THE DISPATCHER SHELL

The DISPATCHER is a multiprogramming system with a four-Level priority process dispatcher for System processes and User processes.

Four-Level Priority Dispatcher

The dispatcher operates at four priority levels:

1. System processes must be run immediately on a first-come-first-served (FCFS) basis, preempting any other processes running with lower priority. These processes are run until completion.

2. Normal User processes are run on a three-level feedback dispatcher (Figure 1). The basic timing quantum of the dispatcher is 1 second. This is also the value for the time quantum of the feedback scheduler.

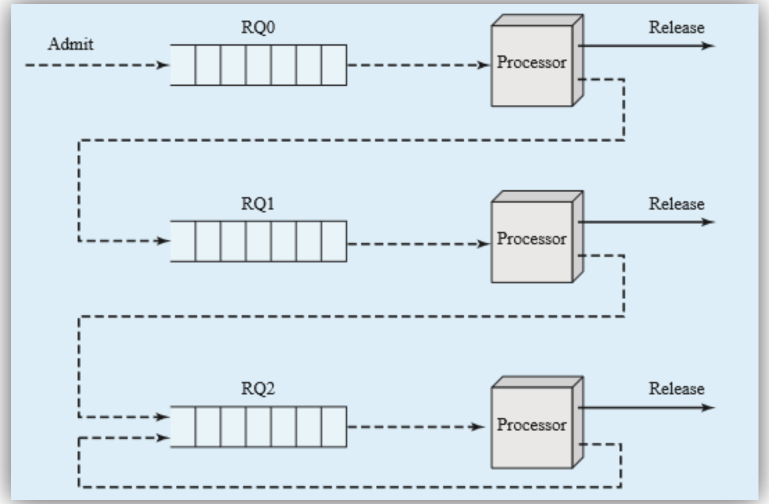

Figure 1 Three-Level Feedback Scheduling

The dispatcher needs to maintain the two queues—System and User priority— fed from the job dispatch list (detailed later). The dispatch list is examined at every dispatcher tick and jobs that "have arrived" are transferred to the appropriate queue. The queues are then examined; any System jobs are run to completion, preempting any other jobs currently running.

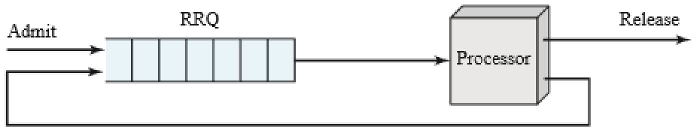

Figure 2 Round-Robin Dispatcher

The System priority job queue must be empty before the lower priority feedback dispatcher is reactivated.   Normal operation of a feedback queue will accept all jobs at the highest priority level and degrade the priority after each completed time quantum. However, this dispatcher has the ability to accept jobs at a lower priority, inserting them in the appropriate queue. This enables the dispatcher to emulate a simple round robin dispatcher (Figure 2) if all jobs are accepted at the lowest priority.

When all "ready" higher priority jobs have been completed, the feedback dispatcher resumes by starting or resuming the process at the head of the highest priority nonempty queue. At the next tick the current job is suspended (or terminated) if there are any other jobs "ready" of an equal or higher priority.

The logic flow should be as shown in Figure 3 (and as discussed subsequently in this project assignment).

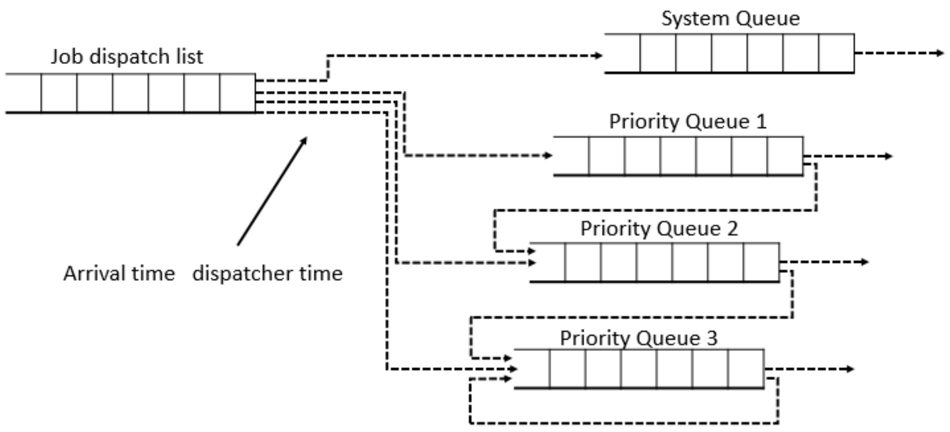

Figure 3 Dispatcher Logic Flow

Processes

Processes are simulated by the dispatcher creating a new process for each dispatched process. This process is a generic process `(supplied as "process" - source: sigtrap.c (see makefile` – it will be compiled to `process.exe))` that will be used to mimic any priority process and be dispatched by the dispatcher. It actually runs itself at very low priority, sleeping for 1-second periods and displaying the following:

1. A message displaying the process ID when the process starts;
2. A regular message every second the process is executed; and
3. A message when the process is Suspended, Continued, or Terminated.

The process will terminate of its own accord after 20 seconds if it is not terminated by your dispatcher (This is because a process will give a burst time by input). The process prints out using a randomly generated color scheme (already written in `sigtrap.c)` for each unique process, so that individual "slices" of processes can be easily distinguishable. Use this process rather than your own.

The life cycle of a process is as follows:

1. The process is submitted to the dispatcher input queues via an initial process list that designates the arrival time, priority and processor time required (in seconds).
2. A process is "ready-to-run" when it has "arrived".

3. Any pending System jobs are submitted for execution on a first-come-first-served basis.

4. Lower priority User is transferred to the appropriate priority queue within the feedback dispatcher unit.

5. A job is started by calling the `fork` function which returns a `pid`. Store the `pid` for each process. We then call `(execvp("./process",..))` to run the process.

6. A System process is allowed to run until its time has expired when the dispatcher kills it by sending a SIGINT signal to it. `(kill(process->pid, SIGINT))`

7. A low priority User job is allowed to run for one dispatcher tick (one second) before it is suspended (`SIGTSTP`) or terminated (`SIGINT`) if its time has expired. If suspended (`kill(process->pid, SIGTSTP)`), its priority level is lowered (if possible) and it is requeued on the appropriate priority queue as shown in Figures 1 and 2. To retain synchronization of output between your dispatcher and the child process, your dispatcher should wait for the process to respond to a SIGTSTP or SIGINT signal before continuing (`waitpid(process->pid, &status, WUNTRACED)`).

8. Provided no higher-priority System jobs are pending in the submission queue, the highest priority pending process in the feedback queues is started or restarted (SIGCONT) (`kill(process->pid, SIGCONT)`).

9. When there are no more processes in the dispatch list, the input queues and the feedback queues, the dispatcher exits.

Dispatch List

The Dispatch List is the list of processes to be processed by the dispatcher. The list is contained in a text file that is specified on the command line. That is,

```
>./dispatcher dispatchlist.txt
```

Each line of the list describes one process with the following data as a "comma-space" delimited list:

```
<arrival time>, <priority>, <processor time>
```

where, a System process has the highest priority of 0; and a User process can have a priority from 1-3 with 3 the lowest. For example,

12, 0, 1

12, 1, 2

13, 3, 6

would indicate the following:

1st Job: Arrival at time 12, priority 0 (System) and requiring 1 second of processor time.

2nd Job: Arrival at time 12, priority 1 (high priority User job) and requiring 2 seconds of processor time

3rd Job: Arrival at time 13, priority 3 (lowest priority User job) and requiring 6 seconds of processor time.

The submission text file can be of any length, containing up to 1000 jobs. It will be terminated with an end-of-line('\n') followed by an end-of-file (EOF) marker.

Project Requirements

1. Design a dispatcher that satisfies the above criteria

2. Implement the dispatcher using the C language.

3. The source code MUST be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code.

4. Create separate function to terminate process (name it *`terminateProcess`*), to suspend user process (name it *`suspendProcess`*), to start process (name it *`startProcess`*) and finally to restart suspended user process (name it *`restartProcess`*) .

5. The submission should contain only source code file(s) and/or include file(s). Sample *`makefile`* is attached. If you create your own version of makefile, please include it to your submission. No executable program should be included. If the submitted code does not compile, it cannot be graded.

6. The source code with main function should be name `dispatcher.c`

7. The makefile should generate the binary executable file `dispatcher` (all lowercase, please).

8. Use the provided source code for *`sigtrap.c`*.

9. include "`signal.h`" "`sys/wait.h`" to use waitpid function in your source code.

10. A test input file is offered. But you should test the operation of your dispatcher extensively.

11. As usual, DO NOT ZIP your submission.