Paged Virtual Memory Simulation

The objective of this assignment is to add a simulation of a demand-paged virtual memory system to the scheduling simulation implemented in the previous project.

Memory Manager Details

The memory manager has fixed, specified amounts of physical and virtual memory to manage. Each addressable location in physical and virtual memory is a single byte, capable of storing one character.

The memory manager will maintain index structures for physical and virtual memory so that it can efficiently determine in which frame a given virtual page resides (if any) and which virtual page a given frame contains (if any). The index structures may also store additional information about the state of the pages or frames.

When a process is created, it will be allocated precisely enough contiguous[1] pages of virtual memory to store the process image. If there is not a contiguous free block of virtual memory large enough to hold the process, the memory manager will not allocate any memory for the process. In that case, the process should be placed into a hold state until it is possible to allocate virtual memory to it. If there is more than one contiguous free block that is large enough to hold the process, the memory manager will use the first-fit policy to allocate space for the process in virtual space.

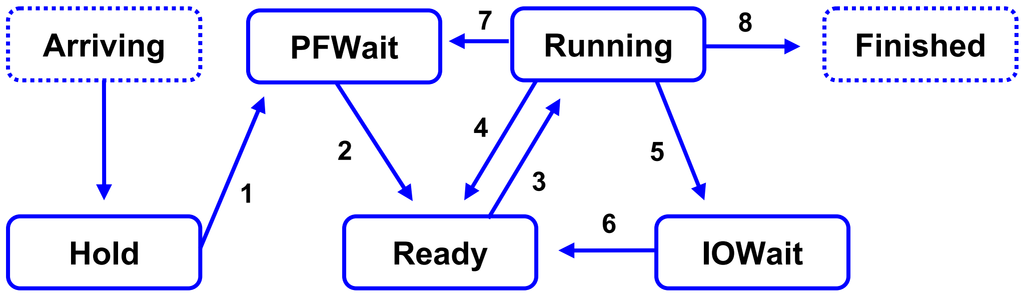
The memory manager uses a static frame allocation strategy[2]. If a process is allocated v pages of virtual memory, that process is assigned a frame allocation limit of $\lceil \alpha v \rceil$ (frames of physical memory), where α is a constant specified in the simulation configuration file. (Obviously, α will always be between 0 and 1.)

The memory manager uses demand paging. Before a process is first moved to the ready state, it is allocated one frame of physical memory to hold the page containing its entry point. If there are no free frames, the process must remain in a wait[3] state until it is possible to do this. We will assume the entry point is always on page 0 of the process. As the process executes, additional process pages will be loaded as needed. If the process has not reached its frame allocation limit, the free frame with the lowest frame number will be allocated to the process (if there are free frames available). If the process has reached its allocation limit, or if there are no free frames, then the memory manager will replace one of the process's resident pages.

The memory manager uses the pure local LRU page replacement policy. How you implement this is entirely up to you, but note that you are not using an approximation of LRU.

When a page in physical memory is replaced, its contents must be written back to the appropriate location in virtual memory if (and only if) the page was modified while it was in physical memory. Reading a page from virtual memory to a frame and writing a page from a frame back to virtual memory each take two units of system time.

The addition of virtual memory to the simulation requires modifying the process state transition diagram:



State transition rules

In reference to the state diagram above:

1. A job makes the transition from Hold to PFWait when its required virtual memory is allocated.
2. A job makes the transition from PFWait to Ready when the page it has requested has been loaded.
3. A job makes the transition from Ready to Running if it is at the head of the ready list and a context switch occurs.
4. A job makes the transition from Running to Ready if its time quantum expires.
5. A job makes the transition from Running to Blocked if it issues an I/O request.
6. A job makes the transition from Blocked to Ready if the I/O request it is waiting for is completed.
7. A job makes the transition from Running to PFWait if it makes an access request for an address on a page that is not currently loaded.
8. A job makes the transition from Running to Finished if it has accumulated the necessary CPU time or if it is killed because it made an illegal memory access request.

Priority rules for event ordering

It is possible that more than one event may fall at the same time; for example, a new job may arrive at the same time that a running job terminates, freeing memory that could be used to move another job from IOWait to PFWait. Many other scenarios for simultaneous events are possible. If the order in which the simulation handles updating the states of entities within the system is not fully specified then there may be multiple end results that are consistent with the system definition. We would prefer to reduce the number of acceptable variations. So…

When the system time is updated, the simulator will update the state of the system in the following order:

1. notify all relevant processes that the clock has ticked
2. add new arrivals to the hold state, if necessary
3. check the state of the process in the run state, if any
   a. check for process termination
   b. check for process I/O event
   c. check for process memory access and possible page fault
   d. check for quantum expiration
4. check for waiting processes whose page faults have been serviced
5. check for blocked processes whose I/O requests have been satisfied
6. check for held processes whose memory requests can now be satisfied
7. select new process to run, if necessary

Performing these steps in a different order will produce results that may be logically acceptable in a different scheduling system, but which will not satisfy the requirements of this assignment.

System Operation

Your system should accept two command-line arguments, which are the names of an input file and an output file:

```
simvm <input file name> <output file name>
```

The input file contains all specifications concerning arrivals, I/O requests, memory activities, and display events.

Each line in the input file conforms to one of the specifications given below:

Comment

Any line beginning with a `'#'` character is a comment.

System initialization

The first non-comment line in the input file will always be of the form:

```
sysinit <SZ> <PP> <VP> <FF> <TQ>
```

Key:

`SZ`	the size of a memory page, in bytes
`PP`	the number of pages of physical memory
`VP`	the number of pages of virtual memory
`FF`	the frame limit factor (α)
`TQ`	the time quantum

For this simulation, addresses will always be expressed in base 10. Therefore, the page size will be a power of 10. There is no stated limit on the total size of virtual or physical memory, but it will be restricted in testing to a value that is reasonable given the typical memory configuration of contemporary systems.

As before, the system should be initialized to include the specified number of pages of primary and virtual memory. Each page will be of the specified size. A total of 16 pages are reserved for the kernel memory. This implicitly means that the primary memory should always have more than 16 pages. The `<time quantum>` refers to the time-quantum used for the round robin algorithm that is used for the short-term scheduling. There will always be exactly one system initialization specification in the script file.

Before the scheduler starts managing user processes, the OS must be loaded. As stated above, the OS will occupy 16 pages of memory. The OS image will be loaded into the first 16 pages of virtual memory, and into the first 16 frames of physical memory. Unlike other processes, the OS is not subject to a frame allocation limit (other than its size), and the OS will never incur page faults. The OS is not specified by an arrival command (below); for uniformity, its PID should be 9999.

Job arrival

The arrival of a new job in the system is signified by a command of the form:

```
arrival <jobnum> <system time> <cpu time> <memory>
```

Here, `<jobnum>` is the unique identifier for this job, `<system time>` is the arrival time of the job, `<cpu time>` is the processing/service time of the job and `<memory>` is the number of memory pages needed by this job.

Job I/O request

A job, while it is in the running state, can perform an input/output operation. This request is in the form:

```
IOrequest <jobnum> <job time> <io time>
```

Here, `<jobnum>` is the unique identifier of the job doing the i/o request, `<job time>` is the time the request is made, and `<io time>` is the time for the i/o operation. Note that the job time is relative to the accumulated CPU time of the job in question.

Job memory access

A job, while it is in the running state, can attempt to access the contents of a specific virtual address. This access will be in the following form:

```
MemAccess <jobnum> <job time> <address>
```

Here, `<jobnum>` is the unique identifier of the job doing the i/o request, `<job time>` is the time the request is made, and `<address>` is the relative address (from the perspective of the process) to be accessed. Note that the job time is relative to the accumulated CPU time of the job in question.

The given relative address should be resolved to a virtual address (virtual page number and offset), which should be logged. If the process can legally access the virtual address, the corresponding physical address (frame number and offset) should be produced and logged. If not, the process should be immediately terminated.

System time advance

The system can be instructed to advance the simulation to a specified time by a command of the form:

```
simto <system time>
```

The simulation will perform all actions that should occur up to the specified system time. No additional commands will be read from the input script until this is carried out.

System state display

The user can request a snapshot of the system queues and the status of each job in the system:

```
show
```

The contents of each state and the statistics associated with each job, including the remaining time, should be printed out. The information should be current as of the simulated system time at which the **show** command was read from the script. There are several (additional) variations of this command:

```
show [ hold | IOwait | PFwait | ready | run | terminated ]
```

Display the contents of the specified state, as described for the general **show** command.

```
show <jobnum>
```

Display the state of the specified job. This includes CPU time data, memory allocation data including page/frame map information, the virtual base address of the process, and the current state of the job.

```
show pmt
```

Display a table showing the state of virtual memory allocation. For allocated pages this should include the PID of the process that owns the page, whether the page is resident in physical memory, and the corresponding frame number if the page is resident. It is not necessary, or desirable, to show information for unallocated virtual pages. Data should be listed in ascending order by virtual page number.

```
show fmt
```

Display a table showing the state of physical memory allocation. For allocated frames, this should include the corresponding virtual page number and the PID of the owner. It is not necessary, or desirable, to show information for free page frames. Data should be listed in ascending order by frame number.

Each is similar to the original but results in a display only of the specified state or of the specified job.

System shutdown

Stop processing scheduling decisions at the specified system time. Display the system queues and the status of each job in the system, handle all necessary memory deallocation and other cleanup, and terminate the simulation.

```
shutdown <system time>
```

where `<system time>` refers to the absolute system time at which the system shutdown is done. A full display of the final system status should be carried out before the simulation exits.

There is no guarantee that a shutdown command will be given. If not, the system should detect when the script has run out and terminate the simulation automatically. After system shutdown, the arrival time, finish time, wait time, turnaround time, weighted turnaround time and the page fault rate for each normally terminated[5] job should be printed out in tabular format. For the purposes of this assignment, use the following definitions:

wait time	Time a process spends waiting in the ready state before its first transition to the running state
turnaround time	Time between the moment a process first enters the ready state and the moment the process exits running state for the last time (completed)
weighted turnaround time	Ratio of turnaround time to the total CPU time needed.
page fault	Loading a process page from virtual memory into a frame of physical memory.[4]
page fault rate	The number of page faults divided by the number of explicit memory accesses.

These definitions imply that you do not account for the time that a job might have spent in the hold queue. Additionally, the overall average wait time, average turnaround time, average weighted turnaround time, and average page fault rate should be printed out.

Job numbers and all time values will be given as positive integers. There will never be two specified arriving jobs that are given the same number. Any command in the script file that specifies a system time that precedes the current simulation time when the command is read should be ignored.

The script files are guaranteed to be syntactically correct. All commands that include a system time will be provided in ascending order. Out-of-order commands should be flagged with an error message and then ignored.

Note: All output should be sent to the output file. Assume that the tokens in the commands are separated by one or more spaces and/or tab characters.

Assumptions and error handling

All the assumptions stated for the previous project remain in effect unless explicitly changed or retracted in this specification.

Some changes:

- Jobs that are waiting for a page fault to be serviced should not be placed in the same state as jobs that are waiting for an explicit I/O request to be serviced. There should be a dedicated memory wait state.
- If two jobs are removed from the hold state or the blocked state at the same system time, they should be removed in the order in which they were placed into that state.
- **`show`** commands always result in a display of information that is current as of the system time at which the command was read from the script.

Any syntactic or semantic errors in the input script file (described below) should be handled gracefully, including an error message and continued simulation if that is possible. Note that the point of this assignment is not to test every such scenario, and you are not expected to be particularly creative in designing such error handling.

Test Data and the Log File

Sample input and possible output data will be made available on the class site at least one week before the due date. You should create your own input files for testing your program. It is permissible to exchange test data among yourselves.

You must echo any comments that occur in the script file to the log file, immediately before you echo the corresponding command. (This makes it much easier to browse the log output if useful comments are placed in the script.)

Testing may also be easier if your implementation produces extra logged output, beyond the minimal expectations stated above. You may add as much output as you like, as long as the end result is a log file that is easily scanned by someone who wants to verify its correctness.

In particular, the output produced in response to any particular command, or from reaching a specific system time, should be clearly delimited and labeled.

System Design and Implementation

You are expected to produce and implement an object-centered design for the simulator. This implies your implementation must be in C++, not in C. More to the point, this implies you should produce a sensible collection of classes and that each class should be assigned sensible responsibilities. It is not necessary to use inheritance, although there is no objection to doing so. Association and aggregation relationships will probably abound in a good design.

Certainly all data structures should be templates. You are free to use any STL containers you like.

You should document your implementation internally, according to the requirements given on the Programming Standards page of the course website.

Submitting Your Program

You will submit this assignment to the Curator System (read the *Student Guide*), where it will be archived for grading at a demo with a TA.

For this assignment, you must submit a gzip'd tar file containing all the source code files for your implementation (i.e., header files and cpp files). Submit only the header and cpp files. Submit nothing else.

In order to correct submission errors and late-breaking implementation errors, you will be allowed up to five submissions for this assignment. You may choose which one will be evaluated at your demo.

The Student Guide and link to the submission client can be found at: `http://www.cs.vt.edu/curator/`

Evaluation

Shortly before the due date for the project, we will announce which TA will be grading your project and post signup sheets inside the McB 124 lab. You will schedule a demo with your assigned TA. At the demo, you will perform a build, and run your program on the demo test data, which we will provide to the TAs. The TA will evaluate the correctness of your results. In addition, the TA will evaluate your project for good internal documentation and software engineering practice.

Remember that your implementation will be tested in the McB 124 lab environment. If you use a different development platform, it is entirely your responsibility to make sure your implementation works correctly in the lab.

Note that the evaluation of your project will depend substantially on the quality of your code and documentation. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement provided on the Submitting Assignments page of the course website.

Notes:

1 This assumption allows you to simplify your representation of the mapping of the process image into virtual memory. It is not logically necessary.

2 That is, the maximum number of frames a process may own is fixed.

3 This is logically different from the hold state..

4 Loading the initial page of a process does count as a page fault.

5 Abnormally terminated processes should be included in the list of terminated processes, but they should not be used to compute any of the summary statistics.