CMSC 412 Project #3

Threads & Synchronization

Due March 17th, 2017, at 5:00pm

Overview

User level threads. You will implement user level threads where each thread has it's own stack, but shares the program, globals, and heap data with its peers.

Synchronization Implementation. You will implement semaphores, a simple synchronization primitive. In addition, you will implement a user-level synchronization primitive based on busy waiting with semaphores.

Threads

You will add one new system call that provides the ability to clone a process into another running thread. The system call will be:

```
int Clone(void (*threadFunc)(void), void *childStack)
```

The first argument to the function is a function that is to be invoked when the new thread is started. The function takes no parameters. This function should call Exit to terminate the thread, if it returns the behavior is undefined. The second parameter is a pointer to the stack for the new user thread. The stack is typically a malloc'd space in a threading environment. However, since the user level code for GeekOS lacks a heap at this point, you should pass in a pointer intro a global array to use as the stack. Recall that stacks grow downward in memory on x86 processors and that references to C array start with the lowest element in memory. These two facts mean you will need to pass in the **end** of your global array to the Clone system call. The return value for Clone is pid of the newly created thread.

Only original processes may call clone. Calling clone from a cloned thread should return `EINVALID`.

Implementing Clone will require allocating a kernel stack, copying the parent's kthread structure and changing the appropriate fields. It will also require calling `Attach_User_Context`, adding the newly created thread to the list of all threads via `Add_To_Back_Of_All_Thread_List` and finally making it runnable by calling `Make_Runnable`. The cloned process should have its own kthread struct, but share the user context with its parent.

To get the newly created thread to run the correct user code on the correct stack, you will need to "fixup" the copied kernel stack so that the pushed interrupt (the one the parent user to make the system call Clone) returns back into the first instruction of the supplied main function for the cloned thread.

User Stack Data Selector (data selector)	**User Stack Location**
User Stack Pointer (to end of user's data segment)	
Eflags	**Interrupt_State**
Text Selector (code selector)	
Program Counter (entry addr)	
Error Code (0)	
Interrupt Number (0)	
EAX (0)	
EBX (0)	
ECX (0)	
EDX (0)	
ESI (Argument Block address)	
EDI (0)	
EBP (0)	
DS (data selector)	
ES (data selector)	
FS (data selector)	
GS (data selector)	

Semaphores

You will add system calls that provide user programs with semaphores, to enable thread synchronization among different threads. The system calls (on the user side) will be:

```
int Open_Semaphore(const char *name, int ival)
int P(int sem)
int V(int sem)
int Close_Semaphore(int sem)
```

`Open_Semaphore`

`Open_Semaphore(name, ival)` is a request by the current process to use a semaphore. The user gives a name for the semaphore, as well as the semaphore's initial value, and will get back a *semaphore ID*, an integer between 0 and N - 1. The semaphore ID denotes a particular semaphore data structure in the kernel, which you must implement. The semaphore ID is then passed by the user program to the operations P() and V(), described next, to wait or signal the associated semaphore.

Your operating system should be able to handle **at least** 20 (thus N = 20) semaphores whose names may be **up to** 25 characters long. If there are no semaphores left (i.e., there were N semaphores with unique names already given), ENOSPACE must be returned indicating an error.

The returned semaphore ID is chosen in one of two ways.

1. If this is the first time `Open_Semaphore` has been called for the given name, the kernel should find and return an unused SID, and initialize the value of the associated semaphore datastructure to `ival`.

2. If another thread has made this system call with the same name, and the semaphore has not been destroyed in the meantime (see below), you must return back the same semaphore ID (`sem`) that was returned the first time. The parameter `ival` is ignored in this case.

Think of a semaphore ID as like a file descriptor in UNIX: in that case, when you open a file, you get back a number (the file descriptor) that denotes that file. Subsequent read and write operations take that file descriptor as an argument, and the kernel figures out which file the number is associated with, and then performs the operations on that file. Just the same way, you will implement a semaphore datastructures within the kernel, and refer to them from user programs via their associated semaphore IDs. However, one difference is that file descriptor numbers are re-used for different processes, but semaphore ID's are globally unique.

`P` and `V`

The `P(sem)` system call is used to decrement the value of the semaphore associated with sempahore ID `sem`. This operation is referred to as wait() in the text. Similarly, the `V(sem)` system call is used to increment (signal() in the text) the value of the semaphore associated with `sem`.

As you know, when P() is invoked using a semaphore ID whose associated semaphore's count is less than or equal to 0, the invoking process should block. To block a thread, you can use the `Wait` function in the kernel. Each semaphore data structure will contain a thread queue for its blocked threads. The file `kthread.h` provides a definition of a thread queue. You should look at `kthread.h` and `kthread.c` to see how it is declared and used. To wake up a thread/all threads waiting on a given semaphore, i.e. because of a `V()`, you can use `Wake_Up_One()/Wake_Up()` routines from `kthread.h`.

A process may only legally invoke P(sem) or V(sem) if sem was returned by a call to Open_Semaphore for that process (and the semaphore has not been subsequently destroyed). If this is not the case, these routines should return EINVALID.

`Close_Semaphore`

`Close_Semaphore(sem)` should be called when a process is done using a semaphore; subsequent calls to P(sem) and V(sem) (and additional calls to Close_Semaphore(sem) by this process) will return EINVALID.

Once all processes using the semaphore associated with a given semaphore ID have called Close_Semaphore, the kernel datastructure for that semaphore can be destroyed. A simple way to keep track of when this should happen is to use a reference count. In particular, each semaphore datastructure can contain a count field, and each time a new process calls Open_Semaphore, the count is incremented. When Close_Semaphore is called, the count is decremented. When the count reaches 0, the semaphore can be destroyed.

When a thread exits, the kernel should close any semaphores that the thread still has open. In your code, both the Sys_Close_Semaphore() function and at least some function involved in terminating user threads should be able to invoke the "real" semaphore-closing function.

Spin Locks

On multi-core systems, often times semaphores (which require going into the kernel and blocking processes) are too slow. As a result, in parallel programs often time "spin" locks are used. A spin lock checks a shared memory location to see if another thread is currently using the lock. If no other thread is using it, the lock function marks the lock as in use. If there is another thread holding the lock, it keeps trying to see if the lock is free (i.e., it spins).

In this project, you will implement spin lock primitives. The functions are:

```
int Is_Locked(User_Spin_Lock_t *lock)
```

Returns 1 is the spin lock is locked and 0 otherwise

```
void Spin_Lock_Init(User_Spin_Lock_t *lock)
```

initializes the spin lock data structure

```
void Spin_Lock(User_Spin_Lock_t *lock)
```

Locks a spin lock. If another thread has the lock, busy wait until it is available.

```
int Spin_Unlock(User_Spin_Lock_t *lock)
```

Unlock a spin lock. Returns -1 if the lock is not currently held, and 0 if the is successfully unlocked.

Since spin locks are implemented entirely in user space, the code should be added to the C library (src/libc) in a new function called spin.c. To correctly implement the spin locks, you will need to write at least part of these routines in assembly code and use the atomic memory operations. The atomic swap instruction on x86 looks like "lock xchg eax, [ebx]" where eax is a register value to atomically swap into the memory location pointed to by ebx. Look in the file spin.c at the supplied code for Spin_Lock to see how to get a C variable into a specific register.

The following two references provide a good introduction to using inline assembly with the GNU compiler:

http://wiki.osdev.org/Inline_Assembly

http://ericw.ca/notes/a-tiny-guide-to-gcc-inline-assembly.html

You should write test code for the spin lock using the clone function (described above) to create two threads with shared memory between them.

Notes

In order not to clutter syscall.c with too much functionality, you must put your semaphore implementation in two new files sem.h and sem.c. Semaphore operations need to be implemented within a critical section, so that operations execute atomically.

In this, and other projects, you will rely heavily upon a list data structure. For this reason an implementation has been provided to you in `list.h` file. Please familiarize yourself with its syntax and functionality. It could be a little tricky to understand the syntax since functions are written using `#define`. Naturally you are always free to extend, modify, or write your own implementation that would better suit your needs.

The wait system call should work with cloned threads. A background process can clone and wait for its child (just like a background process can fork a child that is in the foreground).

Summary: New System Calls

Kernel Function	User Function	Effect

Sys_Clone	`Int Clone(`
	`  void(*threadFunc(void),`
	`  void *childStack)`
		A new thread is created in the parent's address space and starting running threadFunc on stack childStack.

Sys_Open_Semaphore	`int Open_Semaphore(`
	`  const char *name,`
	`  int ival)`
		if *name* is longer than 25 characters, return ENAMETOOLONG. If a semaphore with this *name* doesn't exist, create it and return its SID; if it exists, return its SID; note that SID must be >= 0

Sys_P	`int P(int sem)`	might block (textbook wait() semantics) returns EINVALID if `sem` invalid returns 0 on success
Sys_V	`int V(int sem)`	**never** blocks (textbook signal() semantics) returns EINVALID if `sem` invalid returns 0 on success
Sys_Close_Semaphore	`int Close_Semaphore(` `  int sem)`	**never** blocks returns EINVALID if `sem` invalid returns 0 on success

Testing your code

The files we provided can be used to test your semaphores:

- **sem-ping.c** and **sem-pong.c** create a nice effect when you launch them concurrently:

```
% /c/sem-ping.exe &
% /c/sem-pong.exe
```

Final Notes

We do not require that your earlier projects worked; you should be able to implement this project directly from the base kernel, without using the earlier kernel features.