**Lab 1: Process Management**

*50.005 Computer System Engineering*

***Due date: 25 Feb 08:30 AM (Week 5)***

[Outline](#_pckct1z1y98f)

[Getting Started](#_ayjir8e06gjo)

[Grading -- [100 pts]](#_knltmoc6mpjv)

[Submission Procedure](#_rrx7joq8cqav)

[Download Materials, Check](#_6az1k1bxuipt)

[The Task](#_462lmgyfphys)

[Implementation Guide](#_bys7gcwrhzgn)

[TODO#1: Implement the setup function.](#_oah6hkk12rbf)

[TODO#2: Implement createchildren()](#_49s37cyd3cii)

[TODO#3: Implement job\_dispatch(), modify task()](#_pwtceq1p716r)

[TODO#4: Implement main\_loop() busy wait for job creation and termination](#_6c5lk5eus00v)

[TODO#5: Implement cleanup() function](#_z7agq34hqw96)

[Validation](#_tvz2qt24ag56)

[Test Files](#_zhj96lwl5s58)

[Run through checker script](#_h94dwz11r3zu)

[Submission Rules](#_8grvdj69vz4y)

## Outline

In this lab, you are tasked to manage a pool of UNIX processes.

The lab is written entirely in C. At the end of this lab, you should be able to:

* Understand how to perform fork() and handle its return value
* Manage a set of processes
* Manage inter-processes communications using shared memory
* Protect shared resources with semaphores
* Managing process pool:
  + Checking the status of child processes
  + Dispatch jobs to child processes
  + Respawn (prematurely) terminated children processes
* Perform basic I/O operations

## Getting Started

Clone the files:

git clone https://github.com/natalieagus/50005Lab1.git

then **closely follow the instructions** given in this handout. You are only required to **modify** the starter code and header files. **DO NOT create your own script for submission. DO NOT modify the makefile either.**

## Grading

The points awarded in this lab are written in each of the sections below. The total marks for this lab is **100.**

## Submission Procedure

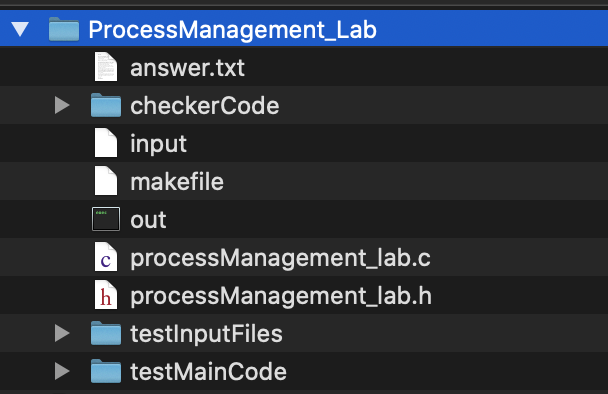
1. **This is an individual assignment.**
2. Write a short README note on how you tackle [this segment.](#_6c5lk5eus00v)
3. **Zip (ZIP, not RAR, or whatever other format)** back the folder containing the modified starter code + README note in (2) with your answer:
   1. **Upload** to @csesubmitbot telegram bot using the command /submitlab1
   2. **CHECK** your submission by using the command /checksubmission
4. Carefully read the submission rules [here](#_8grvdj69vz4y).

## Download Materials, Check

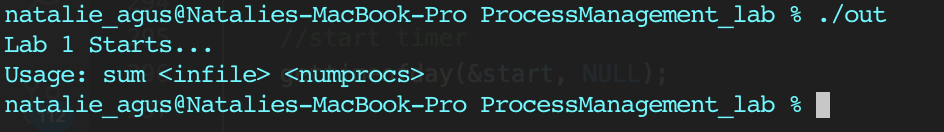
Clone the starter code from github:

git clone https://github.com/natalieagus/50005Lab1.git

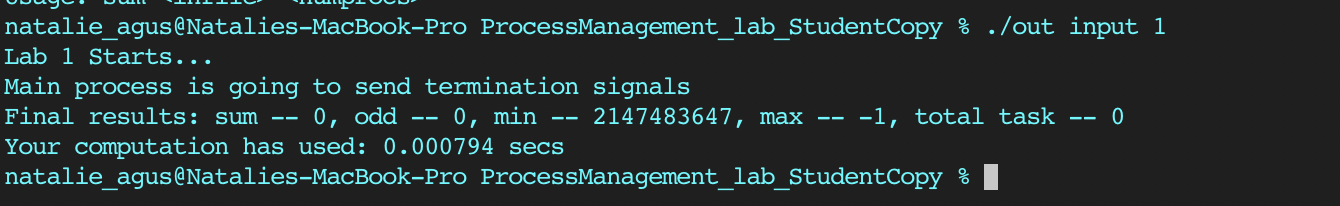
You should see the following files given to you:



To compile, type make and you will have an executable called out. You can execute it as follows: ./out and notice the welcome message:



Re-run the program with: ./out input 1



If you see the above output, then you’re all set. Let’s begin with the description of the lab first.

## 

## The Task

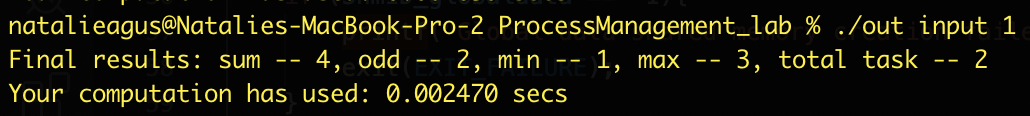
The program that we have to make for this lab must read an input file, and execute jobs based on the content of the input file by spawning N number of processes where N is the last argument value given when executing the program as ./out input N.

Open ‘input’ given to you, and you will find it has the following contents. If it is not, modify it to contain these. 

This input file resembles the input **jobs** that your program must do. The format of the input file is as follows:

* Each line consists of two characters: t/w/i and followed by a number N that means the unit time
* For each line beginning with t, it means ‘task’, that is executed for N unit time
* For each line beginning with w, it means ‘wait’ for N unit time
* Task that starts with i means an illegal task. Attempts to execute these illegal tasks causes the process to crash.
* There will **be no other alphabet given in the input file**. So do not worry.

Your program must be executed in the following format: ./out input <num of processes>

**And printout the following statistics if the program has been implemented properly.** 

**The computation of the statistics is simple:**

* **Sum:** indicates the total time spent performing ‘task’. In the input above, we only have t1 and t3, and therefore sum results in 4.
* **Odd:** indicates the number of tasks that requires an odd number of unit time to execute. Both 1 and 3 are odd, hence odd results in 2.
* **Min:** indicates the minimum amount of time to do a task that the program has encountered, that is 1 unit time caused by t1.
* **Max:** indicates the maximum amount of time to do a task that the program has encountered, that is 3 unit time caused by t3.
* **Total task:** as the name itself states, how many tasks have been performed by our program.

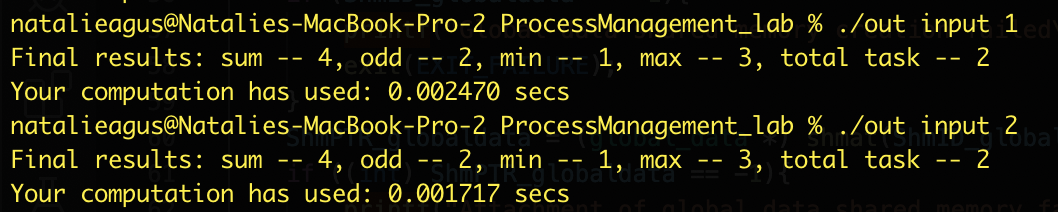
Of course when <num of processes> is 1, it is equivalent to doing the tasks consecutively. We spawn 1 worker process using fork() and this worker process will:

1. Perform task for 1 unit time,
2. Then wait for 2 unit time,
3. And perform the task again for 3 unit time.
4. **The next task is the illegal task i1, the worker process crash upon attempting to execute this.**
5. The main process respawns another worker process. This new worker process executes i3 and crashes again. Repeat for i4. **The main process only keep 1 worker process alive at any time.**
6. The total time to complete the entire input is 6 unit time + overhead of creating the worker process 3 times.

The reason we spawn worker process to execute the task is that since it is isolated from the main process, the main process does not crash when the worker crashes. **This is the benefit of using fork()** to do multiple jobs. **If these jobs were to be done using a single process then the entire process crash whenever theres ‘illegal’ task encountered,** and we will never succeed in executing the entire jobs written in the input file.

In this simple case we can simply choose not to “crash” when the first letter is *i*. However, we perform this ‘crash’ to simulate a real-world scenario where we are tasked to do a series of jobs from input / user and we do not know if it will cause our system to crash due to bugs in these jobs. Hence the spawning of worker process prevents this entire system from crashing in the event that a buggy input is encountered. **Recall: the multi-process web browser tabs scenario from our lecture notes.**

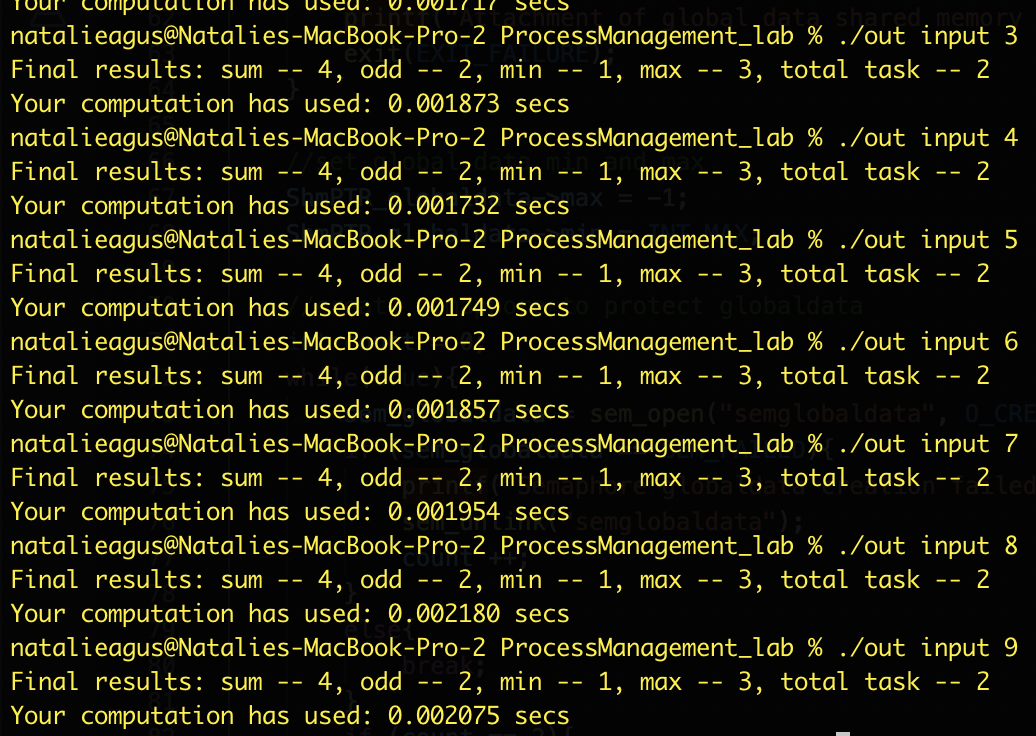
If we increase the number of processes into 2, we can finish the task in almost half the time.



One possible combination is:

1. Process 1 execute t1 and Process 2 execute w2 in parallel
2. When Process 1 is done, it begins executing t3. Process 2 is still waiting halfway.
3. When Process 2 is done, it will crash when attempting to execute i1.
4. Main process respawn process 2 and a repeated crash + respawning happens when it attempts to execute i3 and i4.
5. **In any case, there’s only at maximum of 2 worker processes being alive at a time.**
6. The legal tasks are done in 4 unit time + overhead, faster than when only 1 worker process is used.

Since we only have two effective tasks, increasing the process count does not speed things up any faster because we suffer from overhead of process creation:



You can see more of the detailed sample output format in [this](#_tvz2qt24ag56) section.

## Implementation Guide

You are given only two files, processManagement\_lab.c and its header file processManagement\_lab.h:

* **DO NOT modify the header file processManagement\_lab.h**
* **Fill up your answer in the spaces indicated as “TODO#” in processManagement\_lab.c**

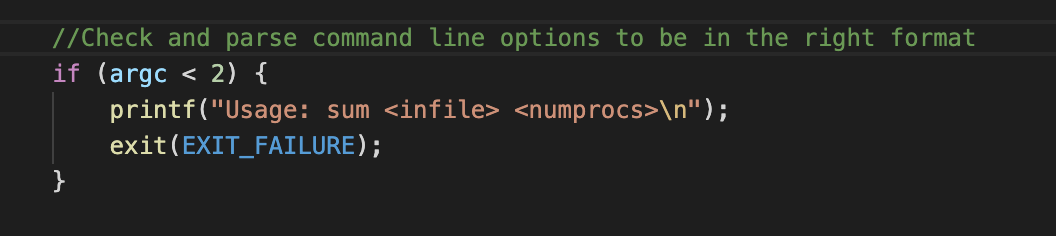
The steps below go through the starter code as well as what you should do to implement the program.

First thing first, the main process spawn N children worker when executed in the following format:

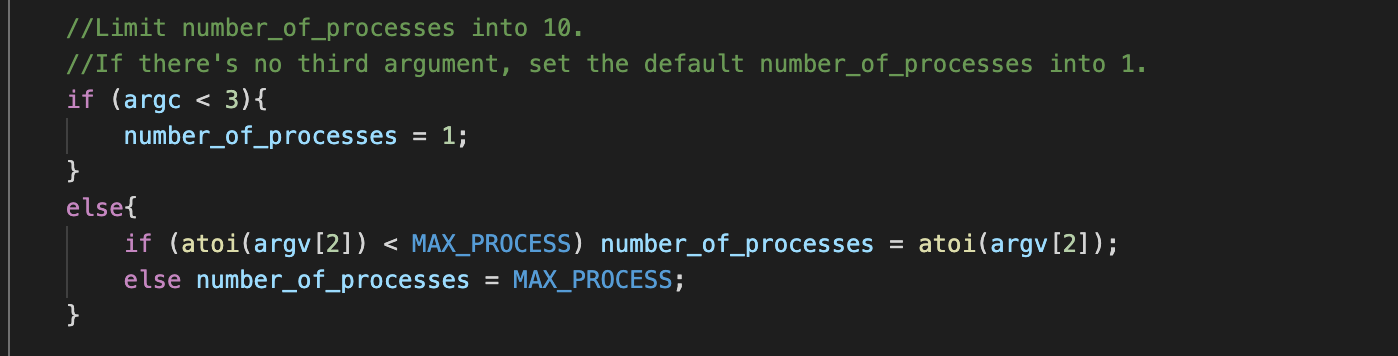
./out input N

Let's go through the main() function in procsesManagement\_lab.c first. At the beginning, there’s a little setup for keeping track of computation time. You can ignore that.

Then, we need to check that the argument format is correct, which is already done here in the main() function:

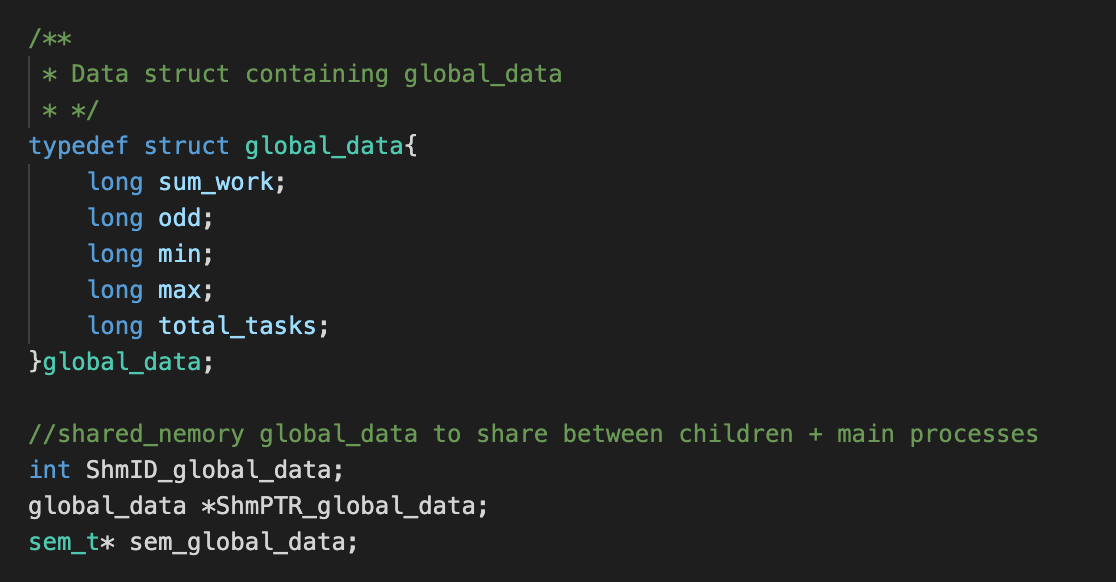


And also limit the amount of N into 10, storing it into number\_of\_processes:

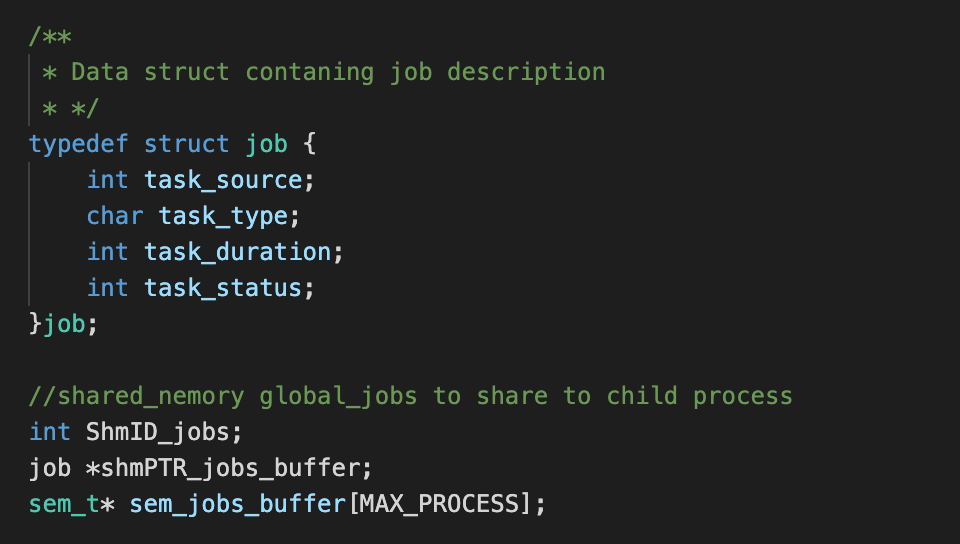


In the next line, the main process calls the setup() function. Now go to the setup() function, and you will find you need to utilise these declared variables:

* 1. a global\_data struct that contains all the required stats (see processManagement\_lab.h). **DO NOT use your own struct.**



* 1. A shmPTR\_jobs\_buffer of size N as shared memory that can contain job descriptions. The job description uses job struct given (see processManagement\_lab.h). **DO NOT use your own struct.**

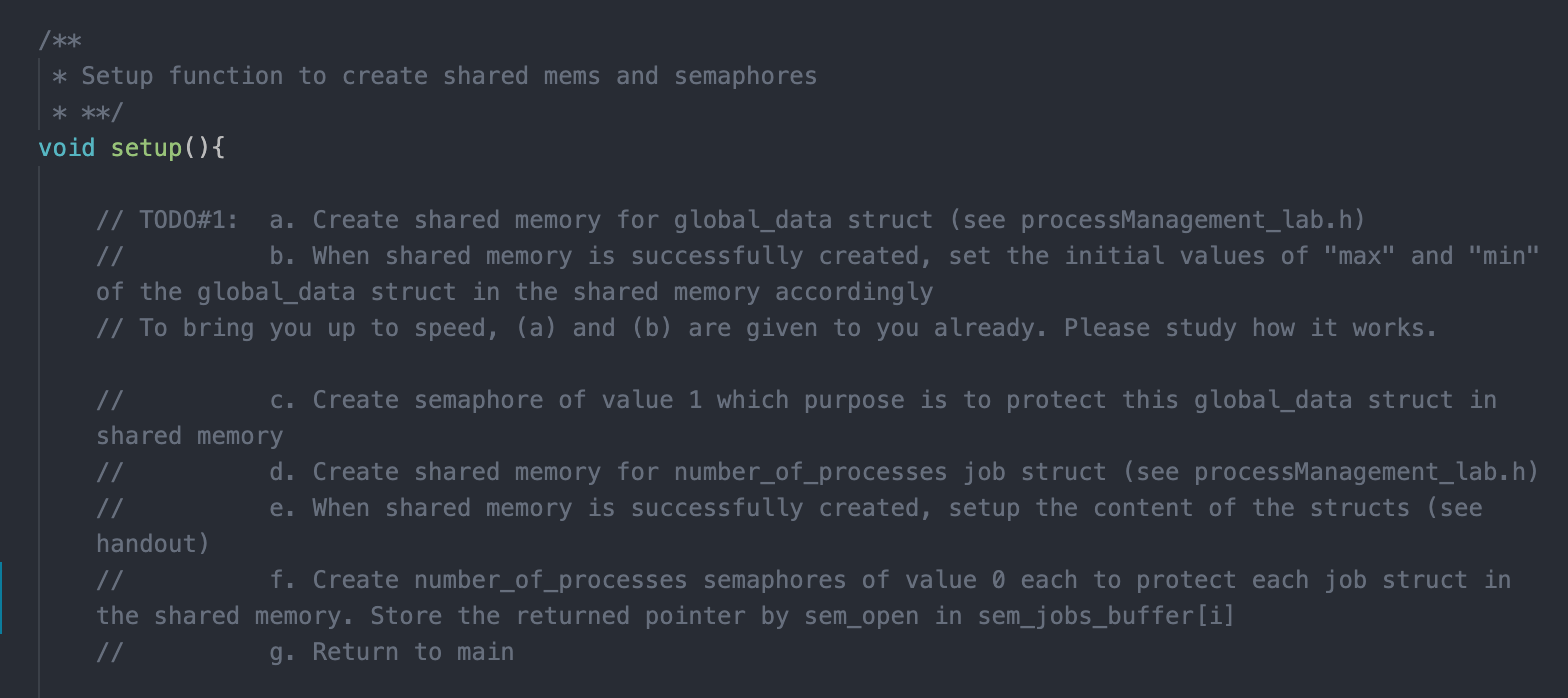
****

* 1. As shown above, a **mutex** called **sem\_global\_data** to **protect** the global\_data shared memory , and **N job mutexes** called sem\_jobs\_buffer to protect each shmPTR\_jobs\_buffer entry. (see processManagement\_lab.h). **DO NOT create your own mutexes.**

Obviously, although these structs can be declared as global variables, we cannot share them among processes. One way to perform inter-process communication is by creating a shared memory space, and initializing our structs there. The next section guides you on how to do this.

### TODO#1: Implement the setup function.

It contains the following requirements:

****

**You need to implement the setup() function and utilise all the pre-declared variables in the header file when realising (a), (b), (c), and (d) above for this function.**

**To avoid any hiccups and get you started, (a) and (b) are implemented for you already. Please study how it works.**

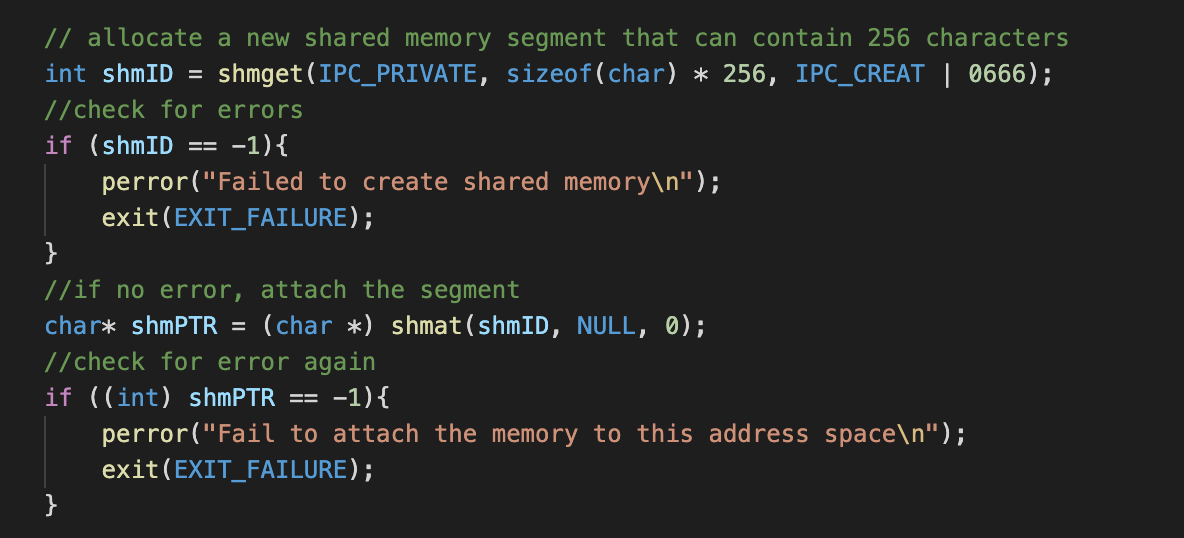
To create shared memory segments, you should use **system calls shmat() and shmget():**

* shmget() is used to allocate new shared memory location when the first argument is IPC\_PRIVATE. You need to specify the **total size** of the memory segment and pass it as its second argument. It returns the identifier (int) for the shared memory segment.
* shmat() *attaches* the shared memory segment to the calling process’ address space (meaning map the allocated segment in physical memory to the VM space of the calling process), and returns the address (pointer type, you can cast it to the type you want) to that shared memory segment.
* Documentations can be found here <http://man7.org/linux/man-pages/man2/shmget.2.html> and <http://man7.org/linux/man-pages/man2/shmat.2.html>
* For part (e) setup the content of job structs in the shared memory, see this [section](#_pwtceq1p716r) to understand what each field in job struct mean. You may want to initialize task\_status as 0 first for now, but it is UP TO your implementation. To initialize, you need to loop through shmPTR\_jobs\_buffer[i] and set each job struct in it.
* It is important to **detach** and **delete** the shared memory using **shmctl()** and **shmdt()** after usage, otherwise you’ll run out of space (If you are still confused, see how its implemented in the test code given in the file testMainCode/testmain\_todo1.txt). You will implement this in TODO#5.

For now, it is already done for you in the test main function given, so you don’t need to do anything. If your code **crashes** in the middle many times and your computer refuses to create more shared memory, you can:

* + Restart your computer, OR
  + Type ipcs in the terminal, and manually delete the shared memory one by one using ipcrm -m <index>
* Please use the shared memory global variables that are **already declared for you in the header file to store the return value of shmget and shmPTR.** DO NOT re-initialize it in the setup() because it will make them *local variable* and the other functions won’t be able to access it.
* **Usage example (THIS IS JUST AN *EXAMPLE*, DO NOT JUST COPY PASTE THIS CODE BLINDLY. SIMPLY STORE THE RETURN VALUE OF shmget and shmat to:**
  + **ShmID\_jobs, ShmID\_global\_data**
  + **shmPTR\_jobs\_buffer, ShmPTR\_global\_data**

**WITHOUT redeclaring it SINCE THESE VARIABLES are ALREADY DECLARED FOR YOU IN THE HEADER FILE!!)**

****

The size of the shared memory segment shmPTR\_global\_data is: **sizeof(global\_data)** and the size of the shared memory segment shmPTR\_jobs\_buffer is **sizeof(job) \* num\_of\_processes.**

To create semaphores, you need to utilise the system call **sem\_open():**

* The **first** argument to sem\_open is the semaphore’s name. The **last** argument is the initial value of the semaphore. You can treat the middle two as fixed.
* If you are interested, you can read more about it in the documentation here: <http://man7.org/linux/man-pages/man3/sem_open.3.html>
* If existing semaphore with the same name is present, it will return SEM\_FAILED, otherwise, it returns a semaphore pointer.
* Therefore, use a loop to check if it failed, do sem\_unlink(semaphore\_name) and attempt sem\_open again
* **The global VARIABLE and SEMAPHORE NAME that you MUST use to store the return value of sem\_open system call are also already DECLARED for you in header file, please just use it straight:**
  + sem\_global\_data: “semglobaldata”
  + sem\_jobs\_buffer[i] : “semjobsi” where i = 0, …, N-1
* **Usage example (THIS IS JUST AN *EXAMPLE*, DO NOT JUST COPY PASTE THIS CODE BLINDLY. SIMPLY STORE THE RETURN VALUE OF SEM\_OPEN to sem\_global\_data/sem\_jobs\_buffer[i] WITHOUT redeclaring it SINCE THESE VARIABLES are ALREADY DECLARED FOR YOU IN THE HEADER FILE!!):**



**Note on semaphore function:**

We are using *named semaphore,* where we can identify the semaphore by name such as “mysemaphore” above. POSIX has implementations of unnamed semaphore as well (using sem\_init and sem\_destroy) but it is **deprecated** since OS X. It can still be used in some other UNIX system. For named semaphore, simply ensure that you unlink the exact semaphore name as when you created it with sem\_open.

**CHECK:** To check if your implementation of shared memory and semaphore is proper, temporarily replace the entire main() function with the main() function given in the file testMainCode/testmain\_todo1.c.

You should should observe the following output given the executions:

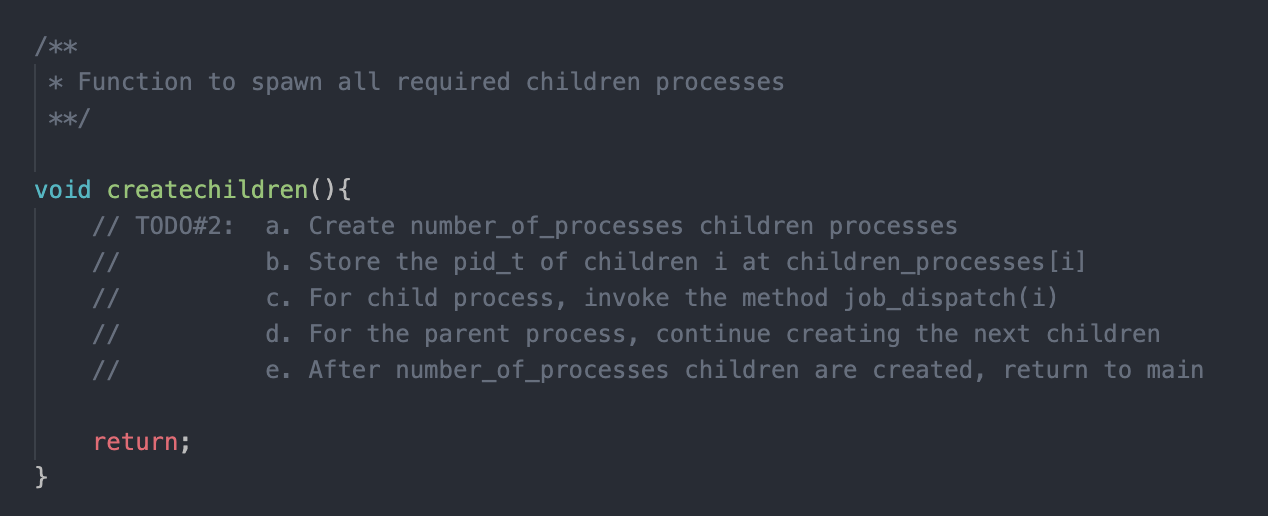


Notice how the parent and child processes are able to communicate with one another, and are synchronized to print in order, i.e: *“parent write job i” comes before “child receives job i” message. “Job i cleared by children” comes before “semaphore close..” message,* ***for the same i.***

Note that we cannot control the order for different i, since these processes are concurrent.

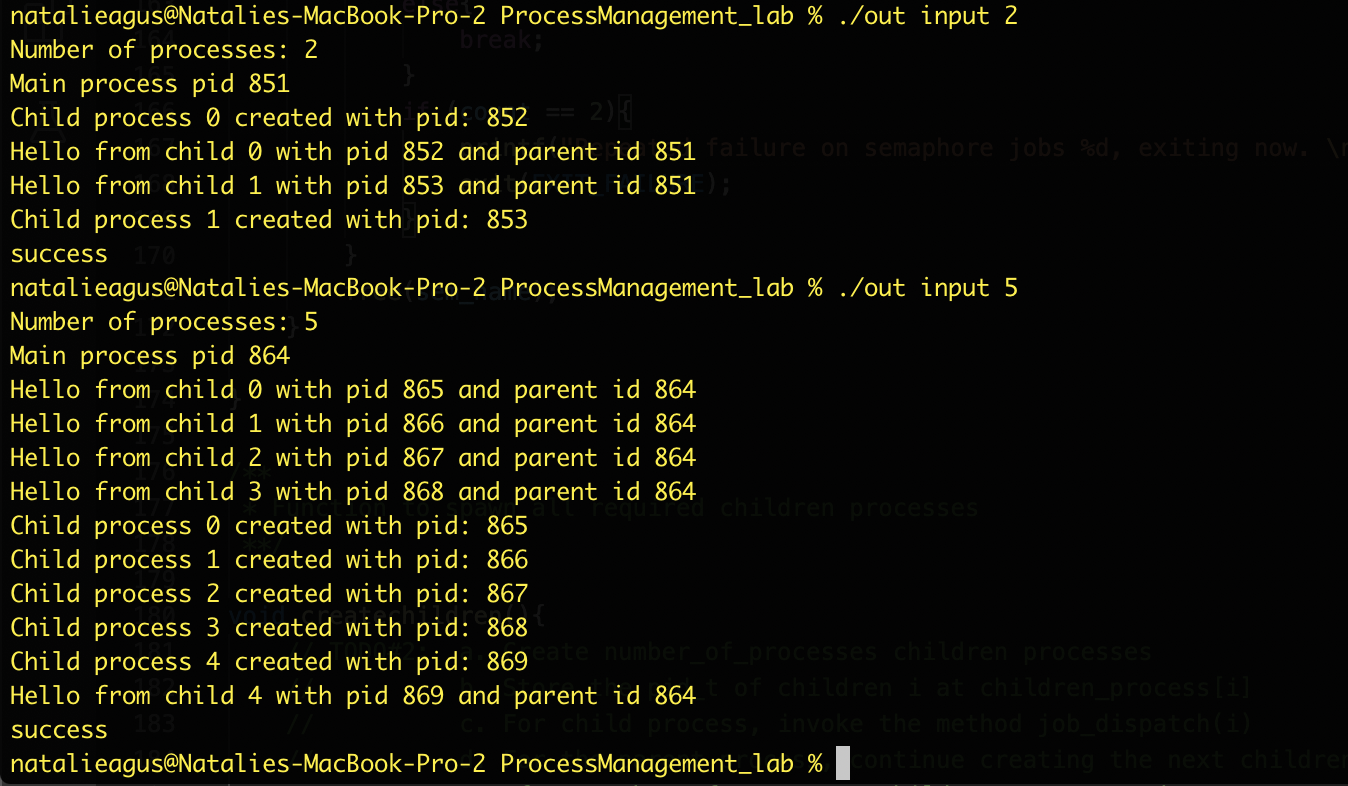
### TODO#2: Implement createchildren()

After setup(), the main process calls createchildren() where it spawns N worker processes using fork().

If you forgot how fork() works, refer to our lecture material. The requirements are written below:

**CHECK:** To check if your implementation of shared memory and semaphore is proper, temporarily replace the entire main() function with the main() function given in the file testMainCode/testmain\_todo2.c.

You should should observe the following (similar, there might be some interleaving in the print) output given the executions:



Notice how N child processes are invoked accordingly, knowing its own index i.

### TODO#3: Implement job\_dispatch(), modify task()

After spawning worker processes, the main process will open the input file, read it, and then start putting jobs into the global shmPTR\_jobs\_buffer. Each *‘slot’* indicates a location where the main process and *respective worker process i can communicate.*

Concurrently, each worker process i will execute the instructions in job\_dispatch() function.

This function instructs the worker process i to get a job from its corresponding buffer slot i and execute it. The job is a struct which contains the following variables:

* int task\_duration: the time needed to do this job
* char task\_type: the type of this job. You will only need to handle job type ‘t’, ‘w’, ‘z’, or ‘i’
* Int task\_status: state of the job, will be either 0, 1, or -1

Once each worker finishes executing, this worker process should clear its corresponding shmPTR\_jobs\_buffer[i] by modifying the task\_status of the job struct residing in shmPTR\_jobs\_buffer[i] to 0.

Here is the meaning of the task\_status field:

* shmPTR\_jobs\_buffer[i].task\_status == 0: no job or job cleared
* shmPTR\_jobs\_buffer[i].task\_status == 1: new job present
* shmPTR\_jobs\_buffer[i].task\_status == -1: termination job

Then, the worker process *loops* to examine shmPTR\_jobs\_buffer[i] again for the presence of a new job. If there’s no job, the worker process calls sem\_wait(sem\_jobs\_buffer[i]).

**IMPORTANT REQUIREMENT:** Worker process **is NOT allowed to *busy wait,* e.g: loop over and over to check for a new job.** Use the worker child process’ semaphore sem\_wait(sem\_jobs\_buffer[i]) for this. **If you do busy wait, -20 pts.**

**How to handle each job type:**

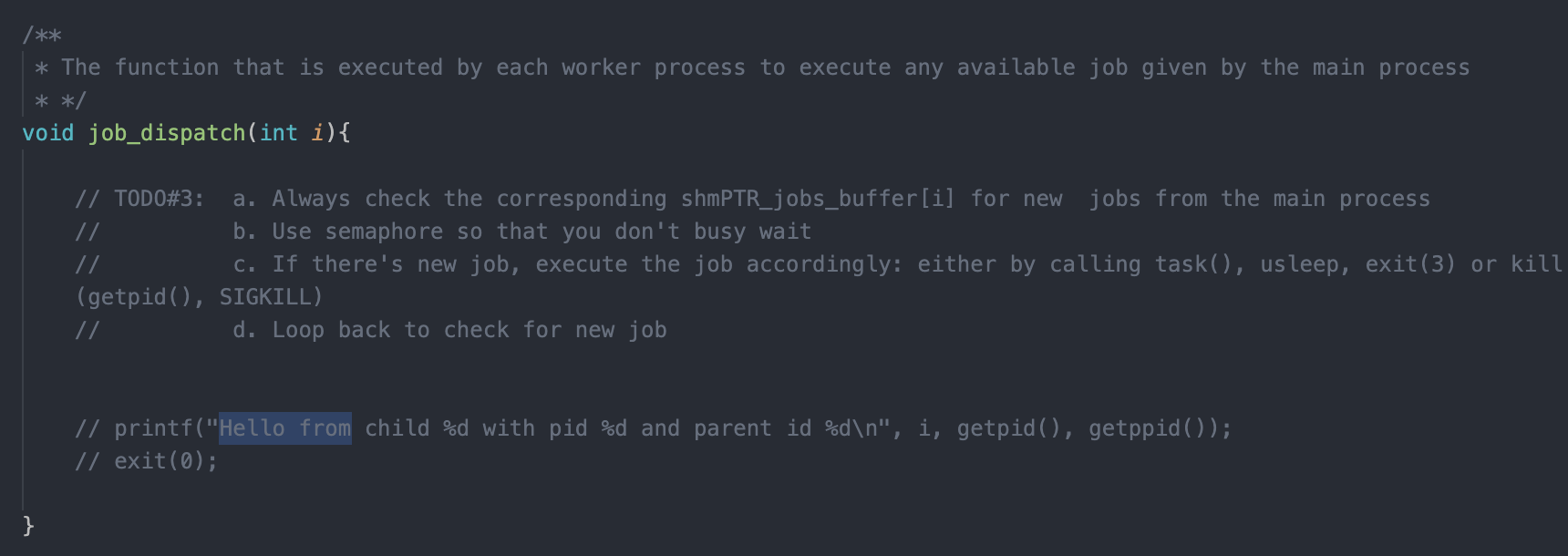
* A job that starts with ‘t’ (shmPTR\_jobs\_buffer[i].task\_type == ‘t’) causes the worker process to execute the function task(int time) that’s been given to you for the designated amount of time as given to the worker process in shmPTR\_jobs\_buffer[i].task\_duration
* A job that starts with ‘w’ causes the worker process to simply sleep using usleep(time \* TIME\_MULTIPLIER) for the designated amount of time obtained from shmPTR\_jobs\_buffer[i].task\_duration
* A job that starts with ‘z’ causes the worker process to exit legally using system call exit(3). **Exiting this way will cause the STATUS (not return status) of the child to be pid of the child when waitpid/wait is called.**
* A job that starts with ‘i’ causes the worker process to terminate itself prematurely using system call kill(getpid(),SIGKILL). **Exiting this way will cause the STATUS (not return status) of the child to be 9 when waitpid/wait is called.** The worker process has to crash this way. **Failure to do this results in -10 pts**.

Inside the task function, the worker process has to **modify** the global stats **shmPTR\_global\_data** using its **mutex solution: semaphore sem\_global\_data. Otherwise, you will face race conditions. If you face ANY race condition, -40 pts.**

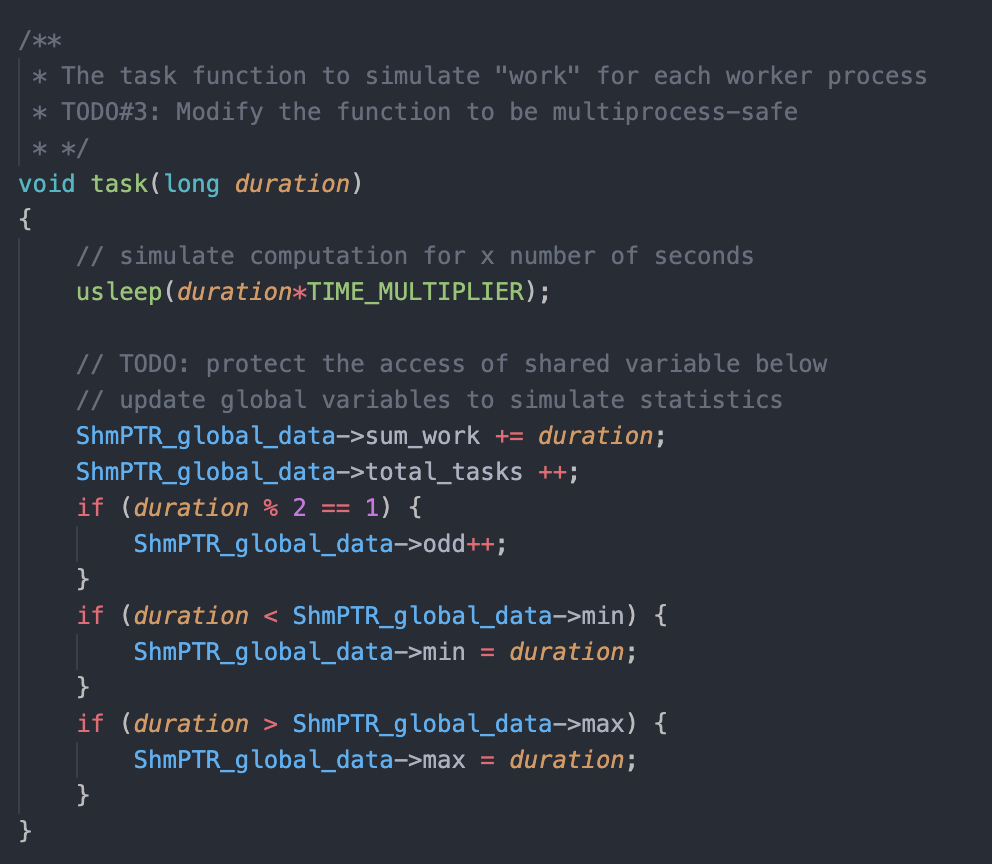
**Check return value of sem\_wait : on success, it returns 0. Else if error, e.g: if you DID NOT initialize it properly in TODO#1, sem\_wait(sem\_t\* semaphore) will return -1.**

**Comment out the given printf and exit(0)**, then write your answer below. As stated in part (b), you can **use semaphore**  sem\_jobs\_buffer[i] to **NOT** busy wait, i.e: not repeatedly check shmPTR\_jobs\_buffer[i] for *new job*. It is your duty to think about how to use semaphore for this child process to avoid busy wait.

*The parent in turn, to increase sem\_jobs\_buffer[i] for child i to proceed, you will do this in TODO#4.*

****

**Don't forget to fix the task() function below to be safe from race condition:**

****

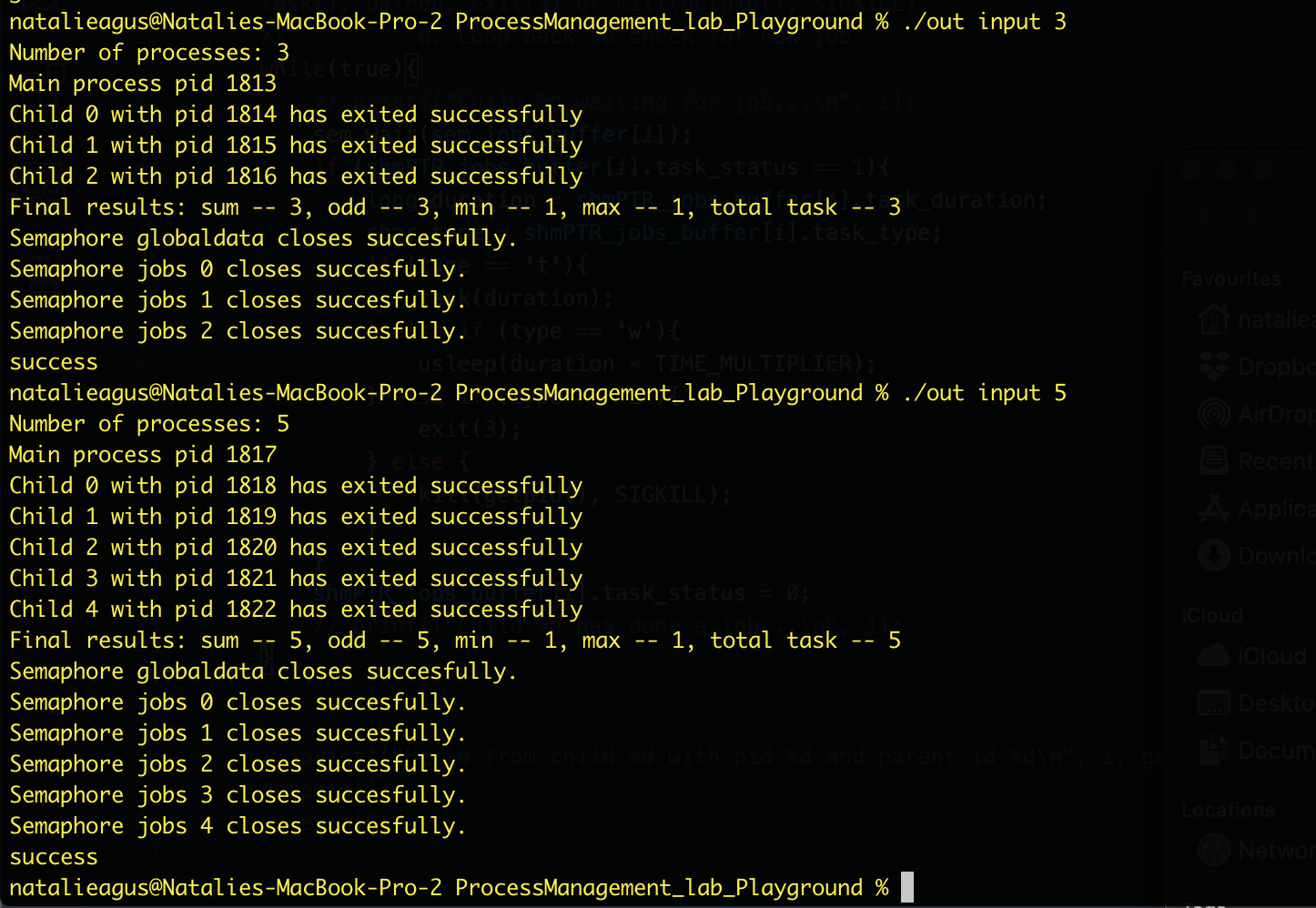
**CHECK:**

* To check if your implementation of shared memory and semaphore is proper, temporarily replace the entire main() function with the main() function given in the file testMainCode/testmain\_todo3.c.

Note: the program will sleep for 3 seconds so don’t panic and wait a little bit after executing the program for any output.

You should see such output. Also if the order of printing of the child is not the same in your machine, that is okay as well, because these processes are concurrent and we cannot control their order of execution.

**The key is to check if the global stats are correct:**



Don’t worry that the global stats does not reflect what’s in the input file yet. **We hardcoded the test case in this checker code** so the output should look exactly as the above if you do TODO#3 correctly.

### TODO#4: Implement main\_loop() busy wait for job creation and termination

Concurrently, the main process busy waits (yes we allow main process to busy wait) by examining each of the N slots of shmPTR\_jobs\_buffer in rotation to check if its empty (i.e: shmPTR\_jobs\_buffer[i].task\_status == 0), which means the worker process has executed it / no new job in the slot for as long as there’s still new jobs from input file. If there’s still an **empty** slot **and the process is still alive,** the main process fills it up with the next available job.

The main process can inspect if the worker process i is *still alive* using waitpid with WNOHANG option to not let it hang for children that’s still alive:

| int alive = waitpid(children\_processes[i], NULL, WNOHANG); |
| --- |

When a child worker process i is still alive, the above waitpid will return with 0. The main process will give a new job for that worker process i if the shmPTR\_jobs\_buffer[i] is done (task\_status == 0), or whatever other way you designed the code so that the child process can tell parents that it has done the assigned job.

IF you want to CHECK the **return status** of the child, then you should make use of the **SECOND** argument:

int status;

int alive = waitpid(children\_processes[i], &status, WNOHANG);

// status == 9 if the child exits prematurely, otherwise you can

// reap the return value:

if (WIFEXITED(status)) // if the process exits NORMALLY, this evaluates to TRUE.

{

int es = WEXITSTATUS(status);

//Hence print return status

printf("Exit status was %d\n", es); // this will give 3 when exit(3) caused

// the child to terminate

}

Some fact:

* status stores the return value of the child in its lower order 8 bits, plus other information in the higher bits.
* waitpid() returns the pid of the child if the status of the child process is available (meaning the child has exited)
* If WNOHANG is specified and there are valid processes to report on, but there is no status available, a value of 0 is returned.
* If WNOHANG is **not specified** and there are valid processes to report on, **but there is no status available,** the calling process is **blocked** until status becomes available, at which time the return code is set to the pid of the process for which status is available.
* The macro WEXITSTATUS does the following:

WEXITSTATUS(*stat\_val*)

If the value of WIFEXITED(*stat\_val*) is non-zero, this macro evaluates to the low-order 8 bits of the status argument that the child process passed to *\_exit()* or *exit()*, or the value the child process returned from *main()*.

If the worker process i is not alive, the main process **spawn a new worker** process to execute jobs from this buffer[i], and pass this new job to shmPTR\_jobs\_buffer[i].

**You are free to design some kind of worker process revival process, e.g:**

1. **Naive way: always spawn new worker process that’s abruptly terminated, regardless on whether there’s new job or not**
2. **Better way: Spawn new worker process only if all other worker processes are busy. If there’s no job, don’t need to revive terminated worker process, and just send ‘z0’ signals to the remaining processes that are alive**

**Your design here will affect how you terminate the worker process in the next paragraph.**

**Failure to revive worker processes do this results in -20 pts.**

After the main process finishes reading the entire input file, it will loop through once more shmPTR\_jobs\_buffer and it will put a termination type ‘z’ job in each **empty / no job** shmPTR\_jobs\_buffer[i] i.e, when shmPTR\_jobs\_buffer[i].task\_status == 0 with job duration 0.

**Design a way to legally terminate ALL worker processes that are still alive** before exiting the main process. Remember, any worker process i may prematurely terminate at any time depending on the content of input file and execution order, and may not be revived because there’s no more new jobs. **Failure to do this results in -10 pts.**

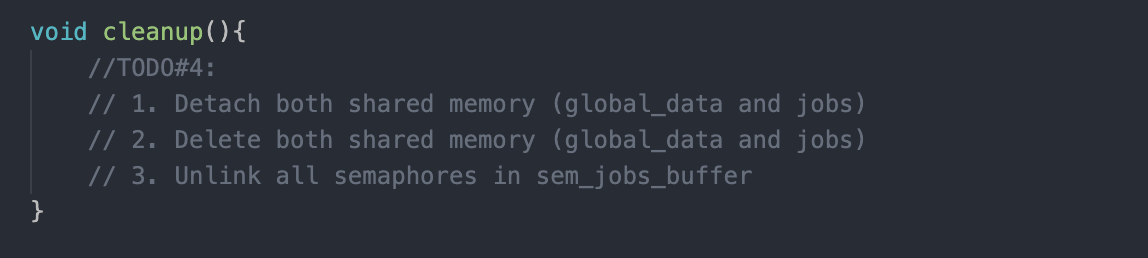
**Be sure to highlight this in your submission README notes.**

### 

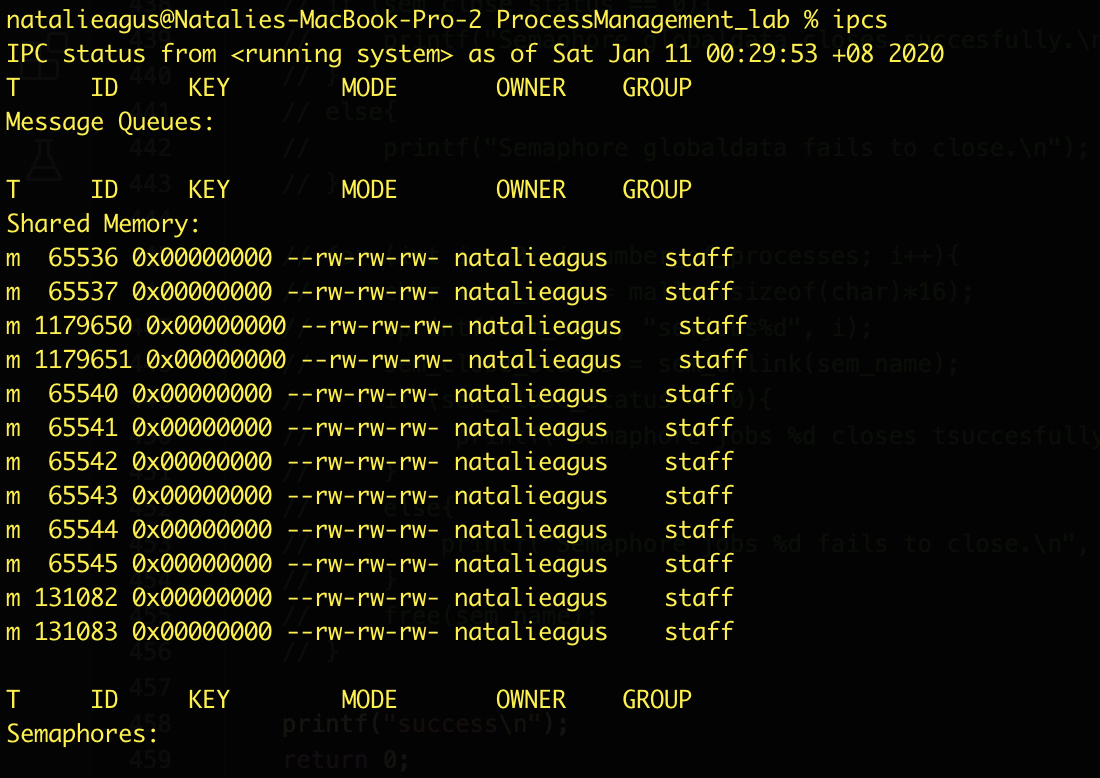
### TODO#5: Implement cleanup() function

**There’s no check file for TODO#4. Finish up the final TODO#5 (don’t worry this last one is ultra easy) and you can check your entire program.**

If all worker processes have successfully terminate, the main process **cleans up all the global variables / shared memory segment,** and **close all the semaphores** by calling cleanup() function:



The answer to this has actually been given to you wholesale in one of the testmain\_todo files given. If you haven’t cleaned up semaphores / segment memory segment properly, you will see it when you run the command ipcs:

****

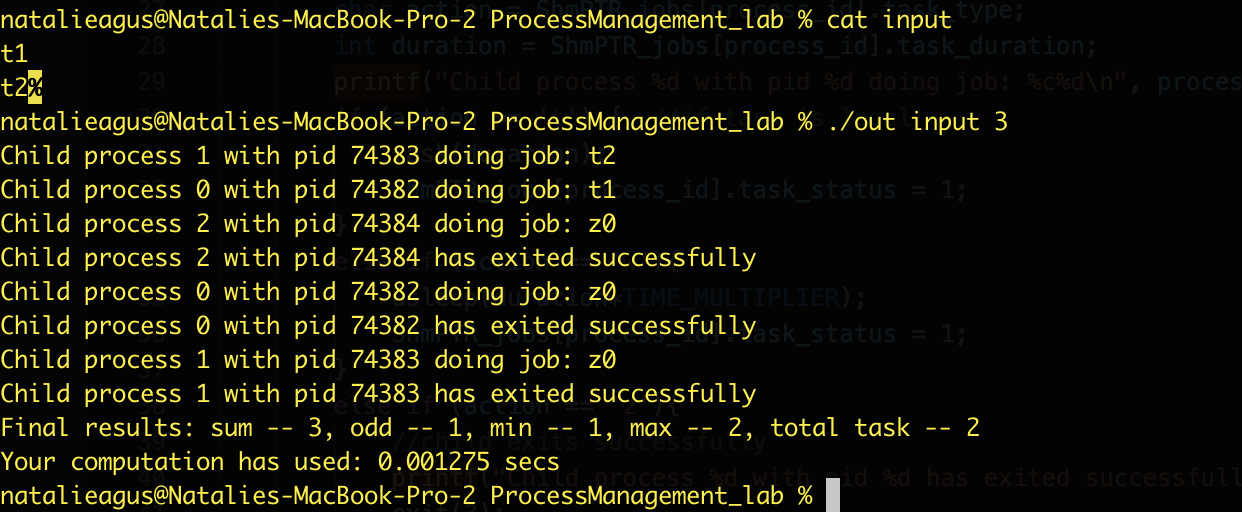
Don’t worry though, you can easily remove these by restarting your machine (or using other commands, you can Google it depending on your machine).

Paste back the original main() function (we give you in testMainCode/realmain.c) and you’re good to check the entire program with the various input files. See [next](#_zhj96lwl5s58) section.

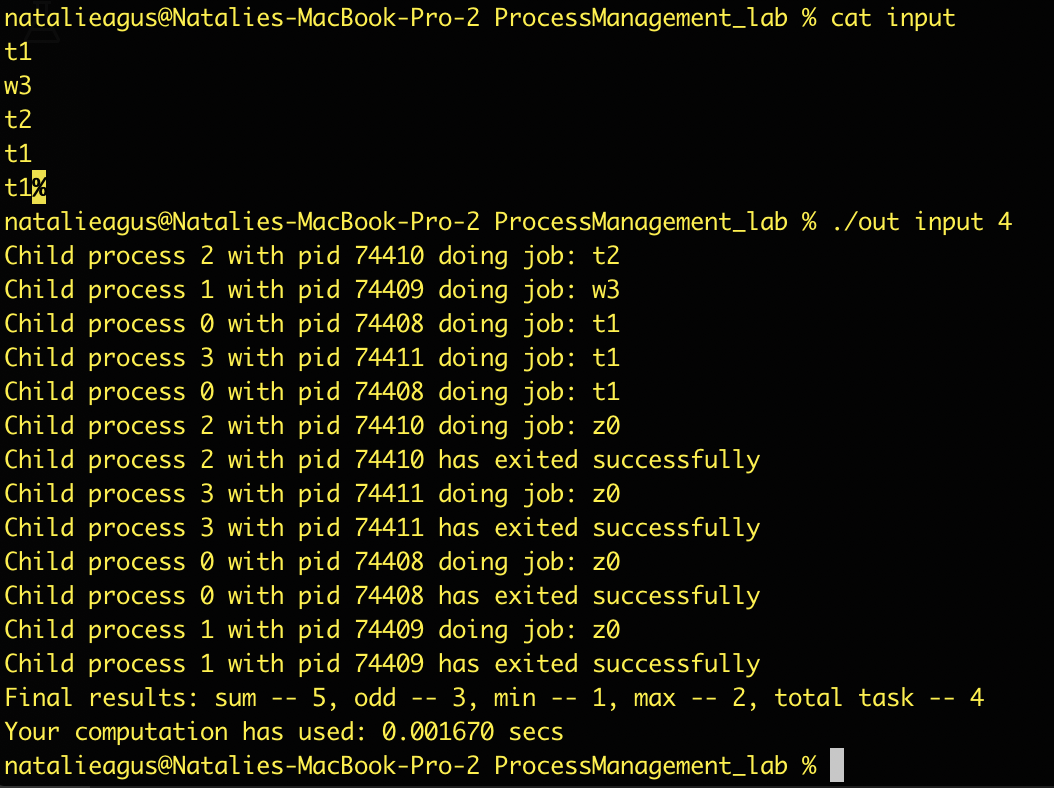
### Validation

Below are the screenshots of the output from various input contents that you can modify yourself to ensure correctness. **Note that the other print messages other than final results and computation time are for debugging and illustration purposes only for this handout.** You don’t have to print the same debug messages if you don’t wish to. **PLEASE REMOVE all debug print messages for submission.** **Failure to do this results in -10 pts.**

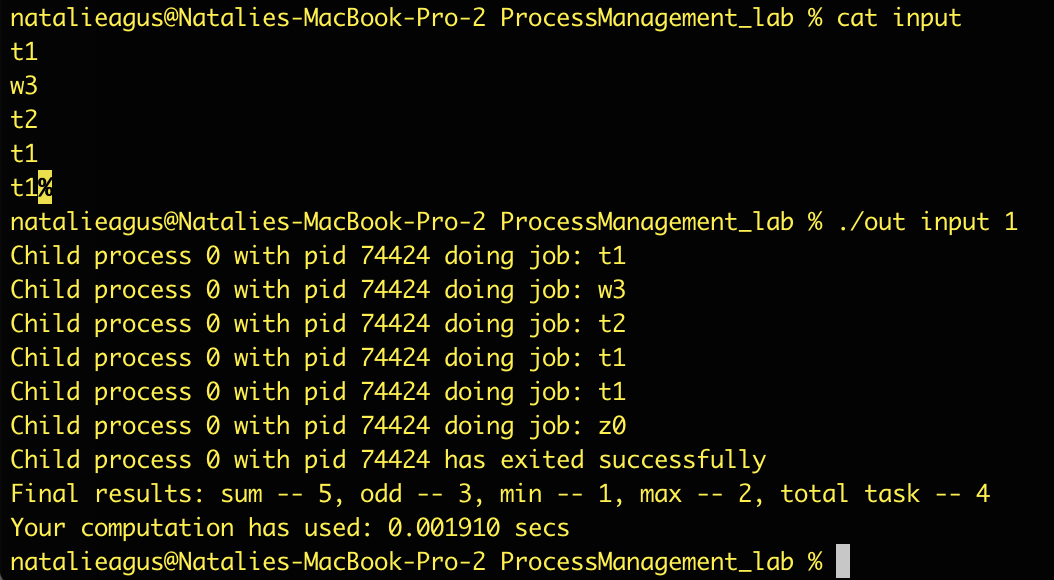
1. An input file with two tasks: t1 and t2, and executing the program with 3 worker processes.
   * Three children worker processes are spawn, locally named as i=0,1,2. P0 does t1, and P1 does t2, while P2 idles.
   * After input is read, the main process disseminate z0.
   * We can see that some degree of parallelism (job dispatching by main process and job execution by worker process) is apparent as P2 (who’s idling) exits *first*.
   * P1 exits last as it does the longest task.



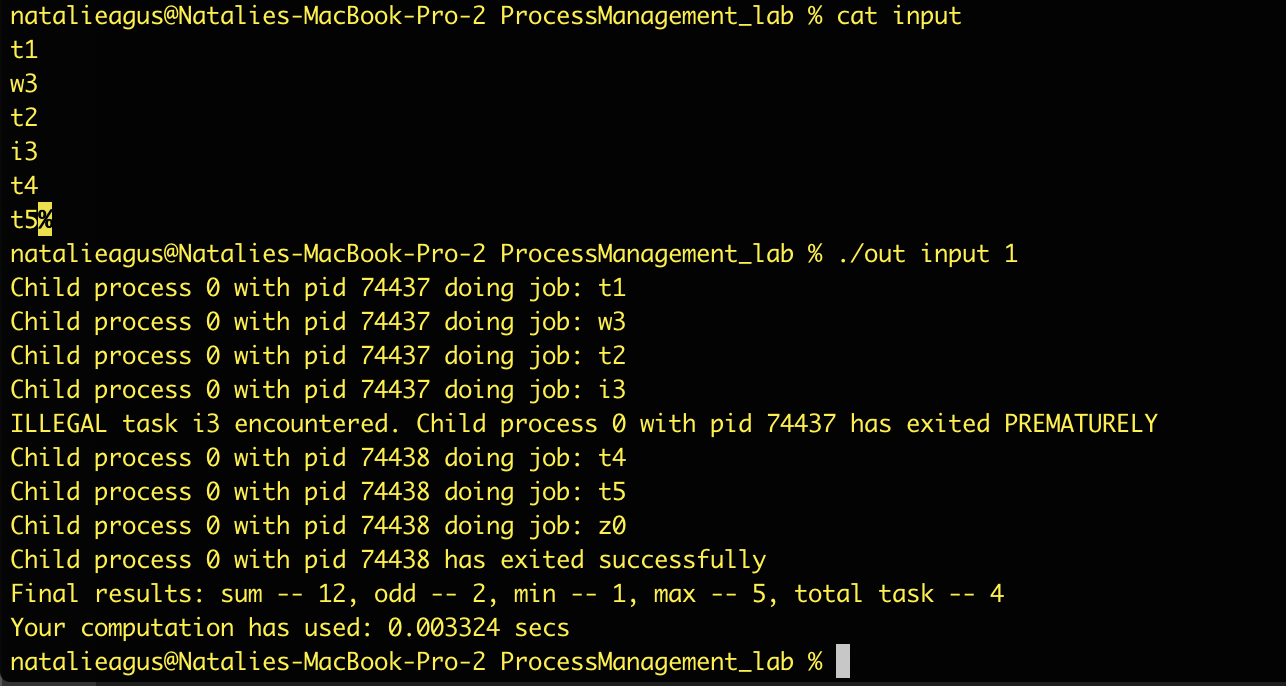
1. Another example with new input and 4 worker processes.



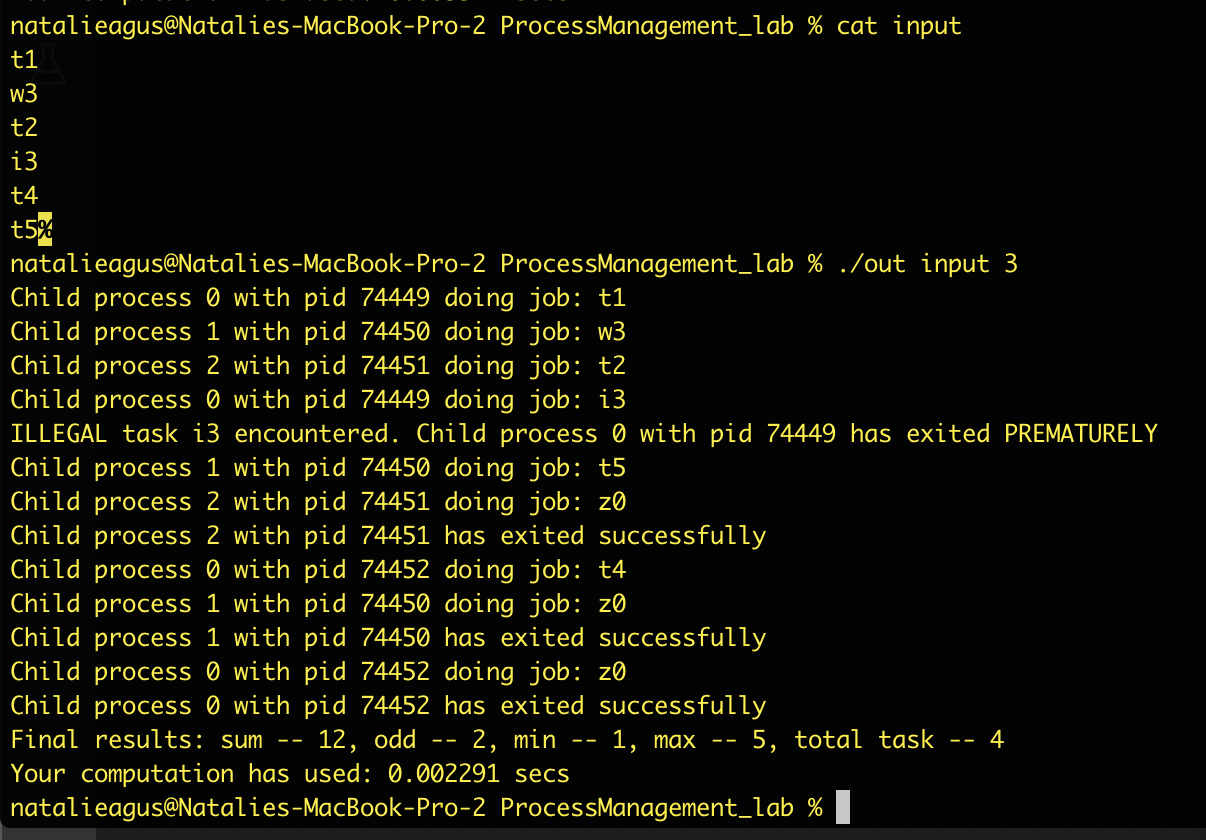
1. The same input in (2) with 1 worker process only.



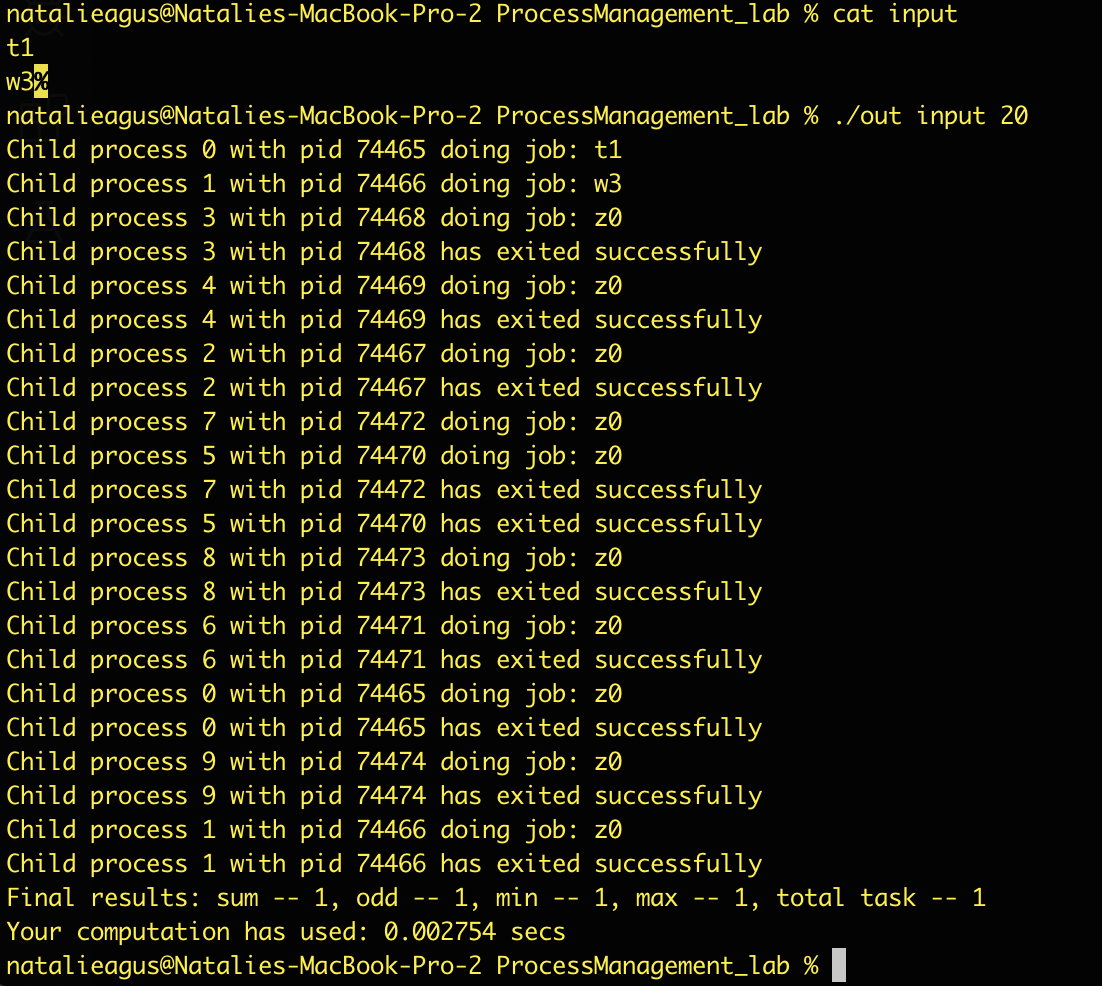
1. An example with illegal input and 1 worker task:



1. The same input file as (4) containing illegal task, and 3 worker processes.



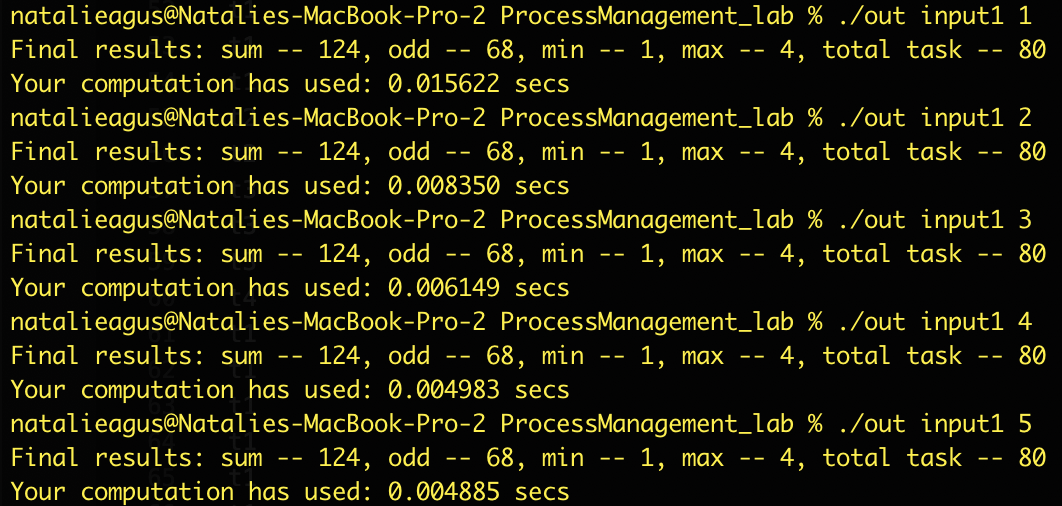
1. Limit the max number of spawned worker process into 10.



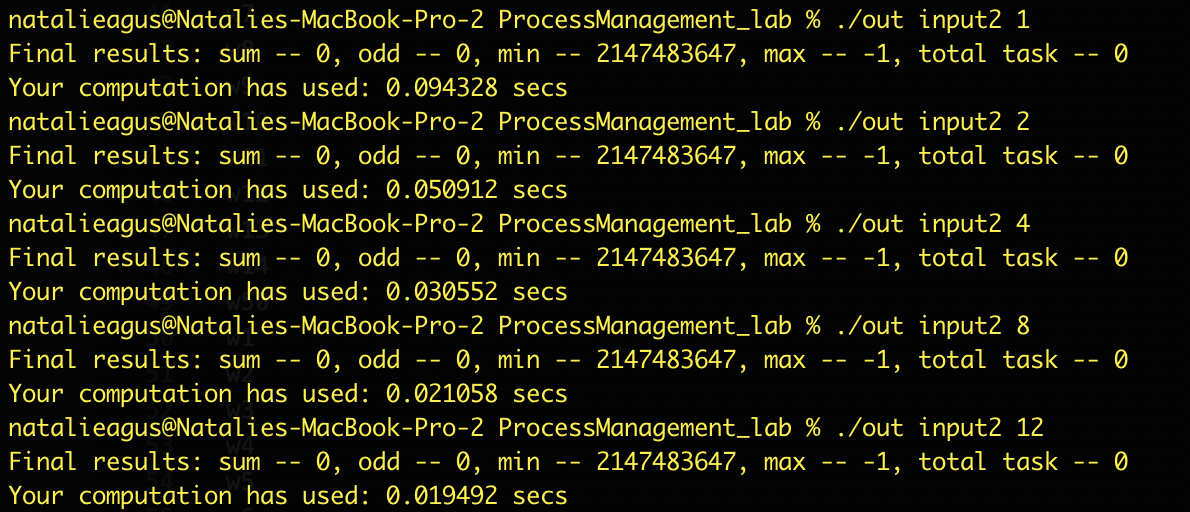
## Test Files

After completing your program implementation and trying out the short input files yourself as shown in the screenshot yourself, see the content of the folder testInputFiles. It contains 5 different input files: input1, input2, input3, input4, and input5. Move them to the ProcessManagement\_lab folder and test them as follows:

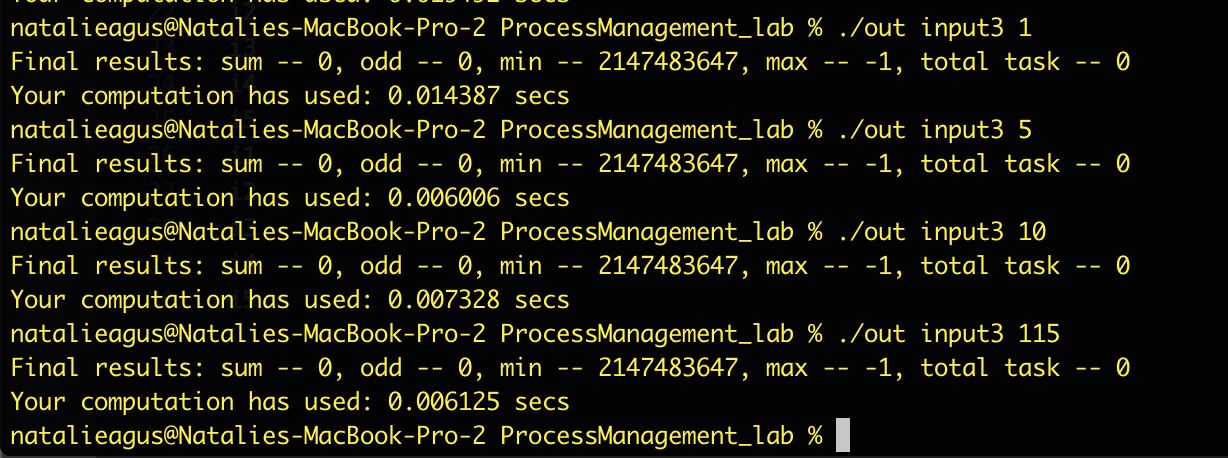
**With input1:**

****

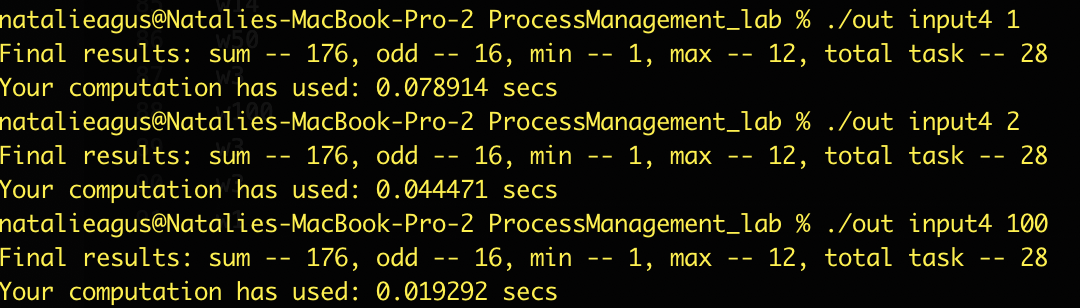
**With input2:**

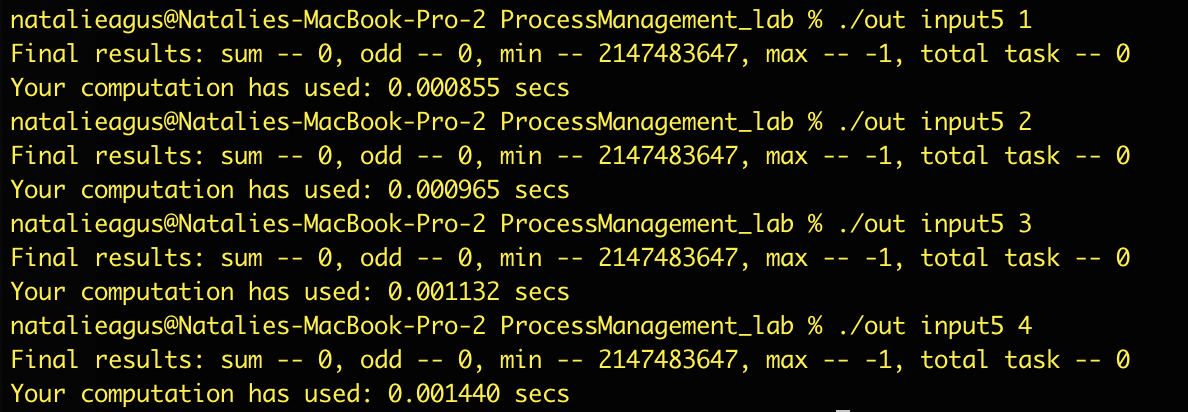
****

**With input3:**



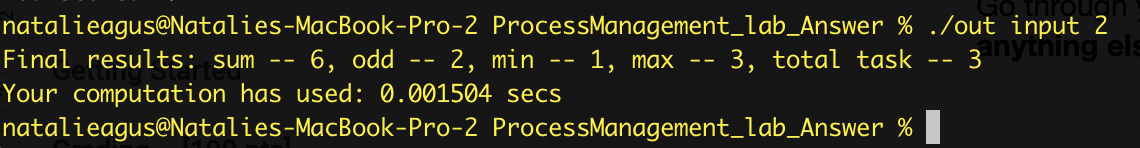
**With input4:**



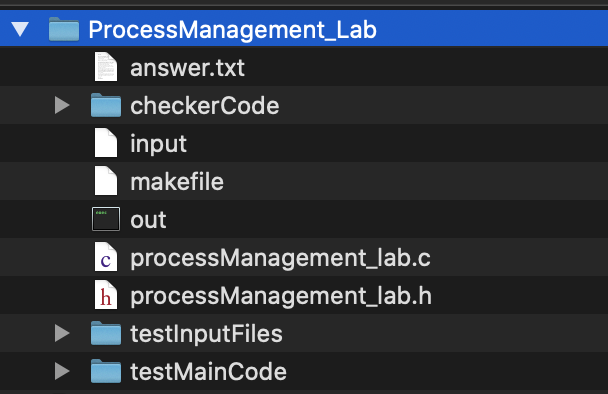
**With input5 (yes this is just an empty file)** 

## Run through checker script

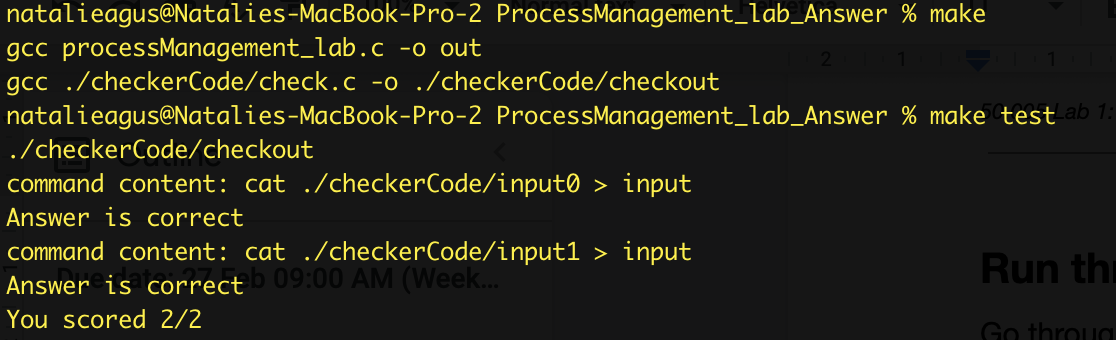
Go through your script and comment out all print statements except THESE TWO BELOW **THAT WAS GIVEN TO YOU IN THE BEGINNING**. Make sure **you don’t print anything else except these two default messages OTHERWISE YOU WILL FAIL THE CHECKER SCRIPT.**

****

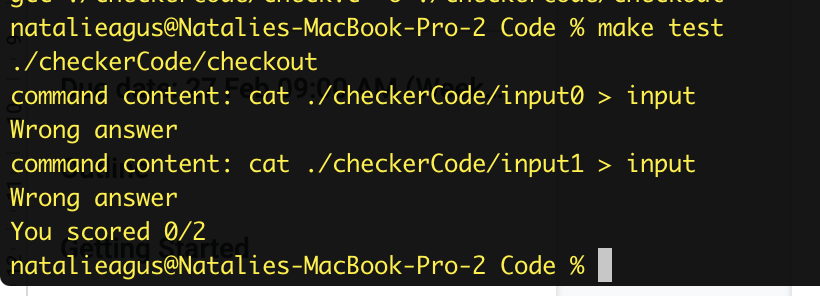
Navigate to the ProcessManagement\_Lab directory containing these files (ensure your current working directory is the ProcessManagement\_Lab), type make to recompile again, and then type make test to run the test script.



Upon success, you will see the following:



If you don’t pass the test case, this is what you’ll see. Most likely is because your program still prints other output. Please carefully comment out **all unnecessary print**. **The checker code checks the strings outputted by your program for correctness.**

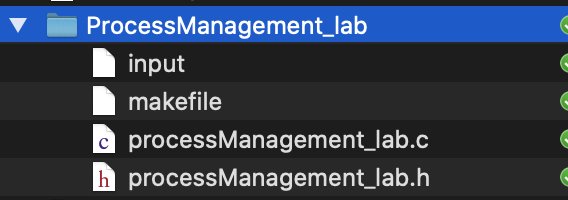


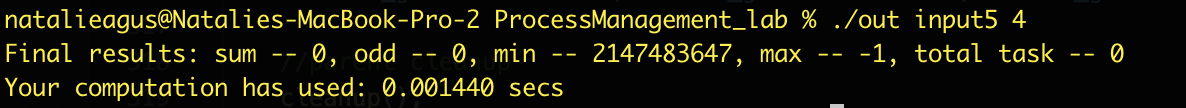
**To grade your file, we will run this checker script with more input test files. It is important to ensure that your script works with this checker. DO NOT modify anything inside the checkerCode file.**

## Submission Rules

**Don’t overcomplicate your solution. My solution contains only about 350 lines, where half of it are comments. Also adhere to all the rules below**

1. **DO NOT modify the makefile and as a result NO other script submission is allowed. We are going to automate the checking of your code with various test input files.**
2. **In case it’s not clear, your submission should ONLY contain these things + README file, zipped together.**



1. **DO NOT MODIFY existing function names and arguments. However, you may add helper functions for your own implementation.**
2. **FOLLOW the implementation guide closely.**
3. **REMOVE all your custom print messages for submission. Only two things must be printed upon each execution of ./out input <num> exactly in these format (in other words, do not change the given printf functions in main):**