Programming Project Two
The HOST Dispatcher Shell

William Stallings

Supplement to
Operating Systems, Seventh Edition
Prentice Hall 2011
ISBN: 013230998X
http://williamstallings.com/OS/OS7e.html

The Hypothetical Operating System Testbed (HOST) is a multiprogramming system with a four level priority process dispatcher operating within the constraints of finite available resources.

Four-Level Priority Dispatcher

The dispatcher operates at four priority levels:

1. Real-time 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 P2.1). 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 needs to maintain two submission queues - Real Time and User priority — fed from the job dispatch list. The dispatch list is examined at every dispatcher tick and jobs that "have arrived" are transferred to the appropriate submission queue. The submission queues are then examined; any Real Time jobs are run to completion, preempting any other jobs currently running.

The Real-Time priority job queue must be empty before the lower priority feedback dispatcher is reactivated. Any User priority jobs in the User job queue that can run within available resources (memory and I/O devices) are transferred to the appropriate priority queue. 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 P2.2) if all jobs are accepted at the lowest priority.

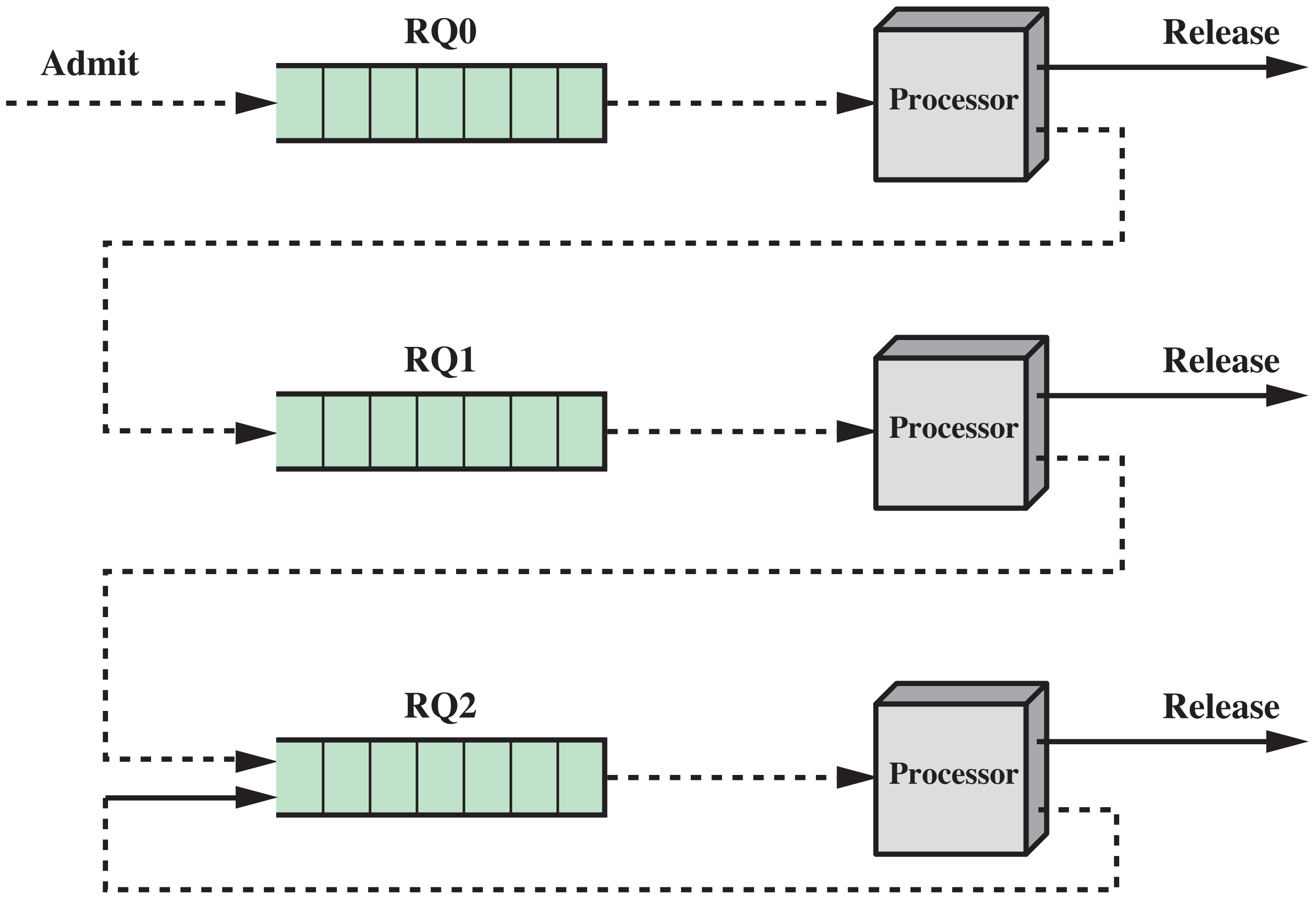

Figure P2.1 Three-Level Feedback Scheduling

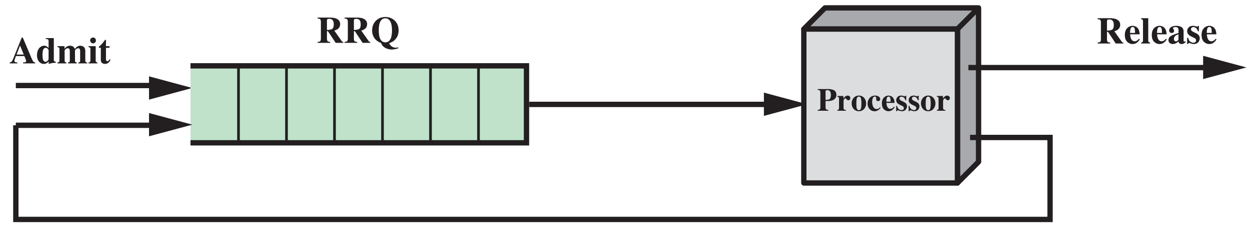

Figure P2.2 Round-Robin Dispatcher

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 and its resources released) if there are any other jobs "ready" of an equal or higher priority.

The logic flow should be as shown in Figure P2.3 (and as discussed subsequently in this project assignment).

Resource Constraints

The HOST has the following resources:

- 2 Printers
- 1 Scanner
- 1 Modem
- 2 CD drives
- 1024 Mbyte memory available for processes

Low-priority processes can use any or all of these resources, but the HOST dispatcher is notified of which resources the process will use when the process is submitted. 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 to the Priority 1-3 queues through to process completion, including intervening idle time quanta.

Real-Time processes will not need any I/O resources (Printer, Scanner, Modem, CD), but will obviously require memory allocation - this memory requirement will always be 64 Mbytes or less for Real-Time jobs.

Memory Allocation

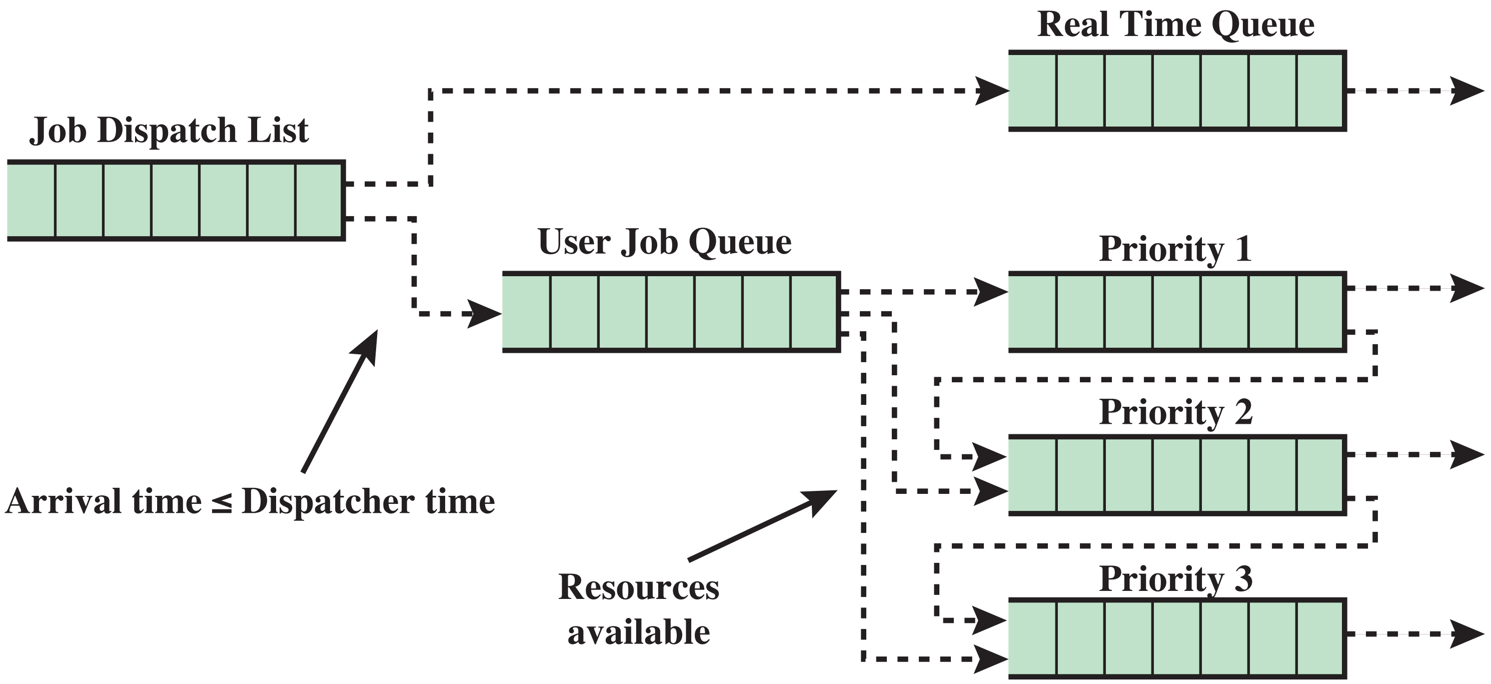

Figure P2.3 Dispatcher Logic Flow

For each process, a **contiguous** block of memory must be assigned. The memory block must remain assigned to the process for the lifetime of the process.

Enough contiguous spare memory must be left so that the Real-Time processes are not blocked from execution — 64 Mbytes for a running Real-Time job, leaving 960 Mbytes to be shared among "active" User jobs.

The HOST hardware MMU cannot support virtual memory so no swapping of memory to disk is possible. Neither is it a paged system.

Within these constraints, any suitable variable partition memory allocation scheme (First Fit, Next Fit, Best Fit, Worst Fit, Buddy, etc.) may be used.

Processes

Processes on HOST 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`) that can be used for any priority process. It actually runs itself at very low priority, sleeping for one-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. The process prints out using a randomly generated color scheme 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, processor time required (in seconds), memory block size, and other resources requested.

2. A process is "ready-to-run" when it has "arrived" and all required resources are available.

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

4. If enough resources and memory are available for a lower priority User process, the process is transferred to the appropriate priority queue within the feedback dispatcher unit, and the remaining resource indicators (memory list and i/o devices) updated.

5. When a job is started (`fork` and `exec("process",...))`, the dispatcher will display the job parameters (Process ID, priority, processor time remaining (in seconds), memory location and block size, and resources requested) before performing the `exec`.

6. A Real-Time process is allowed to run until its time has expired when the dispatcher kills it by sending a `SIGINT` signal to it.

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, its priority level is lowered (if possible) and it is requeued on the appropriate priority queue as shown in Figures P2.1 and P2.3. 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(p->pid, &status, WUNTRACED)`). To match the performance sequence indicated in the comparison of scheduling policies (see Figure 9.5), the User job should not be suspended and moved to a lower priority level unless another process is waiting to be (re)started.

8. Provided no higher-priority Real-Time jobs are pending in the submission queue, the highest priority pending process in the feedback queues is started or restarted (`SIGCONT`).
9. When a process is terminated, the resources it used are returned to the dispatcher for reallocation to further processes.
10. 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,

```
>hostd dispatchlist
```

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

```
<arrival time>, <priority>, <processor time>, <Mbytes>,
<#printers>, <#scanners>, <#modems>, <#CDs>
```

Thus,

```
12, 0, 1, 64, 0, 0, 0, 0
12, 1, 2, 128, 1, 0, 0, 1
13, 3, 6, 128, 1, 0, 1, 2
```

would indicate the following:

1st Job: Arrival at time 12, priority 0 (Real-Time), requiring 1 second of processor time and 64 Mbytes memory — no I/O resources required.

2nd Job: Arrival at time 12, priority 1 (high priority User job), requiring 2 seconds of processor time, 128 Mbytes of memory, 1 printer, and 1 CD drive.

3rd Job: Arrival at time 13, priority 3 (lowest priority User job), requiring 6 seconds of processor time, 128 Mbytes of memory, 1 printer, 1 modem, and 2 CD drives.

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

Dispatcher input lists to test the operation of the individual features of the dispatcher are described subsequently in this project assignment. It should be noted that these lists will almost certainly form the basis of tests that will be applied to your dispatcher during marking. Operation as described in the exercises will be expected.

Obviously, your submitted dispatcher will be tested with more complex combinations as well!

A fully functional working example of the dispatcher will be presented during the course. If in any doubt as to the manner of operation or format of output, you should refer to this program to observe how your dispatcher is expected to operate.

Project Requirements

1. Design a dispatcher that satisfies the above criteria. In a formal design document,

a. Describe and discuss what memory allocation algorithms you could have used and justify your final design choice.

b. Describe and discuss the structures used by the dispatcher for queueing, dispatching, and allocating memory and other resources.

c. Describe and justify the overall structure of your program, describing the various modules and major functions (descriptions of the function "interfaces" are expected).

d. Discuss why such a multilevel dispatching scheme would be used, comparing it with schemes used by "real" operating systems. Outline shortcomings in such a scheme, suggesting possible improvements. Include the memory and resource allocation schemes in your discussions.

The formal design document is expected to have in-depth discussions, descriptions and arguments. The design document is to be submitted separately as a physical paper document. The design document should NOT include any source code.

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. Properly commented and laid out code is much easier to interpret and it is in your interests to ensure that the person marking your project is able to understand your coding without having to perform mental gymnastics.

4. Details of submission procedures will be supplied well before the deadline.

5. The submission should contain only source code file(s), include file(s), and a `makefile`. No executable program should be included. The marker will be automatically rebuilding your program from the source code provided. If the submitted code does not compile, it cannot be marked.

6. The `makefile` should generate the binary executable file `hostd` (all lowercase please). A sample `makefile` would be as follows

```
# Joe Citizen, s1234567 - Operating Systems Project 2
# CompLab1/01 tutor: Fred Bloggs
hostd: hostd.c utility.c hostd.h
gcc hostd.c utility.c -o hostd
```

The program `hostd` is then generated by typing `make` at the command line prompt. Note: The fourth line in the above `makefile` **MUST** begin with a `tab`.

Deliverables

1. Source code file(s), include file(s), and a `makefile`.
2. The design document as outlined in Project Requirements section 1 above.

Submission of code

A `makefile` is required. All files will be copied to the same directory; therefore, *do not include any paths in your makefile*. The `makefile` should include all dependencies that build your program. If a library is included, your `makefile` should also build the library.

Do not submit any binary or object code files. All that is required is your source code and a `makefile`. Test your project by copying the source code only into an *empty* directory and then compile it with your `makefile`.

The marker will be using a shell script that copies your files to a test directory, performs a `make`, and then exercises your dispatcher with a standard set of test files. If this sequence fails due to wrong names, wrong case for names, wrong version of source code that fails to compile, nonexistence of files, etc., then the marking sequence will also stop. In this instance, the only further marks that can be awarded will be for the source code and design document.