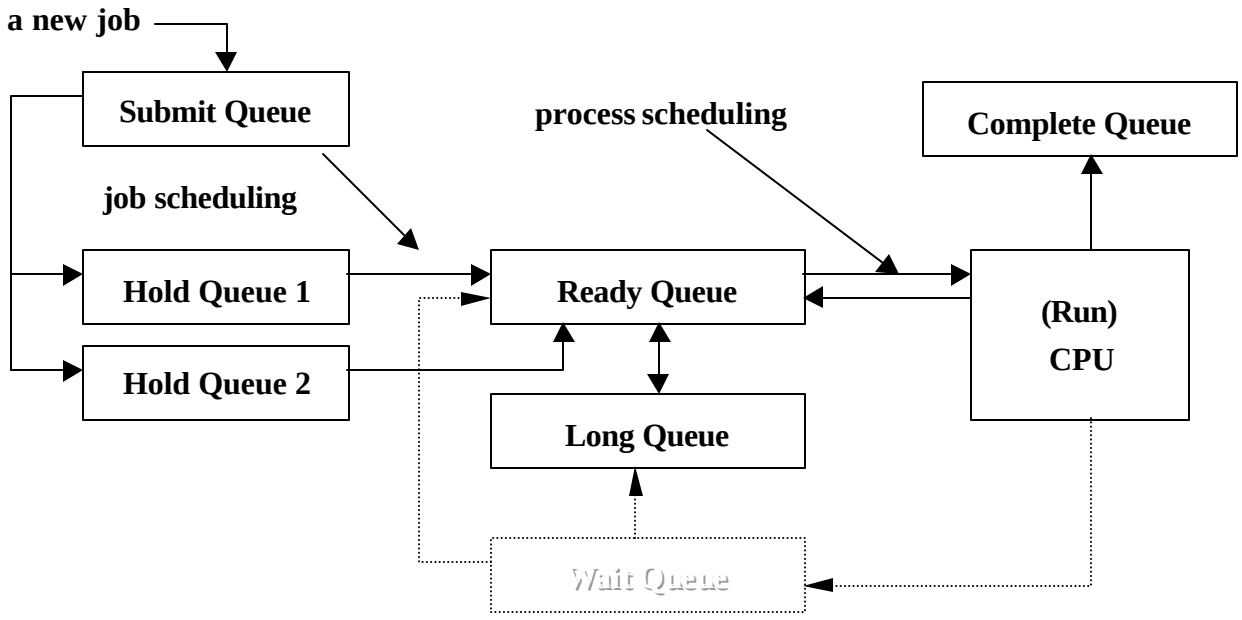
CS 3204 – Operating Systems
Programming Project #1 – Job / CPU Scheduling
Dr. Sallie Henry – Spring 2000
Due Date: 17 February 2000 at 8:00am

Design and implement a program that simulates some of the job scheduling and CPU scheduling of an operating system. Your simulator must conform to the criteria established in these specifications. A **detailed design** is due on February 4, 2000 in class.

The input stream to the program describes a set of arriving jobs and their actions. The following diagram describes Job and process transitions.

A graphic view of the simulator



When a job arrives, one of three things may happen:

1. If there is not enough *total* main memory or total number of devices in the system for the job, the job is rejected never gets to one of the **Hold Queue**s.
2. If there is not enough *available* main memory or available devices for the job, the job is put in one of the **Hold Queues**, based on its priority, to wait for enough available main memory **(Preallocation)**.
3. If there is enough main memory and devices for the job, then a process is created for the job, the required main memory and devices are allocated to the process, and the process is put in the **Ready Queue**.

When a job terminates, the job releases any main memory. The release of main memory may cause one or more jobs to leave one of the **Hold Queues** and move to the **Ready Queue**.

Assume that all the two **Hold Queues** are based on priority. There are two external priorities: 1, and 2 with 1 being the highest priority. **Priority is only used for the job scheduler**:

- Job scheduling for **Hold Queue 1 is Shortest Job First (SJF).**
- Job scheduling for **Hold Queues 2** is **First In First Out (FIFO).**

Process scheduling will be **Limited Round Robin**. Once a job has exceeded the **threshold** of CPU time, it is considered a long job and may not run until the **Ready Queue** is empty. At that time the **Long Queue** becomes the **Ready Queue** and **Round Robin (with a quantum)** process scheduling is used.

Input specification

The input to your program will be text. Each line in the file will contain one of the commands listed below. Each command consists of a letter in column one followed by a set of parameters. Each text file contains multiple type "C" (system configuration) commands. (A sample for the format of input file will be given out on the web.) All input will be *syntactically and symantically* correct, but you should detect and report other types of errors to the GTA. There will always be *exactly* one blank after each number in the input file.

The input will be syntactically and semantically correct, however, due to possible human error, you are requested to please check for the following errors, which may occur in the input data:

- The time on a command is either less than or equal to the time on an earlier command.

1. System Configuration:

C 9 M=45 L=6 S=12 Q=1

The example above states that the system to be simulated starts at time 9, and that the system has a main memory consisting of 45; a time excess (threshold) of 6 determines long jobs; 12 serial devices; and a time quantum or time slice of 1.

2. A job arrival:

A 10 J=1 M=5 S=4 R=5 P=1

The example above states that job number 1 arrives at time 10, requires 5 units of main memory, holds no more than 4 devices at any point during execution, and runs for 5.

3. **A display of the current system status in *Readable* format (with headings and properly aligned):**

D 11

The example above states that at time 11 an external event is generated and the following should be printed:

1. A list of each job that has entered the system; for each job, print the state of the job (e.g. running on the **CP**U, in the **Hold Queue**, **Long Queue** or finished at time 11), the remaining service time for unfinished jobs and the turnaround time and weighted turnaround time for finished jobs.

2. The contents of each queue.

3. The system turnaround time and system weighted turnaround only at the last display. Assume that the input file has a "D ∞" command at the end, so that you dump the final state of the system. ∞ is to be taken as a large number (i.e. the very last event of the input file and subsequently of the system. Infinity will be denoted by 999999.

Helpful Hints

Let *i* denote the time on the next input command, if there is still unread input: otherwise *i* is infinity. Let *e* denote the time of the next internal event, which will be the time at which the currently running job terminates or experiences a time quantum expiration. The "inner loop" of your program should calculate the time of the next event, which is the *minimum* of the *i* and e. If *i* = e, then process the *internal event before the external event* (input file event). Notice that if this is not strictly followed, your results will not match the expected output to grade your project!

Your simulation must have a variable to denote the "current time". This variable must always be advanced to the time of the next event by a single assignment. (The variable cannot be "stepped" by a fixed increment until it reached the time of the next event.)

You may also wish to use a .h file; put your #defines, global variables, and global type and structure definitions in to improve the code readability.

Notice that unlike the examples in class, we will be giving you **all integer values** to simplify the problem.

You must turn in the following

- A copy of a printout of your source code, containing a comment with your name, compiler name and version, hardware configuration, operating system and version. NOTICE: You must use standard C++ / Java. Do not use machine / compiler dependent constructs. Your code must compile and execute in the McBryde 116/118 lab.

- A copy of the program output on each test data file.

- A floppy disk, labeled with your name, and instructions for running your program, containing
  - source code
  - executable file
  - execution output for all test cases

A set of input data files will be made available for distribution on the 12th of February. Attend class for further details, answers to questions, and hints on this assignment. **You will be responsible for all that your instructor says about this assignment in class.**

Implementation Hints

1. Implement the **Hold Queues** as sorted linked lists. On what key are they sorted?
2. Implement the process table as an array of size 100; the D command dumps the process table. (Do not confuse this table with the PCB's.)

3. You will be graded in part on the maintainability of your code. Therefore use #define to avoid embedding numeric constants in the code.

4. The end of a time slice is an internal event. You may assume a context switch will take zero time.

Other Hints:

If there is a completion of a job, check the 2 **Hold Queues** before the **Long Queue**.

When a job completes, it releases main memory and implicitly releases the devices. Now, check the 2 **Hold Queues** and then the **Long Queue**.

Priority is only meaningful in the 2 **Hold Queues,** and **NOT** in the **Long Queue**.

Priority is job scheduling not process scheduling.

The only constraints to move from one of the **Hold Queues** to the **Ready Queue** are main memory and devices.

If more resources are needed than the system contains **(NOT** available, **Actually** contains) then **do not** even consider the jobs. (Kick it out and do not "count" the job for t or w).

If jobs have same run-time and same priority, use FIFO scheduling (FIFO within SJF).

Handle **all** internal events before external.

Do not use an array of size 100 for jobs.

A display event is external.

Information hiding is important.

Any new job entering the **Ready Queue** sends a **Long Queue** job back to the **Long Queue** (i.e. the **Ready Queue** must be empty for a job on the **Long Queue** to get on the **CP**U.)

When a process is in the **Long Queu**e, it does **not** give up its resources.

There will never be two external events at the same time.

Do **NOT** use a **MEGANOD**E!!! Use information hiding unless you plan on losing points.

Hint: The MEGANODE issue can be solved through inheritance. Think about it.

Absolutely under no circumstances do you want to read the entire input file in the beginning of the program (i.e. pre-process the input file).