CS 367 Project #1 – Spring 2021:
CPU Process Scheduler
Due: Friday, February 12 at 11:59pm

This is to be an individual effort. No partners.
No late work allowed after 48 hours; each day late automatically uses up one of your tokens.

Key Topics: Basic C Programming, Singly Linked List Operations, Bitwise Operations

1 Introduction

You will finish **scheduler.c** to implement functions that will be called from an Operating System Simulator. Your functions will be used to schedule processes to run on a virtual CPU. The Simulator will create processes and handle inputs, but **it will call your functions** for scheduling.

Problem Background *(You'll study schedulers like this in CS471!)*

A major feature of most operating systems is to allow multiple **processes** (running programs) to run on a single CPU. The OS manages these processes by putting them into one of several **scheduler queues**, which are implemented as singly linked lists.

For each time unit on the computer, the OS chooses the next process to run from the **ready queue**, which contains all ready processes. This process will run for a set amount of time, which uses a hardware timer to keep track of.

When the timer expires, if the running process did not finish, it will go back to the **ready queue** to get scheduled again. A process may also need to **wait** for data to be transferred to or from an Input/Output (I/O) device, so a running process may need to be moved to the **waiting queue** until that data is ready. Of course, a process also may be finished, so it can just be terminated.

One other thing is that the user may press Control-Z to **stop** a running process, this would move into the **stopped queue**, where it won't run. If someone calls **continue** on that process, then it can go back into the ready queue, so it can be selected again.

Organization (How to read this specification)

- Section 2 is what the whole program does for context and process struct details.
- Section 3 is how to build the whole program.
- Section 4 details what you have to write in your **scheduler.c** file.
- Section 5 is tips on how to approach this project.
- Section 6 is Trace Files, which let you see the simulator running for debug purposes.
- Section 7 is Submission Instructions.

2 Project Overview

Project Particulars

Your project will implement several of the operations related to the CPU Scheduler. This project will require the implementation and use of **three singly linked lists** to create and store process structs, to move process structs between lists, to add/search/remove process structs from the lists, and to perform basic bitwise operations to manage flags and combined data.

Your code (**scheduler.c**) will work with pre-written files from an OS simulator to implement several of these operations. You will be maintaining all three singly linked lists (**ready queue**, **stopped queue**, and **waiting queue**). The structs for these lists are all defined in **structs.h**.

The bottom line is the simulator will call your functions, and your code will do that operation.

2.1 OS Simulator Overview

The OS Simulator simulates running an Operating System by running processes. The way a real OS does this is by selecting a ready process to run, running it for a small period of time, and then putting that process back into the ready queue and selecting a new process to run.

The OS Simulator will call your functions in **scheduler.c** to do some of these tasks. You will not have to design a simulator, that's already done for you. You will just need to implement a series of functions to make the simulator work.

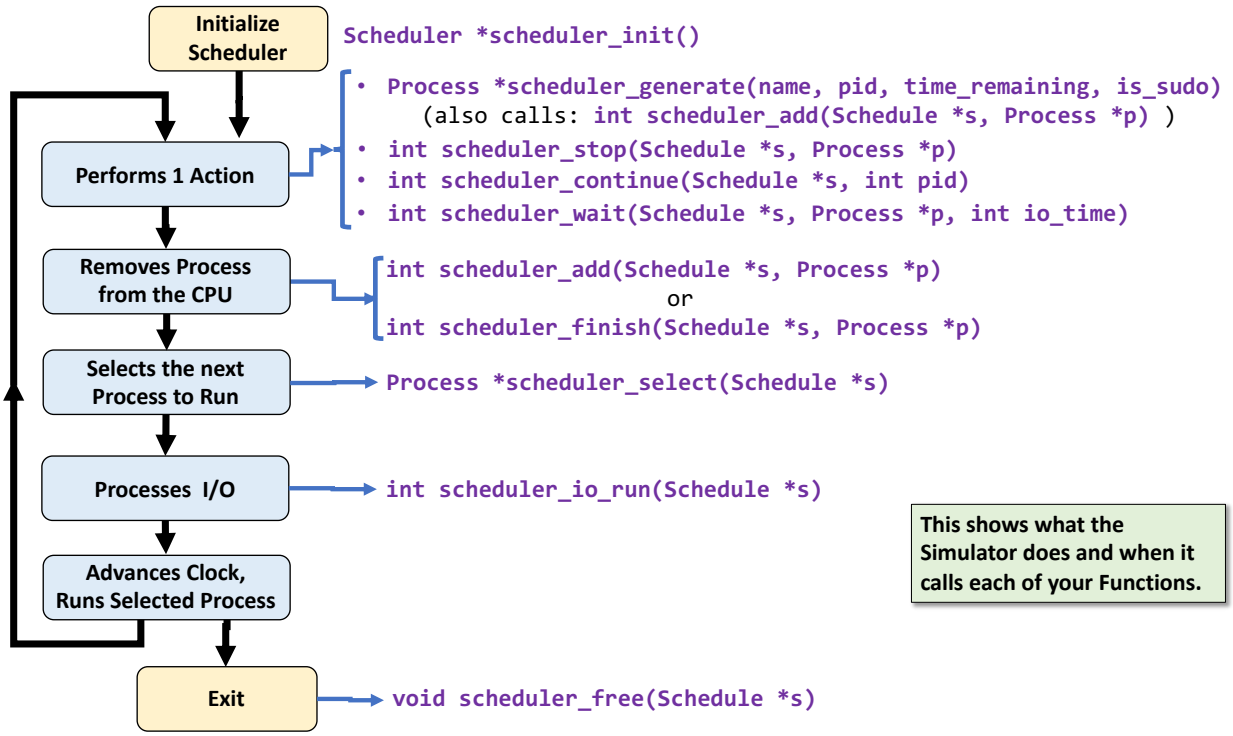
You will be creating and maintaining three linked lists for this project. Each of these are already defined for you in **structs.h** and are detailed here in Section 2.5. The **Schedule struct** has pointers to your three queues (linked lists) in it. The **Queue struct** has a pointer to the head, which is the first node of your linked list (no dummy nodes), and a count of the number of nodes in the list. The **Process struct** contains all the data for each process.

Each Process has a Process ID field called `pid`, which is an int. This is the **Process ID** number that the OS uses to uniquely identify a specific process; each process will have its own unique `pid`. Some of your functions will use the pid field to perform an operation on a process.

This simulator works by a simple loop of operations that is based on a clock. The **Clock** is a simple counter that starts at 1 and runs until the OS is halted with an exit command. Each cycle through the loop takes 1 time unit, and all of the processes use `time_remaining` in terms of the same time units. The Simulator will adjust the clock when it runs your process. All you need to do is look at the `time_remaining` value of each process to make some of your decisions. These are described in each function.

2.2 **OS Simulator Operational Loop** *(How does the Simulator Work at the High Level?)*

The OS Simulator is written for you and uses the following loop. For each step in this loop (represented by boxes), the OS simulator will call one of your functions, as appropriate. The function prototypes on the right are the ones you will be writing for this project.



Example: Let's say that we have only one process running on the CPU and nothing in the ready queue at the end of Time 2. The following events will happen:

- At the beginning of the cycle, on Time 3, the Simulator will do one of the actions to run and call your function for that action.
  - As an example, it may call the **`scheduler_continue()`** function.
- Next, the Simulator will call your **`scheduler_add`** function to add the running process back into your queue. *(Details about how to write each function is coming)*.
- After that, it will call your **`scheduler_select`** function to have you choose the next process for it to run on the CPU for 1 time unit.
- Then, it will call your **`scheduler_io_run`** function to have you update the waiting time of the first process in the waiting queue and, if there is no wait time left on that process, your function will move that process back to the Ready Queue.
- Then, the Simulator itself will run the process **`scheduler_select`** returned on the CPU for 1 time unit and update all the clocks and the **`time_remaining`** on the running process.
- Finally, it will either go back to the top to pick a new action or exit.

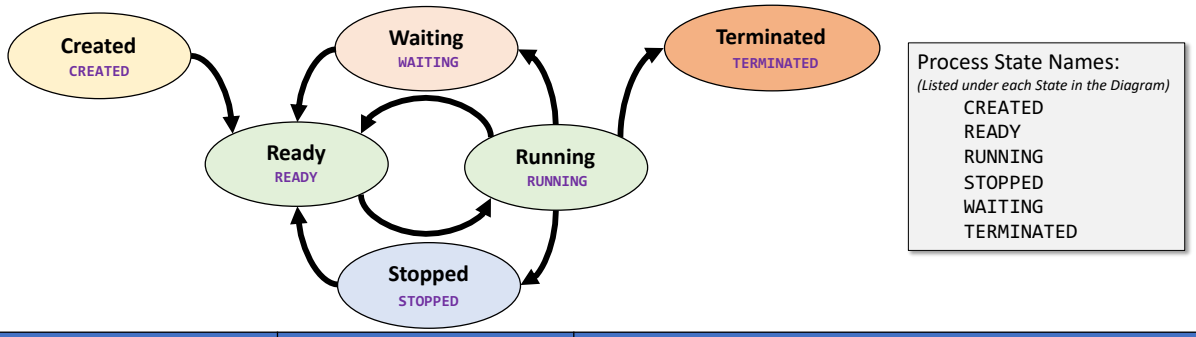
Remember, you are not implementing any of the Simulator code or any of this main cycle. You will be writing a series of functions that will be called by the Simulator. Your functions will add new features to an existing Simulator.

The file you will be finishing, **scheduler.c**, is already started for you, with a signature (stub) for each function you need to write. You will need to write these functions (and any other functions that you may want in order to fit your design).

2.3 Process States *(How does a process struct know what status it is in?)*

Each process will be in one of your three linked lists (queues), but each process also has its own internal state, which represents how it was last running on the CPU.

Every process can be in exactly one of six possible **states** at any given time. The following diagram shows the six states of a process, and how they can change to other states.



Simulator Calls your Function	Process State Change	Queue Action
`scheduler_add()`	`Created -> Ready`	Process is Added to the Ready Queue
`scheduler_select()`	`Ready -> Running`	Process removed from Ready Queue and Returned
`scheduler_add()`	`Running -> Ready`	Process is Added to the Ready Queue
`scheduler_wait()`	`Running -> Waiting`	Process is Added to the Waiting Queue
`scheduler_stop()`	`Running -> Stopped`	Process is Added to Stopped Queue
`scheduler_continue()`	`Stopped -> Ready`	Process is Moved from Stopped to Ready Queue
`scheduler_io_run()`	`Waiting -> Ready`	Process may Move from Waiting to Ready Queue
`scheduler_finish()`	`Terminated (No Change Needed)`	None. Simulator sets TERMINATED when the process finishes and then calls scheduler_finish for you to free it. Not in a Queue.

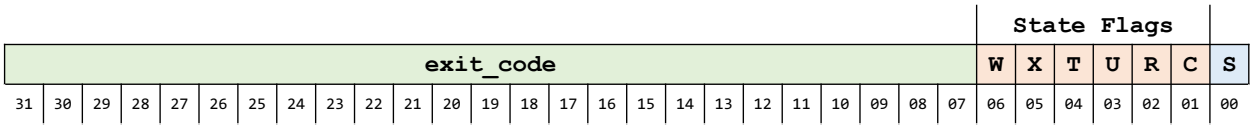
This diagram shows the states that each Process can be in and which function that changes the states. As an example, if a Process is in the Running State (`RUNNING`), and the simulator calls your `scheduler_stop()` function, then that Process will change to the Stopped State (`STOPPED`).

Each state generally also matches where the process is in the different queues. For example, a Ready process will be in the **Ready Queue**. A Waiting process will be in the **Waiting Queue**, and a Stopped process will be in the **Stopped Queue.**

For the other states, a Created Process is not in any queue, but will be passed to you when the Simulator calls your `scheduler_add` function. A Running process is also not in any queue but is loaded on to the CPU by the Simulator. A Terminated process is also not in any queue as it will be freed by your `scheduler_finish` function.

2.4 Process Flags *(Ok, but there's no state field in the struct. How are they stored?)*

The `Process` structs maintain their current state using the `flags` field. This is a 32-bit int that contains pieces of information, which have been combined together using bitwise operations.



(Note: the numbers on the bottom just show the bit position, so you can see each of the 32 bits)

Bit 0 is a 1-bit flag saying if the process was run with super user privileges (Called using sudo)
S = `SUDO`

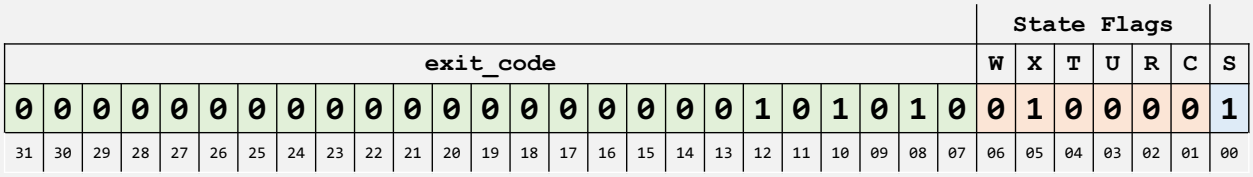
Bits 1-6 represent the current state of the Process.
These are stored as 1-bit values (0 for No, 1 for Yes). (Hint: Use bitwise operations!)
Note: Only one of these bits will be set to 1 at any time. All of the others must be a 0!

C = `CREATED`	R = `READY`	U = `RUNNING`
T = `STOPPED`	X = `TERMINATED`	W = `WAITING`

Bits 7-31 are the lower 25 bits of the Exit Code when the process finished running on the CPU.
(Note: When a process exits, its exit code is stored in the process struct in this field by the Simulator.)

Example: A process run with **sudo** that is in `TERMINATED` State with an `exit_code` of 42 will have `flags` = **0x00001521**

(Note: Remember Binary and Hex! 0x is the Hex prefix. Each two hex digits is 4 bits in binary)



(What this example flags field looks like in Binary)

For the flags field, remember your **bitwise operators**!

- **&** is the Bitwise AND. This is very useful for setting a given bit to 0.
- **|** is the Bitwise OR. This is very useful for setting a given bit to 1.

Hint: Write helper functions to set a given bit to a 0 and to set a given bit to a 1.

2.5 Schedule and List struct overviews (structs.h)

Schedule Struct

The overall struct of type **`Schedule`** is used for holding all of the lists. You will dynamically allocate and return the pointer in **`scheduler_init()`**, and will be passed in to other functions.

```
/* Schedule Struct Definition */
typedef struct schedule_struct {
  Queue *ready_queue;   /* Ready Processes */
  Queue *stopped_queue; /* Stopped Processes */
  Queue *waiting_queue; /* Waiting Processes */
} Schedule;
```

Schedule contains pointers to three **`Queue`** type structs, called **`ready_queue`**, **`stopped_queue`**, and **`waiting_queue`**. Each of these three linked lists must be allocated and initialized as well.

Queue Struct

Each **`Queue`** struct contains a pointer to a **`Process`** struct called **`head`**, which is the first node of a singly linked list of Processes, and **`count`**, which tracks the number of processes in the list.

```
/* List Struct Definition */
typedef struct queue_struct {
  Process *head; /* Singly Linked List */
  int count;     /* Number of process structs in the queue */
} Queue;
```

Process Struct

Each **`Process`** struct contains all of the information you need to properly run your functions.

```
/* Process Struct Definition */
typedef struct process_struct {
char *command; /* Process Command (OS Generates this, what the user ran) */
int pid;             /* Process ID (OS Generates this, it is unique) */
int flags;           /* Process Flags (State bits and exit_code are in here) */
int time_remaining;  /* OS Modifies This, the Time Units Left to Execute */
int time_last_run;   /* The Last Time the Process was Selected (or Time Created) */
int wait_remaining;  /* You will modify this in scheduler_io_run.  I/O Time Remaining */
struct process_struct *next;
} Process;
```

Each process has a pointer to a string called **`command`**, which will be the full command being executed, such as "**`ls -al *.dat`**".

Every process also has a unique Process ID (PID), which is stored here as an int called **`pid`**.

You will also have a 32-bit int **`flags`**, which contains several pieces of data that are combined together using bitwise operations, as specified in Section 0.

The next field, **time_remaining**, contains the number of time units needed for that process to complete. Each cycle through the main OS loop will run the process on the CPU for 1 time unit.

The last field, **wait_remaining**, contains the amount of time a process has left when waiting for I/O data in the **waiting queue**. Your **scheduler_io_run** function will decrement this and make decisions about what to do with the process based on this value.

3 Building and Running the OS Simulator

You will receive the file **project1_handout.tar**, which will create a handout folder on Zeus.

```
kandrea@zeus-1:handout$ tar -xvf project1_handout.tar
```

In the handout folder, you will have **scheduler.c**, which is the only file you will be modifying and submitting, **Makefile**, and a **traces** folder. You will also have very useful header files (scheduler.h, constants.h, and structs.h) along with other files for the simulator itself.

3.1 Building the OS Simulator

To build the OS Simulator, run the make command.

```
kandrea@zeus-1:handout$ make
gcc -g -Og -Wall -Werror -std=gnu99 -o scheduler context.c scheduler.c sys.o clock.o
```

3.2 Running the OS Simulator

To start the OS Simulator, run **./scheduler <tracefile>**

The tracefiles, which are located in the traces folder, are described in subsequent sections. The other PDF file on Blackboard has a large number of sample runs you can compare against.

3.3 Compiling Options

There is one very important note about our compiling options that may differ from what you used in CS262 (or other C classes). We're using **-Wall -Werror**. This means that it'll warn you about a lot of bad practices and makes almost* every warning into an error.

To compile your program, you will need to address all warnings!

**The only exception is for unused variables, which will stay as warnings.*

4 Implementation Details *(The part you were waiting for.)*

You will complete all of the functions in **scheduler.c**. You are encouraged to create any additional functions that you like, but you cannot modify any other files or the Makefile.

You will only be submitting one file, **scheduler.c**

The next section describes what each of your functions needs to accomplish.

4.1 Function API References *(aka. what you need to write)*

`Schedule *scheduler_init();`

Your init function needs to create a Schedule struct that is fully allocated (**`malloc`**). All allocations within this struct will be dynamic, also using **malloc**. Take note that Schedule contains pointers to Queue structs, which themselves also need to be allocated.

Each Queue struct contains a pointer to the head of a singly linked list and a count of the number of items in that list, which must be initialized to 0. All head pointers must be initialized to NULL.

On any errors, return NULL, otherwise return your new Schedule.

`int scheduler_count(Queue *ll);`

Return how many processes are in the given queue, or -1 on any error.

`Process *scheduler_generate(char *command, int pid, int time_remaining, int is_sudo);`

Create a new Process with the given information using malloc.

- Dynamically allocate memory for the string **`command`** and copy the command argument into the new process.
- Set the fields for **`pid`** and **`time_remaining.`**
- For the **`time_last_run`** field, call the provided **`clock_get_time()`** function.
  - Prototype: **`int clock_get_time(); // Returns current Time`**
  - This is already written for you, you can simply call this function.
- For the **`wait_remaining`** field, set it to 0.
- If **`is_sudo`** is 1, set the **`SUDO`** bit to a 1 in the flags integer, otherwise set to 0.
- Set the **`CREATED`** bit to 1 and all other state bits to a 0.
- Set the **`exit_code`** portion of the flags to 0.

Return the pointer to the new Process on success, or NULL on any error.

Errors may include getting passed in a NULL pointer, an empty command string, getting a malloc error, or any error on copying the string.

Remember to use memory safe functions: e.g. **`strncpy`** instead of **`strcpy`**.

`int scheduler_add(Schedule *schedule, Process *process);`

This needs to add the given process to your Schedule, but you will be adding it differently, depending on its State (in the flags).

Do the operations based on which of these states the process is in. *(The bit will be a 1)*

- **`CREATED`**, **`READY`**, or **`RUNNING`**
  Change the state to **`READY`**.
  Then, insert it in ascending PID order in your **Ready Queue**.
  The Ready Queue should always be in ascending PID order:
  (e.g. head -> PID:1 -> PID:3 -> PID:4 -> PID:5 -> NULL)

- Other States
  If the process is in any other state, it is an error. You don't have to do anything with the process, but you will need to return -1.

Return 0 on success or -1 on any error.

`int scheduler_wait(Schedule *schedule, Process *p, int io_time);`

This function will take the process and insert it into the waiting list.

- Change the State of the Process to **`WAITING`**.
- Set the **`wait_remaining`** field of the Process to **`io_time`**.
- Insert it at the end of the **Waiting Queue**.
  - For example, if your Waiting Queue looks like this:
    head -> PID:1 -> PID:5 -> PID:4 -> PID:3 -> NULL
  - And you insert PID:2, then your Waiting Queue should look like this:
    head -> PID:1 -> PID:5 -> PID:4 -> PID:3 -> PID:2 -> NULL

Return 0 on success or -1 on any error.

`int scheduler_finish(Process *p);`

This function will terminate a finished process. (Called if **`time_remaining`** is 0)
*Note, the process will already be set to TERMINATED by the Simulator.

- Extract the **`exit_code`** from the flags field.
  - This will be set for you by the Simulator before this is called.
- Free the process completely.
- Return the **`exit_code`** that you extracted.
  - All exit codes on this system will be >= 0.

Returns the extracted `exit_code` on success or -1 on any error.

`Process *scheduler_select(Schedule *schedule);`

For this system, each selected process will run on the CPU for 1 time unit only. If that process isn't finished yet, it will be put back in the Ready Queue, so another process can have a chance to run. This idea is called multitasking processes. The role of the scheduler you are writing in this Project is to pick the next process to run.

In this function, you will choose the best process from the Ready Queue to run on the CPU. When you select a process, you will be **removing** that process struct from the Ready Queue linked list and returning a pointer to the same struct. This means you will also need to set its next pointer to NULL before returning it, since it's no longer in a list.

The algorithm you are implementing here is a real scheduler algorithm called **Shortest Remaining Time First (SRTF)**. Each process will have some amount of time that it takes to run. This is stored in your **`time_remaining`** field in the process struct. The SRTF algorithm will choose the process that has the smallest **`time_remaining`** to run next. This gives your computer higher throughput, since we're running short processes first, then they will also finish sooner.

However, what if you have a really long process and lots of really short ones? The long process may never run as long as there are plenty of short processes. When this happens, it is called **starvation**, since the long running process will never be picked.

So, we fix this problem by checking each process to see when they last ran on the CPU. If a process has not run in **`TIME_STARVATION`** (given constant) of time, then it is starving!

For this algorithm, we'll pick the process that has the lowest **`time_remaining`**, unless there is a starving process. If a process is starving, we'll pick that one instead.

The algorithm you will use is as follows:

- Iterate through all processes in the **Ready Queue**.
  - Select the process with the lowest **`time_remaining`**.
    - If there is a tie, then select the one with the lowest PID.
  - However, if any process is in starvation, then select that one instead.
    - Call the **`clock_get_time()`** function to get the current time.
    - If the difference between **`clock_get_time()`** and **`time_last_run`** on the Process is greater or equal to **`TIME_STARVATION`**, then the process is starving.
    - If you see any starving process, do not check any other processes, you can stop iterating and select it.
- Remove the selected process from the list.
  - Remember to also set its next pointer to NULL when removing it.
- Set the State of the selected process to **`RUNNING`**.

Return the selected Process on success or NULL on any error.

```
int scheduler_stop(Schedule *schedule, Process *process);
```

This will stop a running process and move it to the Stopped Queue. This is what happens on your computer whenever a user enters CTRL-Z on the keyboard.

- Set the process' state to `STOPPED`
- Insert it in ascending PID order in your **Stopped Queue**.
    - The Stopped Queue should always be in ascending PID order: **(e.g. head -> PID:1 -> PID:3 -> PID:4 -> PID:5 -> NULL)**

Return 0 on success or -1 on any error.

```
int scheduler_continue(Schedule *schedule, int pid);
```

This needs to find the process with matching pid from the **Stopped List**.

- Once found, remove the process from the Stopped List.
- Set the process' state to `READY`
    - Insert that process in ascending PID order in your **Ready Queue**.
        - The Ready Queue should always be in ascending PID order: **(e.g. head -> PID:1 -> PID:3 -> PID:4 -> PID:5 -> NULL)**

Return 0 on success or -1 on any error.

```
int scheduler_io_run(Schedule *schedule);
```

The I/O device on the computer will do work for first process in the Waiting Queue. Your function will decrement the `wait_remaining` field of this first Process in the Waiting Queue to represent that work was done. If no waiting time is left, it'll go back to Ready.

Only work with the first process in the **Waiting Queue**.

- Decrement the process `wait_remaining` by 1.
- If the `wait_remaining` is now 0, then do the following:
    - Remove the first process from the **Waiting Queue**
    - Change the process state to `READY`
    - Insert that process in ascending PID order in your **Ready Queue**.
        - The Ready Queue should always be in ascending PID order: **(e.g. head -> PID:1 -> PID:3 -> PID:4 -> PID:5 -> NULL)**
- If there was no process in the **Waiting Queue**, you can return success.

Return 0 on success or -1 on any error.

`void scheduler_free(Schedule *schedule);`

Frees all memory allocated with a Schedule and then the schedule itself.

It is possible to run this program with no memory leaks!

5 Notes on This Project

Primarily, this is an exercise in working with structs and with multiple singly linked lists. All of the techniques and knowledge you need for this project are things that you should have learned in the prerequisite course (CS 222 or CS 262 at GMU) in C programming. Chapter 2.1-2.3 of our textbook also describes the use of Bitwise operations in Systems Programming. This project involves basic use of bitwise operations to set and clear flags and to extract data from flags. You will use bitwise operations in programming more extensively for the next project.

In our model, your selection uses an algorithm called **Shortest Remaining Time First (SRTF)** to select the process from the ready list that has the lowest remaining time (which is the value closest to 0) to go next. This is an algorithm for selecting the next process in OS Design. ***(You'll study this algorithm extensively in CS471!)***

This project also involves processes being created, running for a little bit of time on the CPU and then being removed from the CPU to run again later, as well as processes being stopped and continued. We'll go through the theory on all of these concepts later this semester! For now though, you only need to know about linked lists to finish this project.

5.1 Error Checking

If a value is passed into any of your functions through a given API (such as we have here), then you need to perform error checking on those values. **If you are receiving a pointer, always make sure that pointer is not NULL, unless you are expecting it to be NULL.** Failing to check the validity of inputs can, and often will, result in SEGFAULTS.

5.2 Memory Checking

To check for memory leaks, use the **valgrind** program.

```
kandrea@zeus-2:handout$ valgrind ./scheduler traces/trace1.dat
```

You are looking for a line that says:

```
All heap blocks were freed -- no leaks are possible
```

All of the traces (including any you write for testing) should run with no memory leaks.

6 Trace Files

Trace files simulate user input on an Operating System. Since we don't have real users starting programs (at least not until our last project in CS 367), the Trace File gives the Simulator a series of instructions to tell it what programs to run, how long they will run for, and any other user-input. You don't need to write code to parse these, the Simulator already does that for you.

Sample Trace files are given to help you test your code. You are encouraged to write your own!

A trace file is just a collection of user actions, one per line, in a text file. Each iteration of the main loop has a stage that will execute one action. The action to execute is provided by the trace files from the following list:

`command [X,Y,Z]`

This format is used to **create a new process** with a given command.
Inside the square brackets are three numbers we will use when creating the process:

[**`PID`**, **`time_remaining`**, **`exit_code`**]

Example: (Creates a process with command **`ls -al`**, with PID 1, that will run for 4 time units, and will exit with the exit_code 42)

```
ls -al [1,4,42]
```

`sudo command [1,2,3]`

This format is exactly the same as the above one, but when the process is generated, the **`is_sudo`** argument will be a 1. This lets you run your program as a super-user.

`kill -STOP X`

This format creates an action to stop process with PID X

`kill -CONT X`

This format creates an action to continue process with PID X

`wait X`

This format tells the currently running process to wait for X time units.

`pass`

This format lets you take no actions (pass) during that iteration

`exit`

This exits the Simulator

7 Submitting and Grading

Submit this assignment electronically on Blackboard. **Make sure to put your G# and name as a commented line in the beginning of your program.**

Note that the only file to submit is scheduler.c

You can make multiple submissions; but we will test and grade ONLY the latest version that you submit (with the corresponding late penalty, if applicable).

Important: Make sure to submit the correct version of your file on Blackboard! Submitting the correct version late will incur a late penalty; and submitting the correct version 48 hours after the due date will not bring any credit. Your code must compile to receive any points.

Questions about the specification should be directed to the CS 367 Piazza Forum.

Your grade will be determined as follows:

- **20 points - Code & Comments.** Be sure to document your design clearly in your code comments. This score will be based on reading your source code.
    - If you do not use bitwise operations when working with flags or do not use linked lists in this project, you will lose points in this section.

- **80 points - Automated Testing (Unit Testing).** We will be building your code using your submitted code and our provided Makefile and running unit tests on your functions.
    - It must compile cleanly to earn any points.
    - Each function will be tested independently for correctness by script.
    - Partial credit is possible.
    - **Border cases and error cases will be checked!**

  The testing will be done at the function level. This means we will load up our Simulator to a known state and then call one of your functions to do a particular task.
    - If that operation runs as expected, then that test will pass will full points.
    - If part of the operation does not work as expected, that test will fail, but with partial credit.
    - If the operation does not work at all, or breaks the program, the test will fail without any points.

  The trace files you have are used to help you see bugs in your program but will not be used for grading. Make sure to test your functions well!

  The other PDF file is a reference showing how each of the provided traces should look when they are executed.