Programming Assignment 1

Due 11:59pm Sunday, January 10

This programming assignment consists of 6 exercises, labelled A through F. The first five are to help you get started and to understand the basic concepts. You do NOT have to hand anything in for these exercises, just do them as they will help you. However, for the last exercise F, you will make a modification to the operating system in a file called `mycode1.c`, and the contents of this file constitutes your solution to this assignment. Hence, all that you need to turn in is `mycode1.c`, and nothing else. Each of the programs below can be found in separate files, `pa1a.c`, `pa1b.c`, ..., `pa1f.c`. You may modify them to your liking; they will not be viewed
or submitted.

To do this assignment, you must first install it using your account on ieng9:

1. Log in to ieng9.ucsd.edu using your class account.

2. Enter in the command, "`prep cs120w`". This configures your account for this class. You should always do this when doing CSE120 work. Note that this command may run automatically when you log in, and if so, it is not necessary to repeat it (but won't hurt if you do).

3. Enter in the command, "`getprogram1`". This will create a directory called "`pa1`" and will copy the relevant files into that directory. It is important that you do ALL your work in your pa1 directory, as UMIX only works in that directory (for this programming assignment). If you change its name, or if you copy its contents to another directory and try to work there, UMIX will simply not run. You may certainly save your work by copying your pa1 directory to another directory, e.g., to make a record or "snapshot" in time; you simply cannot work there. In summary, ALL of your latest work with any changes you make should be in, and all your executions should be run from within, pa1.

4. To compile, enter the command "`make`" (from within the pa1 directory). This will compile all of the programs. You can compile a particular program by specifying it as a parameter to make, e.g., "`make pa1a`".

5. To turn in the assignment, make sure that the `mycode1.c` file that you want to turn in is in the current directory and enter the command "`turninprogram1`".

Note: As you are developing your program, you may have many processes that are lingering from previous runs that never completed. If you run up against ieng9's maximum number of processes that you can have running, you may encounter strange behavior. So, periodically

check for existing processes by typing the command "`ps -u <yourloginname>`". If you see any processes labeled "`pa1`..." you can kill those by typing "`kill -9 <pid1> <pid2> …`" where `<pidn>` is the process id of the runaway process. A convenient command that does this is:

```
ps -u <yourloginname> | grep pa1 | awk '{print $1}' | xargs kill -9
```

Notes on Grading

1. You will be graded primarily on whether your code works and how well it works (is it efficient in time and/or space). While you will not be graded specifically on code structure and commenting/documentation, it is to your benefit to use good software engineering principles in structuring and commenting your code. This will especially benefit you if your code requires visual inspection by a grader. If a problem arises and we can't understand your code, or it is difficult to do so, your grade may suffer.

2. Unless indicated otherwise, you should NOT use any library functons or any code that you did not write yourself other than the functions given to you. For example, if you need a data structure like a linked list or queue, you should create it yourself rather than relying on a C library.

3. You should NOT use dynamic memory allocation library functions, such as `malloc()`, in your kernel. The reason is that since any dynamic memory allocator may fail (if all the memory is used), the kernel cannot depend on it (otherwise the kernel itself might fail, which would be catastrophic).

(The last two points above are especially relevant for the next assignments, as this assignment is simple and the idea of using library functions or dynamic memory allocation should not even enter your mind. But since these constraints are important for kernel implementation in this class, it's good to know about them sooner rather than later).

4. It is your responsibility to proactively come up with YOUR OWN tests that you think are necessary to convince yourself that your kernel is doing what is asked for in the specification. If your code passes tests provided by anyone on the CSE 120 teaching staff, you should not assume that your code completely works and that you are done. What ultimately matters as far as what your code is expected to do will be based on the specification you are given (i.e., these instructions). It is up to YOU to interpret it and devise any test cases you think are applicable. This mimics the experience of a real operating system designer/implementer, who has no idea what kind of applications will be written for their operating system, and how their code will be exercised. The real operating system implementer must test for anything and everything as best they can. So, you must test robustly, and be creative in coming up with tests. You are free to ask questions about cases that you think may matter, and even post tests you think are worthy of sharing.

5. All your code must be contained in `mycode1.c`. ALL OTHER CHANGES YOU MAKE TO ANY OTHER FILES WILL BE COMPLETELY IGNORED! Consequently, do not put declarations, function or variable definitions, defined constants, or anything else in other files that you expect to be part of your solution. We will compile your submitted code with our test programs, and so your entire solution must be contained in what you submit. If your code does not compile with our test programs because you made changes to other files that required inclusion, YOU WILL RECEIVE ZERO CREDIT, so be careful!

6. There should be no `Printf` statements in your final version of `mycode1.c` (it's OK to have `Printf`s in your other files, as you won't be handing them in anyway), as if there are, it will disrupt the autograder's ability to check your program and you may lose points. For debugging YOUR KERNEL CODE, i.e., `mycode1.c`, you may wish to use `DPrintf`, which is used and works exactly like `Printf`, except that it generates no output during the autograder's testing, so it is very safe to use.

Programming Assignment 1: Exercise A

In this first exercise, you will learn how to use the UMIX (which stands for "User Mode UNIX") CSE 120 instructional operating system and how to create new processes. Programs for this operating system must be written * in C. UMIX is similar to UNIX, but there are of course differences.

One cosmetic difference is that the main function is capitalized, i.e., "`Main`". The same is true of all system calls. In addition, it is recommended that you use `Printf`/`DPrintf`, rather than printf, as the former immediately flushes buffered output, thus avoiding the potential of unusual sequences of combined output from multiple processes (you may wish to experiment with trying both printf and `Printf`/`DPrintf` to see this behavior).

This system uses UNIX-style processes: each process has a single thread of control and its own private memory. The **`Fork()`** system call is used to create a process. The process that calls `Fork()` is called the parent, and the new process is called the child. `Fork()` creates the child with a memory that is almost identical to that of the parent, with the child starting its execution by returning from `Fork()`. In other words, after the call to `Fork()`, there are two processes, and each has just returned from `Fork()`.

The main (and important) difference is that, in the child process, the return value from `Fork()` is 0, while in the parent, the return value is the process identifier (pid) of the child. This is illustrated in the simple program below. Notice the additional system calls, including **`Getpid()`** which returns the pid of the calling process, and **`Exit()`** which causes the calling process to be destroyed.

(Note that if `Fork()` fails, e.g., an attempt to create too many processes, it will return -1 to the calling process. In general, system calls in UMIX and UNIX use the convention of returning -1 if the system call fails.)

Run the program below. To do this, use the supplied Makefile and run make. You must run this from within your CSE 120 account, as this will cause the compiler to use special libraries that work for the CSE 120 operating system.

Things to think about

1. For this OS, the parent always continues to run after a call to `Fork()` (despite that the child process exists and can potentially run in competition with the parent), but this is simply an artifact of the implementation. For most UNIX OS's systems, once `fork()` is called, the choice of which process actually runs is arbitrary and is determined by the OS scheduler. In the verson of UMIX for this first assignment, the choice happens to be to always run the parent. In some OS's, the choice can be virtually random.

2. Notice the call to `Exit()` in the child. If `Exit()` were not called, the child would continue running beyond the if clause (executing the statements that print the parent's identity, etc.).

3. When `Fork()` is called, the child's memory looks just like that of the parent. Thus, since the value of pid is 0 when `Fork()` is called, the child will inherit this variable and its value (0). Note that pid is set AFTER `Fork()` returns. In the parent, it will be set to the process identifier of the child, and in the child, it will be set to zero, because these are the semantics of `Fork()`. Thus, for the child to learn its identifier, it must call `Getpid()`.

Questions

1. Change the program to print the value of pid in the code executed by the child. What does it print, and why?
2. Remove the Exit() statement. What happens, and why?

```
#include <stdio.h>
#include "aux.h"
#include "umix.h"

void Main()
{
      int pid;

      if ((pid = Fork()) == 0) {

            /* child executes here */
            Printf("I am the child, my pid is %d\n", Getpid());
            Exit();
      }

      Printf("I am the parent, my pid is %d\n", Getpid());
      Printf("I just created a child process whose pid is %d\n", pid);
}
```

Programming Assignment 1: Exercise B

Now we will expand the program of Exercise A to call `Fork()` multiple times.
Run the program below and answer the following questions:

Questions

1. Can you explain the order of what gets printed based on the code?
2. Why do you think the first child executes before the second child?
3. Move the two print statements executed by the parent to just after where it says HERE. How can you print the pid of the first child?

```
#include <stdio.h>
#include "aux.h"
#include "umix.h"

void Main()
{
      int pid = 0;

      if ((pid = Fork()) == 0) {

            /* first child executes here */
            Printf("I am the first child, my pid is %d\n", Getpid());
            Exit();
      }

      Printf("I am the parent, my pid is %d\n", Getpid());
      Printf("I just created a child process whose pid is %d\n", pid);

      if ((pid = Fork()) == 0) {

            /* second child executes here */
            Printf("I am the second child, my pid is %d\n", Getpid());
            Exit();
      }

      /* HERE */

      Printf("I (the parent) just created a second child process whose
pid is %d\n", pid);
}
```

Programming Assignment 1: Exercise C

Now we will learn how to effect the execution sequence of the various processes. We introduce a new system call, **`Yield(pid)`**, which causes the currently running process to yield to the process whose identifier is `pid`. Furthermore, when `Yield(pid)` returns, it returns the identifier of the process that yielded to the one that is now running (and returning from `Yield(pid)`). This is illustrated by the program below, which you should study and run.

Questions

1. Can you explain the order of what gets printed based on the code?

```
#include <stdio.h>
#include "aux.h"
#include "umix.h"

void Main()
{
      int pid = 0, rpid;

      if ((pid = Fork()) == 0) {

            /* first child executes here */
            Printf("I am the first child, my pid is %d\n", Getpid());
            Exit();
      }

      Printf("I am the parent, my pid is %d\n", Getpid());

      Printf("About to yield to child process whose pid is %d\n", pid);
      rpid = Yield(pid);    // yield to first child before continuing
      Printf("Process %d just yielded to me (the parent)\n", rpid);

      if ((pid = Fork()) == 0) {

            /* second child executes here */
            Printf("I am the second child, my pid is %d\n", Getpid());
            Exit();
      }

      Printf("About to yield to child process whose pid is %d\n", pid);
```

```
    rpid = Yield(pid);    // yield to second child before continuing
    Printf("Process %d just yielded to me (the parent)\n", rpid);
}
```

Programming Assignment 1: Exercise D

Let's continue using `Yield(pid)` to effect a variety of executions sequences. Using the code below, the sequence of the print statements will be ABP:

First Child (A)
Second Child (B)
Parent (P)

Questions

1. See if you can cause a change in the sequence as specified below just by ADDING Yield statements (i.e., do not remove any of the ORIGINAL Yield statements, just add extra ones anywhere you wish):
   - BAP
   - BPA
   - PAB
   - PBA
   - APB

```
#include <stdio.h>
#include "aux.h"
#include "umix.h"

void Main()
{
      int ppid, pid1, pid2;

      ppid = Getpid();

      if ((pid1 = Fork()) == 0) {

            /* first child executes here */
            Printf("I am the first child, my pid is %d\n", Getpid());
            Exit();
      }

      Yield(pid1);

      if ((pid2 = Fork()) == 0) {

            /* second child executes here */
```

```
        Printf("I am the second child, my pid is %d\n", Getpid());
        Exit();
    }

    Yield(pid2);    // yield to second child before continuing

    Printf("I am the parent, my pid is %d\n", Getpid());
}
```

Programming Assignment 1: Exercise E

Study the program below. This will be used for your next and final exercise, so make sure you thoroughly understand why the execution sequence of the processes is the way it is.

Questions

1. Can you explain the order of what gets printed based on the code?

```
#include <stdio.h>
#include "aux.h"
#include "umix.h"

#define NUMPROCS 3

void handoff(int p);

void Main()
{
      int i, p, c, r;

      for (i = 0, p = Getpid(); i < NUMPROCS; i++, p = c) {
            Printf("%d about to fork\n", Getpid());
            if ((c = Fork()) == 0) {
                  Printf("%d starting\n", Getpid());
                  handoff(p);
                  Printf("%d exiting\n", Getpid());
                  Exit();
            }
            Printf("%d just forked %d\n", Getpid(), c);
      }

      Printf("%d yielding to %d\n", Getpid(), c);
      r = Yield(c);
      Printf("%d resumed by %d, yielding to %d\n", Getpid(), r, c);
      Yield(c);
      Printf("%d exiting\n", Getpid());
}

void handoff(p)
      int p;                        // process to yield to
```

```
{
      int r;

      Printf("%d yielding to %d\n", Getpid(), p);
      r = Yield(p);
      Printf("%d resumed by %d, yielding to %d\n", Getpid(), r, p);
      Yield(p);
}
```

Programming Assignment 1: Exercise F

We are finally ready to modify the UMIX operating system kernel! Up until now, you have studied and modified only user programs that make system calls, but not the kernel. Here, you will make your first addition to the kernel by implementing the all-important mechanism of context switching.

Whenever `Yield(p)` is called, the kernel is entered and eventually calls **`MySwitchContext(p)`**, which causes a context switch to process p, and returns the process ID of the process that just called `Yield(p)`. This function can be found in the file `mycode1.c`, which will contain all of YOUR modifications to the kernel. If you look at `MySwitchContext(p)`, you will see that it calls yet another function, **`RefSwitchContext(p)`**, which is a "reference" working version of context switching that is already built in to the kernel. `RefContextSwitch(p)` is what actually causes a context switch to occur, and its return value is then what `MySwitchContext(p)` will return. Note that without `RefSwitchContext(p)`, the kernel would not have worked properly up until now; we needed it to be able to run all the programs discussed so far.

In this exercise, you will REMOVE the call to `RefSwitchContext(p)`, which will break the kernel, and in its place, you will add your own code to make the kernel work properly again. (When your code is tested during grading, `RefSwitchContext(p)` will be deactivated, so you cannot use it anyway; if you mistakenly leave it in your code, it will do nothing.)

`MySwitchContext(p)` should cause a context switch from the currently running process to process `p`. To implement `MySwitchContext(p)`, you are given three utility functions:

`void SaveContext(CONTEXT *c)`
Saves the context of the currently running process into a `CONTEXT` variable (`CONTEXT` is a special data type in UMIX) pointed to by `c`. This context comprises the current values of registers, including the SP (stack pointer) and the PC (program counter), though you need not be concerned with these details.

`RestoreContext(CONTEXT *c)`
Restores the context of the process whose previously saved context is in the CONTEXT variable pointed to by `c`. This saved context is the result of either (1) a call to `SaveContext(c)` at some point in the past (and whatever process was running at that point will be the one restored), or (2) a call to `NewContext(p, c)` that was called when the process was created but has not had a chance to run yet. `NewContext(p, c)` is described below.

`int GetCurProc()`
Simply returns the ID of the currently running process.

Consider what happens when a process's context is restored. It begins executing from wherever the PC was pointing to when its state was saved, specifically somewhere within `SaveContext(c)`. Therefore, `SaveContext(c)` will return TWICE, even though it was called only once! The first time corresponds to when `SaveContext(c)` was explicitly called, and the second time is when the process's context is restored (as a result of calling `RestoreContext(c)`).

YOUR JOB is to find a way of distinguishing between returns so that your code can determine whether or not `RestoreContext(c)` should be called. *(Hint: study the Lecture notes on context switching. If you understand the notes, you will know how to write this code).*

As both `SaveContext(c)` and `RestoreContext(c)` make use of contexts, you are provided with a data type (a typedef structure in C) named `CONTEXT`, defined in `sys.h`. You can use this to declare variables to store process contexts as follows. First, declare a variable of type `CONTEXT`:

```
CONTEXT ctxt;
```

You can then call `SaveContext(&ctxt)`, which will store the context of the currently running process into the variable `ctxt`. If you wish to restore the context of a process whose context was previously saved into `ctxt`, you can then call `RestoreContext(&ctxt)`. If you find it more convenient to pass a pointer, you may do the following:

```
CONTEXT ctxt;
CONTEXT *c = &ctxt;

SaveContext(c);       // which is equivalent to
SaveContext(&ctxt)
```

Note that you do not need to (nor should you) modify the actual `CONTEXT` structure defined in sys.h, all you need to do is declare a variable of type `CONTEXT` and use it by passing the variable to `SaveContext(c)` and `RestoreContext(c)`.

You will need to keep track of ALL active processes, and so each will require its own CONTEXT variable. It's up to you as to what data structure to use for this purpose (a typical data structure is a table, i.e., an array). The following two facts about UMIX will be helpful:

1. *There is a maximum number of processes that UMIX will support*; this number is defined by the constant `MAXPROCS`, which you may use in declaring your data structure.

2. *Process IDs range from 1 to `MAXPROCS`.* Every time a new process is created, an ID that is currently not in use in that range will be assigned to that process.

Finally, we must deal with the issue of how to restore the context of a process that is newly created (i.e., the result of a `Fork()` system call) but has not run yet. Since it has not run, there was been no call to `SaveContext(c)`, and so there is no saved context variable from which to restore. This is solved with the function **`NewContext(p, c)`:**

`NewContext(int p, CONTEXT *c)`
This function is called by the kernel whenever a new process is created. The process ID is `p`, and the INITIAL context is pointed to by `c`. `NewContext(p, c)` is part of `mycode1.c`, and so it is a function that you get to implement. The kernel is basically providing a notification that a new process was created by calling `NewContext(p, c)`, and this is your opportunity to record that process `p` is a new process and that its corresponding context is pointed to by `c`.

You should take this opportunity by copying the context pointed to by `c` into whatever data structure you are maintaining for the contexts of all processes, making sure to associate it with process ID `p`, so that a future call to `RestoreContext(c)` (where `c` points to the context of this new process) will work. Once `NewContext(p, c)` returns, you cannot count on `c` pointing to process `p`'s context, so make sure you make a copy of it while in `NewContext(p, c)`.

You now have all the tools to implement `NewContext(p, c)` and `MySwitchContext(p)`. Here are some hints to help implement them:

NewContext
You MAY use the `memcpy(a, b, n)` library function, which copies `n` bytes from address `b` to address `a`. You may use `sizeof(CONTEXT)` to determine the number of bytes for a `CONTEXT` structure variable.

MySwitchContext
Remove the call to `RefSwitchContext(p)` so that the body of `MySwitchContext(p)` is now empty, and then replace with YOUR code. The only functions you will need for this are `SaveContext(c)`, `RestoreContext(c)`, and `GetCurProc()`. You should review how variables get allocated in the data and stack memory areas (and how to effect this via declarations in C, including the use of the keyword "static"), favoring variables of minimal scope. (Think about why this is important, especially for a large complex program like an operating system where many programmers may eventually modify it). Also, you should NOT make any system calls from within `MySwitchContext(p)`, since system calls are called by processes from outside the kernel, and `MySwitchContext(p)` is called from inside the kernel.

Next, make sure that `MySwitchContext(p)` RETURNS THE PROPER VALUE, which is the ID of the process that just called `Yield(p)`. For example, say process A yields to process B, thus resulting in a context switch from A to B. The return value will allow B to learn that it was A that was running just prior to the context switch. Getting this right can be a bit tricky! Verify that the output using your version of `MySwitchContext(p)` matches the output of the original unmodified `MySwitchContext(p)`, INCLUDING output based on the return value. You should test your kernel by seeing if it will work with the program below, as well as OTHER TESTS YOU DESIGN (and which you may share with others).

Good luck!

```
#include <stdio.h>
#include "aux.h"
#include "umix.h"

#define NUMPROCS 3

void handoff(int p);

void Main()
{
      int i, p, c, r;

      for (i = 0, p = Getpid(); i < NUMPROCS; i++, p = c) {
            Printf("%d about to fork\n", Getpid());
            if ((c = Fork()) == 0) {
                  Printf("%d starting\n", Getpid());
                  handoff(p);
                  Printf("%d exiting\n", Getpid());
                  Exit(0);
            }
            Printf("%d just forked %d\n", Getpid(), c);
      }

      Printf("%d yielding to %d\n", Getpid(), c);
      r = Yield(c);
      Printf("%d resumed by %d, yielding to %d\n", Getpid(), r, c);
      Yield(c);
      Printf("%d exiting\n", Getpid());
}

void handoff(p)
      int p;                        // process to yield to
{
      int r;

      Printf("%d yielding to %d\n", Getpid(), p);
      r = Yield(p);
      Printf("%d resumed by %d, yielding to %d\n", Getpid(), r, p);

      Yield(p);
}
```