right manner in Unix by separating targets with the vertical bar `|`. Less formally, a pipe can be viewed as an area in which information can be exchanged.

As a basic example, suppose we have a program `EngToSpa` and `SpaToFre`, which are English-to-Spanish and Spanish-to-French translators. Now suppose we want to translate the word, "dog" from English to French. Instead of writing a new English-to-French translator, we can just utilize what we already have with piping. Typing `./EngToSpa | ./SpaToFre` in Unix and typing in "dog" would produce our desired result (it would use the Spanish output and use it as the input for the second program).

But, this is a little bit inconvenient for the user. If we've got a lot of languages, the user's going to need to keep on typing vertical bars to try and find a short path to a language when a direct path might not exist. The programmer can simplify the amount of work necessary on the user's end by piping inside of a C program.

In C, there's a `pipe()` function which takes in an integer array of size two. The first entry in the array needs to be the file descriptor for the read end of the pipe, whereas the second entry acts as the write end of the pipe. If a process tries to read before anything is written to the pipe, the process is suspended until something is written.

24 Friday, July 19, 2019

More on Pipes

Recall that the goal of piping is to exchange data between two processes. This is particularly helpful when we're exchanging data between a parent process and a child process. In particular, we can create a pipe and fork the process so that the child gets the pipe as well. A pipe is completely determined by two file descriptors: a read end, and a write end. When we call the `pipe()` function in C, we pass in an array whose first entry is the read end and second entry is the write end.

Below is an illustrative example of how piping works:

Listing 62: Piping Example 1

```c
#include <stdio.h>
#include <sys/wait.h>
#include <sysexits.h>
#include <err.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <fcntl.h>

#define MAX_LEN 80

void print_powers() {
   int i = 0, limit = 4;

   for(i = 0; i <= limit; i++) {
      printf("%d\n", i * i);
   }
}

int main() {
    pid_t child_pid;
    int pipe_fd[2];
    char filename[MAX_LEN + 1];

    pipe(pipe_fd);
    child_pid = fork();

    if (child_pid) { /* parent code */
       close(pipe_fd[0]); /* closing pipe's read end */
       printf("Enter filename for results: ");
       scanf("%s", filename);  /* reading filename */
       /* sending filename */
       write(pipe_fd[1], filename, strlen(filename) +
           1);
       close(pipe_fd[1]); /* closing pipe's write end
           */
       wait(NULL); /* reaping */
    } else { /* child code */
       int fd;

       close(pipe_fd[1]); /* closing pipe's write end
           */
       read(pipe_fd[0], filename, MAX_LEN + 1); /*
           reading file name */
       close(pipe_fd[0]); /* closing pipe's read end */
       fd = open(filename, O_CREAT | O_WRONLY, 0666);
       dup2(fd, STDOUT_FILENO); /* redirecting */
       close(fd); /* releasing resource */

```

```
        print_powers();
    }

    return 0;
}
```

In this program, we're forking the parent process, and we're sending a file name to the child. Subsequently, the child uses this file name as the destination for its processing (perfect squares up to 16). The `pipe()` call on Line 25 is what initializes our pipe.

In the parent code, note that we're closing the read end of the pipe. The reason why this is done is because it's sending data to the child (thus, the read end is not necessary). Subsequently, we prompt the user for a file name, and we send it over to the child by using the pipe. Once we're done sending it over with the `write()` call, we'll close the write end of the pipe, and we'll wait for the child to reap.

From there, the child's code executes. The child will be reading data, so it doesn't have any use for the write end of the pipe. Thus, the write end of the pipe is closed. Subsequently, it reads from the read end of the pipe and closes it. It can now process with the file name provided.

Here's an example that combines `dup2` and `pipe` calls.

Listing 63: Piping Example 2

```
#include <stdio.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <unistd.h>
#include <err.h>
#include <sysexits.h>

#define MAX_STR_LEN 80

int main() {
    int pipe_fd[2];
    pid_t child_pid;
    char value[MAX_STR_LEN + 1];

    printf("Enter number: ");
    fgets(value, MAX_STR_LEN + 1, stdin);

    if (pipe(pipe_fd) < 0) { err(EX_OSERR, "pipe
        error"); }
    if ((child_pid = fork()) < 0) { err(EX_OSERR, "fork
        error"); }

    if (child_pid) { /* parent code */
       close(pipe_fd[0]);   /* closing read end */
       write(pipe_fd[1], value, MAX_STR_LEN + 1); /*
           placing data in pipe */
       close(pipe_fd[1]);   /* closing write end */
       wait(NULL);  /* reaping */
    } else { /* child code */
       close(pipe_fd[1]);   /* closing write end */

       if (dup2(pipe_fd[0], STDIN_FILENO) < 0) {
           err(EX_OSERR, "dup2 error"); }
       close(pipe_fd[0]);
       execlp("./table", "table", NULL);
       err(EX_OSERR, "exec error");
    }

    return 0;
}
```

This is similar to the previous example. Pretty much, we're reading a number into `value`, we're creating a pipe, and we're forking. The parent closes the read end of the pipe (it has nothing to read!), writes the inputted value to the write end of the pipe (for the child), and closes the write end of the pipe. It then waits for the child to finish execution.

The child closes the write end of the pipe (it has nothing to write), and it maps its standard input to read from the read end from the pipe. Consequently, it closes the read end of the pipe (this is allowed because we've already mapped our standard input), and the child executes the program "table." Now, if the program "table" takes in a value from standard input, it will instead read the value that was put into the pipe.

At this point, we should be able to implement an EngToFre program that uses piping with the outputs of EngToSpa and SpaToFre (create a pipe and two children; the first child represents EngToSpa, and the second child represents SpaToFre).

Introduction to Concurrency

Concurrency is the ability to use different parts of a program in an out-of-order sequence without affecting the final desired result. A **thread** is a lightweight process that specifies an execution sequence in a process. We've already briefly introduced threads—recall that the minimal representation of a thread is a stack and a program counter. By quickly switching between threads, we can make it seem as if different execution sequences are executing at the same time. An example of how threads might be used is a GUI displaying clocks in different timezones.

If we have multiple threads, there are some things that they share. In particular, threads share heap memory, global/static memory, open files, shared libraries, and virtual addresses.

How do we use threads in C? With the `pthread.h` library.

This library includes a data type called `pthread_t`, which allows us to represent thread IDs. There's also a built-in `pthread_create` function which is used to initialize (but not start) a thread's process. Finally, there's a `pthread_join` function which allows the thread to begin executing.

Here is a basic example:

Listing 64: Threads Example 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

struct point {
   int x, y;
};

void *print_point(void *pointp);

int main() {
   pthread_t tid; /* thread id */
   struct point pt = {3, 5};

   if (pthread_create(&tid, NULL, print_point, &pt) !=
       0) {
      fprintf(stderr, "pthread_create failed\n");
      exit(1);
   }

   /* reaps thread blocking until thread terminates */
   printf("Waiting for child to finish\n");
   if (pthread_join(tid, NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
25     }

       printf("In main after thread finished\n");

       return 0;
30  }

    /* code executed by the thread */
    void *print_point(void *pointp) {
       struct point arg = *(struct point *)pointp;
35
       sleep(2); /* simulating some work */
       printf("Point: (%d, %d)\n", arg.x, arg.y);

       return NULL;
40  }
```

In the above example, we've declared a thread called `tid`. This is initialized with the `pthread_create` function, where the thread ID is passed in as an out parameter. In this class, we'll always have the second argument of `pthread_create` equal to `NULL` (the second argument allows us to use custom initializations). Subsequently, the third parameter of `pthread_create` specifies what the task the thread should be executing (here, it's the function `print_point`), and it is followed by any parameters that the function might need.

The function prototype of the task that a thread is performing will always return a void pointer, and it will always take in a void pointer. This function header cannot change.

In our program, if this initialization succeeds, we'll use the `pthread_join()` function (again, in our class, the second parameter will always be `NULL`). This function will tell the thread to finish executing. Finally, we'll print the `printf` statement on Line 27.

We'll continue with concurrency next class.

25 Monday, July 22, 2019

Retrieving Values from Threads

Last time, we saw how we can initialize a thread and make it perform a task. But, what if the thread computes some important value and we want to retrieve it?

Here is an example:

Listing 65: Retrieving Values

```c
   #include <stdio.h>
   #include <stdlib.h>
   #include <pthread.h>

 5 void *get_square(void *args);

   int main() {
      pthread_t tid;
      void *result_ptr = NULL;
10    int argument;

      printf("Input value to compute square: ");
      scanf("%d", &argument);

15    pthread_create(&tid, NULL, get_square, &argument);
      pthread_join(tid, &result_ptr); /* notice use of &
          */
      printf("Square of %d is %d \n", argument, *(int
          *)result_ptr);

      free(result_ptr);
20
      return 0;
   }

   void *get_square(void *args) {
25    int argument = *(int *)args;
      int *answer_ptr = malloc(sizeof(int));

      *answer_ptr = argument * argument;

30    return answer_ptr;
   }
```

Like we mentioned last class, the function that specifies the task that the thread will be performing will always have the same function header: it always takes in a void pointer, and it will always return a void pointer.

If the thread's function always takes in a pointer, how do we perform tasks that require more than one parameter? This is solved by declaring a pointer to a structure, where the structure contains all of the necessary fields. This is exactly what we've done in the above example.

On Line 8, we've defined a thread that will be used to perform the `get_square` task. The `pthread_create()` call on Line 15 initializes `tid`, and it also specifies the task that the thread will be computing. Moreover, as we mentioned last class, the second argument of the function will always be `NULL`. Finally, the last parameter specifies the parameter of the function we're initializing the thread to.

Now, the `pthread_join()` function will tell the program to execute the task that the thread was assigned. If we want to keep the value that the function is returning, we need to allocate memory and return that value. Now, how do we retrieve the value? We pass an out-parameter when we're calling the `pthread_join()`

function (in this case, `result_ptr` acts as our out-parameter). Thus, after Line 16, `result_ptr` will store the returned value. The caller is responsible for freeing the dynamically allocated memory.

When we're creating multiple threads, it's good practice to create all of the threads at first and join them all afterwards (rather than creating one, joining it, creating another, joining it). Why? Because otherwise, we'd be executing our program sequentially, which doesn't actually use concurrency.

Here is an example which uses more than one thread:

Listing 66: Multiple Threads

```c
   #include <stdio.h>
   #include <stdlib.h>
   #include <pthread.h>
   #include <time.h>
 5 #include <unistd.h>

   #define THREAD_CT 8

   void *print_stuff(void *ptr) {
10    int k, id = *(int *) ptr;

      for (k = 0; k <= 4; k++) {
         printf("Thread %d, ", id);
         fflush(stdout);
15       printf("loop %d\n", k);
         fflush(stdout);
         sleep(rand() % 2);  /* sleep 0 or 1 seconds */
      }
      printf("Thread %d exiting\n", id);
20
      return NULL;
   }

   int main() {
25    pthread_t tids[THREAD_CT + 1];
      int i, ids[THREAD_CT + 1];

      for (i = 1; i <= THREAD_CT; i++) {
         ids[i] = i;
30       pthread_create(&tids[i], NULL, print_stuff,
             &ids[i]);
         printf("Thread 0 created thread %d\n", i);
      }

      for (i = 1; i <= THREAD_CT; i++) {
35       pthread_join(tids[i], NULL);
         printf("Thread 0 reaped thread %d\n", i);
      }

      return 0;
40 }
```

In our main, we create `THREAD_CT` threads, and we keep track of each of their IDs in our array `tids`. The second loop between Lines 34 and 37 will print some details about the thread, and we'll finally terminate the program.

Note that when we're creating the threads, we use `&id[i]` as the last parameter of `pthreads_create()` rather than `&i`. Why? Because if we were to use `&i` as the parameter to `print_stuff`, there would be a **data race**. Since `i` is changing inside of the loop and multiple threads depend on this same variable, we'll have unexpected output. The solution is to make sure that each variable has its own in-parameter.

Locks, Mutexes, and Semaphores

Suppose we want to use multiple threads to compute one value (such as the maximum in an array). The variable that would store the array maximum would need to be "shared" among all of the threads. That is, all of the threads should be able to modify the array with a new maximum it finds. However, it's important to make sure that only one thread accesses the shared variable at once — otherwise, we might have a data race. In general, **when one thread is accessing common variable, no other thread should be accessing it.**

Here's another example in which locks might be important. Consider the following code segment:

Listing 67: Accessing Critical Section without Synchronization

```c
#include <stdio.h>
#include <stdlib.h>

#define LOOPS 10000000

static int count = 0;

void *counter(void *args) {
   int i;

   for (i = 0; i < LOOPS; i++) {
      count++;
   }
   printf("Executed %d times\n", i);

   return NULL;
}

int main() {
   pthread_t tids[2];

   pthread_create(&tids[0], NULL, counter, NULL);
   pthread_create(&tids[1], NULL, counter, NULL);
   pthread_join(tids[0], NULL);
   pthread_join(tids[1], NULL);
   printf("Count: %d\n", count);

   return 0;
}
```

This program creates two threads, each of which execute the function `counter`. The function `counter` increments the global variable `count` ten million times. Finally, we print out the value of `count`. While one might expect `count` to have a value of $20,000,000$, it turns out that this is not the case. Each time we run the program, we should expect to get a quantity between $10,000,000$ and $20,000,000$. Like the array maximum problem described, this discrepancy is caused by a data race.

How do we control the threads' access to a variable? With locks, mutexes, semaphores. First, we'll distinguish between the three:

1. A **lock** only permits one thread to access data that is locked. This lock is not shared with any other processes.
2. A **mutex**[3] is just like a lock; however, it can be shared by multiple processes.
3. A **semaphore** is a same as a mutex; however, it permits a predefined number of threads to access the shared data space.

[3]Short for mutual exclusion

Also, the data section that only one active thread should be accessing at once (in our example, this would be the `count` variable) is formally known as the **critical section**.

How do we use these in C? Let's first look at the code segment that solves our `count` issue:

Listing 68: Accessing Critical Section with Synchronization

```
   #include <pthread.h>
   #include <stdio.h>
   #include <stdlib.h>

 5 #define LOOPS 10000000

   static int count = 0;
   static pthread_mutex_t mutex;

10 void *counter(void *args) {
      int i;

      for (i = 0; i < LOOPS; i++) {
         pthread_mutex_lock(&mutex);
15       count++;
         pthread_mutex_unlock(&mutex);
      }
      printf("Executed %d times\n", i);

20    return NULL;
   }

   int main() {
      pthread_t tids[2];
25
      pthread_mutex_init(&mutex, NULL);

      pthread_create(&tids[0], NULL, counter, NULL);
      pthread_create(&tids[1], NULL, counter, NULL);
30    pthread_join(tids[0], NULL);
      pthread_join(tids[1], NULL);
      printf("Count: %d\n", count);

      pthread_mutex_destroy(&mutex);
35
      return 0;
   }
```

Here, we see that there's a data type called `pthread_mutex_t`. This is used to declare mutexes (mutexes sounds strange – maybe mutices? mutexi? mutii?).

How do we use this mutex? Whenever we're going to access the critical section, we'll need to acquire the lock by using the `pthread_mutex_lock()` function. Once we're done accessing the critical section, we'll let go of the lock with the `pthread_mutex_lock()` function. Both of these functions take in a pointer to a `pthread_mutex_t` type (which makes sense – the functions would need to modify this type to signal that the critical section shouldn't be accessed).

When one thread has acquired the lock, no other thread can access the critical section. More specifically, suppose two threads are trying to modify `count` at the same time. One thread will execute Line 14 before the other (and thus acquire the lock). When the second thread gets to Line 14, it'll realize that the lock has already been acquired. This thread will now be forced to wait until the first thread has finished accessing the critical section (which occurs after Line 16 is executed).

The con of synchronizing our code with mutexes, locks, and semaphores is time efficiency. This code segment runs slower than the previous (incorrect) code segment. One reason why might be that one thread needs to wait for the other before it can perform its task.

System and Unix Time

There are many different ways to measure time on a computer, a couple of which are described below:

1. **Wall time** (also known as "**elapsed real time**") is the actual time taken from the start of a computer program to the end. Internally, it's calculated by subtracting the ending time of a program by the starting time of a program.

2. **Process time** is the time your program was running without accounting for the time the program stopped for other programs or the time the program needed to wait for I/O.

Process time can further be divided into two categories: **user time**, which represents the amount of time the operating system is running your code, and **kernel time**, which represents the time the operating system is running system code (i.e. when we're handling system calls, like `fork()`).

We can obtain measurements on how long a program takes to execute using the `time` command in Unix. The general syntax for this command is `time [executable]`.

As an example, consider the following code:

Listing 69: Time Example

```c
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>

/*
 * Execute "time sleeper"
 */
int main() {
   long x = 900000000;

   while(x-- > 0);
   printf("About to sleep for 5 seconds\n");
   sleep(5);
   printf("Done\n");

   return 0;
}
```

If we were to compile the file and place an executable with the name "sleeper" in our directory, typing `time sleeper` would produce a result similar to what follows below:

```
1.844u 0.001s 0.06.85 ... ... ... [other information] ... ... ...
```

The first entry, `1.844` denotes the user time in seconds. The second entry, `0.001` denotes the kernel time in seconds. The third entry, `0.06.85`, denotes the wall time in seconds. This output is kind of expected since we have a `sleep()` call in our function, which does nothing for five seconds.

Date and Time Functions

26 Friday, July 26, 2019

Today, we're wrapping up multithreading.

Thread Safety

Recall that when we have multiple threads, we need to make sure our code is **thread safe**. As we've already seen, two threads attempting to modify a shared variable can lead to data races.

Here's another example of a function that threads might use, which would be considered unsafe:

Listing 70: Thread Unsafe Function

```
char *itoa(int n) {
    static char buffer[50];
    sprintf(buffer, "%d", n);
    return buffer;
}
```

Why is this thread unsafe? Suppose two threads are using this same function. When the first thread calls the function with an integer, we'll store the string-equivalent of that integer into `buffer`. The function returns a pointer to the buffer (note that this is actually fine since the `buffer` variable is static). Now if a second thread calls this same function, it'll modify the same memory address, which means that the first thread and second threads would both be pointing to the second result.

How do we solve this issue? We can protect calls to this function with a binary semaphore and make deep copies before allowing other threads to access the function. Alternatively, we can use out-parameters to initialize the return value.

A function that can be interrupted and resumed at a later time without hampering its earlier course of action is said to be **reentrant**. There are some conditions that a function must satisfy to be reentrant, like not using any global or static data. Also, a reentrant function cannot call another non-reentrant function.

Every reentrant function is thread safe; however, not every thread safe function is reentrant. As an example, the `itoa` function could be modified to include locks. In that case, `itoa` still wouldn't be reentrant (it has a static variable!), but it would be thread safe.

Libraries

A **library** is a collection of object files that provide compiled functions to perform some related task. Libraries are linked into programs either prior to execution or during program execution.

There are a few options that we have when we're sharing code:

- We can give out the source code. This gives the client access to everything you wrote. The downside is that it needs to be recompiled and re-linked by every user, and it also exposes implementation details.
- We can give out the object code. This won't require recompilation of the object code; however, it'll require re-linking of the application that'll be using it.

Surprisingly, giving out libraries is even easier than both of these options. Giving out libraries doesn't even require re-linking of the application using it.

Another benefit of using libraries is that they only include what is being used into the executable. For instance, if a library contains hundreds of functions, but we only use one, the library will only compile the ones that are being used. The linker has to search through an object file and find each function being used. However, this process can be sped up with indexing.

The `nm` Unix command has general syntax `nm [.o file]`, and it returns a list of functions and other symbols being used in the compiled *object* code. This is helpful because it looks only at machine code rather than C code.

The primary types of libraries are summarized below:

1. An **archive** library is linked into a program as a part of the linking phase of compilation. It requires space in each executable that uses them, and a benefit of using them is their ease of use. These types of libraries consist of only one `.a` file (similar to a `.zip` file), where everything is stored.

2. A **shared library** allows different executables to share the same library code, ultimately saving disk and memory space. Shared object files have the file extension `.so`. These typically function more efficiently than archive libraries.They are linked either at program startup or during execution, and only one copy is needed for the entire system. Using command line, we can use the `ldd` command, which has syntax `ldd [executable file]`, to tell us which shared libraries an executable file depends on.

The `ln` command in Unix can be used with the flag `-s` to to create symbolic links between two or more files. The general syntax is `ln -s [file_name] [link_name]`.

What are links used for? Essentially, they're used to redirect one file to another piece of data that already exists. This can help us save space when we're copying files to other locations in memory. Say we have a file called `file1.txt` that has the contents "File 1." We can now create a new file called `file2.txt`, and executing `ln -s file1.txt file2.txt` will make File 2 "point" to the contents of File 1. Thus, if we were to execute `cat file2.txt`, we'd see "File 1" as output.

Symbolic linking can also be done with other types of files, including directories.

27 Monday, July 29, 2019

Dynamically-loaded Libraries

Last class, we introduced two types of libraries: shared libraries and static (or archive) libraries. A **dynamically-loaded library** is a different way to use a shared library; however, it is NOT a new type of library.

Functions in dynamically loaded libraries can be loaded into an application during runtime, not just at program startup[4]. Dynamically loading a library requires more work for the programmer; however, it makes the program more convenient for the user.

It's important to remember that static libraries cannot be dynamically loaded.

If static libraries requires space in every executable AND cannot be dynamically loaded, why do we even use them? First, static libraries allow for a quicker startup of the program since we don't need to load the functions needed at runtime. Essentially, it permits us to "pay" a price (in time) at compile time for a faster startup at runtime.

Nelson says that we should be able to name the two types of libraries and list their advantages/disadvantages for the final exam.

Introduction to Optimization

In machine code, not all instructions take the same amount of type. A classical example of this is dividing integers versus floating point numbers—dividing an integer quantity by a constant is significantly faster (by a factor between 5 to 10) than dividing a floating point number. There are other optimizations that the compiler automatically performs. It's helpful to understand what compilers can and can't do, as well as the time it takes on the hardware.

First off, processors use **caching**, which is a method of keeping copies of recently accessed memory locations in fast storage. Consequently, future requests for that data are served up faster than is possible by accessing the data's primary storage location. Efficiency is maximized if the same cache items are used multiple times. Two principles of locality used by computer architecture when performing caching are listed below:

1. One of the two principles is based on **temporal locality**. This principle states that recently referenced items are likely to be referenced again in the near future.

2. The second principle is based on **spatial locality**. This principle states that items with nearby addresses tend to be referenced close together in time (like items in an array).

Next, processors perform **pipelining**, which allow parts of multiple instructions to execute simultaneously. For example, the processor might start to decode one instruction while loading the next one from memory. Some superscalar processors can execute two or more instructions at once. Pipelining can further be optimized with **branch prediction**, which is a process by which the processor guesses which way a branch (e.g. a conditional statement) will go, ultimately allowing the pipeline to stay full.

When conducting efficiency measurements, typically we'll run the program a set number of times and take the mean of the K fastest runs.When conducting these tests, it's important to have representative input samples. Why? Because some inefficient algorithms, like one that runs in $\mathcal{O}(n^2)$, might appear to be efficient for small values of n.

What are the sources of performance problems? There are a lot of reasons why some code might be inefficient. The following list names a few:

[4]This is how browsers allow for skins, plug-ins, etc

1. I/O Operations that are too small. (e.g. it is inefficient to read a file one character at a time; it is much faster to read entire strings or lines at once).

2. Poor algorithm implementations. Using an $\mathcal{O}(n^2)$ (or worse) algorithm is bad when n is large.

3. Caching memory that isn't reused.

Optimization is not about algorithms but rather how to convert algorithms into efficient code and how to refine code to make it run faster.

Compilers can be told to "optimize" your code. In gcc, the `-O` flag enables the optimizer, which makes modifications to the compiled program wherever possible. Optimization will **never** break your code; the actions taken by enabling the `-O` flag will only be safe changes. However, optimizing code might reveal latent bugs. Naturally, there are some limits on compiler optimization. The compiler has a limited understanding of the program, and there is a need to compile programs quickly.

Types of Optimizations

There are a few types techniques that compilers use to optimize code. Here, we'll discuss a few.

Code Motion

The following example demonstrates one way in which the compiler can optimize a code segment.

First, consider the original, pre-optimized code segment:

Listing 71: Unoptimized Compiler Code

```c
int i, j, k;
....
for (i = 0; i < 200; i++) {
    a[2 * i * j] = j * k + i;
}
```

Now here's the compiler optimized equivalent:

Listing 72: Optimized Compiler Code

```c
int i, j, k;
....
int prod_2j, prod_jk;
prod_2j = 2 * j;
prod_jk = j * k;
for (i = 0; i < 200; i++) {
    a[prod_2j + i] = prod_jk + i;
}
```

Why is this faster? In the original code, it's unnecessary to repeatedly multiply `j` and `k` since they're constant values. Likewise, it's unnecessary to keep on multiplying `2` and `j`. It's also important to remember that the compiler **automatically does this for us**. This second code segment is just showing how the compiler could potentially be interpreting this code.

This type of optimization (e.g. using variables to store repeatedly used constant values) is known as **code motion**. Another good example is here[5].

[5] https://stackoverflow.com/questions/5607762/what-does-code-motion-mean-for-loop-invariant-code-motion

Loop Unrolling

Loop unrolling is a type of processor optimization which allows for the contents of a loop body to be executed faster.

Here's an unoptimized code segment:

Listing 73: Unoptimized Compiler Code

```
/* Assuming n is even. */
for (i = 0; i < n; i++) {
    c[i] = a[i] + b[i];
}
```

A post-optimized equivalent might look something like this:

Listing 74: Optimized Compiler Code

```
for (i = 0; i < n - 1; i += 2) {
    c[i] = a[i] + b[i];
    c[i + 1] = a[i + 1] + b[i + 1];
}
```

We've "unrolled" the loop and now we only perform half the number of iterations. Why is this faster than the unoptimized segment? Because, as previously mentioned, modern processors can perform several parts of instructions simultaneously.

Dead Code Elimination and Other

This is really simple, so there's no example. Pretty much, the compiler will eliminate chunks of code that never get executed. So, if we added had something like, `if (false) { .... .... }`, the compiler wouldn't need to look at that. This is called **dead code elimination**.

We should also attempt to reduce the number of function calls we have since it takes time. This can be done by eliminating short functions (like, one line functions), and replacing them with parametrized macros (`#DEFINE`) or **inline functions** (with the `inline` keyword in C), which are used to tell the compiler that the function is a short one.

Amdahl's Law

Amdahl's Law is a formula which gives the theoretical speedup of a task that can be expected of a system whose resources are improved.

Let T denote the execution time of a program. Now suppose we have a function that takes α fraction (so we have $0 \le \alpha \le 1$) of the program execution time, and we can make this function k times faster. Then, the following equality holds:

$$T_{new} = (1 - \alpha)T_{old} + (\alpha T_{old})/k.$$

To get a better understanding of what this equation is saying, let's look at the edge cases:

- If we have a function that occupies the entire runtime of a program (that is, $\alpha = 1$), then we'll simply have $T_{new} = T_{old}/k$, which makes sense if we're making the function k times faster.
- If we have a function that occupies none of the runtime of a program (that is, $\alpha = 0$), then we'll have $T_{new} = T_{old}$ since making the function faster won't do anything if our program never executes it.

28 Wednesday, July 31, 2019

Recall that two-dimensional arrays in C are layed out in row-major order. Thus, it's more efficient to process a 2D array row-by-row (instead of column-by-column) in order to efficiently use the cache.

Memoization

Another optimization technique is to store intermediate values in computationally difficult tasks. This is known as **memoization**. The classical example is computing Fibonacci numbers using DP:

Listing 75: DP Fibonacci

```c
#include <stdio.h>
#include <stdlib.h>

unsigned long fib(unsigned int n) {
    static unsigned int table[100] = {0};
    if (n == 0 || n == 1) {
        return 1;
    }
    if (table[n]) {
        return table[n];
    }
    return table[n] = fib(n - 1) + fib(n - 2);
}
```

Without memoization, a recursive Fibonacci function runs in $\mathcal{O}(\phi^n)$, which is really bad. To do better, once we've computed a Fibonacci number, the number is stored for later use in the static array `table`. Since the variable is static, the variable will be shared with subsequent function calls. This reduces the number of recursive calls made when we use the function a lot.

Parametrized Macros

We've already seen that we can use `#define` to blindly substitute text in one place to another. However, `#define` can also be used to create **parametrized macros** where parameters are substituted along with the text substitution.

The general syntax for a parametrized macro is `#define(parameter-list) text`.

As an example, consider the macro `#define SUM(a, b) a + b`. We could then write something like `x = SUM(2, 6)`, and it would blindly substitute this text to become `x = 2 + 6`. Long parametrized macros can be extended to a new line using a backslash.

It is important to be careful with parametrized macros since they are essentially just substituting text. In particular, we need to pay attention to the precedence of operators as well as issues with post and pre-incrementation.

Let's look at an example in which a parametrized macro might fail to do what we want it to do:

The macro `#define SQUARE1(x) x * x` will fail when we call the macro with `SQUARE1(5 + 1)`. While we might want the answer to be 36, the preprocessor will instead perform a blind substitution, and we will compute $5+1\cdot 5+1 = 11$. The solution is to change the parametrized macro to `#define SQUARE2(x) (x) * (x)`.

Below are a few important things to remember about parametrized macros:

1. Macros are textually replaced. Consequently, they are much faster than function calls, which have overhead.

2. Macros cannot be recursive.
3. Macros cannot be type-checked by the preprocessor.
4. Macros might result in difficult bugs.

Virtual Memory

A computer has **virtual memory** and **physical memory** which are of importance when writing code. Virtual memory can be thought of as our entire memory space, which ultimately gives the programmer an illusion of having infinite memory. On the other hand, the physical memory space contains memory addresses that are actively being used.

For instance, a programmer is using data (i.e. creating a variable), parts of the virtual memory is mapped to physical memory, where the data is stored. For this reason, virtual memory is typically larger than physical memory.

These concepts of physical and virtual memory explain why when we call `fork()` on a program in C, the memory addresses of the child and parent processes are initially the same. The reason why is that the child has not been mapped to a physical memory address space yet.

Signals

Nelson says that we don't need to write code with signals on the final, but we need to understand signals conceptually.

A **signal** is a method of communication between two or more processes. There are many ways to send signals, some of which are listed below:

- We can send signals via the keyboard. The `CTRL + C` and `CTRL + Z` send the `SIGINT` (terminate process) and `SIGTSTP` (suspend process execution) signals, respectively.
- The `kill` command in Unix can be used to send signal to a process to terminate it.
- Software errors can also produce signals. For example, if we get a segmentation fault, a `SIGSEGV` signal is sent in order to indicate a segmentation violation.
- The `SIGCHLD` signal is sent to a parent process when the child process terminates.

29 Friday, August 2, 2019

Today is the last day of new content. After this, we'll just be reviewing for the final.

Back to Assembly

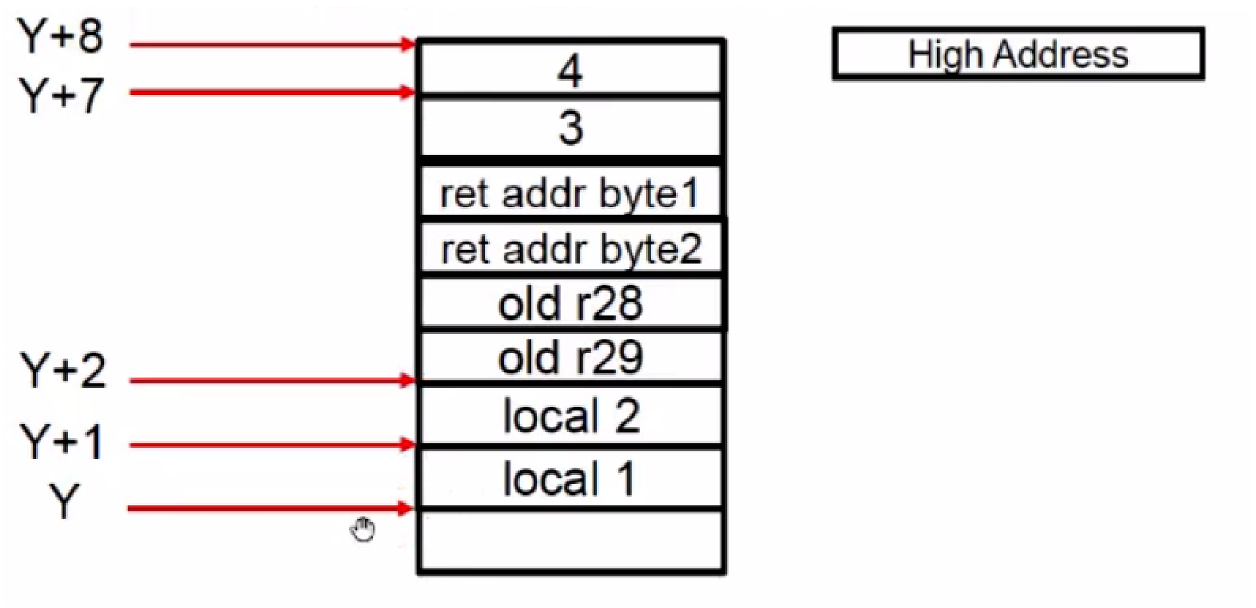
Recall that in Assembly, we use registers to pass and return values to and from a function. But, what if we have enough arguments to make us run out of registers? Likewise, recall that we use registers to keep track of local variables. What happens if we exhaust all of the registers, and we still want to store more variables?

In either of these scenarios, the solution is to use the stack. We can use the stack to hold arguments and local variables.

A **stack frame** is an area of memory in the stack that supports the execution of a function. If the function were recursive, each subsequent call to the function would

AVR Stack Frame

In AVR Assembly, we can define the `Y` register pointer to act as our frame pointer. We will be able to access local variables relative to this pointer. The following image depicts this scenario:



As depicted above, we can find our first local variable at the `Y + 1` (we get here by adding one byte to the `Y` register pointer), another local variable at `Y + 2`, and so on.

Okay, but how can we access these values? We've already seen that we can use `adiw` to increment a register pair. Since the `Y` register pointer resides in the register pair `r29:r28`, we could just perform the correct number of `adiw` operations on `r28`. But this is really slow — what if our stack is really big? The solution is to use the `ldd` instruction.

The `ldd` instruction has general syntax `ldd [register], [register pointer] + q`, where `register pointer` is either `Y` or `X` (we can't use `ldd` on the `X` register pointer), and `q` is some constant integer value. What does this instruction do? It loads the contents of `Y + q` into the specified register.

We can now look at an example of an Assembly program that uses the `ldd` example. For simplicity, let's look at the equivalent C code first:

Listing 76: C Program for Sum of Squares

```
int sum_sys(int a, int b) {
    int sum1 = a * a;
    int sum2 = b * b;
    return sum1 + sum2;
}
```

Essentially, we're going to be computing two squares separately, and we'll return their sum. Now here's the corresponding Assembly code:

Listing 77: Assembly: Factorial

```
;;; Example - A function that uses the stack to pass
    parameters
;;; and defines local variables in the stack.
;;; Program implements sum_sqrs(a, b) =  a * a + b * b

        .set SPH, 0x3e          ; stack pointer (high
            byte)
        .set SPL, 0x3d          ; stack pointer (low
            byte)

;;; Global data
        .data

pctd:   .asciz "%d "
a:      .byte 3
b:      .byte 4

;;; Program code
        .text

.global main
main:
        ;; calls sum_sqrs(3, 4), printing 25 as result
        call init_serial_stdio

        lds r22, a              ; pushing parameters (a
            consider first parameter)
        lds r23, b
        push r23                ; pushing parameters; b
            goes in first, a second
        push r22
        call sum_sqrs
        pop r22                 ; removing parameters
        pop r23
        call pint               ; printing the result
            left in r25:r24 by sum_sqrs
        call prt_newline

    cli                     ; stopping program
        sleep

        ret

sum_sqrs:
        ;; computes a * a + b * b
                                ; SETTING FRAME
        push r28                ; saves old frame /
            base pointer
        push r29
        in r28, SPL             ; retrieving stack
            pointer value and initializing Y
        in r29, SPH             ;
        sbiw r28, 2             ; allocating space for
            2 local variables; a byte each
        out SPL, r28            ; adjusting stack
            pointer for local variables
        out SPH, r29            ; END SETTING FRAME


        ldd r24, Y+7            ; first parameter (a)
        mul r24, r24            ; a * a
        mov r24, r0             ; result in r24
            (ignoring r25)
        std Y+1, r24            ; storing in first
            local variable

        ldd r24, Y+8            ; second parameter (b)
        mul r24, r24            ; b * b
        mov r24, r0             ; result in 24
            (ignoring r25)
        std Y+2, r24            ; storing in second
            local variable

        ldd r24, Y+1            ; accessing first local
            variable
        ldd r25, Y+2            ; accessing second
            local variable
        add r24, r25            ; sum
        clr r1                  ; we must always leave
            it as 0
        clr r25                 ; r25:r24 has the result



                                ; THROWING AWAY FRAME
        adiw r28, 2             ; throws away locals
        out SPL, r28            ; adjusting stack
            pointer
        out SPH, r29
        pop r29                 ; restoring old frame /
            base pointer
        pop r28                 ; END THROWING AWAY
            FRAME

        ret

pint:
        ;; prints an integer value, r23:r22 have the
            format string
        ldi r22, lo8(pctd)      ; lower byte of the
            string address
        ldi r23, hi8(pctd)      ; higher byte of the
            string address
        push r25
        push r24
        push r23
        push r22
        call printf
        pop r22
        pop r23
        pop r24
        pop r25

        ret

prt_newline:
        ;; prints newline
        clr r25
        ldi r24, 10
        call putchar

        ret
```

Since we haven't done Assembly in a while, we can review the basics again:

The `.set` directive allows us to define a symbolic constant. On Lines 5 and 6, we're declaring `SPH` and `SPL` to be the hexadecimal numbers `0x3e` and `0x3d` respectively. Why are we declaring these constants? The hexadecimal numbers `0x3e` and `0x3d` are special memory addresses, representing the high and low bytes of the stack pointer. We should know these values.

Next, the `.data` directive indicates that we're going to be initializing data. We've initialized the `pctd` label to be a null-terminated string by using the `.asciz` directive. Likewise, we've used the labels `a` and `b` to store 3 and 4.

Now we will indicate that our program code is beginning with the `.text` function. The `.global` directive is equivalent to the `extern` keyword in C. It allows the main function to be accessible from outside of the current file. What

Finally, we can start our `main` function. What happens when we reach this function? Whatever value is at the top of the stack is used to initialize the program counter (which indicates the next instruction to be executed). Thus, it's important to be careful — if we use the stack for another purpose, it's important to perform the appropriate number of pops. When `ret` is executed, whatever is at the top of the stack is removed, and the program counter will continue executing whatever is left. So far, we've been using the stack, but we haven't had to do any initializing of it.

In our main function, we first load the contents of `a` and `b` (which are 3 and 4) into registers `r22` and `r23`, respectively. There's nothing special about these registers — we could have used any pair of registers. Next, we push the two registers in reverse order (since `a` is our first argument and `b` is our second argument to the `sum_sqrs` function, we'll push `b` and then push `a`). Why do we push in reverse order? It's just a matter of convenience so that our first argument can be found at `Y + 7`, and the second argument can be found at `Y + 8` (refer to the diagram on the previous page). Now, we can call our function.

We've now called our `sum_sqrs` function. Let's look at what's going on in there.

A The Make Utility

The Make Utility allow us to simplify the process of compiling code. When we increase the size of our software, we'll likely have several files we need to deal with. It would be pretty inefficient to compile *all* of our files for a small change in a *single* file. So, this is where our utility come into play: they let us keep track of what's been modified and what needs to be re-compiled.

Suppose we have a program called `puzzles.c` and we want to run a public test named `public01.c`. Typically, we'd execute `gcc puzzles.c public01.c` and run the executable `a.out`. But, it turns out that we could have compiled these files separately with the `-c` flag (which is used to create the `.o` object file). That is, we could've run `gcc -o puzzles.c` and `gcc -o public01.c` separately to produce the `puzzles.o` and `public01.o` object files. The key takeaway here is that we can compile `.c` files individually, even if they don't have a main (However, at least one file needs a main).

So, what are the advantages to being able to compile files individually? If we're only modifying one file, we'll only need to re-compile that one file. However, we will *always* need to re-link the object files.

Now what? Now, we need to link these object files in order to create our executable. We can do this by typing `gcc -o public01 public01.o puzzles.o`, which will produce an executable called `public01` from the two object files we have.

The make utility uses something called a **Makefile**, which we can modify with any text editor. The Makefile provides a set of rules that identifies what needs to be compiled, A good way to understand how a Makefile can help us is through the following example:

Suppose we've got a driver file called `publicX.c`, which makes use of some of the functions defined in `puzzles.c`. Now, there's also a `puzzles.h` file, which contains the headers for the functions in the `puzzles.c` fie. Both `publicX.c` and `puzzles.c` include the `puzzles.h` file.

Before we create our Makefile, we need to understand our "tree" of dependencies. There are some basic dependency rules we need to understand:

1. Executables depend on all of the object files that could compose the program.
2. Executables are made by linking object files.
3. Object files depend on their respective source files (`x.o` depends on `x.c`) and any header files included in the source files.
4. Object files are created by compiling `.c` files with the `-c` flag.

Now, in our Makefile, we list compilation rules in pairs of two lines. These are referred to as **rules**. Each rule has a **target** (a file name followed by a colon). After the colon comes a list of that file's dependencies. On the subsequent line, a command is provided, which specifies how to compile the program. The lines containing the commands must begin with a tab character.

In our example, we'd have the following Makefile:

Listing 78: Makefile 1

```
publicX: publicX.o puzzles.o
	gcc -o publixX publicX.o puzzles.o

publicX.o: publicX.c puzzles.h
	gcc -c publicX.c

puzzles.o: puzzles.c puzzles.h
	gcc -c puzzles.c
```

This Makefile specifies, for example, that if `publicX.c` is modified, then `publicX.o` will need to be re-linked. It's important to remember that the second line of a rule **must** begin with a tab character.