Programming Assignment 1 [Last Revision 1/11/15, 10:45am]

Due 10:00pm Sunday, January 18

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 mykernel1.c, and the contents of this file constitutes your solution to this assignment. Hence, all that you need to turn in is mykernel1.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.

Check the web site or bulletin board for this class for actual directions on setting up your environment to do this assignment, and on how to turn it in.

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.

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 mykernel1.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. All you will be graded on is whether your code works, and how well it works (is it efficient in time and/or space). You will not be graded specifically on commenting/documentation. This doesn't mean you should not add comments as, if the grader can't figure out what your code does, the comments will often help. But, it's up to you.

2. Unless indicated otherwise, you should NOT use any library routines or any code that you did not write yourself other than the routines 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 routines, 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 routines 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 read 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
"works" and you are done. What ultimately matters as far as what your code
is expected to do is the specification you are given. 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. In fact, we encourage this!

5. All your code must be contained in *** mykernel1.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 you code does not compile with our test programs because you made changes
to other files that required inclusion, YOU WILL RECEIVE ZERO CREDIT, so
please be careful!

```
/* 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 differences.  One cosmetic
 * difference is that the main function is capitalized, as in "Main".
 * The same is true of all system calls.  In addition, it is recommended
 * that you use Printf, 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 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 only (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.
 *
 * 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. The parent runs (to completion) before the child.  For this OS, the
 * parent always continues to run after a call to Fork (), but this is simply
 * an artifact of the implementation.  For most Unix OS's, once Fork () is
 * called, the choice of which process actually runs is arbitrary and is
 * determined by the OS scheduler.  In this OS, the choice happens to be to
 * always run the parent.  In some OS's, the choice may be 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 ().
 *
 * Review 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:
 *
 * 1. Can you explain the order of what gets printed based on the code?
 *
 * 2. Why does the first child execute 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 (not present in Unix) called
 * 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 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):
 *
 * a. BAP
 * b. BPA
 * c. PAB
 * d. PBA
 * e. 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;
{
        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 now 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 function of
 * context switching.
 *
 * Whenever Yield (pid) 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 (pid).
 * This function can be found in the file mykernel1.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 SwitchContext (p), which
 * is an internal kernel function that actually does the context switch, and
 * returns whatever SwitchContext (p) returns.  Without SwitchContext (p),
 * the kernel would not have worked properly up until now, and we needed it
 * to be able to run all the programs shown so far.  In this exercise, you
 * will REMOVE the call to SwitchContext (p), which will break the kernel,
 * and in its place, add your own code to make the kernel work properly again.
 *
 * 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:  SaveContext (p), RestoreContext (p),
 * and GetCurProc ().
 *
 * SaveContext (p) saves the context of process p, which MUST be the pid of
 * the currently running process (just prior to entering the kernel), storing
 * the state of the CPU registers, including the SP (stack pointer) and lastly
 * the PC (program counter), into a table in the kernel indexed by the pid
 * (you need not be concerned about this table).
 *
 * RestoreContext (p) loads the CPU registers with the context that was
 * previously saved for process p.
 *
 * 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 of SaveContext (p).  (In fact, it is saved
 * just before returning from SaveContext (p) - why is this?) Therefore,
 * SaveContext (p) will return TWICE, even though it was called only once!
 * The first time corresponds to when SaveContext (p) was actually called,
 * and the second time when the process's context is restored.  YOUR JOB
 * is to find a way of distinguishing between returns so that your code
 * can determine whether or not RestoreContext (p) should be called.
 * (Hint: study the Lecture notes on context switching.  If you
 * understand the notes, you will know how to write this code).
 *
 * You now have all the tools to implement MySwitchContext (p).  Remove the
 * call to SwitchContext (p) so that the body of MySwitchContext (p) is now
 * empty, and then replace with YOUR code.  The only functions you will need
 * are SaveContext (p), RestoreContext (p), and GetCurProc ().  As a hint,
 * 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.
 *
 * Finally, make sure that MySwitchContext (p) returns the proper value.
 * Getting this right is a bit tricky!  Make sure 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 test cases you design.  You should test it
 * thoroughly, as it will be graded by seeing if it successfully runs with
 * other test programs (not available to you).
 *
 * 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;
{
        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);
}
```