Patrick Goddard

CS 4323 - DESIGN AND IMPLEMENTATION OF OPERATING SYSTEMS I

Phase I (Due: March 23) - You are to design and implement a simulated multiprogramming operating system with the following characteristics.

SCHEDULING ALGORITHM: Multi-level feedback queue.

MEMORY ALLOCATION STRATEGY: For Phase I, the memory is viewed as 512 allocatable units. The particular details of memory management will be part of Phase II and will be discussed in the specification for Phase II.

PROCESS MANAGEMENT: Every job in the SYSTEM (the name used to refer to the driver of the simulation) is represented by a Process Control/Context Block (PCB). There are a total of at most 15 PCBs in this system. A PCB includes at least the following information:

a. job ID
b. job size
c. number of CPU bursts predicted at the start of the job
d. current burst length (clock += 1 →)
e. time the job entered the SYSTEM
f. cumulative CPU time used by the job
g. time of completion of the current I/O operation

Once in the SYSTEM, each job may be in one of the following states:

running: A job is running if it is in charge of the CPU (executing).

ready: A job is ready when it is waiting to be dispatched. Ready processes are kept in the ready queue (or list) (to be called ready_Q).

blocked: A process is blocked when it is awaiting I/O completion. Blocked processes are kept in the blocked list (blocked_Q), which is organized in ascending I/O completion times.

This program (i.e., Phase I) is to consist of a main program (referred to as SYSTEM) to drive the simulation, a subprogram mem_manager to simulate the memory management functions, a loader to load the "programs" into memory, a process scheduler to dispatch jobs and maintain the ready and blocked queues, and a number of other subroutines which will be discussed shortly.

MEMORY MANAGER: The subprogram mem_manager simulates the memory management functions of the system. This subprogram is to have four entry points.

allocate: Which is given the required allocation length and, depending on whether or not the request is honored, it will return a 1 to indicate a success and a 0 otherwise.

release: Which is given the size of a deallocation and, as a result, the total available memory will be incremented by the given size.

setup: Which is used for the initialization purposes.

stats: Which outputs the memory stats every 200 virtual time units to the system output file called SYS_LOG.

LOADER: The loader will be responsible for loading jobs into memory (i.e., for job scheduling or long-term scheduling). To do so, the loader will call mem_manager(allocate) to acquire memory. If enough memory is available, the loader will call scheduler(setup) to have a PCB created and inserted in the ready queue. The loader will try to load jobs i) as long as there is enough memory available and the number of PCBs is fewer than 15, and ii) as long as there are incoming job requests, as explained below.

You can define your own limit as to when memory is considered "full". As one can imagine, it might not be practical to keep on searching for a job whose memory request will utilize the memory 100%. Therefore, one might decide that if the amount of free memory drops to, for example, 5 units, then memory is "full" (to reduce the overhead incurred in searching for a perfect match). A call to mem_manager(allocate) will send an appropriate signal to the loader to indicate this situation.

A job size of 0 in our simulation input "data" (i.e., the simulated list of arriving jobs) will signify that there are no new job arrivals at that time. As a result, at that point in time the loader will stop loading jobs.

If the loader encounters a new arrival that cannot be loaded immediately, the request will be queued (i.e., placed on the Job_Q or the disk, which will have no specified size or management for Phase I) for later consideration. Note that every time the loader is called to load a job, it will first consider the load requests on the Job_Q (i.e., disk) before considering any new arrivals.

Once the loader has finished loading jobs, control is returned to the SYSTEM which calls the process scheduler for dispatching jobs.

SCHEDULER: The process scheduler will be responsible for dispatching jobs and maintaining ready_Q and blocked_Q. Other than normal entry points needed (discussed in items 1-3 below), the scheduler has three special entry points named setup, update, and stats, as explained below.

scheduler(setup): Which is called by the loader and is responsible for creating a PCB for a newly-loaded job and inserting that PCB in the appropriate place on the ready_Q.

scheduler(update): Which is called by the SYSTEM to update the information maintained in a PCB every time there is a need for such an update.

scheduler(stats): Which is called every time a job terminates. The collected statistics are to be output to the JOB_LOG file.

The scheduler will dispatch the job at the head of the highest priority non-empty subqueue. Once a job is running, one of three things can happen:

→Time Slice

1. If a job uses all its quantum and still needs more CPU time to complete its current CPU burst, the job loses the CPU and control is transferred to the SYSTEM which will call the scheduler. The scheduler will append the job (PCB) that lost the CPU to the appropriate subqueue within the ready_Q. → end of ready_Q

2. If a job terminates, the CPU is released and control is transferred to the SYSTEM. The SYSTEM will call scheduler(stats) to output job termination statistics to the JOB_LOG file. The SYSTEM will then call mem_manager(release)

to deallocate the job's memory.

After each job termination, the following information should be output to

loader calls memory manager → scheduler → dispatcher → PCB

the JOB_LOG file:

a. job ID
b. time job entered the system
c. time job is leaving the system
d. execution time (time actually spent in control of the CPU + 10 time units per I/O request)
e. number of CPU shots used by the job (how many time slices)

3. If a job requests I/O, it releases the CPU and control is transferred to the SYSTEM. The SYSTEM will initiate I/O (in this simulation no action actually takes place). The SYSTEM will then call the scheduler to place the PCB of the job in the I/O blocked queue (blocked_Q). The job will remain on the blocked list for 10 virtual time units (time for I/O completion).

Whenever a job relinquishes the CPU and appropriate actions have been carried out (as described above), the scheduler will check to see if any job in the blocked_Q has completed its I/O and is ready to run again. If such a job is found, it is taken from the blocked_Q and placed in the ready_Q. Subsequently, the scheduler will look for another job to run. → Join Ready_Q

WHEN TO CALL THE LOADER: Other than the system start-up, at which time as many jobs as can be accommodated are loaded into memory, the loader will be called in the following situations:

1. When a job is completed and leaves the system (i.e., at job terminations). → all IDs 0

2. If the scheduler tries to find a job in the ready_Q unsuccessfully (because there are no "ready" processes at the time), or the waiting time (the virtual I/O time) for the first job in the blocked list,i.e., the blocked_Q, is not over yet AND there is room in memory, then a new job (if any) will be initiated. Once the loader is called, it will attempt to load as many jobs as it can fit into the available memory until memory is "full".

Only if no other job can be found to run, will the CPU have to sit idle waiting for the completion of the I/O of a job. In such a situation, increment the CLOCK by the duration of time the CPU has to sit idle. This is the only occasion when the CLOCK is incremented other than during job "execution" in the CPU.

Your main program, SYSTEM, drives the simulation of "memory needs" as follows:

(1) To simulate memory demands, the file /home/opsys/OS-I/SP17/17s-ph1-data should be opened and read. This is the file of "arriving jobs" in terms of their memory and CPU requests. A "job" is composed of: Job ID as the first parameter, the number of allocatable memory units required as the second parameter, and an unknown number of CPU bursts as predicted by each respective job. It should be clear that between successive CPU bursts there is I/O activity.

(2) If an attempt to acquire memory cannot be honored, the request must be preserved for later service.

(3) Every 200 virtual time units, statistics concerning the utilization and status of the system is collected. These statistics include:

(a) number of allocated units of memory
(b) number of free units of memory
(c) number of jobs in the Job_Q

(d) number of jobs in the blocked_Q
(e) number of jobs in the ready_Q
(f) number of jobs delivered

The gathered data is to be written (neatly formatted) to the SYS_LOG file.

(4) The simulation is to last for as long as there are processes on the disk (i.e., the job_Q).

(5) the following is a high-level description of the main program. This is only a suggested outline. For brevity, a number of relevant and important details, such as the limit of 15 PCBs and a job size of 0, have not been included.

```
while not eof do
  as long as there are jobs on disk AND there is room in memory
        load jobs
  call scheduler
  call CPU
  if used up quantum, then place in ready queue
  else if I/O encountered, then place in blocked queue
  else if finished last CPU burst, then output results and release memory
  if virtual clock = 200, 400, ..., then collect statistics
endwhile
```

Once the simulation is completed, output the SYSTEM clock at termination in the SYS_LOG file.

PROCESS SCHEDULING: The process scheduling algorithm used for this simulation is a system of multi-level feedback queues. Upon exit from the CPU (due to job completion, I/O request, or quantum timeout), you will do the following.

1. Move all eligible jobs from the blocked_Q to the appropriate subqueues.
2. Load all jobs from the job_Q for which there is room in your system.
3. Pick the job from the front of the lowest number non-empty subqueue to run next.

The ready queue will be divided into four subqueues with the following residency rules.

	subqueue1	subqueue2	subqueue3	subqueue4
# of turns	3	5	6	---
quantum	20	30	50	80

The residency rules further specify that if a process places an I/O request before its time quantum expires, the process remains in the same subqueue and the count of turns in that particular subqueue will be reset to zero. Otherwise, if the I/O request is made at the end of a time quantum, the count of turns will be maintained and incremented in the next CPU burst. A process will remain in the last subqueue (subqueue4) until it terminates, or until it places an I/O request. When an I/O request is placed, the process will be moved at once to the top subqueue (subqueue1).

NOTES

As can be seen from the above specification, you will need to add a few new data items to the PCB of jobs (e.g., current subqueue, number of turns spent in that subqueue, etc.).

The LOADER subsystem will read/load as specified above. There will be no preprocessing/parsing/checking of the job file or "data" file, since it is "in the future".

You must keep up with class discussions concerning the project specification. As a result of the discussions in class, there may be some changes in the project specification as outlined in this document.

The driver of the first phase of the project is to be named SYSTEM if possible (this depends on the language of implementation). Other descriptive/mnemonic names must also be used as specified in this document.

Design Note: Conceptually, your simulation is a virtual machine. Conventional architecture and operating system component functionalities and restrictions should be adhered to, i.e., your code should be properly modularized not by size but by function. There is ample opportunity for tailoring/customizing or personalizing/individualizing your simulation during implementation. This refers to the relative internal implementation latitude vs. strict external functional behavior. Nonetheless, any significant departure from the specification must be approved by the instructor.

CSX is the Computer Science Department's main instructional computer. You must develop your program on CSX, or upload your program for submission. In the latter case, you should be aware of and handle potential language and compiler incompatibility problems between your development platform and the submission platform. In short, your project (simulation program) must compile and run on CSX.

When reporting ~~averages and~~ utilization numbers, if applicable, pay special attention to the number of meaningful significant digits.

All deliverables are to be hard copy (i.e., paper copy) submissions. No soft copy submissions such as email, CDs, or flash drives will be accepted. No handin submissions are required either.

↗ Conceptualization (what a method does)

DOCUMENTATION GUIDELINES: Your simulation program must include external and internal documentation. External documentation (for conceptualization) appears in the form of comments as header blocks and is an explanation of the functionality of the respective program/subprogram/function/method/module, a description of the global variables, and a general high-level discussion of the implementation approach. Internal documentation (for readability) is the documentation that is mixed with the code and is used to clarify potentially obscure segments of code such as case statements, loops, or conditions. Use meaningful names and blank lines to enhance the readability and understandability of your code. DO NOT pollute the user's environment with unnecessary echos and prompts. You are to follow the last section of this document titled "DOCUMENTATION GUIDELINES FOR ALL PROGRAMMING ASSIGNMENTS".

A handout outlining the "procedure to obtain a final copy of your program/output" will be distributed in the near future.

54 units of time
10 units of time for I/O

==========
==========

ID ...		[	I/O		I/O		I/O		I/O		I/O		CPU bursts]	1 Job is a line
36	32	[54		84		54		44		15		15		
37	30	67		56		15		34		56		56	44	
38	10	16		92		56		14		56		4	56	
39	14	14		44		15		54		44		14	15	
40	32	4		54		4		54		15		54	189	90
41	31	14		12		74		44		15		17	1	

```
42 105    16  15     4    54    49    44
 0   0 no job
 0   0 no job
 0   0 no job
43 185     4  14    15    90    15   150
44  10    14  16    15    54     4
45  25    57   4    44    15     5    85
46  13    44  25     4    15    56    44    15    54
47  32    54  79   128    16    54     4    44
48  32    56  97    43    14    55    16    59
49  14    14  14    45    12    14    56    15
 0   0 no job
 0   0 no job
50  30    44  56    15    12
51  10    15  15    56    15    16    15    16
52 112    15 180    54    14    44
53  31    56  14    14    16     4    87    16
 ...
================================================
================================================
```

GENERAL GUIDELINES FOR ALL PROGRAMMING ASSIGNMENTS

1. External Documentation - At the beginning of the main routine, there should be documentation containing items a through g below. At the beginning of all other routines (i.e., subprograms, functions, methods, modules, classes, etc.), there should be documentation containing items f and g only.

a. Your name.

b. Course number.

c. Assignment title or number.

d. Date

e. A brief description of the global variables, if any.

f. A brief description of what the routine does, i.e., a short explanation of the functionality of the program/subprogram/function/method/module/class.

g. If applicable, depending how thoroughly and accurately the implementation reflects the specification, a critique of the program/subprogram/function/method/module/class indicating the ways in which it does not meet the programming assignment's specification.

2. Internal Documentation - This is the documentation that is "mixed" or interspersed with the code to clarify the potentially obscure segments of the code, e.g., case/switch statements, loops, conditional statements, and procedure calls/returns.

a. All major variables should be commented in the declarations except those that are patently self-explanatory.

b. The code should also be commented, not too much though. As a general rule, when in doubt, use a comment.

c. Use meaningful names and blank lines to enhance the understandability of your code. Blank lines help isolate code segments as

spatial localities.

3. Program Layout and Data Structures -

a. The code should be well-structured, with indentation showing the general program logic and flow. GOTOs (for implementation languages that actually include GOTO among their instructions) and other unconditional transfers of control should be used only for a fairly good reason, otherwise they should be avoided.

b. The program should be modular. Program modules generally should not be larger than the size of a typical monitor screen in terms of the number of lines.

c. Choose your data structures carefully, e.g., linked lists are generally inefficient and should be avoided, use them only if their use is justifiable.

4. Input and Output -

a. Use prompts and echos only when they contribute to the overall user-friendliness of your program. Avoid unnecessary prompts and echos.

b. Your output should be well-formatted, commented, and understandable. To enhance comprehensibility, arrange your output in rows, columns, blocks, etc. with appropriate explanatory headers, within the framework delineated in the programming assignment's specification.

c. The submitted program and the output files should not include any debugging code or debugging comments.

Notes:

(1) Consider the probability that the system (i.e., CSX) could be down, or have system hardware/software problems, a few days before the due date.

(2) Going through the "design/desk-check/redesign cycle followed by coding" is more effective and efficient than going through the "quick design followed by code/output-check/re-code cycle".

(3) It should go without mentioning that a clean compilation with no warnings is called for (applicable to compiled languages and not interpreted ones).

(4) Keep a copy of the deliverables that you submit for your own reference.

==========