Lab 1: Process Management

50.005 Computer System Engineering

Due date: 27 Feb 09:00 AM (Week 5)

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 -- [100 pts]

The points awarded in this lab are written in each of the sections below. This total makes up 4% of your grade.

Submission Procedure

1. **This is an individual assignment.**

2. Write a short README note on how you tackle this segment.

3. Zip back the folder containing the modified starter code + README note in (2) with your answer and upload it to eDimension.

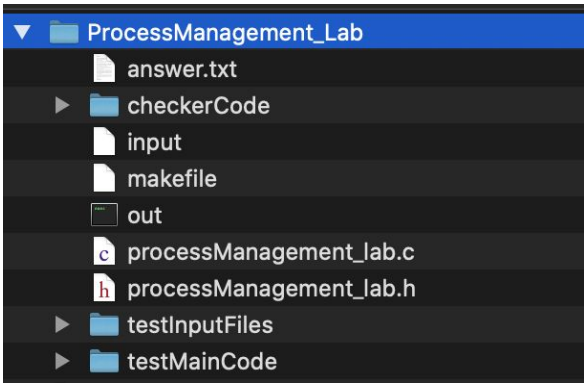
4. **Proceed to edimension to do the quiz.** Treat this like an "online checkoff". **IMPORTANT: If you do not complete the quiz, you will obtain a ZERO grade.**

5. Carefully read the submission rules here.

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:

```
natalie_agus@Natalies-MacBook-Pro ProcessManagement_lab % ./out
Lab 1 Starts...
Usage: sum <infile> <numprocs>
natalie_agus@Natalies-MacBook-Pro ProcessManagement_lab %
```

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

```
natalie_agus@Natalies-MacBook-Pro ProcessManagement_lab_StudentCopy % ./out input 1
Lab 1 Starts...
Main process is going to send termination signals
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.000794 secs
natalie_agus@Natalies-MacBook-Pro ProcessManagement_lab_StudentCopy %
```

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.

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % cat input
t1
w2
t3
i1
i3
i4
```

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 illegal task. Attempts to execute these illegal task 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.

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 1
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.002470 secs
```

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 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 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 respawn another worker process. This new worker process executes i3 and crash 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 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.

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 1
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.002470 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 2
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.001717 secs
```

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 attemtps 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:

```
Your computation has used: 0.001717 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 3
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.001873 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 4
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.001732 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 5
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.001749 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 6
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.001857 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 7
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.001954 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 8
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.002180 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 9
Final results: sum -- 4, odd -- 2, min -- 1, max -- 3, total task -- 2
Your computation has used: 0.002075 secs
```

You can see more of the detailed sample output format in this 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 goes 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
```

Lets 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:

```c
//Check and parse command line options to be in the right format
if (argc < 2) {
    printf("Usage: sum <infile> <numprocs>\n");
    exit(EXIT_FAILURE);
}
```

And also limit the amount of N into 10, storing it into `number_of_processes`:

```c
//Limit number_of_processes into 10.
//If there's no third argument, set the default number_of_processes into 1.
if (argc < 3){
    number_of_processes = 1;
}
else{
    if (atoi(argv[2]) < MAX_PROCESS) number_of_processes = atoi(argv[2]);
    else number_of_processes = MAX_PROCESS;
}
```

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:

a. a `global_data` struct that contains all the required stats (see `processManagement_lab.h`). **DO NOT use your own struct.**

```
/**
 * Data struct containing global_data
 * */
typedef struct global_data{
    long sum_work;
    long odd;
    long min;
    long max;
    long total_tasks;
}global_data;

//shared_nemory global_data to share between children + main processes
int ShmID_global_data;
global_data *ShmPTR_global_data;
sem_t* sem_global_data;
```

b. 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.**

```
/**
 * Data struct contaning job description
 * */
typedef struct job {
    int task_source;
    char task_type;
    int task_duration;
    int task_status;
}job;

//shared_nemory global_jobs to share to child process
int ShmID_jobs;
job *shmPTR_jobs_buffer;
sem_t* sem_jobs_buffer[MAX_PROCESS];
```

c. 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 variable, we cannot share it among process. 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:

```
/**
 * Setup function to create shared mems and semaphores
 * **/
void setup(){

    // TODO#1:  a. Create shared memory for global_data struct (see processManagement_lab.h)
    //          b. When shared memory is successfully created, set the initial values of "max" and "min"
    of the global_data struct in the shared memory accordingly
    // To bring you up to speed, (a) and (b) are given to you already. Please study how it works.

    //          c. Create semaphore of value 1 which purpose is to protect this global_data struct in
    shared memory
    //          d. Create shared memory for number_of_processes job struct (see processManagement_lab.h)
    //          e. When shared memory is successfully created, setup the content of the structs (see
    handout)
    //          f. Create number_of_processes semaphores of value 0 each to protect each job struct in
    the shared memory. Store the returned pointer by sem_open in sem_jobs_buffer[i]
    //          g. Return to main
```

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 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!!):

```
// allocate a new shared memory segment that can contain 256 characters
int shmID = shmget(IPC_PRIVATE, sizeof(char) * 256, IPC_CREAT | 0666);
//check for errors
if (shmID == -1){
    perror("Failed to create shared memory\n");
    exit(EXIT_FAILURE);
}
//if no error, attach the segment
char* shmPTR = (char *) shmat(shmID, NULL, 0);
//check for error again
if ((int) shmPTR == -1){
    perror("Fail to attach the memory to this address space\n");
    exit(EXIT_FAILURE);
}
```

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!!):**

```c
//create a semaphore with name "mysemaphore" of init size 1. Semaphores must be unique by name.
sem_t* semaphore = sem_open("mysemaphore", O_CREAT | O_EXCL, 0644, 1);
//check if failed,
while (true){
    if (semaphore == SEM_FAILED){
        //try to unlink, chances are it failed because there's already a semaphore of such name (maybe from your
        previously executed program, etc)
        sem_unlink("mysemaphore");
        //try to open again
        semaphore = sem_open("mysemaphore", O_CREAT | O_EXCL, 0644, 1);
    }
    else{
        break;
    }
}
```

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:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 2
Number of processes: 2
Parent write job 0 with duration 0, status 0
Parent write job 1 with duration 2, status 0
Child receives job duration from parent: 0, status 0
Child receives job duration from parent: 2, status 0
Job 0  cleared by children. Duration: -1, status -1
Job 1  cleared by children. Duration: -1, status -1
Semaphore globaldata closes succesfully.
Semaphore jobs 0 closes succesfully.
Semaphore jobs 1 closes succesfully.
success
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 3
Number of processes: 3
Parent write job 0 with duration 0, status 0
Parent write job 1 with duration 2, status 0
Parent write job 2 with duration 4, status 0
Child receives job duration from parent: 0, status 0
Child receives job duration from parent: 2, status 0
Child receives job duration from parent: 4, status 0
Job 0  cleared by children. Duration: -1, status -1
Job 1  cleared by children. Duration: -1, status -1
Job 2  cleared by children. Duration: -1, status -1
Semaphore globaldata closes succesfully.
Semaphore jobs 0 closes succesfully.
Semaphore jobs 1 closes succesfully.
Semaphore jobs 2 closes succesfully.
success
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab %
```

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:

```
/**
 * Function to spawn all required children processes
 **/

void createchildren(){
    // TODO#2:  a. Create number_of_processes children processes
    //          b. Store the pid_t of children i at children_processes[i]
    //          c. For child process, invoke the method job_dispatch(i)
    //          d. For the parent process, continue creating the next children
    //          e. After number_of_processes children are created, return to main

    return;
}
```

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:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 2
Number of processes: 2
Main process pid 851
Child process 0 created with pid: 852
Hello from child 0 with pid 852 and parent id 851
Hello from child 1 with pid 853 and parent id 851
Child process 1 created with pid: 853
success
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 5
Number of processes: 5
Main process pid 864
Hello from child 0 with pid 865 and parent id 864
Hello from child 1 with pid 866 and parent id 864
Hello from child 2 with pid 867 and parent id 864
Hello from child 3 with pid 868 and parent id 864
Child process 0 created with pid: 865
Child process 1 created with pid: 866
Child process 2 created with pid: 867
Child process 3 created with pid: 868
Child process 4 created with pid: 869
Hello from child 4 with pid 869 and parent id 864
success
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab %
```

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 input file, read it, and starts 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` are executing `job_dispatch()` function.

This function instructs 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 it finishes executing, this worker process clears 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, it examines `shmPTR_jobs_buffer[i]` again for new jobs. If there's no job, the worker process calls `sem_wait(sem_jobs_buffer[i])`. Worker process **cannot busy wait, e.g: loop over and over to check for new job.** Use the worker child process' semaphore `sem_wait(sem_jobs_buffer[i])` for this. **If you 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 condition. If you face ANY race condition, -40 pts.**

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.

```
/**
 * The function that is executed by each worker process to execute any available job given by the main process
 * */
void job_dispatch(int i){

    // TODO#3:  a. Always check the corresponding shmPTR_jobs_buffer[i] for new  jobs from the main process
    //          b. Use semaphore so that you don't busy wait
    //          c. If there's new job, execute the job accordingly: either by calling task(), usleep, exit(3) or kill
    (getpid(), SIGKILL)
    //          d. Loop back to check for new job


    // printf("Hello from child %d with pid %d and parent id %d\n", i, getpid(), getppid());
    // exit(0);

}
```

Dont forget to fix the task() function to be safe from race condition:

```
/**
 * The task function to simulate "work" for each worker process
 * TODO#3: Modify the function to be multiprocess-safe
 * */
void task(long duration)
{
    // simulate computation for x number of seconds
    usleep(duration*TIME_MULTIPLIER);

    // TODO: protect the access of shared variable below
    // update global variables to simulate statistics
    ShmPTR_global_data->sum_work += duration;
    ShmPTR_global_data->total_tasks ++;
    if (duration % 2 == 1) {
        ShmPTR_global_data->odd++;
    }
    if (duration < ShmPTR_global_data->min) {
        ShmPTR_global_data->min = duration;
    }
    if (duration > ShmPTR_global_data->max) {
        ShmPTR_global_data->max = duration;
    }
}
```

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`.
- **UPDATE 16/02/2020: PLEASE DOWNLOAD THE NEW `testmain_todo3.c` file.** The old one didn't print the print output and mistakenly set new `task_status` jobs as `0` instead of `1`, so if you checked that in the child process even after calling sem_wait, it might cause your code to stuck.

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:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab_Playground % ./out input 3
Number of processes: 3
Main process pid 1813
Child 0 with pid 1814 has exited successfully
Child 1 with pid 1815 has exited successfully
Child 2 with pid 1816 has exited successfully
Final results: sum -- 3, odd -- 3, min -- 1, max -- 1, total task -- 3
Semaphore globaldata closes succesfully.
Semaphore jobs 0 closes succesfully.
Semaphore jobs 1 closes succesfully.
Semaphore jobs 2 closes succesfully.
success
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab_Playground % ./out input 5
Number of processes: 5
Main process pid 1817
Child 0 with pid 1818 has exited successfully
Child 1 with pid 1819 has exited successfully
Child 2 with pid 1820 has exited successfully
Child 3 with pid 1821 has exited successfully
Child 4 with pid 1822 has exited successfully
Final results: sum -- 5, odd -- 5, min -- 1, max -- 1, total task -- 5
Semaphore globaldata closes succesfully.
Semaphore jobs 0 closes succesfully.
Semaphore jobs 1 closes succesfully.
Semaphore jobs 2 closes succesfully.
Semaphore jobs 3 closes succesfully.
Semaphore jobs 4 closes succesfully.
success
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab_Playground %
```

Don't worry that the global stats does not reflect what's in 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 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 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:

```
void cleanup(){
    //TODO#4:
    // 1. Detach both shared memory (global_data and jobs)
    // 2. Delete both shared memory (global_data and jobs)
    // 3. Unlink all semaphores in sem_jobs_buffer
}
```

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`:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ipcs
IPC status from <running system> as of Sat Jan 11 00:29:53 +08 2020
T     ID     KEY        MODE       OWNER    GROUP
Message Queues:

T     ID     KEY        MODE       OWNER    GROUP
Shared Memory:
m  65536 0x00000000 --rw-rw-rw- natalieagus   staff
m  65537 0x00000000 --rw-rw-rw- natalieagus   staff
m 1179650 0x00000000 --rw-rw-rw- natalieagus    staff
m 1179651 0x00000000 --rw-rw-rw- natalieagus    staff
m  65540 0x00000000 --rw-rw-rw- natalieagus    staff
m  65541 0x00000000 --rw-rw-rw- natalieagus    staff
m  65542 0x00000000 --rw-rw-rw- natalieagus    staff
m  65543 0x00000000 --rw-rw-rw- natalieagus    staff
m  65544 0x00000000 --rw-rw-rw- natalieagus    staff
m  65545 0x00000000 --rw-rw-rw- natalieagus    staff
m 131082 0x00000000 --rw-rw-rw- natalieagus    staff
m 131083 0x00000000 --rw-rw-rw- natalieagus    staff

T     ID     KEY        MODE       OWNER    GROUP
Semaphores:
```

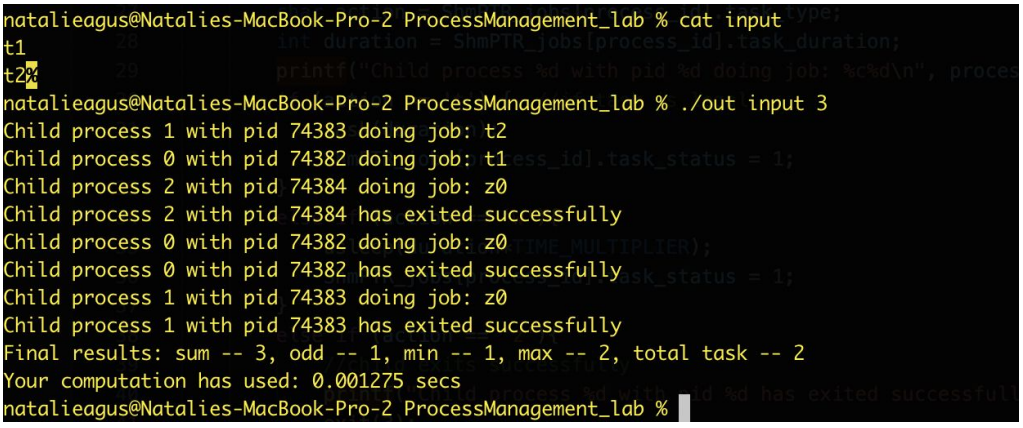
Don't worry though, you can easily remove these by restarting your machine (or using other commands, you can Google it depending on your machine).

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 section.

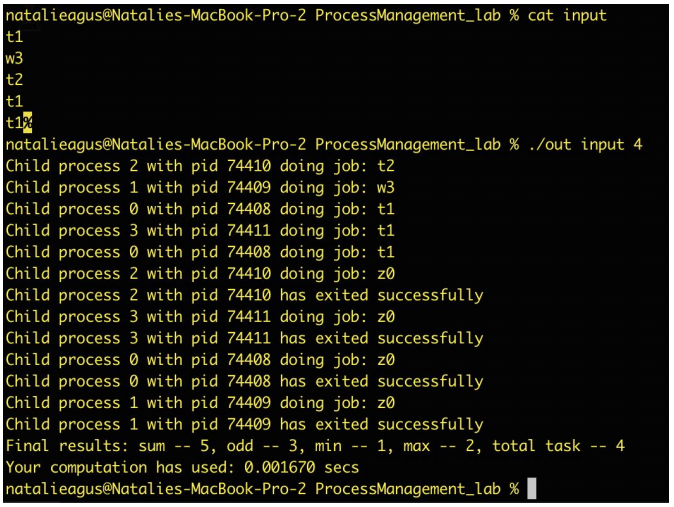
Validation

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.



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



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

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % cat input
t1
w3
t2
t1
t1%
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 1
Child process 0 with pid 74424 doing job: t1
Child process 0 with pid 74424 doing job: w3
Child process 0 with pid 74424 doing job: t2
Child process 0 with pid 74424 doing job: t1
Child process 0 with pid 74424 doing job: t1
Child process 0 with pid 74424 doing job: z0
Child process 0 with pid 74424 has exited successfully
Final results: sum -- 5, odd -- 3, min -- 1, max -- 2, total task -- 4
Your computation has used: 0.001910 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab %
```

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

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % cat input
t1
w3
t2
i3
t4
t5%
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 1
Child process 0 with pid 74437 doing job: t1
Child process 0 with pid 74437 doing job: w3
Child process 0 with pid 74437 doing job: t2
Child process 0 with pid 74437 doing job: i3
ILLEGAL task i3 encountered. Child process 0 with pid 74437 has exited PREMATURELY
Child process 0 with pid 74438 doing job: t4
Child process 0 with pid 74438 doing job: t5
Child process 0 with pid 74438 doing job: z0
Child process 0 with pid 74438 has exited successfully
Final results: sum -- 12, odd -- 2, min -- 1, max -- 5, total task -- 4
Your computation has used: 0.003324 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab %
```

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

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % cat input
t1
w3
t2
i3
t4
t5%
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 3
Child process 0 with pid 74449 doing job: t1
Child process 1 with pid 74450 doing job: w3
Child process 2 with pid 74451 doing job: t2
Child process 0 with pid 74449 doing job: i3
ILLEGAL task i3 encountered. Child process 0 with pid 74449 has exited PREMATURELY
Child process 1 with pid 74450 doing job: t5
Child process 2 with pid 74451 doing job: z0
Child process 2 with pid 74451 has exited successfully
Child process 0 with pid 74452 doing job: t4
Child process 1 with pid 74450 doing job: z0
Child process 1 with pid 74450 has exited successfully
Child process 0 with pid 74452 doing job: z0
Child process 0 with pid 74452 has exited successfully
Final results: sum -- 12, odd -- 2, min -- 1, max -- 5, total task -- 4
Your computation has used: 0.002291 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab %
```

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

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % cat input
t1
w3%
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input 20
Child process 0 with pid 74465 doing job: t1
Child process 1 with pid 74466 doing job: w3
Child process 3 with pid 74468 doing job: z0
Child process 3 with pid 74468 has exited successfully
Child process 4 with pid 74469 doing job: z0
Child process 4 with pid 74469 has exited successfully
Child process 2 with pid 74467 doing job: z0
Child process 2 with pid 74467 has exited successfully
Child process 7 with pid 74472 doing job: z0
Child process 5 with pid 74470 doing job: z0
Child process 7 with pid 74472 has exited successfully
Child process 5 with pid 74470 has exited successfully
Child process 8 with pid 74473 doing job: z0
Child process 8 with pid 74473 has exited successfully
Child process 6 with pid 74471 doing job: z0
Child process 6 with pid 74471 has exited successfully
Child process 0 with pid 74465 doing job: z0
Child process 0 with pid 74465 has exited successfully
Child process 9 with pid 74474 doing job: z0
Child process 9 with pid 74474 has exited successfully
Child process 1 with pid 74466 doing job: z0
Child process 1 with pid 74466 has exited successfully
Final results: sum -- 1, odd -- 1, min -- 1, max -- 1, total task -- 1
Your computation has used: 0.002754 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab %
```

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:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input1 1
Final results: sum -- 124, odd -- 68, min -- 1, max -- 4, total task -- 80
Your computation has used: 0.015622 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input1 2
Final results: sum -- 124, odd -- 68, min -- 1, max -- 4, total task -- 80
Your computation has used: 0.008350 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input1 3
Final results: sum -- 124, odd -- 68, min -- 1, max -- 4, total task -- 80
Your computation has used: 0.006149 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input1 4
Final results: sum -- 124, odd -- 68, min -- 1, max -- 4, total task -- 80
Your computation has used: 0.004983 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input1 5
Final results: sum -- 124, odd -- 68, min -- 1, max -- 4, total task -- 80
Your computation has used: 0.004885 secs
```

With input2:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input2 1
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.094328 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input2 2
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.050912 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input2 4
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.030552 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input2 8
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.021058 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input2 12
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.019492 secs
```

With input3:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input3 1
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.014387 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input3 5
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.006006 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input3 10
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.007328 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input3 115
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.006125 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab %
```

With input4:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input4 1
Final results: sum -- 176, odd -- 16, min -- 1, max -- 12, total task -- 28
Your computation has used: 0.078914 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input4 2
Final results: sum -- 176, odd -- 16, min -- 1, max -- 12, total task -- 28
Your computation has used: 0.044471 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input4 100
Final results: sum -- 176, odd -- 16, min -- 1, max -- 12, total task -- 28
Your computation has used: 0.019292 secs
```

With input5 (yes this is just an empty file)

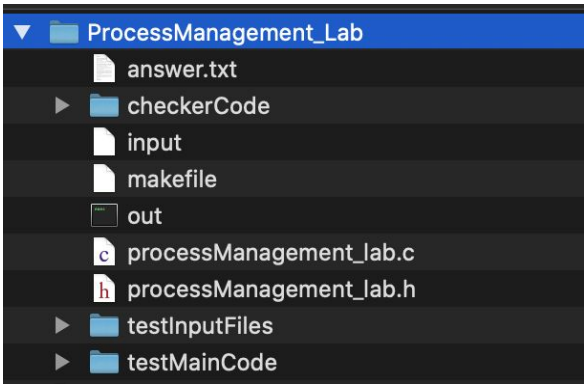
```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input5 1
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.000855 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input5 2
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.000965 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input5 3
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.001132 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input5 4
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.001440 secs
```

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.**

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab_Answer % ./out input 2
Final results: sum -- 6, odd -- 2, min -- 1, max -- 3, total task -- 3
Your computation has used: 0.001504 secs
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab_Answer %
```

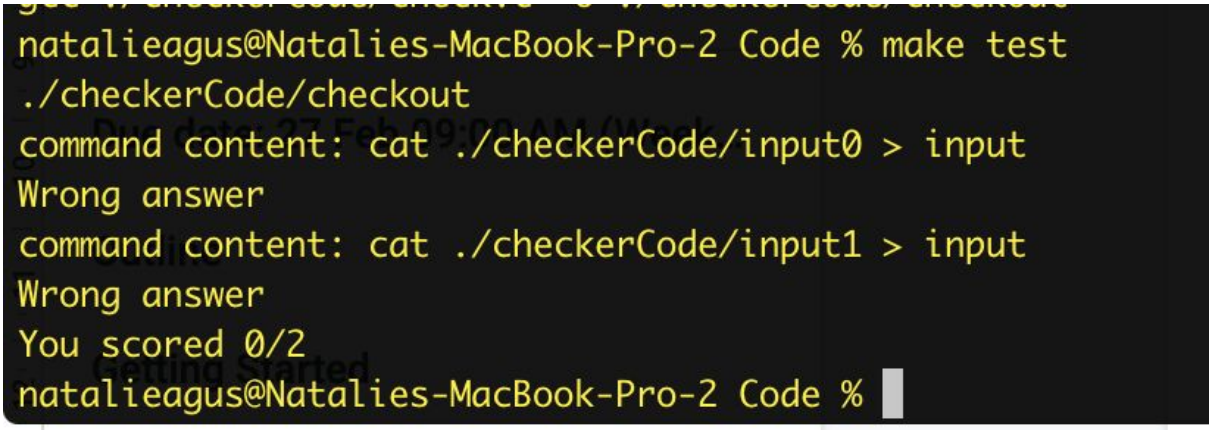
Navigate to the ProcessManagement_Lab directory containing these files (ensure your current working directory is the ProcessManagement_Lab), type `make` to recompile again, and then type `make test` to run the test script.



Upon success, you will see the following:

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab_Answer % make
gcc processManagement_lab.c -o out
gcc ./checkerCode/check.c -o ./checkerCode/checkout
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab_Answer % make test
./checkerCode/checkout
command content: cat ./checkerCode/input0 > input
Answer is correct
command content: cat ./checkerCode/input1 > input
Answer is correct
You scored 2/2
```

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.



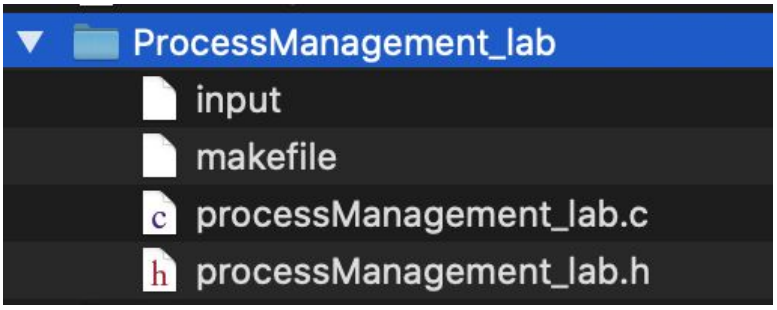
To grade your file, we will run this checker script with more input test files. It is important to ensure that your script works with this checker. DO NOT modify anything inside the checkerCode file.

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.**



3. **DO NOT MODIFY existing function names and arguments. However, you may add helper functions for your own implementation.**

4. **FOLLOW the implementation guide closely.**

5. **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`):**

```
natalieagus@Natalies-MacBook-Pro-2 ProcessManagement_lab % ./out input5 4
Final results: sum -- 0, odd -- 0, min -- 2147483647, max -- -1, total task -- 0
Your computation has used: 0.001440 secs
```