CMSC 412, Spring 2017

Project 1

Due: Wednesday, February 15th, 2017 5:00 pm

Overview

In this project we will be augmenting the GeekOS process services to include (1) being able to kill a process asynchronously from another process, (2) being able to view the currently running processes and their status, and (3) managing processor affinity in scheduling.

Getting/Submitting code

- git clone https://gitlab.cs.umd.edu/cmsc412_sp2017/spring2017.git
- Submissions: run "make submit" to submit your project.

Preliminaries

We will first provide some background on process lifetimes in GeekOS, and some more details on how system calls are implemented. The project requirements are presented below. Following the requirements, we present further background material on how process address spaces are implemented, many details of which are not important for this project, but will be more important for project 4 (so it might be useful to understand at least some of them now).

Process Creation and Termination

As you already know, a user process in GeekOS is essentially a Kernel_Thread with an attached User_Context. Each Kernel_Thread has a field `alive` that indicates whether it has terminated (e.g., whether Exit() has been called). In addition, a Kernel_Thread has a refCount field that is used to indicate the number of kernel threads interested in this thread. When a thread is alive, its refCount is always at least 1, which indicates its own reference to itself. If a thread for a process is started via Start_User_Thread with a `detached` argument of "false", then the refCount will be 2: one self-reference plus a reference from the owner. When `detached` is false, the `owner` field in the new Kernel_Thread object is initialized to point to the Kernel_Thread spawning it (aka the *parent*). Typically, the parent of a new process is the shell process that spawned it.

The parent-child relationship is useful when the parent wants to retrieve the returned result of the new process using the Wait() system call. For example, in the shell (src/user/shell.c), if Spawn_Program is successful, the shell waits for the newly launched process to terminate by calling Wait(), which returns the child process's exit code. The Wait system call is implemented by using thread queues, which we explain below.

When a process terminates by calling Exit, it detaches itself, removing its self-reference. Moreover, when the parent calls Wait, it removes the other reference, bring refCount to 0. When this is the case, the Reaper process is able to destroy the thread, discarding its Kernel_Thread object. Any process that is dead, but has not been reaped, is called a zombie. The reasons for this could be many, one instance being the parent failing to release its refCount: bug or otherwise.

More about process lifetimes: Zombies

A process can be in one of four states on its way from being alive to being dead:

1. refCount=0, alive=false

   This process is a zombie that's "totally dead," as the child has done Exit to reduce its refCount, and if it had a parent at all, it reduced its refCount too. Thus, the process will soon be reaped.

2. refCount=1, alive=false, background=false

   The process has called Exit(), but the parent hasn't called Wait(). In this case, the process is also a zombie, but is not on the graveyard queue.

3. refCount=1, alive=true, background=true

   The process is a background process, and is alive and well. The parent is not allowed to call Wait on this process.

4. refCount=2, alive=true, background-false

   The process is a "foreground" process, and is alive and well.

Thread Queues

As processes enter the system, they are put into a *job queue*. In particular, the processes that are residing in the main memory and are ready and waiting to execute are kept on a list called the *run queue*. It stays there, not executing, until it is selected for execution. Once the process is allocated the CPU and is executing, one of several events can occur:

- The process could issue an I/O request and then be placed in an *I/O queue*. For example, suppose the process makes an I/O to a shared device, such as a disk. Since there may be many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular device is called the device queue. Each device has its own *device queue*.
- The process could create a new subprocess and Wait for the subprocesses termination. In which case it goes into the wait queue for that process (which is defined in the Kernel_Thread struct) by calling the join() routine in kthread.c.
- The process could be removed forcibly from the CPU, as a result of a timer interrupt, and be put back in the run queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then removed from its I/O queue and put back in the run queue. A process continues this cycle until it terminates, at which time it is not present on any queue.

Project Requirements

There are three primary goals of this project:

- Support asynchronous killing of background processes.
- Support printing the process table (i.e., information about the currently-running processes)
- Add call to set and get scheduler affinity

Killing processes

It could be that once a background process, or any other process, starts to run, it may behave badly, or the work it is performing may become irrelevant. Therefore, we would like to have some way for one process to kill another process. To do this, do the following:

- Implement the Sys_Kill() system call (a stub appears in src/geekos/syscall.c) that takes a PID as its argument. The semantics are that the given process is killed immediately. (Note that this is unsafe in a general setting, since that process might hold shared resources, but there aren't any shared resources in GeekOS. Wait until later projects!). There is a second argument to the kill system call (sig) that is not used for this assignment. You will use it in a later assignment.

  This is different from a thread calling Exit(). Note that when a thread is executing it is not in any queue and is only referenced in g_currentThreads[]. Therefore when doing the cleanup of a thread that called Exit by itself it makes sense to not consider any queues. But an asynchronous kill of a process can happen at any time. Particular example scenarios are for instance when the process is waiting for its child process to die and so is in the wait queue for that process. Or when it is in the runQueue of the system. Therefore when doing this asynchronous kill you will need to ensure that you properly remove the process from all queues it is in.

Also consider what should happen with a killed process's child processes: Their parent pointer is now invalid, and so they should be adjusted accordingly. Indeed, the same thing should happen for Exit, but does not in the implementation we provide---it turns out that this field is not used by the child process after its parent dies, so it can be left dangling. However, in your modified code, you will be using the parent pointer to print the process table, so both Exit and Kill should behave similarly, nulling the dangling parent pointer.

There is an interesting design point here: if a parent dies and fails to Wait() on its children, should we also decrement the children's refCount? If this does not happen, the child will remain a zombie, so decrement the counter.

When determining which processes can be killed; a process that falls in category II above (sec "More about process lifetimes: Zombies") should be allowed to be killed. This would happen because a child has died but its parent has not yet done a Sys_Wait, and we want Sys_Kill to be able to clean up the system nonetheless.

User space programs cannot kill kernel threads, only other user processes.

Printing the process table

Now that we can run many processes from the shell at once, we might like to get a snapshot of the system, to see what's going on. Therefore, you will implement a system call that returns the current processes in the system and a program that prints the status of the threads and processes in the system:

- Implement the system call Sys_PS that returns relevant information about the current processes. The return value of the system call is the number of processes in the system. The user should pass a pointer to an array of Process_Info structs, along with the size of the array (in number of elements). The struct is defined as

```
struct Process_Info {
  char name[MAX_PROC_NAME_SZB];
  int pid;
  int parent_pid; /* 0 if no parent */
  int priority;
  int status;
  int affinity;
  int currCore;
  int totalTime;
};
```

Here, pid and parent_pid should be self-explanatory. The "name" part comes from the program argument to Spawn(); for kernel processes this should be the threadName field from kthread . The "status" field should be 0 for runnable threads (ie threads that actually running or not blocked waiting for something), 1 for blocked (ie threads that are waiting on some I/O queue, or the queue of a child process), and 2 for zombie (ie threads that are no longer alive but have not yet been reaped). The proper #defines for these, and the above struct, are set up for you in include/geekos/user.h, which is included from include/libc/process.h. Priority is the priority number of the process, with respect to scheduling. Affinity is the core that the thread wants to be scheduled on (with -1 meaning any core). currCore is the current core that a thread is executing on and -1 if the thread is not currently running. Totaltime is the total time consumed by the thread. You can get this information from the Kernel_Thread and User_Context structs, though you may need to augment them.

When printing out the status of the process, it should be considered a zombie if it falls in category 1 or 2 above (in sec "More about process lifetimes: Zombies") ---that is, a process is a zombie if the alive field is false.

Add a src/user/ps.c user program (we supplied a file with an empty main function for you). This takes no arguments and should execute the Sys_PS system call to extract the process information and print it out. For the status field, print R for runnable or currently running, B for blocked, and Z for zombie. Format the output as in the following:

```
PID PPID PRIO STAT AFF TIME COMMAND
  1    0    5    B   A   15 {Main}
  2    0    0    R   0    0 {Idle-#0}
  3    1    5    B   A    1 {Reaper}
  4    1    5    B   A   37 {IDE}
  5    1    5    B   A    0 {Alarm}
  7    0    0  1 R   1   54 {Idle-#1}
  8    1    5    B   A    0 {NetRecv}
  9    1    5    B   A    0 {Forwarding}
 10    1    1    B   A    4 /c/shell.exe
 11   10    1  0 R   A    0 /c/ps.exe
```

The PS system call stub in user space has been defined for you; its prototype appears in include/libc/process.h. Your process table must have space for at least 50 entries. Please use "%3d %4d %4d %2c%2c %3c %4d %s\n" as the format string to achieve the formatting in the table as shown. Failure to use the format string may cause tests to fail. For running processes, the ps command should print the core they are currently running on just before the 'R' status (see pids 7 and 10 above).

Processor Affinity

When a computer has more than one core (or processor), it needs to decide which core to run a process on. Normally, processes can run on any core in the system. However, for some applications, the programmer might wish to specify which core to use. In GeekOS, each thread as a field called affinity that controls which cores it can run on. If the value of the affinity variable is -1, it can run on any core. If the value is 0 to CPU_Count-1, then that process can only run on that core number.

In this project, you will add a system call to GeekOS to set and get the affinity of a process. There are two system calls:

int Set_Affinity(int pid, int core)
    Set the affinity of pid to be core

int Get_Affinity(int pid)
    Return the affinity of the passed process id (pid)

The GeekOS scheduler already knows how use the affinity field, when it makes scheduling choices.

These routines should return EINVALID as the return code if any of the parameters are not valid.

After implementing the affinity system calls, you should run the affinity user program that has been supplied with this project. The affinity program takes one argument (either all1 or split). It will then run two programs either with their affinity set to core #1, or with each one set to a different core. The program prints the elapsed wall time for each of the programs. After running both options to affinity, look at the differences in time. In the file WRITEUP in the geekos directory, write a couple of sentences that explains why the two results are different. This file will be submitted with your project and graded like a test case (by hand by the TAs after the deadline).