CS 367 Project #0 - Fall 2020:
CPU Process Scheduler
Due: Friday, September 11 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 be writing a single file, called **scheduler.c,** to implement a series of functions as specified in the Scheduler API to perform a series of basic operations for an OS Simulator. 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 as if each process had its own dedicated CPU to run on. The OS implements this by putting each running process (program) into one of several **priority queues**, which are implemented as singly linked lists.

The OS starts by choosing the next process to run from the **ready queue**, which contains all of the processes that are ready to begin, or continue, running are. At the end of a set amount of time (usually ~10ms), the OS will stop running that process and put it back into a queue.

When a timer expires, if the process did not finish running in that time, it will go back to the **ready queue** to get scheduled again. If it did finish, it will go into the **defunct queue**, where all the exited processes get cleaned up from. Finally, if it was stopped (eg. user presses Control-Z), then it would move into the **stopped queue**, to show that 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.

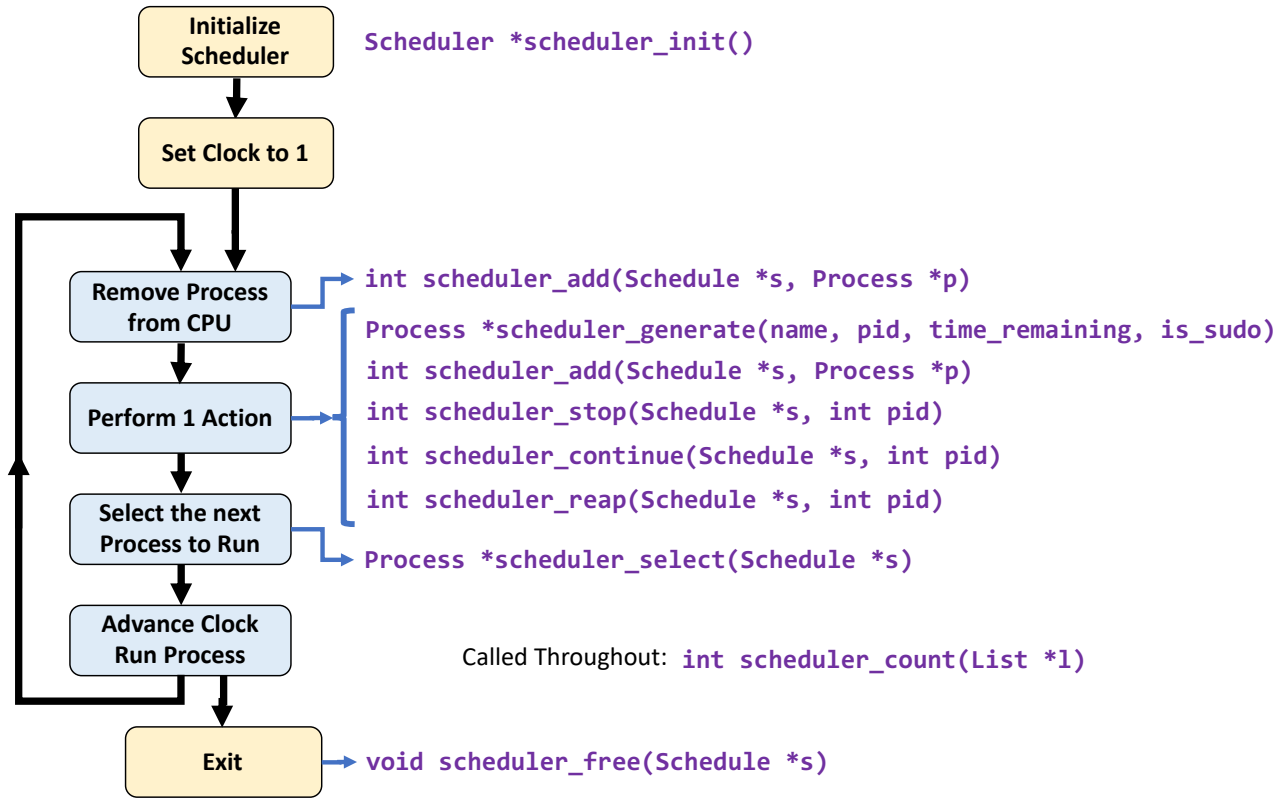
Your code (**scheduler.c**) will work with pre-written files from an OS simulator to implement several of these operations. You will be maintaining all three singly linked lists (**ready queue**, **stopped queue**, and **defunct queue**). The structs for these lists are all defined in **structs.h**.

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

2.1 OS Simulator Overview *(How does the Simulator Work at the High Level?)*

The OS Simulator is written for you and uses the following loop. For each box, 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: In the "Remove Process from CPU" step, the OS calls your **`scheduler_add`** function.



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. So, if a process has a `time_remaining` value of 3, it will need to run on the CPU three more times for it to complete.

Each process struct that you store in the linked lists has a field called "pid", which is an int. This is the Process ID number that the OS uses to uniquely identify a specific process. Most of your functions will use the pid field to perform an operation on a process.

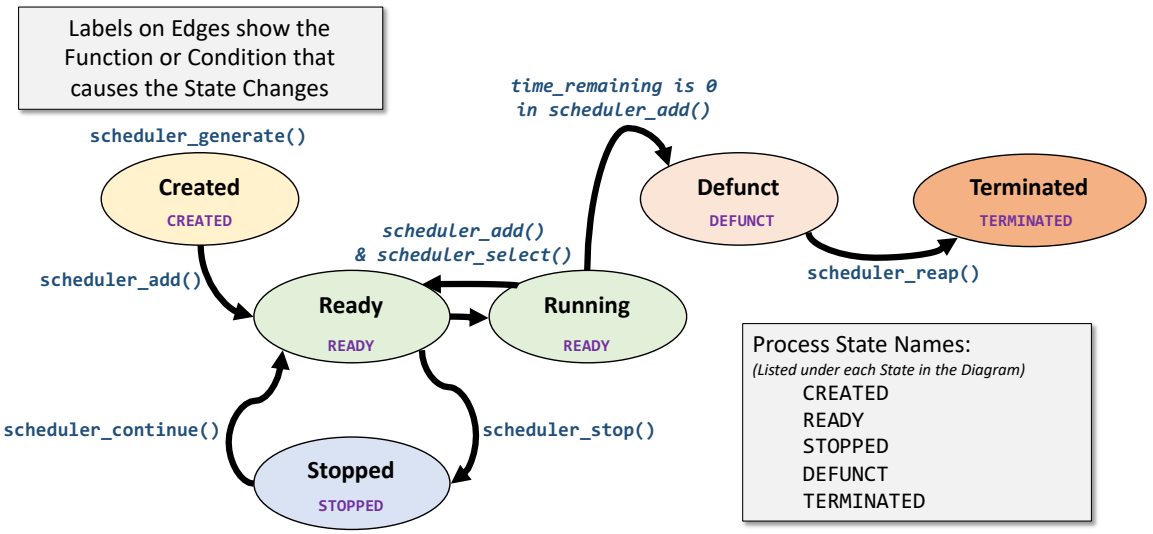
Example: Let's say that we have only one process running on the CPU and nothing in the ready queue at the end of Time 2. At the beginning of the cycle, on Time 3, the simulator will call your `scheduler_add` function to add the running process back into your queue. *(Details about how to write each function is coming)*. It will then choose one of the actions to run and call the function for that action. In this case, let's say it calls your `scheduler_stop` function. Then it will call your `scheduler_select` function to have you choose the next process to run. Finally, it advances the clock and 'runs' that process for 1 time unit.

2.2 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 five possible **states** at any given time. This is separate from which queue it's in. For example, if it is ready to run, it's state will be `READY`.

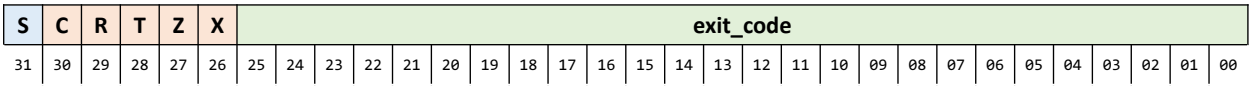
The following diagram shows the five states of a process, and how they can change to other states. *(Note: Ready and Running both are in the same state,* `READY`*).*



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 Ready State (`READY`), and the simulator calls your `scheduler_stop()` function, then that Process will change to the Stopped State (`STOPPED`).

2.3 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)

Bits 0-25 are the lower 26 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.)

Bits 26-30 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` T = `STOPPED`
Z = `DEFUNCT` X = `TERMINATED`

Bit 31 is a 1-bit flag saying if the process was run with super user privileges (Called using sudo)
S = `SUDO`

> **Example:** A process run with **sudo** that is Defunct with an `exit_code` of 3 will have `flags` = **0x88000003**
> *(Note: Remember Binary and Hex! 0x is the Hex prefix. Each hex digit is 4 bits in binary)*

2.4 Schedule and Queue struct overviews (structs.h)

Schedule Struct

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

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

Schedule contains pointers to three `Queue` type structs, called `ready_queue`, `stopped_queue`, and `defunct_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 queue.

```
/* Queue 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 base_priority;  /* Base Process Priority Level (0 to 139) – What it is run as */
int cur_priority;   /* Process Priority Level (0 to 139) – What it is modified to */
int time_remaining; /* OS Modifies This, the Time Units Left to Execute */
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 2.3.

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.

3 Building and Running the OS Simulator

You will receive the file **project0_handout.tar**, which will create a handout folder on Zeus.

```
kandrea@zeus-1:handout$ tar -xvf project0_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 every warning into an error.
To compile your program, you will need to address all warnings!

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

You will complete all of the functions in **scheduler.c**. You may 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 queue, 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 base_priority, 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 in to the new process.
- Set the fields for **`pid`**, **`base_priority`** and **`time_remaining.`**
- For the **`cur_priority`** field, set it to the same value as **`base_priority.`**
- If **`is_sudo`** is 1, set the **`SUDO`** bit to a 1 in the flags integer.
- Set the **`CREATED`** bit to 1 and all other state bits to a 0.
- Make sure the **`exit_code`** portion of the flags starts at 0.

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

`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`**
  Change the state to **`READY`**.
  Then, insert it into the **front** of the **Ready Queue**. (e.g. head)
- **`READY`**
  You need to check its **`time_remaining`**. If the **`time_remaining`** > 0, then insert it into the **Ready Queue** (insert to the **front** of the queue).
  Else, if the **`time_remaining`** is 0, then set its state to **`DEFUNCT`** and insert it into the **Defunct Queue** (insert to the **front** of the queue).
- **`DEFUNCT`**
  You can just insert it into the **front** of the **Defunct Queue**.

If the process arrives in any other state, you can return as an error.
Return 0 on success or -1 on any error.

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

This function cleans up the processes that are finished.

- Find the process with matching pid from the **Defunct Queue**.
  - Once found, remove the process from the Defunct Queue.
  - Set the process' state to **`TERMINATED.`**
  - Extract its **`exit_code`** from the flags.
  - Free the process.

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

`Process *scheduler_select(Schedule *schedule);`

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 need to set its next pointer to NULL before returning it, since it's no longer in any list.

The algorithm you are writing is called **Priority Scheduling**. Each process in Linux has a priority number (0-139). Priority 0 is the highest priority, and 139 is the lowest. The lower the value of `cur_priority` (eg. closest to 0) means a greater priority level.

Your function will find the process with the lowest `cur_priority` and return it to run next. However, a process with a very high `cur_priority` (close to 139) might always be skipped if there are a lot of other processes. When this happens, it is called **starvation**.

So, we fix this by first removing the process with the lowest `cur_priority`, and then subtracting 1 from every remaining process' `cur_priority` in the Ready Queue, so eventually a process that has been waiting a long enough time will be highest priority!

The algorithm you will use is as follows:

- Remove the process with the lowest `cur_priority` in the **Ready Queue**.
  - If a tie for lowest `cur_priority,` remove the first of them in the queue.
- For that process you removed, set its `cur_priority` equal to its `base_priority.`
  - This resets its priority to its normal level before it runs.
- Then subtract 1 from `cur_priority` of all remaining structs in the **Ready Queue.**
  - The minimum `cur_priority` value is 0, so do not reduce below this.

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

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

This needs to find the process with matching pid from the **Ready Queue**.

- Once found, remove the process from the Ready Queue.
- Set the process' state to `STOPPED`
- Insert that process into the front of the **Stopped Queue**.

If this process is not in the ready queue, it is an error.
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 Queue**.

- Once found, remove the process from the Stopped Queue.
- Set the process' state to `READY`
- Insert that process into the front of the **Ready Queue**.

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 (CS262 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 some 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 **Priority Scheduling** to select the process from the ready queue that has the highest priority (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!)

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 are provided to help you test your code. You are encouraged to write your own!

A trace file is just a collection of 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 [1,2,3,4]`

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`**, **`base_priority, exit code`**]

Example: (Creates a process with command **`ls -al`**, PID 1, that will run for 4 time units, with a priority of 100 and will exit with the exit_code 0)

```
ls -al [1,4,100,0]
```

`sudo command [1,2,3,4]`

This format is exactly the same as the above one, but when the process is generated, the is_sudo parameter will be a 1.

`kill -STOP 1`

This format creates an action to stop process with PID 1

`kill -CONT 1`

This format creates an action to continue process with PID 1

`reap 1`

This format creates an action to reap (clean up) a defunct process with PID 1

`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!**
    - There are no partial points for these, however, several of the provided traces will be run, and their output is given on the other PDF file for reference.

  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!