# Programming Assignment 2

Due 11:59pm Sunday, January 24

## Introduction

---

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 mycode2.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. 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, "getprogram2". This will create a directory called "pa2" and will copy the relevant files into that directory. It is important that you do all your work in your pa2 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 pa2 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, pa2.
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 mycode2.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. 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 must 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).

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 mycode2.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 mycode2.c (it's OK to have Printfs 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., mycode2.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 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 function 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 can be 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 mycode2.c. It contains a rudimentary process table data structure, "proctab[]", and a set of functions THAT ARE CALLED BY OTHER PARTS OF THE KERNEL (the source of which you don't have nor require access to). Your ability to modify the scheduler is via these functions, 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 the functions, as the rest of the kernel depends on them. Also, your system must support up to MAXPROCS existing 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 actually exist at any single point in time).

Let's look at the functions:

***InitSched()*** is called at system start-up time. Here, the process table is initialized (all the entries are marked invalid). It also calls SetTimer(t), 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 function.

***StartingProc(int p)*** is called by the kernel when process p starts up. Specifically, whenever Fork() is called by a process, which causes entry into the kernel, StartingProc(p) will be called from within the kernel with parameter p, where p 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 the kernel when process p is ending. Whenever a process calls Exit() (implicitly if there is no explicit call), which causes entry into the kernel, EndingProc(p) will be called from within the kernel with parameter p which is 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 sys.h). The scheduling policy can be set by calling the kernel function SetSchedPolicy(p), where p is the policy ID (one of the above constants), which is called in InitSched(). SchedProc() should return a PID, or 0 if there are no processes to run. This is where you implement the core of the various scheduling policies. The current code for SchedProc implements ARBITRARY, which simply happens to find 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 additional functions that are part of the kernel (but not in mycode2.c), that you may call from within the above functions:

***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). MANY STUDENTS HAVE DIFFICULTY UNDERSTANDING WHY THIS FUNCTION IS NECESSARY, so please read carefully. When you write your code for the above functions (***StartingProc, EndingProc***, ...), there may be a point where you would like to cause the kernel to make a scheduling decision regarding which process should run next. This is where you should call ***DoSched()***, which tells the kernel that it should EVENTUALLY call ***SchedProc()***, as soon as the kernel determines it can do so. Why not immediately? Because other code, including the remaining code of the function you are writing, may need to execute before the kernel is ready to select a process to run. As this is a common confusion, we emphasize that ***DoSched()*** does NOT directly call ***SchedProc(). DoSched()*** simply tells the kernel to record that ***SchedProc()*** should be called eventually. When the kernel is at a point where it is appropriate for it to consider making a scheduling decision, if ***DoSched()*** was indeed called in the past, the kernel THEN calls ***SchedProc().*** (And by the way, only a new call to ***DoSched()*** will cause the kernel to do this again.)

***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) existing 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 actually exist AT ANY SINGLE POINT IN TIME).

## Exercises

1. Implement the FIFO (First In First Out) scheduling policy. 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: 1111111111222222222244444444443333333333 (why this order?) Under FIFO, it should print: 1111111111222222222233333333334444444444 (why this order?).

2. Implement the LIFO (Last In First Out) scheduling policy. 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: 4444444444222222222233333333331111111111 (why this order, and not 4444444444333333333322222222221111111111?).

```
#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 mycode2.c, and study the functions 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(t)**. 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 p, int n)** is called by the kernel whenever a process with PID p calls **RequestCPUrate(int n)**, which is a system call that allows a process to request that it should be scheduled to run with an allocation of n > 0 out of every 100 quantums (i.e., n% of the time), or to run at no fixed allocation if n = 0. By default, processes run at whatever rate is determined by the scheduler (which generally depends on what other processes are in the system and what they are doing), and so a process does not generally have any guarantee of its rate of progress over time. **RequestCPUrate(n)** addresses this by giving the calling process such a guarantee. Allowable values of n are 0 to 100, where positive (non-zero) values indicate a request to run at n% of the CPU's execution rate, and 0 indicates that it should run at no specific guaranteed rate (the same as the default situation, when a process first starts running). The request with n = 0 can be used to "undo" a process's previous request where some guaranteed rate was in place; after the n = 0 request is made, the guaranteed rate is no longer in effect (if no previous n > 0 request was in effect, than the n = 0 request has no effect). Note that it does NOT mean that the process is requesting to actually get no CPU time (as otherwise, if this request were granted, it would never be able to run)! It just means the process is considered one that has not requested a fixed CPU rate.

Consider the following example. If a process wants to run at 33% of the CPU's execution rate, it can call **RequestCPUrate(33)**, thus asking that it run 33 out of every 100 quantums. What happens in the kernel is that when a process p calls **RequestCPUrate(n)**, the kernel is entered, and the kernel calls your **MyRequestCPUrate(p, n)** in mycode2.c, giving you the opportunity to implement the way in which proportional share is to be achieved. Continuing with the example, if the process were to later call **RequestCPUrate(0)**, the process no longer gets the 33% proportional rate; it just gets whatever CPU is made available (more about this below).

**MyRequestCPUrate(p, n)** should return 0 if successful, and -1 if it fails. **MyRequestCPUrate(p, n)** should fail if n is invalid (n < 0 or n > 100). It should also fail if a process calls **RequestCPUrate(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%. If the call fails (for whatever reason), **MyRequestCPUrate(p, n)** should have NO EFFECT, as if the call to **MyRequestCPUrate(p, n)** were never made; thus, it should not affect the scheduling of other processes, nor that of the calling process. 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(n)** with a different value for n > 0, or proceed with an unallocated rate with n = 0.

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(n)**. A good policy for allocating the unrequested portion is to spread it as evenly as possible 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%
- B requests 0%: this actually means it wants to give up its 40% guaranteed rate, and go back to competing for whatever cycles it can get; C still keeps its 50%, and A and B compete for the remaining 50%, thus each getting 25%.
- B requests 40% again; just as when it made this request earlier, since there is at least 40% unrequested (in fact, there is 50% unrequested which it is sharing with A), B gets 40%; 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%)

- B requests 30%: it had 40%, but it is OK to make another request, which in this case causes B to get 30% from this point, and 10% is made available to the unrequesting processes

- A exits, and so E gets the remaining 20%

- E exits, and now there are only processes B (which is getting 30%) and C (which is getting 50%). These processes have no expectation of additional CPU time, so the remaining 20% 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(t)*** is made. Thus, to cause a new interrupt to go off in the future, the timer must be reset by calling ***SetTimer(t).***

***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. (Note that the calls to ***RequestCPUrate(n)*** will be ignored since this function is not relevant to these policies.) 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 small. With the TIMERINTERVAL set to 1 (the smallest possible value), you should then see that the outputs of the processes will be interleaved. For example, if four processes arrived in the simple increasing order of 1, 2, 3, 4, (which, by the way, is NOT the same as the order of arrival of the four processes in this exercise), then the output should be something like 123412341234123412341234... (not necessarily perfectly ordered as shown. Why? Hint: Distinguish between a fixed amount of time and the time needed to execute the instructions to print out a character, which cannot be expected to precisely match.)

3. Try larger values for TIMERINTERVAL, such as 10, 100, and 1000. What is the effect on the interleaving of the output, and why?

4. Implement the PROPORTIONAL scheduling policy. This allows processes to call ***RequestCPUrate(n)*** to receive a fraction of CPU time equal to n%, i.e., n out of every 100 quantums. For example, consider the four processes in the program below, where process 1 requests 40% of the CPU by calling ***RequestCPUrate(40)***, process 2 requests 30% of the CPU by calling ***RequestCPUrate(30)***, process 3 requests 20% of the CPU by calling ***RequestCPUrate(20)***, and process 4 requests 10% of the CPU by calling ***RequestCPUrate(10)***. With TIMERINTERVAL set to 1, this should ideally 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:
12131241231213124123121312412312131241231213242323242334334334444444444

*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(n).*** You should allow any process to call RequestCPUrate(n) multiple times, which would change its share. ***RequestCPUrate*** should fail if n < 0 or n > 100, or if n would cause the overall CPU allocation to exceed 100%. If the call fails, then it should have no effect (as if the call were never made). For any process that does not call ***RequestCPUrate(n)***, that process should get any left-over cycles (unless 100% were requested, then it would get none).

**A valid solution MUST have the following properties:**

1. After a process successfully calls ***RequestCPUrate(n)***, that process should utilize at least 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 previous request remains in force).

2. 100 quantums will be used as the maximum allowable time over which the target n% CPU utilization (regarding processes that have been granted their requests via ***RequestCPUrate(n)***) must be achieved. Furthermore, you will be allowed a 10% slack in how close you come to n% from the low end, meaning that your solution will be considered correct if the actual utilization of EACH AND EVERY process is at least 90% of its requested n%. For example, if 3 processes request and are allocated 50%, 30% and 10%, respectively, their measured utilizations MUST be at least 45% (since 10% of 50% is 5%, and 50%-5% = 45%), 27%, and 9%, respectively. There is no limit as to how much any of the requesting processes can get above their requests, as long as (1) all the other requesting processes get at least their lower bound, and (2) if the sum of the requests is below 100% and there are k non-requesting processes, then each of those processes should get a chance to run at least once every (k * 100) quantums.

3. Unused CPU time should be EQUALLY distributed to any remaining non-requesting processes that have not requested CPU time. "Equally distributed" means that each process should get a chance to run before any of the others gets another chance.

4. We will test to determine correct target CPU utilizations (regarding processes that have been granted their requests via ***RequestCPUrate(n)***) during periods of "steady state," defined as a period of 100 OR MORE quantums during which no special events occur, AND that comes after a period of at least 100 quantums during which no special events occur, where a special event is a process creation, exit, or rate request. For example, consider a period of 200 quantums where no process creation, exit, or rate request occurs. Those first 100 quantums are ignored (to allow the system to stabilize) and the following 100 quantums qualify as a testing period to check whether target CPU utilizations are achieved. This rule APPLIES TO THE CHECKING OF TARGET UTILIZATIONS FOR PROPORTIONAL FOR REQUESTING PROCESSES; non-requesting processes must still follow Rule 2 such that they "get a chance to run at least once every (k * 100) quantums OVER ALL TIME, even during non-steady state periods. This rule also applies to ROUNDROBIN (where target utilizations can be interpreted to be 1/n, where n is the number of processes). This rule does not apply to other scheduling disciplines, for which there are separate tests as to whether correct operation occurs, such as correct ordering of processes for FIFO and LIFO, accounting for new processes, process exits, and correct return values for the various scheduling functions, amongst all other things.

5. Starvation must be avoided over all periods of time (steady state or non-steady state), if possible.

6. You should only use integer operations; NO FLOATING POINT ALLOWED.

You must turn in your version of mycode2.c, with all the scheduling policies implemented. You must set TIMERINTERVAL to 1, which must work for ALL of your policies.

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

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

            if (Fork() == 0) {

                  RequestCPUrate(10);                 // process 4
                  SlowPrintf(7, "44444444444444444");
                  Exit();
            }

            RequestCPUrate(30);                       // process 2
            SlowPrintf(7, "22222222222222222");
            Exit();
      }
```

```
	if (Fork() == 0) {

		RequestCPUrate(20);			// process 3
		SlowPrintf(7, "33333333333333333");
		Exit();
	}

	RequestCPUrate(40);				// process 1
	SlowPrintf(7, "11111111111111111");
	Exit();
}
```