Userland Stride Scheduler

In this project you will build a stride-based simulated scheduler in userland. Your scheduler will read a data file containing information simulating timer interrupts, job arrivals, job terminations, etc. Your output will be graded against known good results. Any deviation (even by a single space) may be counted as wrong, so follow the specification completely.

You will be able to run tests against my grading script to debug any textual differences.

Stride Scheduler

Refer to chapter 9 of OSTEP.

Command Line Options

Your program will take just one argument - the name of the data file to execute. It is an error if the file is not given or cannot be opened for reading.

Platform

This project can be coded natively on the Mac.

This project must be coded under WSL on Windows or on a Linux machine.

Input

Your program will take a data file as its command line argument. It contains comma separated lines with the following syntax:

opcode	argument 1	argument 2	meaning
newjob	NAME	PRIORITY	A new job with specified PRIORITY and NAME has arrived
finish			The currently running job has terminated - it is an error if the system is idle

opcode	argument 1	argument 2	meaning
interrupt			A timer interrupt has occurred - the currently running job's quantum is over
block			The currently running job has become blocked
unblock	NAME		The named job becomes unblocked - it is an error if it was not blocked
runnable			Print information about the jobs in the runnable queue
running			Print information about the currently running job
blocked			Print information about the jobs on the blocked queue

My test input is guaranteed to have correct syntax so you do not have to check for errors.

Operation

newjob

A new job is entered into the system. Its name and priority are given. Assume all job names are unique. A new job's arrival does not cause a rescheduling unless the system was idle.

```
New job: C added with priority: 200
```

finish

The currently running job has completed and should be removed from the system.

```
Job: B completed.
```

If the system is idle, it is an error.

```
Error. System is idle.
```

interrupt

The currently running task has completed its quantum. Adjust your bookkeeping. The scheduler needs to run again.

```
Job: C scheduled.
```

It is an error if `interrupt` is received when the system is idle.

```
Error. System is idle.
```

block

The currently running task has become blocked. Perhaps it is asking for an I/O. The task will remain blocked until it is `unblock`ed.

```
Job: A blocked.
```

It is an error if the system is idle.

```
Error. System is idle.
```

unblock

The named job has become unblocked.

```
Job: A has unblocked. Pass set to: 5000
```

It is an error if the named job was not blocked.

```
Error. Job: C not blocked.
```

Unblocked jobs return to the runnables. The scheduler is not run unless the system was idle.

runnable

The runnables, if any, are listed. Example:

```
Runnable:
NAME    STRIDE  PASS  PRI
```

```
C       500     1000  200
A       500     1500  200
```

These must be listed in the order they would be scheduled.

running

The running task is described (if system is not idle). Example:

```
Running:
NAME    STRIDE  PASS  PRI
B       1000    1000  100
```

blocked

The blocked tasks are listed, if any. Example:

```
Blocked:
NAME    STRIDE  PASS  PRI
A       500     2000  200
```

Picking the next task to schedule

The task with the lowest pass value is chosen to run next. In the event of a tie, sort alphabetically. If `A` and `B` have the same pass value, `A` will run.

```
Job: C scheduled.
```

Running the tests

The results of the stride schedule are deterministic. Given the same input, you will get the same output. Therefore, automatic testing is possible.

I provide a `bash` script for testing purposes. The script takes two command line options.

option	required?	argument	purpose
a	no	progname	specifies name of your executable - defaults to ./a.out
i	yes	testroot	specifies the root name of the test to run

The folder containing the test script is meant to contain the folder `tests` where the test data is actually stored. So, if you specify `-i test1`, the actual data file will resolve to `tests/test1.input.txt`.

Here is sample output from a test:

```
./expected_output_test.bash -i test1
Test input file:       tests/test1.input.txt
Expected output file:  tests/test1.expected_output.txt
Expected output (must match letter for letter):
Runnable:
None
Your output:
Runnable:
None
Differences:
PASSED
Test finished
hyde pk_stride_scheduler $>
```

Here is output from a more sophisticated test:

```
./expected_output_test.bash -i test5
Test input file:       tests/test5.input.txt
Expected output file:  tests/test5.expected_output.txt
Expected output (must match letter for letter):
New job: A added with priority: 100
Job: A scheduled.
New job: B added with priority: 100
Job: B scheduled.
Job: A scheduled.
Job: B scheduled.
Job: A scheduled.
Job: B scheduled.
Job: A scheduled.
Job: B scheduled.
Your output:
New job: A added with priority: 100
Job: A scheduled.
New job: B added with priority: 100
Job: B scheduled.
Job: A scheduled.
Job: B scheduled.
Job: A scheduled.
Job: B scheduled.
Job: A scheduled.
Job: B scheduled.
Differences:
PASSED
Test finished
hyde pk_stride_scheduler $>
```

You will be given only *some* of the tests I will use for grading. Some, I hold in reserve to test corner cases, etc. If you pass all the tests you are given, you have a reasonable change of a high score but this is not a certainty.

Feel free to read the source code of the test script to see an example of `bash` scripting.