ASSIGNMENT 5: Introduction to Semaphore and IPC

CS 69001: Computing Lab 1

August 19, 2016

Introduction:

In this assignment you will be introduced with the concepts of shared memory, system V IPC and multiple processes.

Objective:

- To get acquainted with communication between processes and synchronization between them, and
- To efficiently share resources between different processes and avoid deadlocks.

Story:

Mr. X owns a big ice cream factory. It manufactures thousands of different kind of ice creams daily. There are several steps to manufacture a single bar/scoop of ice cream from milk and other ingredients through a sequence of steps like boiling milk, mixing sugar and butter, freezing, wrapping etc. There are different machines to perform different types of tasks (i.e. boiler, mixer, freezer, wrapper). Mr. X owns multiple number of machines for each type of task. Each machine can perform several variants of tasks (i.e. boiler can boil milk as well as water, wrapper can wrap bar ice cream as well as Cornetto). But each machine can work with only one task at a time. So Mr. X puts same category machines in a row. He sets up a job-queue (assume unlimited buffer) for each row that contains the sequence of tasks that the machines of that row need to perform. Each machine takes a task from the job-queue and starts processing it. If there are no task in the job-queue, the machines of that row waits for next task to be assigned. Whenever there are multiple machines waiting, they will be assigned tasks based on "longest waiting time first" basis (the machine that is waiting most gets the first task to process when a new task arrives in the job queue). Each ice cream has a label that contains the sequence of the tasks to be performed to produce that ice cream. Machines will read this label and process accordingly.

For each ice cream, there is a fixed sequence of tasks (i.e. first boil the milk for 7minutes, then mix vanilla powder & sugar, then freeze it, and at last pack it and wrap it). When a machine finishes a task for a ice cream, that partially produced ice cream is sent to a controller that puts it to an appropriate queue to perform the next task, or removes it from the factory if it has been processed completely. Mr. X puts the whole system in such a way so that he can input ice cream manufacturing procedure from a central location, and the machines will perform all the tasks accordingly.

Your job is to emulate this whole procedure using multiple processes and inter process communication.

Problem:

You need to develop an master-slave program. Master will assign tasks in job-queue and slave will perform tasks from the job-queue. Upon completion of a task, slave will inform master that it has finished the task, and then it looks at the queue for the next task.

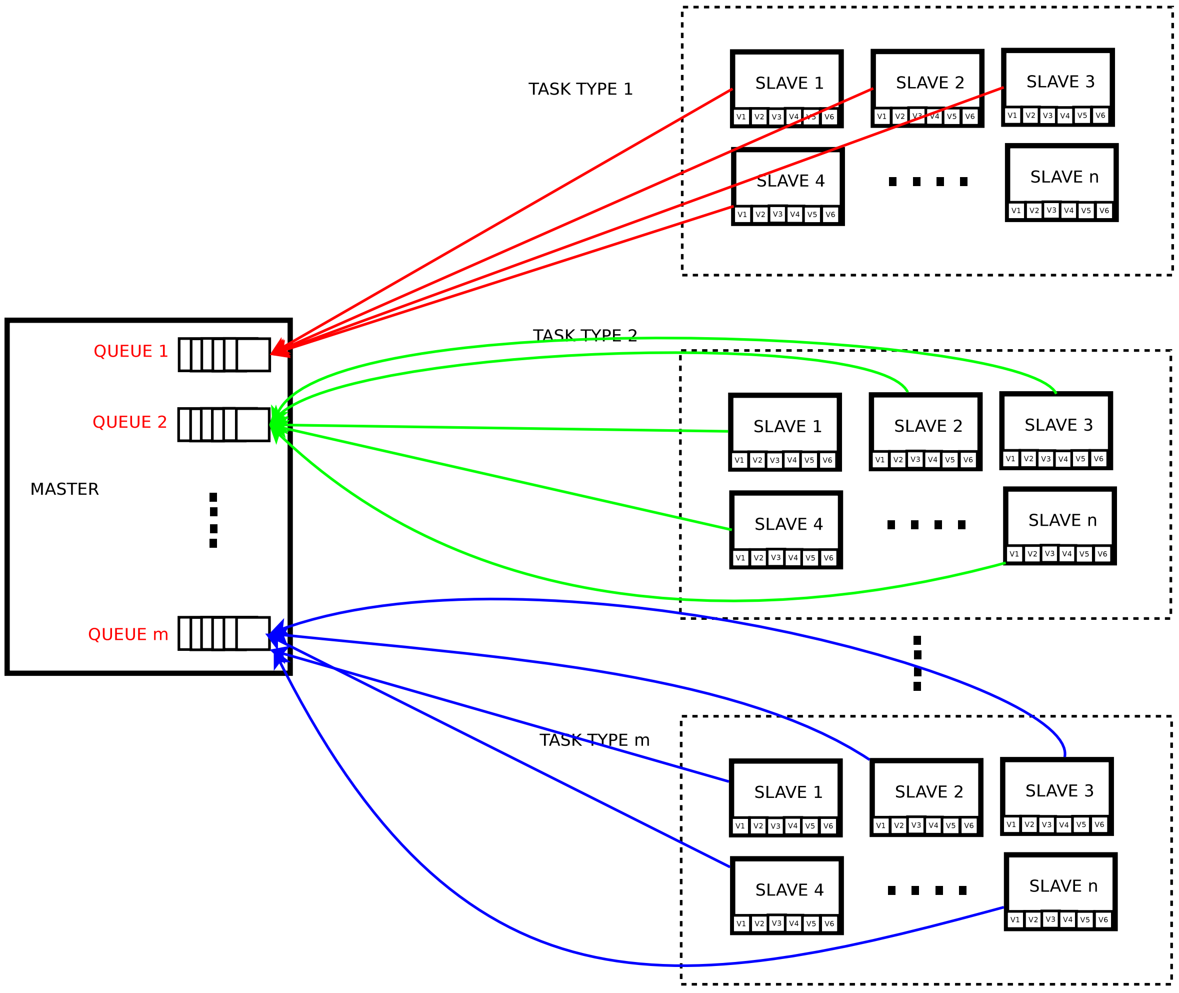

Figure 1: Caption

TASK

In your emulation, you can implement a task by giving a pause (sleep) for some specified duration. For example, for emulating “boiling water for 7 minutes”, the corresponding process sleeps for 7 minutes.

JOB

A job is an ordered list of tasks to be performed sequentially. It is the duty of the master program to put a job in appropriate slave to make it complete. Say, job $vanillaice$ have ordered list of tasks as $boilmilk7min$ followed by $mix - vanilla$.To complete a job $vanillaice$, master first have to put it to slave for $boilmilk7min$, then to $mix - vanilla$.

Program: MASTER

Master has two files: *slave.info* and *job.info*. Master first reads *slave.info* and spawns slave process as per described in *slave.info*.Then it loads jobs from *job.info* and stores in a list (use your favorite data-structure for list). Master will use this list repeatedly. Master then takes first n job and place them in job-queue. Each job contains multiple tasks in order. They have to be performed in order only. After that, whenever a single job finishes, master gets the next job and puts it to appropriate queue. When slave reports that it has completed the task, master again puts that job to another appropriate queue for the next task. After finishing m jobs, master will signal (using ipc or semaphore) every slave to terminate the process. And finally it terminate it-self. Master will give several outputs. We will discuss about output later.

Program: SLAVE

Slave runs different variant of tasks it can perform. It will set itself accordingly. Then it check corresponding job-queue if there is any task for it. If there is, it takes that job and start perform it. After finishing this task, slave informs the server that it has finished the task and move on. A slave can perform multiple variants of a task. But it cannot perform multiple tasks at a time.

Input-Output

For this assignment, you have to follow strict input-output format. Both master and slave take input as command line argument only. No input can be given via *STDIN*.

MASTER

Master program takes exactly 5 arguments as flows:
`$master <path to slave.info file>`[1] `<path to job.info file> <path to slave executable> <n> <m>`
First argument is *slave.info* file. Second argument is *job.info* file. m is the number of job it have to perform before exiting. $n \leq m$.

File Format: *slave.info*

This is a text file. Every line starts with the task category and slave count followed by task variant definition. Task variant is defined by variant name and the time it required (in **millisecond**) to perform. You have to wait this much time before you inform master about your status. There will be one or more task variants for each type of task. Format for a single line in slave.info is as follows:
`<task-type> <count> <variant-name> <time> [<variant-name> <time> [<variant-name> <time> ...]]`[2]
task-type: Type of the task a group of slave can perform.
count: Number of slave processes need to be for this type of task.
variant-name: Variant of tasks, like boiling is a task. But boiling-water or boiling-milk is variants of that task type. Boiler a slave. There can be multiple (defined by count parameter) boiler slave. Every boiler slave can perform all variants of boiling task. But a slave can not boil two things at the same time.

[1]enclosed with <> means mandatory field/parameter/argument
[2]enclosed with [] means optional.

time: Time required to performed the task. For emulation purpose, it will be in ms (millisecond).

Master program have to create a slave process for each line of this file.

Example of *slave.info* file:

```
boil 2 boilmilk7min 7 heating 2 boilwater 10
wrap 3 wrapbar 5 wrapcup 20
mix 2 vanilla 5 butter 20 chocolatepowder 11
freeze 1 medium 6 extreme 50 slow 2
```

Explanation:

- **Line 1:** There will be two slave processes to perform the task of type boil. Both the slave processes can perform every variant of boil-type task i.e boilmilk7min, heating, and boilwater. Among these variants, boilmilk7min will take 7ms time, heating will take 2ms, and boilwater will take 10ms. There will be a single queue for boiling type task in the master process. Two slaves will cater this queue only.
- **Line 2:** There will be 3 slave processes to perform task of type wrap. All the slave processes can perform every variant of wrap, type task i.e wrapbar and wrapcup. wrapbar and wrapcup will take 5ms and 20ms time, respectively. There will be a single queue for wrap type task in the master process. Three slaves will cater for this queue only.

I leave line 3 & 4 for you to understand. There will be total 8 slave processes and 4 task queues in the master process.

File format: *job.info*

Each line in this file is a job description. Each line contains few space separated words. First word is the name of the job. From second word onwards, each word represents a task. Task is composed of two words – task-name and task-variant, separated by :. task-name and task-variant will also be there in *slave.info*. Format for each line in *job.info* is as follows:
`<job-name> <task-type>:<variant-name> [<task-type>:<variant-name> [<task-type> :<variant-name> ..]]`

Example of *job.info* file:

```
vanillaice boil:boilmilk7min mix:vanilla
chocolatebar boil:boilmilk7min mix:chocolatepowder freeze:extreme
```

Explanation:

There are 2 jobs described in this file, Vanillaice and Chocolatebar. To complete the vanillaice job, the system needs to be performed two tasks – boilmilk7min of "boil" type task and vanilla of "mix" type task. Similarly for chocolatebar, it needs to perform 3 tasks.

Same job can appear in this file multiple times with the same task list.

N.B. This format is important. A task name can be similar with job name or task variant name.

SLAVE

Master spawns slaves based on the information given in *slave.info* file.

Output

Master will give following outputs:

- Single line output after it creates a slave. The output is as following:
  `<task-type> <pid>`
  task-type: Name of task-type mentioned in *slave.info*.
  pid: Process id of the slave.

- Single line output after finishing each task by a slave, as follows:
  `<job> <task> <pid> <remaining_task_count> <status>`
  job: Name of the job as mentioned in *job.info* file.
  task: Combination of task-type and task-variant separated by ":"(without quote).
  remaining_task_count: How many task to be performed before the job finishes.
  status: Any one of the followings: *waiting* and *finished*. Status will be *finished* only when remaining_task_count is zero. Otherwise it will be *waiting*.

- Before master terminates, it will give few lines of output. Each line will contain some information of the slave processes. Output format is as follows:
  `<pid> <total number of task it performed> <task-type> <task-variants1> <number of task-variant1 it performed> [<task-variants2> <number of task-variant2 it performed> [<task-variants3> <number of task-variant3 it performed> ...]]`

Sample input-output:

Command:

```
master slave.info job.info ./slave 3 4
```

Contents of slave.info and job.info are as mentioned before.

Output: Check Table 1.

Important notes:

- You have to use interprocess communication.
- You have to use binary semaphore. By default POSIX system provides counter semaphore. You can use your own semaphore or design counter semaphore in such a way that it behaves like binary semaphore.
- Master process should not use any information about slave process apart from pid.
- Be absolute clear about input and output format. We may use automatic script to verify output of your code.
- Don't make any assumption. If anything is not clear ask TA/Instructor.
- You may put optional outputs to STDERR for your won debugging purpose.

Report:

You have to write a report on this assignment. Contents of this report will be as follows:

- A detail description of the Inter-Process-Communication method you have used in this assignment. What are the advantages of this method over other?

Table 1: Sample Output

```
boil 30459
boil 30460
wrap 30461
wrap 30462
wrap 30463
mix 30464
mix 30465
freeze 30466
vanillaice boil:boilmilk7min 30459 1 waiting
chocolatebar boil:boilmilk7min 30459 2 waiting
vanillaice boil:boilmilk7min 30459 1 waiting
vanillaice mix:vanilla 30464 0 finihsed
chocolatebar boil:boilmilk7min 30459 2 waiting
chocolatebar mix:chocolatepowder 30464 1 waiting
vanillaice mix:vanilla 30464 0 finihsed
chocolatebar mix:chocolatepowder 30464 1 waiting
chocolatebar freeze:extreme 30466 0 finihsed
chocolatebar freeze:extreme 30466 0 finihsed
30459 4 boil boilmilk7min 4 heating 0 boilwater 0
30460 0 boil boilmilk7min 0 heating 0 boilwater 0
30461 0 wrap wrapbar 0 wrapcup 0
30462 0 wrap wrapbar 0 wrapcup 0
30463 0 wrap wrapbar 0 wrapcup 0
30464 4 mix vanilla 2 butter 0 chocolatepowder 2
30465 0 mix vanilla 0 butter 0 chocolatepowder 0
30466 2 freeze medium 0 extreme 2 slow 0
```

- What are the informations master shared with the child? How does master share those informations with slave (shared variable or argument or something else)?
- Is there any critical section in your implementation? What are the critical sections in your implementation?
- How do you ensures mutual exclusion of those critical section between processes?

Deliverables

For submission, you are required to submit:

1. Source code.
2. The report.