Programming Assignment 2 [Last Revision 2/1/15, 6:03pm]

Due 10:00pm Sunday, February 1, 2015

This programming assignment is on the topic of process scheduling, and consists of 3 parts, A, B, and C. Each part consists of a set of exercises which you should do. However, all you will hand in are your modifications to the operating system in a file called mykernel2.c, and the contents of this file constitutes your solution to this assignment. Each of the programs below can be found in separate files, pa2a.c, pa2b.c, and pa2c.c. You may modify them to your liking; they will not be viewed or submitted. However, you should devise your own test programs to fully test your implementations of the various scheduling policies.

To install this assignment 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.

3. Enter in the command, "getprogram2". This will create a directory called "pa2" and will copy the relevant files into that directory.

4. To compile, enter the command "make" (from within the pa2 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 pa2a".

5. To turn in the assignment, make sure that the mykernel2.c file that you want to turn in is in the current directory and enter the command "turninprogram2".

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 "pa2..." 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 pa2 | 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 in your kernel. For example, you should not use malloc (both because it is a dynamic memory allocator, and it is a C library routine which, as indicated above, you should not use). Since any dynamic memory allocator may fail (if all the memory is used), the kernel cannot depend on it (otherwise it might fail, which would be catastrophic).

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 *** mykernel2.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 2: Exercise A
 *
 * In this first set of exercises, you will study a simple program that
 * starts three processes, each of which prints characters at certain rates.
 * This program will form the basis for experimenting with scheduling
 * policies, the main subject for this programming assignment.
 *
 * The procedure SlowPrintf (n, format, ...) is similar to Printf, but
 * takes a "delay" parameter that specifies how much delay there should
 * be between the printing of EACH character.  The system is calibrated
 * so that n=7 produces a delay of roughly 1 sec per character.  An increase
 * in n by 1 unit represents an increase in delay by roughly a factor of 2
 * (so n=8 produces a 2 sec delay, and n = 5 produces a 250 msec delay).
 *
 * Run the program below.  Notice the speed at which the printing occurs.
 * Also notice the order in which the processes execute.  The current
 * scheduler (which you will modify in other exercises) simply selects
 * processes in the order they appear in the process table.
 *
 *
 * Exercises
 *
 * 1. Modify the delay parameters so that process 1 prints with delay 8
 * (more than process 2), and process 3 prints with delay 6 (less than
 * process 2).  Notice the speed and order of execution.
 *
 * 2. Try other delay values and take note of speeds and orders of execution.
 * What are the smallest and largest values, and what are their effects?
 *
 * 3. Now repeat steps 1 and 2, but this time MEASURE your program using
 * the Gettime () system call, which returns the current system time
 * in milliseconds.  To compute the elapsed time, record the current time
 * immediately before the activity you want to measure (e.g., SlowPrintf)
 * and immediately after, and then take the difference of the former from
 * the latter:
```

```
 *
 *      int t1, t2;
 *
 *      t1 = Gettime ();
 *      SlowPrintf (7, "How long does this take?");
 *      t2 = Gettime ();
 *
 *      Printf ("Elapsed time = %d msecs\n", t2 - t1);
 *
 * Do the times you measure correspond to the observations you made in
 * steps 1 and 2?  What is the resolution of Gettime () (i.e., what
 * is the smallest unit of change)?  Note that this is different from
 * Gettime ()'s return value units, which are milliseconds.
 */

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

void Main ()
{
        if (Fork () == 0) {

                SlowPrintf (7, "2222222222");           /* process 2 */
                Exit ();
        }

        if (Fork () == 0) {

                SlowPrintf (7, "3333333333");           /* process 3 */
                Exit ();
        }

        SlowPrintf (7, "1111111111");                   /* process 1 */
        Exit ();
}

/* Programming Assignment 2: Exercise B
 *
 * In this second set of exercises, you will learn about what mechanisms
 * are available to you to modify the kernel's scheduler.  Study the file
 * mykernel2.c.  It contains a rudimentary process table data structure,
 * "proctab[]", and a set of procedures THAT ARE CALLED BY OTHER PARTS OF
 * THE KERNEL (the source of which you don't have or require access to).
 * Your ability to modify the scheduler is via these procedures, and what
 * you choose to put in them.  You may also modify proctab, or create any
 * data structures you wish.  The only constraint is that you not
 * change the interfaces to these procedures, as the rest of the kernel
 * depends on them.  Also, your system must support up to MAXPROCS active
 * processes at any single point in time.  In fact, more than MAXPROCS
 * processes may be created (which you must allow), but for any created
 * beyond MAXPROCS, there will have been that same number that have exited
 * the system (i.e., so that only MAXPROCS processes are active at any
 * single point in time).
 *
 * Let's look at the procedures:
 *
 * InitSched () is called at system start-up time.  Here, the process table
 * is initialized (all the entries are marked invalid).  It also calls
 * SetTimer, which will be discussed in Exercise C.  Anything that you want
 * done before the system settles into its normal operation should be placed
```

```
* in this procedure.
*
* StartingProc (int p) is called by the kernel when process p starts up.
* Thus, whenever Fork is called by a process, which causes entry into the
* kernel, StartingProc will be called from within the kernel with p, which
* is the pid (process identifier) of the process being created.  Notice
* how a free entry is found in the process table, and the pid is recorded.
*
* EndingProc (int p) is called by kernel when process p is ending.
* Thus, whenever a process calls Exit (implicitly if there is no explicit
* call), which causes entry into the kernel, EndingProc will be called from
* within the kernel with the pid of the exiting process.  Notice how the
* process table is updated.
*
* SchedProc () is called by the kernel when it needs a decision for which
* process to run next.  It determines which scheduling policy is in effect
* by calling the kernel function GetSchedPolicy, which will return one
* of the following: ARBITRARY, FIFO, LIFO, ROUNDROBIN, and PROPORTIONAL
* (these constants are defined in umix.h).  The scheduling policy can be
* changed by calling the kernel function SetSchedPolicy which is called for
* the first time in InitSched, which currently sets the policy to ARBITRARY.
* SchedProc should return a pid, or 0 if there are no processes to run.
* This is where you implement the various scheduling policies (in conjunction
* with HandleTimerIntr).  The current code for SchedProc implements ARBITRARY,
* which simply finds the first valid entry in the process table, and returns
* it (and the kernel will run the corresponding process).
*
* HandleTimerIntr () will be discussed in the Exercise C, and should not be
* modified in this part.
*
*
* There are also procedures that are part of the kernel (but not in
* mykernel2.c), which you may call from within the above procedures:
*
* DoSched () will cause the kernel to make a scheduling decision at the next
* opportune time (at which point SchedProc will be called to determine which
* process to select).  Elaborating on this further, when you write your code
* for the above procedures (StartingProc, EndingProc, ...), there may be a
* point where you would like to force the kernel into making a scheduling
* decision regarding who should run next.  This is where you should call
* DoSched, which tells the kernel to call SchedProc at the next opportune
* time, i.e., as soon as the kernel determines it can do so.  Why at the
* next opportune time?  Because other code, including the remaining code of
* the procedure you are writing, may need to execute before the kernel is
* ready to select a process to run.
*
* SetTimer (int t) and GetTimer () will be discussed in Exercise C.
*
* Finally, your system must support up to MAXPROCS (a constant defined in
* sys.h) active processes at any single point in time.  In fact, more than
* MAXPROCS processes may be created during the lifetime of the system (which
* you must allow), but for any number created beyond MAXPROCS, there will
* have been that same number (or more) that have exited the system (i.e.,
* so that only MAXPROCS processes are active AT ANY SINGLE POINT IN TIME).
*
*
* Exercises
*
* 1. Implement the FIFO scheduling policy, First In First Out.  This means
* the order in which processes run (to completion) is the same as the order
* in which they are created.  For the program below, the current scheduler
```

```
 * will print:  11111111112222222222444444444333333333 (why this order?)
 * Under FIFO, it should print: 11111111112222222223333333334444444444
 * (why this order?).
 *
 * 2. Implement the LIFO scheduling policy, Last In First Out.  This means
 * that as soon as a process is created, it should run to completion before
 * any existing process.  Under the LIFO scheduling policy, the program
 * below should print:  44444444442222222222333333333311111111111 (why
 * this order, and not 44444444443333333333222222222211111111111?).
 */

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

void Main ()
{
        if (Fork () == 0) {

                if (Fork () == 0) {

                        SlowPrintf (7, "4444444444");   /* process 4 */
                        Exit ();
                }

                SlowPrintf (7, "2222222222");           /* process 2 */
                Exit ();
        }

        if (Fork () == 0) {

                SlowPrintf (7, "3333333333");           /* process 3 */
                Exit ();
        }

        SlowPrintf (7, "1111111111");                   /* process 1 */
        Exit ();
}

/* Programming Assignment 2: Exercise C
 *
 * In this third and final set of exercises, you will experiment with
 * preemptive scheduling.  We now return to the file mykernel2.c, and study
 * the procedures that were briefly mentioned but not discussed in detail
 * in Exercise B.
 *
 * HandleTimerIntr () is called by the kernel whenever a timer interrupt
 * occurs.  The system has an interval timer that can be set to interrupt
 * after a specified time.  This is done by calling SetTimer.  Notice that
 * the first thing that HandleTimerIntr does is to reset the timer to go off
 * again in the future (otherwise no more timer interrupts would occur).
 * Depending on the policy (something for you to think about), it may
 * then call DoSched, which informs the kernel to make a scheduling
 * decision at the next opportune time, at which point the kernel will
 * generate a call to SchedProc to select the next process to run, and
 * then switch to that process.
 *
 * MyRequestCPUrate (int pid, int m, int n) is called by the kernel whenever
 * a process identified by pid calls RequestCPUrate (int m, int n), which is
 * a system call that allows a process to request that it should be scheduled
 * to run m out of every n quantums.  For example, if a process wants to run
```

```
* at 50% of the CPU's execution rate, it can call RequestCPUrate (1, 2),
* asking that it run 1 out of every 2 quantums.  When a process calls
* RequestCPUrate (m, n), the kernel is entered, and the kernel calls your
* MyRequestCPUrate (pid, m, n), giving you the opportunity to implement
* how proportional share is to be achieved.
*
* MyRequestCPUrate (pid, m, n) should return 0 if successful, and -1 if it
* fails.  MyRequestCPUrate (pid, m, n) should fail if either m or n are
* invalid (m < 1, or n < 1, or m > n).  It should also fail if a process
* calls RequestCPUrate (m, n) such that it would result in over-allocating
* the CPU.  Over-allocation occurs if the sum of the rates requested by
* processes exceeds 100%.  You may assume m/n will be no smaller than 1%
* (we will not test smaller ratios), and so it is OK for MyRequestCPUrate
* to fail for smaller ratios.  If the call fails (for whatever reason),
* MyRequestCPUrate should have NO EFFECT, as if the call were never made;
* thus, it should not affect the scheduling of other processes, nor the
* calling process.  (The example below illustrates what happens when a
* request fails.)  Note that when a process exits, its portion of the CPU
* is released and is available to other processes.  A process may change
* its allocation by again calling RequestCPUrate (m, n) with different
* values for m and n.
*
* IMPORTANT: If the sum of the requested rates does not equal 100%, then
* the remaining fraction should be allocated to processes that have not
* made rate requests (or ones that made only failing rate requests).  This
* is important, as a process needs some CPU time just to be able to execute
* to be able to actually call RequestCPUrate (m, n).  A good policy for
* allocating the unrequested portion is to spread it evenly amongst processes
* that still have not made (or have only made failed) rate requests.
*
* Here's an example that should help clarify the above points, including
* what to do with unrequested CPU time and what happens when requests fail.
* Consider the following sequence of 5 processes A, B, C, D, E, F entering
* the system and some making CPU requests:
*
*      - A enters the system: A is able to use 100% of the CPU since there
*        are no other processes
*      - B enters the system: B shares the CPU with A; both get an equal
*        amount, 50% each
*      - B requests 40%: since there is at least 40% unrequested (in fact,
*        there is 100% unrequested), B gets 40%; A now gets the remaining 60%
*      - C enters the system: it shares the unrequested 60% with A (both
*        get 30%)
*      - C requests 50%: since there is at least 50% unrequested (in fact,
*        there is 60% unrequested), C gets 50%; A now gets the remaining 10%
*      - D enters the system: it shares the unrequested 10% with A (both
*        get 5%)
*      - D requests 20%: the request fails, and so D is treated as if it
*        never made the request; A and D continue to share 10% (both get 5%)
*      - D requests 10%: since there is at least 10% unrequested (in fact,
*        there is exactly 10% unrequested), D gets 10%; A now gets the
*        remaining 0%, i.e., it does not get any CPU time
*      - E enters the system: it shares the unrequested 0% with A (both
*        get zero CPU time, i.e., neither can run)
*      - D exits, freeing up 10%, which A and E now share (A and E both
*        get 5%)
*      - A exits, and so E gets the remaining 10%
*      - E exits, and now there are only processes B (which is getting 40%)
*        and C (which is getting 50%).  These processes have no expectation
*        of additional CPU time, so the remaining 10% may be allocated any
*        way you want: it can be allocated evenly, proportionally, randomly,
```

```
*        and even not at all! As long as a process gets (at least) what it
*        requested, the kernel considers it satisfied.
*
* SetTimer (int t) will cause the timer to interrupt after t timer ticks.
* A timer tick is a system-dependent time interval (and is 10 msecs in the
* current implementation).  Once the timer is set, it begins counting down.
* When it reaches 0, a timer interrupt is generated (and the kernel will
* automatically call HandleTimerIntr).  The timer is then stopped until a
* call to SetTimer is made.  Thus, to cause a new interrupt to go off in the
* future, the timer must be reset by calling SetTimer.
*
* GetTimer () will return the current value of the timer.
*
*
* Exercises
*
* 1. Set the TIMERINTERVAL to 1, and run the program below using the three
* existing scheduling policies: ARBITRARY, FIFO, and LIFO.  What is the
* effect on the outputs, and why?
*
* 2. Implement the ROUNDROBIN scheduling policy.  This means that each
* process should get a turn whenever a scheduling decision is made.  For
* ROUNDROBIN to be effective, the timer interrupt period must be made small
* enough.  Set the TIMERINTERVAL to 1 (the smallest possible value).  You
* should then see that the outputs of the processes will be interleaved,
* as in: 1234123412341234123412341234123412341234 (not necessarily perfectly
* ordered as shown.  Why?  Hint: Distinguish between a fixed amount of time
* and the execution of enough instructions to print out a character).
*
* 3. Try larger values for TIMERINTERVAL, such as 10 and 100.  What is the
* effect on the interleaving of the output, and why?
*
* 4. Implement the PROPORTIONAL scheduling policy.  This allows processes to
* call RequestCPUrate (m, n) to receive a fraction of CPU time equal to m/n;
* specifically, within n consecutive quantums, m should be allocated to that
* process.  For example, consider the four processes in the program below,
* where process 1 requests 40% of the CPU by calling RequestCPUrate (4, 10),
* process 2 requests 30% of the CPU by calling RequestCPUrate (3, 10),
* process 3 requests 20% of the CPU by calling RequestCPUrate (2, 10), and
* process 4 requests 10% of the CPU by calling RequestCPUrate (1, 10).  With
* TIMERINTERVAL set to 1, this should produce an interleaving of the
* processes' outputs where ratio of characters printed by processes 1, 2, 3,
* and 4, are 4 to 3 to 2 to 1, respectively.  A sample output is as follows:
* 1213124123121312412312131241231213124123121324232324233433433434444444444
* NOTE: THIS IS JUST A SAMPLE, YOUR OUTPUT MAY DIFFER FROM THIS!
*
* Your solution should work with any number of processes (up to MAXPROCS)
* that have each called RequestCPUrate (m, n).  You should allow any process
* to call RequestCPUrate (m, n) multiple times, which would change its share.
* RequestCPUrate should fail if m < 1, or n < 1, or m > n, or if m/n would
* cause the overall CPU allocation to exceed 100%.  In that case, the call
* should have no effect (as if it were never called).  For any process that
* does not call RequestCPUrate, that process should get any left-over cycles
* (unless 100% were requested, then it would get none).
*
* A good solution will have the following properties:
*
* 1. After a process successfully calls RequestCPUrate (m, n), that process
* should utilize m/n of the CPU over the time period measured from when the
* call is made to when the process exits (or when a new successful call is
* made, at which point a new period of measurement begins; if the call is
```

```
 * not successful, then the prevous request remains in force).
 *
 * 2. 100 ticks will be used as the maximum allowable time over which the
 * target m/n CPU utilization must be achieved, and you may limit both
 * m and n to values within the range 1 to 100 (with m < n).  Furthermore,
 * you will be allowed a 10% slack in how close you come to m/n from the
 * low end, meaning that your solution will be considered correct if the
 * actual utilization of each process is at least 90% of its requested m/n.
 * So, for example, if a process requests and is allocated 50%, it is
 * acceptable for the measured utilization to be as low as 45% (and there
 * is no limit as to how much above 50% it gets, especially considering
 * that it may also receive free CPU time not requested by any process).
 * The reason for this specification is that since time is allocated and
 * measured in discrete units of ticks, it takes some minimal amount of
 * time before the target CPU utilization can be achieved.  For example,
 * if m = 2 and n = 7, at least 7 ticks must pass before a process can
 * possibly utilize precisely 2/7 of the CPU by being allocated two out
 * of seven ticks.
 *
 * 3. Unused CPU time should be equally distributed to any remaining processes
 * that have not requested CPU time.
 *
 * 4. You should avoid the repeated use of floating point operations, which
 * can be expensive if done every tick.  Using approximations for your
 * calculations based on using mostly integer operations is recommended.
 * (However, a working solution that uses floating point operations is
 * better than a non-working solution that avoids them, so get something
 * working first, and then optimize it.)
 *
 *
 * You must turn in your version of mykernel2.c, with all the scheduling
 * policies implemented.  You should set TIMERINTERVAL to 1, which should
 * work with ALL of your policies.
 */

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

void Main ()
{
        if (Fork () == 0) {

                if (Fork () == 0) {

                        /* Process 4 */
                        RequestCPUrate (1, 10);
                        SlowPrintf (7, "44444444444444444");
                        Exit ();
                }

                /* Process 2 */
                RequestCPUrate (3, 10);
                SlowPrintf (7, "22222222222222222");
                Exit ();
        }

        if (Fork () == 0) {

                /* Process 3 */
                RequestCPUrate (2, 10);
```

```
                SlowPrintf (7, "333333333333333333");
                Exit ();
        }

        /* Process 1 */
        RequestCPUrate (4, 10);
        SlowPrintf (7, "111111111111111111");
        Exit ();
}
```