COP 4600 – Fall 2011

Project 2 – HOST Dispatcher

I. Project Objectives

In this project, you will write a four-level priority process dispatcher operating within the constraints of finite available memory and I/O resources.

Due Date and Submission Instructions - The due date for Project 2 is Sunday, December 4th, 2011 at 11:59pm on Blackboard. You must submit all of your *.c and *.h files, as well as a makefile and a readme.txt file, which includes basic instructions on compiling your code and the basic functionality of your code. No executable program should be included. The project ***must*** compile on the c4labpc machines. If your submitted code does not compile on c4labpc11.cse.usf.edu-c4labpc29.cse.usf.edu, you will not receive any credit for any portion of the grade that requires compiled code. There are no exceptions to this rule.

Extra Credit: For each part 1-5, you may submit code that satisfies the specified functionality by the listed due date to receive 5% extra credit. We will check the code for functionality, and if it works, you will receive 5% extra credit. If it does not work, you will receive 0% extra credit. Part 6 is due at the same time as the overall project, so no extra credit is possible for Part 6. The due dates for each portion are listed below.

Optional due dates for project components:

Part	**Due Date(all at 11:59PM)**
Part 1 – A Batch Process Monitor	October 30th, 2011
Part 2 – Round Robin Dispatcher	November 6th, 2011
Part 3 – Feedback Dispatcher	November 13th, 2011
Part 4 – Memory Allocation	November 20th, 2011
Part 5 – Process Management	November 27rd, 2011
Final Submission (Required)	December 4th, 2011

II. Project Requirements

Write a four-level priority process dispatcher operating within the constraints of finite available memory and I/O resources. We will refer to this dispatcher as The Hypothetical Operating System Testbed (HOST). HOST operates at four priority levels:

- Real-Time processes must be run immediately on a First Come First Served (FCFS) basis, pre-empting any other processes running with lower priority. These processes are run till completion.
- Normal user processes are run on a time sliced three-level feedback dispatcher. The basic timing quantum of the dispatcher is 1 second. This is also the value for the time quantum of the feedback scheduler.

The dispatcher controls the following resources:

- 2 Printers

- 1 Scanner
- 1 Modem
- 2 CD Drives
- 1024 Mbyte Memory available for processes

The dispatcher is presented with a list of processes along with their arrival times, priority, and requested resources. The dispatcher ensures that each requested resource is solely available to that process throughout its lifetime in the 'ready-to-run' dispatch queues: from the initial transfer from the job queue through to process completion, including any intervening idle time quanta. The executing process is emulated by a supplied program (sigtrap.c) that reports any signals sent to it and 'ticks' once a second while it is running.

The complete project requirements are detailed in Stalling's project description posted on Blackboard under Sneak Preview Project 2.

III. Suggested Components and Optional Submissions for Extra Credit

Part 1 – A Batch Process Monitor

Due Date for +5 EC – Sunday, October 30th, 2011 at 11:59PM

A First-Come-First-Served (FCFS) dispatcher is essentially the same as a Batch Process Monitor. For the second project you will have a list of processes which you will need to schedule for execution depending on the time of submission and the priority of the process. The process list format is defined as follows:

<arrival time>, <priority>, <cputime>, <memory alloc>, <res >, <res2>, <res3>, <res4>

0, 0, 3, 64, 0, 0, 0, 0
2, 0, 6, 64, 0, 0, 0, 0
4, 0, 4, 64, 0, 0, 0, 0
6, 0, 5, 64, 0, 0, 0, 0
8, 0, 2, 64, 0, 0, 0, 0

You will need to create a queue of job structures so that you only dequeue a job when the appropriate time has elapsed and there is no longer a current job running. There is a guarantee that the jobs presented in the input dispatch file will be in chronological order of arrival.

The first thing to do is to define a process structure such as:

```
struct pcb {
        pid_t pid;              // system process ID
        char * args[MAXARGS];   // program name and args
        int arrivaltime;
        int remainingcputime;
        struct pcb * next;      // links for Pcb handlers
};
typedef struct pcb Pcb; typedef Pcb * PcbPtr;
```

This structure can be used for all the queues (linked lists) needed for the project (with a few extra elements in the structure).

Dispatcher algorithm:
The basic First-Come-First-Served (FCFS) dispatcher algorithm can be reduced to the following pseudo-code. Note that the tests may appear to be out of order (shouldn't one be doing the test at 4.i after the sleep rather than before it?). With the current model it makes no difference, but there are reasons for doing it in this order as we shall see when we come to the multiple dispatcher models later on.

1. Initialize dispatcher queue;
2. Fill dispatcher queue from dispatch list file;
3. Start dispatcher timer;
4. While there's anything in the queue or there is a currently running process:
   i. If a process is currently running;
      a. Decrement process **remainingcputime**;
      b. If times up:
         A. Send **SIGINT** to the process to terminate it;
         B. Free up process structure memory
   ii. If no process currently running &&
       dispatcher queue is not empty &&
       arrivaltime of process at head of queue is <=
       dispatcher timer:
      a. Dequeue process and start it (**fork** & **exec**)
      b. Set it as currently running process;
   iii. **sleep** for one second;
   iv. Increment dispatcher timer;
   v. Go back to 4.
5. Exit.

For the purposes of this exercise (and the 2nd project) you should use the process **process** (sigtrap.c)- it will tell you what signals you have sent it as well as ticking away for the length of time you allow it to run. In the structure above, you would initialize the structure to:

```
PcbPtr newPcb = (PcbPtr) malloc(sizeof(Pcb));
...
newPcb->args[0] = "process";
newPcb->args[1] = NULL;
newPcb....
```

and then, when doing the **exec** all you need to do is:

```
execvp(activePcb->args[0], activePcb->args);
```

There are a number of ways to terminate a job. Since we know its process ID, the easiest way to do this is to send the process a terminating signal using the **kill** function. The one we need to use here is **SIGINT**, which is the equivalent of the **Ctrl-C** keyboard interrupt fielded by the shell. You could also send **SIGKILL** but **SIGINT** is preferred. i.e. to terminate the process with Process ID **activePcb->pid**, you would call the function:

```
if(kill(activePcb->pid,SIGINT))
    fprintf(stderr,"terminate of %d failed",
            (int)activePcb->pid);
```

For the queue management, you may find the following function list useful:

PcbPtr startPcb(PcbPtr process)
- start a process
returns:
  PcbPtr of process
  NULL if start failed

PcbPtr terminatePcb(PcbPtr process)
- terminate a process
returns:
  PcbPtr of process
  NULL if terminate failed

PcbPtr createnullPcb(void)
- create inactive Pcb.
returns:
  PcbPtr of newly initialised Pcb
  NULL if malloc failed

PcbPtr enqPcb (PcbPtr headofQ, PcbPtr process)
- queue process (or join queues) at end of queue
- enqueues at "tail" of queue list.
returns:
  (new) head of queue

PcbPtr deqPcb (PcbPtr * headofQ);
- dequeue process - take Pcb from "head" of queue.
returns:
  PcbPtr if dequeued,
  NULL if queue was empty
  & sets new head of Q pointer in adrs at 1st arg

We have used a simple specific queue implementation here rather than a generic one, but there is nothing to stop you from using your own implementation provided that it operates as a proper FIFO queue.

To see the sort of sequence of output you should be getting, run the following command line on the course home directory (this should have the reference **hostd** and **process** applications)

>hostd fcfs.txt

and compare the output with the top example of Figure 9.5 of Stalling's Operating Systems textbook.

Part 2 – Round Robin Dispatcher

Due Date for +5 EC – Sunday, November 6th, 2011 at 11:59PM

Having got the FCFS dispatcher going from last week, we now need to develop a Round Robin dispatcher. To suspend and restart processes as required by the Round Robin dispatcher will require the use of the **SIGTSTP** and **SIGCONT** signals as well as the **SIGINT** signal that we used for the FCFS dispatcher.(see below).

To make the Round Robin dispatcher work in the same way as described in Stallings Figure 9.5 (RR q=1) the logic should be:

1. Initialize dispatcher queues (input queue and RR queue);
2. Fill input queue from dispatch list file;
3. Start dispatcher timer (**dispatcher timer = 0**);
4. While there's anything in any of the queues or there is a currently running process:
   i. Unload any pending processes from the input queue:
      While (**head-of-input-queue.arrival-time <= dispatcher timer**)
      dequeue process from input queue and enqueue on RR queue;
   ii. If a process is currently running:
      a. Decrement process **remainingcputime**;
      b. If times up:
         A. Send **SIGINT** to the process to terminate it;
         B. Free up process structure memory;
      c. else if other processes are waiting in RR queue:
         A. Send **SIGTSTP** to suspend it;
         B. Enqueue it back on RR queue;
   iii. If no process currently running && RR queue is not empty:
      a. Dequeue process from RR queue
      b. If already started but suspended, restart it (send **SIGCONT** to it)
         else start it (**fork** & **exec**)
      c. Set it as currently running process;
   iv. **sleep** for one second;
   v. Increment dispatcher timer;
   vi. Go back to 4.
5. Exit.

Try this with the following job list with a quantum of 1 sec:
0, 3, 3, 64, 0, 0, 0, 0
2, 3, 6, 64, 0, 0, 0, 0
4, 3, 4, 64, 0, 0, 0, 0
6, 3, 5, 64, 0, 0, 0, 0
8, 3, 2, 64, 0, 0, 0, 0
This should produce the sequence shown in Stallings Figure 9.5 (RR q=1)

Signals provide one way of communicating with and controlling processes. A standard process has 'handlers' to field all the UNIX/POSIX signals. The default action of these handlers is sometimes to do nothing, or, for others to suspend or terminate the process with or without a core dump, etc. The program **sigtrap.c** provided is an example of a program that traps some of the major process control signals and reports them on the terminal. This is in fact

the **process** process that is to be executed by your dispatcher for project 2.
sigtrap (& **process**) is similar to **sleepy** from the previous project in that it takes a command line argument to signify how many ticks it should tock. However, when **exec**'ed with no arguments (as for the project) it will 'tick' for a maximum of 20 seconds.

If you execute it as **sigtrap 1000**, and then apply the standard shell job control commands of **^Z**, **fg**, **^Z**, **kill %1**, etc you will get some idea of the signals that the shell is passing to its **exec**ed children.
If you start up another terminal, you can pass your own signals to the **sigtrap** process by using the system utility **kill**. Again if you start up **sigtrap 1000** and it reports a process ID of **nnnnn**, then try the command lines:

```
kill -TSTP nnnnn
kill -CONT nnnnn
kill -INT nnnnn
```

while observing the terminal window running the **sigtrap** process.
You can achieve the same effect inside a process that has forked **sigtrap** as a child:

```
#include <sys/types.h>
#include <signal.h>
    ...
    child_pid = fork();
    ...
    sleep(1);
    kill(child_pid, SIGTSTP);
    sleep(1);
    kill(child_pid, SIGCONT);
    sleep(1);
    kill(child_pid, SIGINT);
```

It is these last three signals we should be using for process control in the HOST dispatcher.

Part 3 – Feedback Dispatcher

Due Date for +5 EC – Sunday, November 13th, 2011

The logic is very similar to the previous exercise except that instead of enqueueing an incomplete process back onto the same (RR) queue that the process came from, the process is enqueued onto a lower priority process queue (if there is one to receive it). Obviously, the lowest priority queue operates in exactly the same way as a simple Round Robin queue.

To make theFeedback dispatcher work in the same way as described in Stallings Figure 9.5 (Feedback q=1) the logic should be:

1. Initialize dispatcher queues (input queue and feedback queues);
2. Fill input queue from dispatch list file;

3. Start dispatcher timer (**dispatcher timer = 0**);
4. While there's anything in any of the queues or there is a currently running process:
   i. Unload pending processes from the input queue:
      While (**head-of-input-queue.arrival-time <= dispatcher timer**)
      dequeue process from input queue and enqueue on highest priority feedback queue (assigning it the appropriate priority);
   ii. If a process is currently running:
      a. Decrement process **remainingcputime**;
      b. If times up:
         A. Send **SIGINT** to the process to terminate it;
         B. Free up process structure memory;
      c. else if other processes are waiting in any of the feedback queues:
         A. Send **SIGTSTP** to suspend it;
         B. Reduce the priority of the process (if possible) and enqueue it on the appropriate feedback queue
   iii. If no process currently running && feedback queues are not all empty:
      a. Dequeue a process from the highest priority feedback queue that is not empty
      b. If already started but suspended, restart it (send **SIGCONT** to it)
         else start it (**fork** & **exec**)
      c. Set it as currently running process;
   iv. **sleep** for one second;
   v. Increment dispatcher timer;
   vi. Go back to 4.
5. Exit.

As you can see, there is really not much difference between this and the Round Robin we completed last week.
Try this with the following job list with a quantum of 1 sec:
0, 1, 3, 64, 0, 0, 0, 0
2, 1, 6, 64, 0, 0, 0, 0
4, 1, 4, 64, 0, 0, 0, 0
6, 1, 5, 64, 0, 0, 0, 0
8, 1, 2, 64, 0, 0, 0, 0
This should produce the sequence shown in Stallings Figure 9.5 (Feedback q=1)

Part 4 – Memory Allocation

Due Date for +5EC – Sunday, November 20th, 2011 at 11:59PM

The various methods of dynamic segment partitioning are described in Stallings chapter 7. If you wish to implement Best Fit, First Fit, Next Fit, or Worst Fit memory allocation policy, it is probably best to do this by describing the memory as a structure in a linked list:

struct mab {

```
        int offset;
        int size;
        int allocated;
        struct mab * next;
struct mab * prev; };
typedef struct mab Mab; typedef Mab * MabPtr;
```

For the Buddy system, you will need to implement this structure as the node for a Tree with Right and Left pointers.

Either way, the following set of prototypes give a guide as to the functionality you will need to provide:

```
MabPtr memChk(MabPtr m, int size);   // check if memory available
MabPtr memAlloc(MabPtr m, int size); // allocate memory block
MabPtr memFree(MabPtr m);            // free memory block

MabPtr memMerge(MabPtr m);           // merge two memory blocks
MabPtr memSplit(MabPtr m, int size); // split a memory block
```

Allocating memory is a process of finding a block of 'free' memory big enough to hold the requested memory block. The free block is then split (if necessary) with the right size block marked as 'allocated' and the remaining block (if any) marked as 'free'.

Freeing a memory block is done by changing the flag on the block to 'free'. After this the blocks on both sides of the newly 'freed' block are examined. If either (or both of them are 'free') the 'free' blocks must be merged together to form just one 'free' block.

The **mab** structure above has been constructed with a reverse link as well as a forward link. This makes it easier to check for adjacent unallocated blocks that need to be merged when freeing up a memory block. Without the reverse link, you will need to do a separate pass through the memory arena after marking a block as free to merge any adjacent free blocks.

You can make up your own test program or go directly to implementing the routines in your Feedback dispatcher. For this you will need to insert a separate queue in between the input queue and the Feedback queues as indicated in the project 2 specification.

Implementing the extra stage, processes are passed from the input queue (Job Dispatch List) to the Feedback pending queue (User Job Queue) when they have "arrived". They are dequeued and enlisted on the appropriate Feedback queue when sufficient memory is available for them in the memory arena - the memory is allocated when they are enqueued and deallocated when they are terminated.

The modified logic will be:

1. Initialize dispatcher queues (input queue, user job queue, and feedback queues);
2. Fill input queue from dispatch list file;
3. Start dispatcher timer (**dispatcher timer = 0**);
4. While there's anything in any of the queues or there is a currently running process:
    i. Unload pending processes from the input queue:
       While (**head-of-input-queue.arrival-time <= dispatcher**

timer)
dequeue process from input queue and enqueue on user job queue;

ii. Unload pending processes from the user job queue:
While (**head-of-user-job-queue.mbytes** can be allocated)
dequeue process from user job queue, allocate memory to the process and enqueue on highest priority feedback queue (assigning it the appropriate priority);

iii. If a process is currently running:
   a. Decrement process **remainingcputime**;
   b. If times up:
      A. Send **SIGINT** to the process to terminate it;
      B. Free memory allocated to the process;
      C. Free up process structure memory;
   c. else if other processes are waiting in any of the feedback queues:
      A. Send **SIGTSTP** to suspend it;
      B. Reduce the priority of the process (if possible) and enqueue it on the appropriate feedback queue

iv. If no process currently running && feedback queues are not all empty:
   a. Dequeue a process from the highest priority feedback queue that is not empty
   b. If already started but suspended, restart it (send **SIGCONT** to it)
      else start it (**fork** & **exec**)
   c. Set it as currently running process;

v. **sleep** for one second;

vi. Increment dispatcher timer;

vii. Go back to 4.

5. Exit.

or something like that!

Try this with the following job list with a quantum of 1 sec and available memory of 1024 Mbyte:

0, 1, 1, 192, 0, 0, 0, 0
0, 1, 3, 32, 0, 0, 0, 0
0, 1, 1, 416, 0, 0, 0, 0
0, 1, 3, 32, 0, 0, 0, 0
0, 1, 1, 160, 0, 0, 0, 0
0, 1, 3, 128, 0, 0, 0, 0
7, 1, 2, 108, 0, 0, 0, 0
7, 1, 2, 256, 0, 0, 0, 0
7, 1, 2, 80, 0, 0, 0, 0

This interesting mixture should run processes 0-5 in the first 6 seconds and leave the memory arena as:

offset	**size**	**allocated**
0	192	FALSE
192	32	TRUE

224	416	FALSE
640	32	TRUE
672	160	FALSE
832	128	TRUE
960	64	FALSE

with the Next Fit pointer at offset 960. This will be the case whichever of the four allocation policies you are using. The next three processes should then be loaded into the queue and dequeued, enlisted and then executed in the order: process 6, 7, then 8.

Under First Fit, the 108 Mbyte, 256 Mbyte, and 80 Mbyte processes will be allocated at offset 0, 224, and 108 respectively.

Under Next Fit, they will be allocated at offset 0, 224, and 480 respectively.

Under Best Fit, they will be allocated at offset 672, 224, and 480 respectively.

Under Worst Fit, they will be allocated at offset 224, 332, and 0 respectively.

N.B. The final version of the **hostd** dispatcher normally reserves the first 64 Mbyte of memory for Real Time processes so that 64 should be added to all of the above offsets. However, when run with the **-mnr** option, **hostd** will not pre-allocate the Real Time memory block. Run the command line:

hostd -h

to find out which version of the dispatcher you have and the command line switches in **hostd** to change the memory allocation model and printout options (your **hostd** does not need to provide this extended functionality).

Part 5 - Process Management

Due Date for +5EC – Sunday, November 27th, 2011

The Resource Management required for the second project is very simple. The Real-Time processes need no i/o resources, so that resource allocation need only be implemented for the Lo-Priority processes. In addition, you know in advance what resources are going to be required by the processes so they can all be allocated before the process is admitted to the feedback queue(s) and marked as READY to run.

The check, allocation and freeing of resources can be done at the same time as the check, etc for memory allocation, so it makes sense to have analogous function entry points (e.g. **rsrcChk**, **rsrcAlloc**, and **rsrcFree**). The only other thing we need to do is merge the two

different dispatchers we have written. To comply with the project specification we will also need to add the capability to insert different priority jobs into the three different Feedback queues - you can now see why the list of jobs in **rr.txt** all have priority **3** so that all the jobs are inserted at the lowest priority feedback queue, effectively limiting our model to a simple Round Robin queue.

The Full HOST Dispatcher

We should now be ready to put all the parts of the HOST dispatcher together:

1. Initialize dispatcher queues (input queue, real time queue, user job queue, and feedback queues);
2. Initialize memory and resource allocation structures;
3. Fill input queue from dispatch list file;
4. Start dispatcher timer (**dispatcher timer = 0**);
5. While there's anything in any of the queues or there is a currently running process:
   - i. Unload pending processes from the input queue:
     While (**head-of-input-queue.arrival-time <= dispatcher timer**)
     dequeue process from input queue and enqueue on either:
       a. Real-time queue or
       b. User job queue;
   - ii. Unload pending processes from the user job queue:
     While (**head-of-user-job-queue.mbytes** can be allocated and resources are available)
       a. dequeue process from user job queue,
       b. allocate memory to the process,
       c. allocate i/o resources to process, and
       d. enqueue on appropriate priority feedback queue;
   - iii. If a process is currently running:
       a. Decrement process **remainingcputime**;
       b. If times up:
           A. Send **SIGINT** to the process to terminate it;
           B. Free memory and resources allocated to the process (user processes only);
           C. Free up process structure memory;
       c. else if it is a user process and other processes are waiting in any of the queues:
           A. Send **SIGTSTP** to suspend it;
           B. Reduce the priority of the process (if possible) and enqueue it on the appropriate feedback queue
   - iv. If no process currently running && real time queue and feedback queue are not all empty:
       a. Dequeue a process from the highest priority queue that is not empty
       b. If already started but suspended, restart it (send **SIGCONT** to it)
          else start it (**fork** & **exec**)
       c. Set it as currently running process;
   - v. **sleep** for one second;
   - vi. Increment dispatcher timer;

vii. Go back to 5.
6. Exit.

Note that there is nothing really new in the above - it is a merging of logic from previous exercises.

Try this with the following example job list:

```
0, 1, 2, 128, 1, 0, 0, 1
1, 0, 1, 64, 0, 0, 0, 0
1, 1, 1, 128, 0, 0, 0, 0
1, 1, 2, 256, 1, 0, 0, 0
2, 1, 3, 256, 0, 0, 0, 2
3, 1, 2, 770, 1, 0, 0, 0
```

The order of execution will be:

```
T = 0   1   2   3   4   5   6   7   8   9  10  11
P0:  PFR  S   S   S   R |
P1:      QR |
P2:      PF   R |
P3:      PF   F   R   S   R |
P4:           P   P   P   P  FR   R   R |
P5:               P   P   P   P   P   P   FR  R |
```

where:

P: in pending queue waiting for resources
Q: queued in Real Time queue - not started
F: queued in feedback queue - not started
R: process running
S: process suspended in feedback queue
|: process terminated

Part 6 – Finishing Your Dispatcher

This week you will be putting the finishing touches to your dispatcher and the documentation for the second project:

As for the first project you are required to provide a **makefile** with your project

makefile Support
Marking the project will entail the marker recompiling your source code from scratch. To automate this process, a requirement of the project is that your shell must be able to be built using a **make** file. If you are using two source files **hostd.c** and **pcb.c**, both of which reference the local **'include'** file **hostd.h**, then your **makefile** should (at the minimum) look like this:

```
# Joe Citizen, s1234567 - Operating Systems Project 2
        hostd: hostd.c pcb.c hostd.h
        gcc hostd.c pcb.c -o hostd
```

Then your dispatcher is re-built by just typing **make** at the command prompt if any of the **.c** or **.h** files have been changed since the last time the shell was built.

Note that this **makefile** names the resulting executable file as **hostd**, another requirement of the project. Note also that the third line of

the above **makefile** - **gcc hostd...** - the *action line* - must start with a tab.

Polishing:

1. Ensure that you will never try to access a NULL pointer (defensive programming)
2. Are your comments such that someone else who has passed Program Design six months ago, but who has done no programming since, could understand what your program and its functions are trying to do and how your code is addressing those requirements? Could they, in the minimum amount of time, make alterations to the code to meet differing requirements?
3. Check your implementation produces the same output as the example **hostd** dispatcher. Note that example scripts have been provided in previous exercises to illustrate the particular functionality of the dispatcher under discussion. Those scripts (amongst others) will be used for marking your project. Ensure that your dispatcher produces the required output.
4. Check your **makefile** works
5. Ensure that all files are named as requested (no upper case characters).

IV. Late Assignments

Per the syllabus, for each day a programming project is late, the grade you receive is reduced by 15%. For example, if you earned a 95% and submitted your assignment one day late, your grade will become an 80%.

V. Academic Honesty

You must work on this assignment by yourself. In Proj0, you got to experience working with MOSS. We will be checking your submitted assignment against all of your classmates to protect the integrity of your grade. Therefore, cheating will be treated according to University policies. If you are caught cheating on this assignment, you will receive an FF grade for the class. Material copied from the Web and submitted as your work *is cheating*.

VI. Advice

- Many of you found out the hard way that working on the code on your favorite C compiler, then transferring it to the C4 Lab machines will cause you considerable headache and wasted time.
- Do try every extra credit. This is done to reward people for good time management! Also, even if you get 0% on an extra credit submission, it is beneficial since you will become aware of errors in your code, which you'll be able to fix, and will improve your project code (and project grade!).
- Ask for help early and often! Our office hours and e-mails are listed on the syllabus.
- E-mail the professor or TAs immediately if you find yourself in a situation in which you are unsure of whether or not something you are doing constitutes academic dishonesty.