Operating Systems - 2010INT

Assignment 2 - The GHOST Dispatcher Shell

Total Marks: 25% of Assessment **Due Date**: Friday 17th October (5 p.m.)

The Griffith Hypothetical Operating System Testbed (GHOST) 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 that 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.
2. Normal user processes are run on a three level feedback dispatcher[1]. The basic timing quantum of the dispatcher is 1 second. This is also the value for the time quantum of the feedback scheduler.

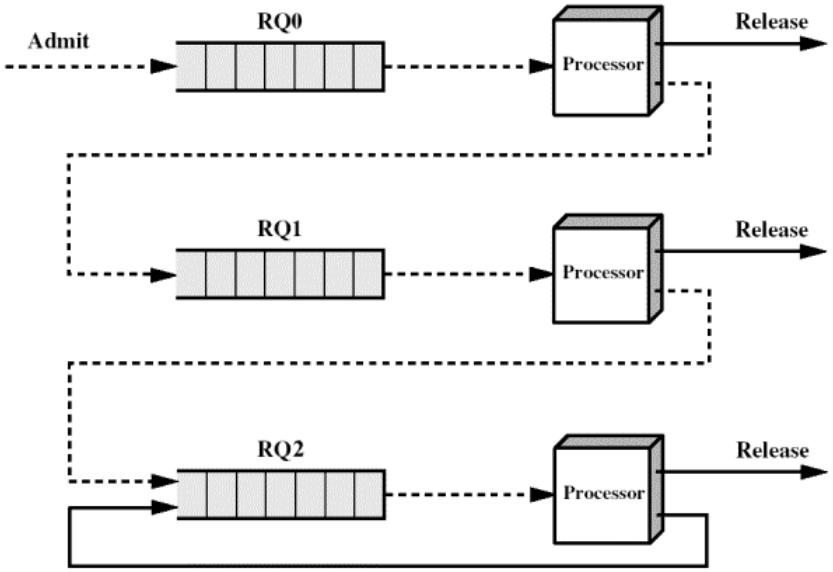

Figure 1. Three Level Feedback Dispatcher

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, pre-empting 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

[1] See "Operating Systems", William Stallings, Prentice Hall, 4th Edition, 2001, pp. 412-414

Robin dispatcher if all jobs are accepted at the lowest priority.

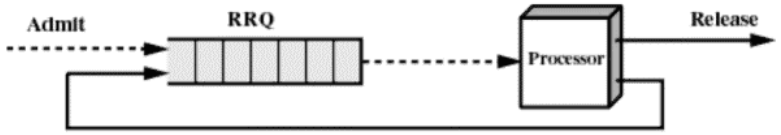

Figure 2. Round Robin Dispatcher

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 non-empty 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 below (and as discussed in tutorials):

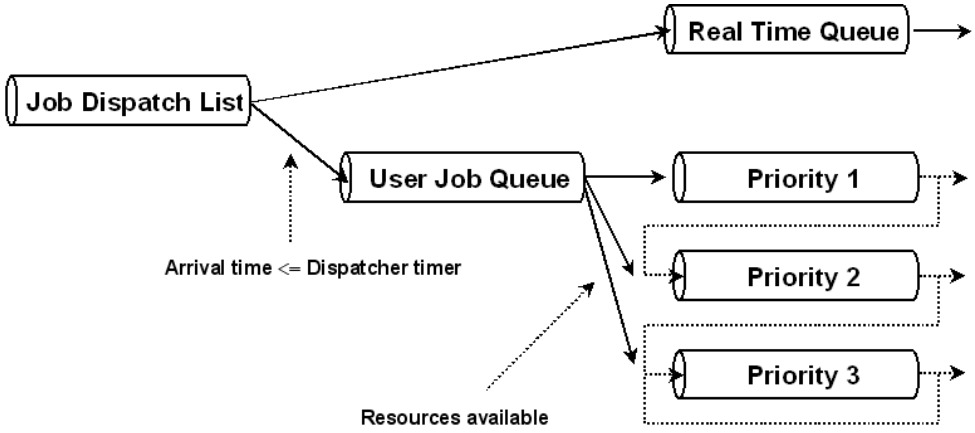

Figure 3. Dispatcher logic flow

Resource Constraints

The GHOST 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 GHOST 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

Memory allocation must be as a **contiguous** block of memory for each process that remains assigned to the process for the lifetime of that 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 amongst "active" User jobs.

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

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

Processes

Processes on GHOST are simulated by the dispatcher creating a new process for each dispatched process. This process is a generic process[2] that can be used for any priority process. It actually runs itself at very low priority, sleeping for one second periods and displaying:

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 colour 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:

1. The process is submitted to the dispatcher input queues via an initial process list which 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 re-queued on the appropriate priority queue as shown in Figure 3 above. To retain synchronisation 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 Stallings' comparison of scheduling policies[3] 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

[2] supplied as "**`process`**" on **`nias:/home2/public/2010int/assign2`**
[3] *ibid* Figure 9.5 p.405

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. i.e.

```
>ghost 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:

1st Job: Arrival at time 12, priority 0 (Real-Time), requiring 1 second of c.p.u. 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 c.p.u. time, 128 Mbytes of memory and 1 printer and 1 CD drive.

3rd Job: Arrival at time 13, priority 2 (lowest priority User job), requiring 6 seconds of c.p.u., 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 in the tutorials and are contained in the **nias:/home2/public/2010int/assign2** folder. It should be noted that these lists will form the basis of tests that will be applied to your dispatcher during marking. Operation as described in the tutorials will be expected.

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

A fully functional working example of a dispatcher is presented as the program: "**ghost**" on **nias:/home2/public/2010int/assign2**. If in any doubt as to manner of operation or format of output, you should refer to this program to observe how your dispatcher is expected to operate.

Assignment 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 queuing, 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 sytems. Outline shortcomings in such a scheme, suggesting possible improvements.

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 (**`cc`** or **`gcc`** compiler on the *nias* Sun platform).
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 assignment is able to understand your coding without having to perform mental gymnastics!
4. The assignment code must be electronically submitted before the official deadline specified above. Details of submission details will be supplied well before the deadline.
5. The electronic 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 can not be marked!
6. The **`makefile`** should generate the binary executable file **`ghost`** (all lower case please). A sample **`makefile`** would be:

```
# Ian Graham a123456 - 2010int assignment 2
ghost: ghost.c utility.c ghost.h
	gcc ghost.c utility.c -o ghost
```

The program **`ghost`** is then generated by just typing **`make`** at the command line prompt.

Note: the third line **MUST** begin with a **tab**

Note: Your name, student number, the course name, your tutorial time and tutor's name must appear in ALL submitted files and on all submitted documentation.

Deliverables

1. Source code file(s), include file(s), and a **`makefile`** to be electronically submitted as described below;
2. The design document as outlined in Assignment Requirements section 1 above must be handed in (or placed in the assignment box) by 5 p.m. on Friday 17th October.

Submission of code

The source code and **`makefile`** for your assignment should be saved to the submission area on *nias* using the **`submitOS`** utility. Please read the **`HowToSubmitOS`** file in **`/home2/public/2010int`** or on the course web site in the week before the assignment is due for precise details about how to submit.

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.

To make this clear: **do not electronically submit any binary code files**. All that is required is your source code and a **`makefile`**. Test your assignment 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, non-existence of files etc then the marking sequence will also stop. In this instance, the only marks that can be awarded will be for the source code and design document.

Marking Criteria

The assignment will be marked out of 170, which will then be scaled to provide 25% of overall assessment.

70 marks will be allocated for the submitted documentation, which will be judged on depth of discussion, readability, maintainability, completeness, and demonstration of an understanding of the problems and your solution:

- Description, discussion and justification of choice of memory allocation algorithms - 20 marks
- Description and discussion of the structures used by the dispatcher - 5 marks
- Description and justification of the program structure and individual modules - 15 marks
- Discussion of dispatching scheme, shortcomings, and possible improvements - 30 marks.

30 marks will be allocated for the source code which will be judged on presentation (incl. complying with requirements), readability, suitability & maintainability of source code and the **`makefile`**.

The balance of the marks (70) will be based on the operation and functionality of your dispatcher and how well it performs against the supplied benchmarks:

- Operation of FCFS high priority scheduler - 10 marks
- Operation of User Feedback scheduler - 10 marks
- Operation of User Feedback scheduler in Round Robin mode - 10 marks
- Mixed scheduler operation - 10 marks
- Resource allocation - 10 marks
- Memory management - 10 marks
- Combination of all - 10 marks

Part marks will be awarded for incomplete assignments or programs only providing a subset of the above requirements. Where only part of the assignment has been attempted a statement to that effect should be included in the submission.

Administration

Your attention is drawn to the following conditions contained in the Study Guide:

1. To be eligible to pass the subject, students are required to complete all forms of assessment and achieve at least 40 (forty) percent in the final item of assessment in order to achieve a grade of "Pass" or above.
2. Non-submission of a piece of assessment will incur a fail grade for the subject.

3. Students may work together in researching their assignments but final submission must reflect the work and original contribution of each individual student.
4. Any dishonest assignments will be dealt with under the rules applying in "The Process of Assessment, Grading and Dissemination of Results" and Statute 8.2 - Student Good Order as defined in the University Calendar.
5. Dishonest assignment includes:
   - deliberate copying or attempting to copy the work of other students;
   - use of or attempting to use information prohibited from use in that form of assessment;
   - submitting the work of another as your own; or
   - plagiarism (i.e. taking and using as your own the thoughts and writings of another with the intent to claim the work as your own).
6. Full and detailed acknowledgment (e.g. notation, and/or bibliography) must be provided if contributions are drawn from the literature in preparation or reports and assignments.
7. All submissions for assessment MUST be word-processed (this does not included electronically submitted source code).
8. Students must be able to produce a copy of all work submitted if so requested.
9. File directories submitted with assignments must only contain files relating to that assignment. Submissions containing irrelevant files will NOT be assessed. Files must be named as advised by the Subject Convenor. Files must have accurate date and time labels attached to them.
10. Assignments MUST be submitted by the due date and time. Extensions may be granted in exceptional circumstances by written application stating the extenuating circumstances (with medical certificate or other evidential documents) and MUST be lodged BEFORE the due date. Before an extension will be granted, a review of the work completed to date MUST be undertaken with the Subject Convenor.
11. Assignments submitted after the due date/time, without an authorised extension, will be penalised as follows:

One day (or part thereof) late	- 10% of marks are deducted
Two days late	- 20% of marks are deducted
Three days late	- 30% or marks are deducted
Four days late	- is considered a fail

12. Assignments submitted without clear student name, subject, tutorial group number and tutor identification will not be assessed.
13. Assignments received by fax or email will NOT be accepted.

Dr Ian Graham
9th September 2003