CS 367 Project 1 - Fall 2024:
Egul CPU Scheduler
Due: Friday, September 13th by 11:59pm

This is to be an individual effort. No partners. No Internet code/collaboration.
Protect your code from anyone accessing it. Do not post code on public repositories.
No late work allowed after 48 hours; each day late automatically uses up one of your tokens.

Core Topics: C Programming Review, Linked Lists, Bitwise Operators, extending Existing Code

1 Introduction

You will be finishing a series of functions for the **Egul CPU Scheduler**. Finish the functions in `src/egul_sched.c` to implement a CPU scheduler for our StrawHat Task Manager (TM).

You will use bitwise operators, structs, and Linked Lists to implement given algorithms in C.

Problem Background *(These related topics come up in CS 367 and in CS 471)*

StrawHat is a lightweight task manager (shell) that runs and swaps between Linux programs with custom **CPU scheduling**. These types of shells are useful for testing complicated interactions between processes (programs being run), for manually scheduling them in certain orders or allow the user to run processes with custom priority orderings, or for evaluating scheduling algorithms.

So, what is CPU scheduling and how does it work? *(What you'll be Writing)*

The Scheduler selects a process (a program being run) from a **Ready Queue** (linked list) to run for a very short amount of time on the CPU. When the time limit is reached, then it is put back into the Queue and another process is selected to run for a short time. This repeats until all processes finish.

The Operating System (OS) runs each process for a tiny amount of time, then swaps it for the next process over and over. This lets you run hundreds of processes on a small number of CPU cores and makes it seem as if they are all running at the same time. This is the main idea of **multitasking.**

Project 1 is to finish functions for the **Egul CPU Scheduler** to manage the Process Scheduling.

Table of Contents: *(What's in this Documentation)*

Project Overview

Like industry, this project involves a lot of code written by other people. You will only be finishing 11 functions of code to add one small feature (Scheduler) to an existing project (StrawHat TM).

You will be finishing code in `src/egul_sched.c` to create, insert, remove, and find process nodes on three singly linked lists (**Ready Queue, Suspended Queue, and Terminated Queue**) and update state with bitwise operations. The struct definitions are defined in `inc/egul_sched.h`.

You are encouraged to add helper functions, but you cannot change how we compile.

The bottom line is our code will call your functions in egul_sched.c to do operations.

Prototypes of Functions for the Egul Scheduler that You Will Be Finishing (Details in Section 4)

Note: All inserts into Queues (linked lists) will be at the **end** of the list.

```
Egul_schedule_s *egul_initialize();
```

- Creates the Egul Schedule struct, which has pointers to all three Linked List Queues.

```
Egul_process_s *egul_create(Egul_create_data_s *data);
```

- Create a new Process node from the data argument and return a pointer to it.

```
int egul_insert(Egul_schedule_s *schedule, Egul_process_s *process);
```

- Insert the Process node to the **Ready Queue** Linked List at the end.

```
int egul_count(Egul_queue_s *queue);
```

- Return the number of Process nodes in the given Queue's Linked List.

```
Egul_process_s *egul_select(Egul_schedule_s *schedule);
```

- Remove and return a pointer to the *best Process* in your **Ready Queue** Linked List. (Details in Section 4)

```
int egul_suspend(Egul_schedule_s *schedule, pid_t pid);
```

- Remove the process with pid from **Ready Queue** and insert in **Suspended Queue** at the end.

```
int egul_resume(Egul_schedule_s *schedule, pid_t pid);
```

- Remove the process with pid from **Suspended Queue** and insert in **Ready Queue** at the end.

```
int egul_exited(Egul_schedule_s *schedule, Egul_process_s *process, int exit_code);
```

- Inserts the Process node in the **Terminated Queue** at the end.

```
int egul_killed(Egul_schedule_s *schedule, pid_t pid, int exit_code);
```

- Move a Process from either **Ready** or **Suspended Queue** to **Terminated Queue** at the end.

```
int egul_get_ec(Egul_process_s *process);
```

- Returns the exit code from the status member for the given process.

```
void egul_deallocate(Egul_schedule_s *schedule);
```

- Free the Egul Schedule, the three Linked List Queues, and all of their Process Nodes.

StrawHat should run with no Memory Leaks

StrawHat (this is the task manager that calls your functions and is already written for you)

The StrawHat TM program is written for you but needs your Egul CPU scheduler code to run. In this manner, Egul is a library that StrawHat uses to do the custom process scheduling.

The general idea of StrawHat is that it runs like a **shell**, which is the command line that you type program names into so you can run them. The difference is that StrawHat runs each program, one at a time, but only for a very small amount of time. If the program isn't finished at the end of that time, StrawHat will pause it and insert it back into the ready queue, so it can continue running later. Then it will select the next process to run, letting programs alternate until finished.

StrawHat runs on top of Linux and implements custom **multitasking** for processes. The basic idea is that every time you start running a new process, like the `ls` program, what's happening is that you're adding that process information to a Node in the **Ready Queue**, which is just a **Linked List**.

The **Ready Queue** is where all processes wait for their turn to run on the CPU. A second queue, the **Suspended Queue**, is used to hold any user suspended processes. When the user resumes a suspended process, it will be moved back to the ready queue again. A third queue, the **Terminated Queue**, holds processes that either exited or that were terminated by the OS.

A sample of one of these linked list Queues is shown below.

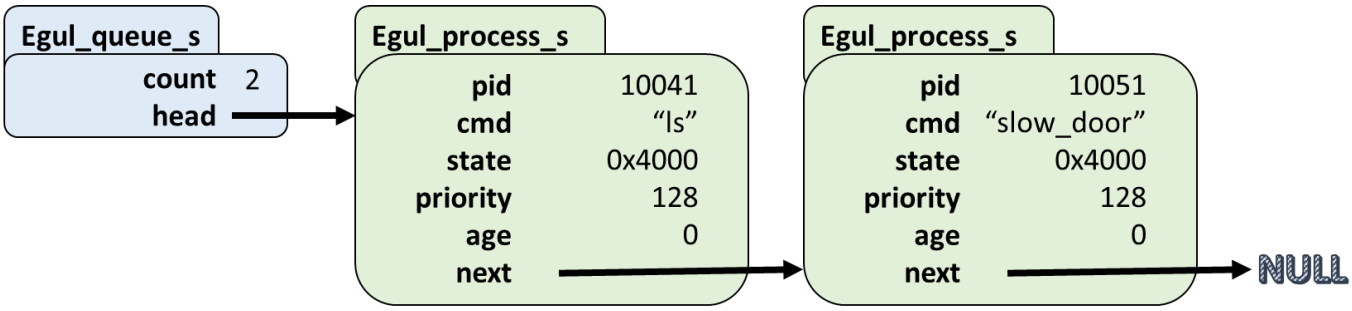


Figure 1: Depiction of a Linked List from a Queue Header.

Each Linked List starts from a Queue Header (`Egul_queue_s *`) node. These headers are used to hold the `head` pointer for each linked list. This means that if the head of the linked list is inside of this queue header, which is passed into many of your functions.

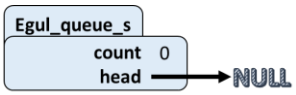
Each **head** pointer will only be pointing to NULL or to a valid node.

You will not use any dummy nodes in this project

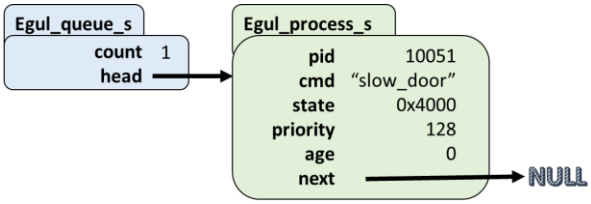
For each of the three Queues, when a new process is added to it, you will always **add them to the end of the linked lists.** All processes will always be inserted at the end of each queue.

All pid values provided by StrawHat are guaranteed to be unique and greater than 1

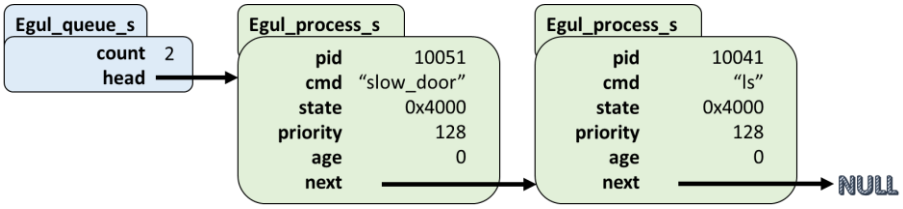
For example, here is a set of operations of starting with an empty **Ready Queue**.



Next, we create a process "ls" with Process ID (PID) 10051, which is inserted as the only node.



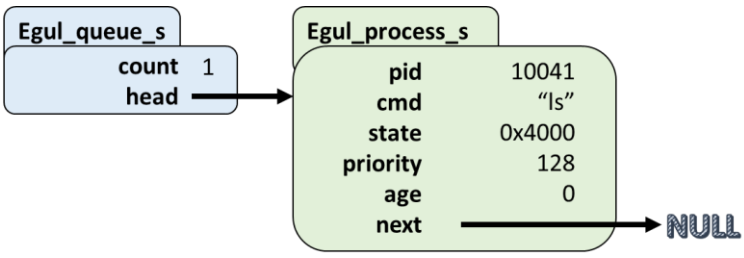
Then we create process "ls" with PID 10041 and insert it in **at the end of the list**.



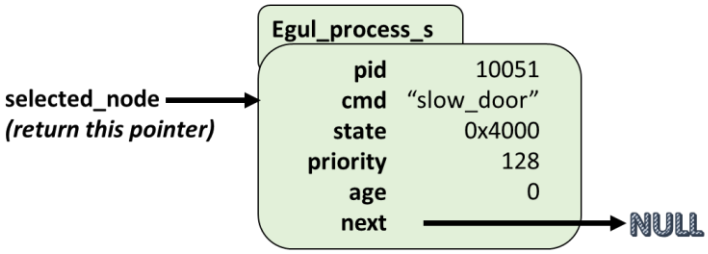
Your code won't need to worry about starting any processes; that's all done by StrawHat, but your code will be called to create the process nodes and manage them in the Linked Lists.

StrawHat will use your Ready Queue to keep track of all the processes that are **ready** to run. When StrawHat needs a next process to execute on the CPU, it will call your `egul_select` function, which will remove the best node (details in Section 4) and return a pointer to it.

For example, the process with PID = 10051 is the best node in the above example. Let's select it.



When the process with PID = 10051 was selected, it was removed from the queue completely, so the queue header now has a count of 1.



The removed process node is now fully separated from the queue and is ready to be returned.

You will always return a pointer to the original struct node that was removed. You will not make copies.

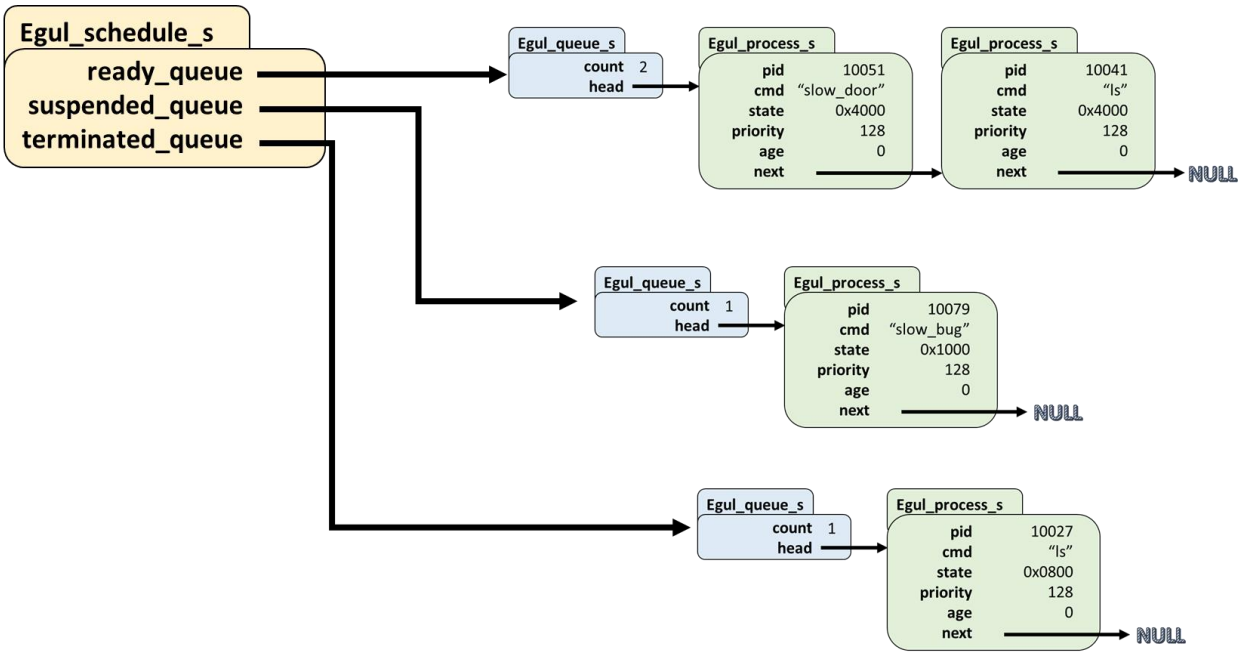
Using a timer, StrawHat will run the process you selected and returned to it for a set amount of time (by default, this is 250ms). It will then give that process to you by either calling your `egul_insert` function to put it back in Ready Queue so it can resume running later, or, if it finished in the middle of the run, StrawHat will instead call your `egul_exited` function to put it into the **Terminated Queue**, which tracks all finished or terminated processes. These will not run again.

When StrawHat is running, it will continue this cycle of calling your functions to get a process that's ready to run, run it for 250ms, and then call your function to put it back into the Ready Queue so it can run again in the future. When multiple processes are all in the ready queue, StrawHat will select and run each of them, one by one, for a small amount of time, then switch to the next one, in a big cycle, until all have completed and ended up in your Terminated Queue.

To help you manage each of your three Linked Lists (Queues), you have another struct, `Egul_schedule_s`**,** that contains pointers to them all. (You won't need any dummy nodes at all.) This is the master **schedule** that will be the argument passed into most of your functions.

You can use the same general idea of classes in C! We have a **schedule** struct that contains references (pointers) to each of the three **queue header** structs, which contain the count and pointers to their linked list.

Here's a picture of what this looks like in action.



Your functions will create the Schedule (`Egul_schedule_s`), then create each of the three Queue Headers (`Egul_queue_s`), and then will be responsible for creating Process Nodes (`Egul_process_s`), inserting, moving, or removing them into one of the three Queues.

2 Project Details

2.1 Source Code Files

When you get started, you will have several subdirectories in the Handout directory. The most important one of these is **src/**, which is where all the source code lives. Next to that is **inc/**, which is where all the headers are, and **obj/**, which contains all of the pre-compiled object files (.o or .a). Because this is a complicated and large project, some of the files needed are already in object format, so you will only see some of the original source code. You will only be writing code in one file in the **src/** directory.

The only file you will be adding code to is `egul_sched.c`

Your code in **`egul_sched.c`** consists of 11 functions that will be called from StrawHat's main code. Each function does one job, as is described in Chapter 4 of this document. You are free to add any number of helper functions that you want, of course.

These 11 functions all have **stubs** present in the file. This just means there is a definition for the function that is already provided for you, but that currently does nothing.

```
/* Returns the number of items in a given Egul Queue (singly linked list).
 * Follow the project documentation for this function.
 * Returns the number of processes in the list or -1 on any errors.
 */
int egul_count(Egul_queue_s *queue) {
  return -2; /* Replace This Line with your Code */
}
```

Example of one of the functions you will be completing in **`egul_sched.c`**

Even though StrawHat is doing many different tasks, your part of this project is just the Egul Scheduler that is used by StrawHat. Each of your functions requires only basic C knowledge from CS 262 or any other introduction to C programming course. Do not think this is just a toy project though; your code will be implementing a full CPU scheduler, even if you are just writing some basic linked list code.

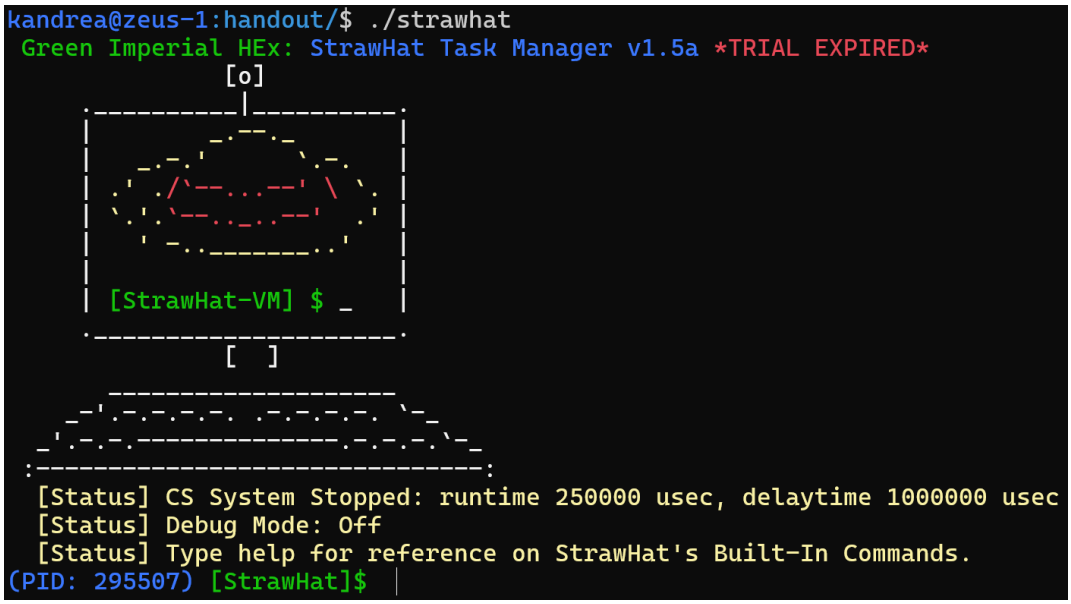
Your functions will primarily be maintaining all three singly linked lists (**Ready Queue**, **Suspended Queue**, and a **Terminated Queue**) and choosing which process the CPU should run next.

Helper Functions are highly encouraged!

The structs for these lists are all defined in **`inc/egul_sched.h`**

2.2 StrawHat Details

StrawHat is not a toy simulator. It is a real program that provides a shell you can use to run normal non-interactive Linux commands like **ls** and **cal**. It can deal with arguments (like **ls -al** or **wc Makefile**) but it cannot deal with any commands with an interactive interface – you cannot use it run **wc** with no arguments or use it to run **vim**, for example.



For more information on StrawHat, including on how to Compile it with **make**, its built-in commands, how to run it with programs, and sample outputs, see **P1_StrawHat_Manual.pdf**.

2.3 Process PID and Priorities

In an Operating System (OS), a program is what we call the executable on your drive. It is an executable object that contains all of the source code in machine code format. When the OS runs the program, it becomes a **process**. This is just your program in execution. Each process is managed by the OS.

Egul Scheduler, which you will be writing, works with processes using a few different concepts. The first is that every process has its own Process ID number (**PID**). The PID is **guaranteed unique** for every process and is generated by the operating system. This is how we identify and work with processes.

The data type for these PIDs is `pid_t`, as is seen in your struct definitions. The **_t** suffix means this is a custom typedef type in the OS. In reality, it is just a typedef alias for an **int**. So, whenever you see `pid_t`, you can treat it the same as an **int**. The `pid_t` should only be used for PID numbers from the OS though.

Each process also has a **Priority** value ranging between 1 and 255 inclusive, with 1 being the **highest possible priority** and 255 being the **lowest possible priority**. Priority will never change during the run.

All processes have a **default priority of 128**, so we don't have anything special to designate them. The Priority Level is used to determine which processes should run first.

2.4 Process Selection *(Which process in the Ready Queue should run next?)*

There are often multiple processes in the Ready Queue that are ready to run, but since there are processes with multiple different priorities, some of them with higher priorities should be selected and run more often than those with lower priorities. This section will overview how we choose which should be selected first when your `egul_select` function runs. **This is just an overview; the details are in Section 4.**

To aid in selection, there are three properties that need to be checked: **Critical**, **Priority Level**, and **Age**.

First: Critical Processes

First, check to see if any process is running with the **Critical** priority flag. If a Process is started with the **Critical** option (**-c** is added on the shell), then it should always be picked to run immediately, even if there are starving or higher priority processes. A critical process always runs exclusively until terminated.

Second: Starving Processes

If there are no critical processes, then we actually have a problem. The highest priority process in the **Ready Queue** linked list would normally be picked in that case. This, however, leads to a major problem in CS called **Starvation**, where a higher priority process may prevent low priority ones from ever running!

To fix this, StrawHat uses a solution called **aging**. Whenever a process is selected and removed from the Ready Queue, you will be incrementing the **age** of all remaining processes in the **Ready Queue**. This lets us know which processes are starving (haven't been run in a very long time). So, we will now select any **starving** process, which is a process that has an **age** greater than or equal to a constant called **STARVING_AGE**. So, if there is such a process, that will get picked to run before even a higher priority process. Of course, when a process is selected to run, you will set its age to 0 again.

Third: Priority Level

If there are still no processes selected, we look at **priority.** In Linux, the highest priority process has priority level equal to 1. The lowest priority process is 255. The default level for new processes is 128. So, if no selections have been made for Critical or Starving, then we select the process with the highest priority, which is the priority level value that is closest to 1. On any ties, we pick the closest to the head.

The result of this is that you will see high priority processes running more frequently than lower ones.

Priority Level of a process will never change during the run. If it was 128 when created, it will stay 128.

Full Algorithm

So, the actual algorithm you'll see in Section 4 when you need to pick the best process to run is you'll first look to see if there are any **Critical** processes in the ready queue. If so, you select the first one you find. Otherwise, you'll see if there are any **Starving** processes in the ready queue. The first starving process will be selected. If none, then select the process in the ready queue with the priority level closest to 1.

Details for all of these are in the subsequent sections and in Section 4.

2.5 Process States

When you are working with Process Nodes (`Egul_process_s` structs), you will see that there is one field in the struct definition called `state`. This is an **unsigned short** (16-bit unsigned integer data type) that we're going to use to hold many different pieces of data within.

Storing multiple smaller data types inside of one larger one is **very common** in Systems Programming.

To understand the `state`, first we need to have a very small discussion on Process States. Each Program – like '**ls**' – that you run in Linux is run as a Process. Each Process starts off by going into a Ready Queue, where it will be in the **Ready State (R)**, indicating it's ready to run whenever it gets scheduled.

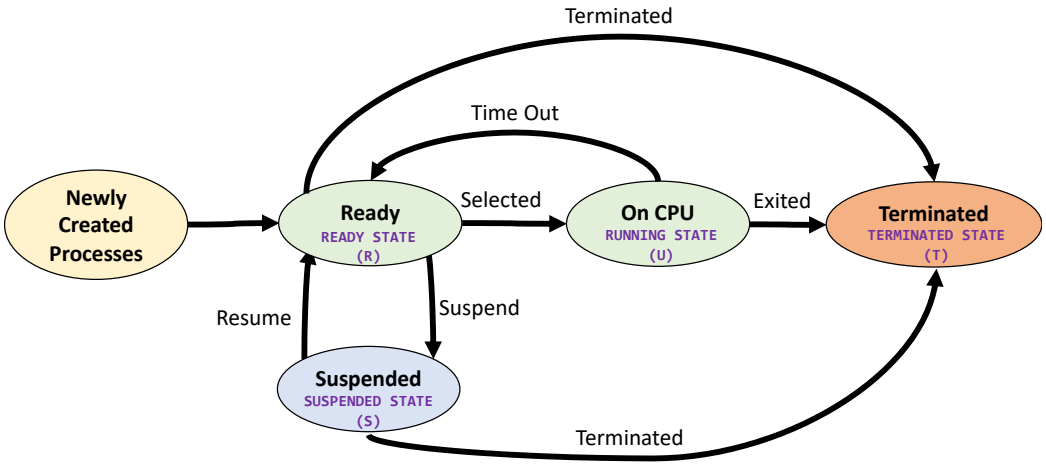
A process that's running on the CPU should be in the **Running State (U)** while it's running. Note that since processes run for such a tiny amount of time, you have to be lucky to see this state on the schedule.

If the user wants to temporarily pause a process, they can suspend it. We will move the process from the ready queue to the **Suspended Queue** and to put it in the **Suspended State (S)**. Now it is waiting for a user to resume it, so we can move it back into the Ready State and put it into the Ready Queue again.

When the process running on the CPU has been running too long, it'll be switched out and put back into the **Ready Queue** and put back into the Ready State, so it can run again later. Then the Scheduler will select the next Process from the **Ready Queue** and put it on the CPU to run for its little piece of time.

When a Process finishes, it is moved to the **Terminated Queue**, which keeps a list of the terminated processes. Anything in the Terminated Queue will be in a **Terminated** (known as a **Zombie**) **State (T)**.

Every process can be in exactly one of four possible states in the TM at any given time.



This shows the states that each Process can be in and when they change. As an example, if a Process is in the Ready Queue in the **Ready State** and your `egul_killed()` function is called on it, then you will change it to the **Terminated State** and move to the **Terminated Queue**.

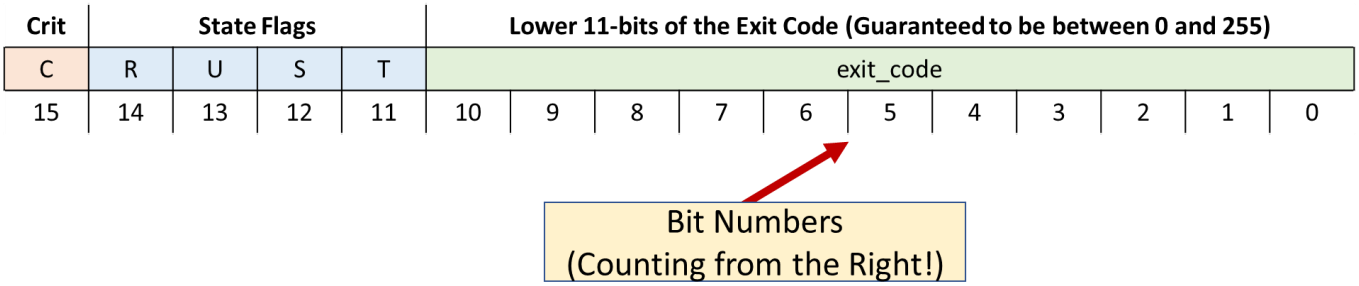
Details for all of these are in the subsequent sections and in Section 4.

2.6 Process Node State

The Process Node (`Egul_process_s`) struct maintains their current state using the `state` member. This 16-bit **unsigned short** contains information that have been combined together using bitwise operations.

You are required to use bitwise operators to change the states.

| & ^ >> <<



Hint: Look at the C Review on Bitwise Operations (Bit Masking) and Shifting!

Critical Bit: Bit 15 is a flag representing if the process was set with Critical Run. (-c)

C = `CRITICAL FLAG`

State Bits: Bits 11-14 represent the current State of the Process. (1-bit values)

R = `READY STATE` U = `RUNNING STATE`

S = `SUSPENDED STATE` T = `TERMINATED STATE`

Bits 0-10 are the lower 11 bits of the Exit Code when the process finished running on the CPU.

When you set these bits, you can assume the Exit Code given to you will always fit without any overflow issues. (eg. The exit code value is guaranteed to be between 0 and 255)

Example: A Process in the Terminated Queue, Terminated State, run as Critical, with `exit_code` = 6

Bin: 1000 1000 0000 0110

Hex: 8 8 0 6

So, `state` = 0x8806

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

2.7 Schedule, Queue, and Process struct overviews (egul_sched.h)

Schedule Struct (*Holds all the* ***Schedule*** *Information****)***

The overall struct of type **`Egul_schedule_s`** is used for holding all the three linked list Queues. You will dynamically allocate (**malloc**) and return the pointer, which will be passed to other functions.

```
typedef struct egul_schedule {
  Egul_queue_s *ready_queue;      // Linked List of Processes ready to Run on CPU
  Egul_queue_s *suspended_queue;  // Linked List of Suspended Processes
  Egul_queue_s *terminated_queue; // Linked List of Terminated Processes
} Egul_schedule_s;
```

Note: As a common programming style in C, non-system custom types may have a **_s suffix, as we have here.*

Queue Header Struct (*Holds all the Information for a single* ***Queue)***

A **Queue Header** (**`Egul_queue_s`**) struct contains a pointer to a **`Process`** struct called **`head`**, which is the first node of a singly linked list of Processes, and **`count`**, to track how many processes are in the list.

*Note: There are no Dummy Nodes here! head **must** point to either NULL or the first Process.*

```
typedef struct egul_queue_header {
  int count;            // How many Nodes are in this linked list?
  Egul_process_s *head; // Points to FIRST node of linked list.  No Dummy Nodes.
  Egul_process_s *tail; // Points to LAST node of linked list.  No Dummy Nodes. Optional.
} Egul_queue_s;
```

Process Node Struct

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

```
typedef struct egul_process_node {
  pid_t pid;            // PID of the Process you're Tracking
  char *cmd;            // Name of the Process being run
  unsigned short state; // 16-bit: Contains the Flags [C,R,U,S,T] AND Exit Code
  int priority;         // The priority of the Process
  int age;              // How long this has been in the Ready Queue since last run
  struct egul_process_node *next; // Pointer to next Process Node in a linked list
} Egul_process_s;
```

Each process has a pointer for a string called **`cmd`**, which will be the name of the command being executed, such as "**`slow_bug`**". Every process on the OS also has a unique Process ID (PID), which is stored here as a **`pid_t`** (an **int**) called **`pid`**. The PID will always be greater than 1.

You will also have a 16-bit unsigned short **`state`**, which contains several pieces of data that are combined together using bitwise operations, as specified in Section 2.6.

3 Building and Running the StrawHat Task Manager (./strawhat)

All compiling and grading will be done on **zeus.cec.gmu.edu** (the class Zeus Server). You may work in any environment of your choosing, but you will need to compile, test, and run your code on Zeus.

You will receive the file **project1_handout_v1_0.tar**, which will create a handout folder on Zeus.

```
kandrea@zeus-1:handout$ tar -xvf project1_handout_v1_0.tar
```

In the handout folder, you will have three directories (**src, inc,** and **obj**). The source files are all in **src/**. The one you're working with is `egul_sched.c`, which is the only file you will be modifying and submitting. You will also have very useful header files (`egul_sched.h`) along with other files for the task manager itself in the **inc/** directory.

3.1 Building StrawHat

To build StrawHat, run the make command:

```
kandrea@zeus-1:handout/$ make
gcc -std=gnu99   -Og -Wall -Werror -Wmaybe-uninitialized -Wno-error=unused-variable -Wno-
error=unused-function -pthread -I./inc -L./obj -g  -c -o obj/strawhat.o src/strawhat.c
gcc -std=gnu99   -Og -Wall -Werror -Wmaybe-uninitialized -Wno-error=unused-variable -Wno-
error=unused-function -pthread -I./inc -L./obj -g  -c -o obj/strawhat_cs.o src/strawhat_cs.c
gcc -std=gnu99   -Og -Wall -Werror -Wmaybe-uninitialized -Wno-error=unused-variable -Wno-
error=unused-function -pthread -I./inc -L./obj -g  -c -o obj/strawhat_shell.o
src/strawhat_shell.c
gcc -std=gnu99   -Og -Wall -Werror -Wmaybe-uninitialized -Wno-error=unused-variable -Wno-
error=unused-function -pthread -I./inc -L./obj -g  -c -o obj/strawhat_support.o
src/strawhat_support.c
...
```

This may be the first time you are building a large project. Your code is just one small piece.

3.2 Running StrawHat

To start StrawHat, run `./strawhat`

Instructions for Running the StrawHat are in the P1_StrawHat_Manual.pdf.

3.3 Compiling Options

There is one very important note about our compiling options that may differ from what you used in CS262/CS222 (or other C classes). We're using **-Wall -Werror**. This means that it'll warn you about bad practices and makes almost every warning into an error. (Unused Variables and Functions are OK).

For this semester, we will be compiling all programs using the **C99** C standard. (--std=gnu99 specifically)

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

4 Implementation Details

You will need to complete all 11 of the functions in **egul_sched.c**. You are encouraged to create any additional functions that you like, but you cannot modify any other files; you only submit **egul_sched.c**

4.1 Error Checking

If a value is passed into any of your functions through a given API (such as the 11 required functions), 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. Remember, you're writing a tiny bit of supporting code for a large project. You never want to be the one who causes a deliverable to SEGFAULT for the customer just because someone else used your library function wrong.

If you have a **static helper function** that takes a pointer, you do not need to check the pointer for being valid. These are internal functions that you have more direct control over how they are being used.

If there is an error on any step (eg. invalid input, malloc returning NULL, or anything that would cause a SEGFAULT), then you should return an error, as specified. **You do not have to free anything on errors.**

4.2 egul_sched.c Function API References (aka. what you need to write)

This section specifies exactly what each of your 11 functions need to do.

Egul_schedule_s *egul_initialize()

Creates a new Egul Scheduler Header with initial values.

- Create an Egul Schedule (`Egul_schedule_s`) struct and initialize it.
  - All allocations within this struct must be dynamic, using **malloc** or **calloc**.
- The Header contains pointers to 3 Queue (`Egul_queue_s`) structs, which themselves **also need to be dynamically allocated and initialized**!
  - Each of the three Queue structs also contains a count of the number of items in its Linked List, which must be initialized to 0.
  - Each of the three Queue structs contain a pointer to the head of a singly linked list, which must be initialized to NULL.
  - Each of the three Queue structs also contains a tail pointer. This will not be checked and you may either ignore this or use it as you see fit.

On any errors, return NULL, otherwise return a pointer to your new Egul Schedule.

Egul_process_s *egul_create(Egul_create_data_s *data)

Create a new Process struct and initialize its members.

*Reference for the **data** struct:*

```
typedef struct create_data {
  char original_cmd[MAX_CMD];    // Original user-input command
  int priority_level;            // Priority level of the Process
  int is_critical;               // 1 if the process is run with critical permissions
  pid_t pid;                     // OS Generated, Guaranteed Unique
} Egul_create_data_s;
```

Note: For this function, you can assume pid, priority, and is_critical are all valid values passed in. PID is also guaranteed to be unique for this process.

- Create a new Process Node (`Egul_process_s`) and initialize it with these values.
  - Set the Ready State bit of the **state** member to a 1.
    - Only one of the four State bits should be set (1) at any given time.
    - This means Running, Suspended, and Terminated bits should be 0s.
  - Set the Critical bit of **state** to be 0 if `data->is_critical` is false (0) or 1 if true.
  - Set the lower 11-bits of **state** to be all 0s.
  - Initialize **priority** to the passed in priority value from `data->priority_level`.
  - Initialize **age** to 0.
  - Initialize the **pid** member to the passed in pid value from `data->pid`.
  - Allocate memory (**malloc**) for the **cmd** member for `data->original_cmd`.
  - String Copy (**strncpy**) `data->original_cmd` into your struct's **cmd** member.
  - Initialize the **next** member to NULL.
- Return a pointer to your new process.

Return a pointer to the process on success, or NULL on any errors.

int egul_insert(Egul_schedule_s *schedule, Egul_process_s *process)

Inserts a Process Node into the Ready Queue and puts it in the Ready State.

- Set the Ready State bit of the **state** member to a 1.
  - Only one of the four State bits should be set (1) at any given time.
  - This means Running, Suspended, and Terminated bits should be 0s.
  - Make sure to set this without changing the Critical bit.
- Insert the Process Node (`Egul_process_s`) struct **at the end** of the Ready Queue.
  - If the head pointer is NULL, then this will be your first Process, and the appropriate Ready Queue's head pointer will point to this process.
- Remember to update the Queue's **count** member.

There will never be any Dummy Nodes in any of your Linked Lists.
ie. **head** Pointers always point to either NULL or to a valid Process in the List.

Return 0 on success or -1 on any error.

int egul_count(Egul_queue_s *queue)

Returns the size of the given Queue

- Return the number of Process nodes in the Linked List of the given Queue.
  - You should have this up to date in your Queue's **count** member.

Return the count on success or -1 on any errors.

Egul_process_s *egul_select(Egul_schedule_s *schedule)

Choose the best process to run, remove it from the Ready Queue, then return its pointer.

- **Algorithm to find the best process to choose**:
  - If there are **Critical** processes, the best process is first one found in the Ready Queue. *(The one closest to the head node)*
  - Otherwise….
    - If there are **Starving** processes (**`age >= STARVING_AGE`**), the best process is the first one found that is starving in the Ready Queue. *(The one closest to the head node)*
    - Otherwise…
      - Select the highest Priority process.
        - The lower the **priority** member, the higher the Priority
        - (eg. priority == 42 is higher than priority == 200)
  - **If there are ties in any category, always select the first one found in the Queue.**
- Once you have found the best process, remove that process from the Ready Queue list.
  - Remember to update the pointers!
- Then set the chosen process' **age** to 0 (it was just picked)
- Set the Running State bit of the **state** member of the selected process to a 1.
  - Only one of the four State bits should be set (1) at any given time.
  - This means Ready, Suspended, and Terminated bits should be 0s.
  - Make sure to set this without changing the Critical bit.
- Set the chosen process' **next** to NULL.
- Increment the **age** member for all processes remaining in the Ready Queue.
- Finally, return a pointer to that **same** process you just removed from the linked list.

Return pointer to the same chosen process on success, or NULL if Queue was empty or Errors.

int egul_suspend(Egul_schedule_s *schedule, pid_t pid)

Remove the process from Ready Queue, change state to Suspended, insert to Suspended Queue.

Note: For this function, you can assume pid is unique and is a valid PID or is 0 when passed in.

- If a process with the same **pid** is in the Ready Queue, remove that process from the List
  - Remember to update the pointers!
- OR, if the **pid** parameter is 0, remove the **first (head)** process from the Ready Queue.
- Set the Suspended State bit of the **state** member of that removed process to a 1.
  - Only one of the four State bits should be set (1) at any given time.
  - This means Ready, Running, and Terminated bits should be 0s.
  - Make sure to set this without changing the Critical bit.
- Set the removed process' **next** to NULL.
- Finally, insert this same process at the end of the Suspended Queue.

Return a 0 on success, or -1 on any errors or if Process was not found.

int egul_resume(Egul_schedule_s *schedule, pid_t pid)

Remove the process from Suspended Queue, change state to Ready, insert to Ready Queue.

Note: For this function, you can assume pid is unique and is a valid PID or is 0 when passed in.

- If a process with the same **pid** is in Suspended Queue, remove that process from the List
  - Remember to update the pointers!
- OR, if the **pid** parameter is 0, remove the **first (head)** process from Suspended Queue.
- Set the Ready State bit of the **state** member of that removed process to a 1.
  - Only one of the four State bits should be set (1) at any given time.
  - This means Suspended, Running, and Terminated bits should be 0s.
  - Make sure to set this without changing the Critical bit.
- Set the removed process' **next** to NULL.
- Finally, insert this same process at the end of the Ready Queue.

Return a 0 on success, or -1 on any errors or if Process was not found.

int egul_exited(Egul_schedule_s *schedule, Egul_process_s *process, int exit_code)

Insert the given Process into the Terminated Queue and return 0 on success.

- *This is called when a process that was on the CPU (not in your queues) finishes.*

Note: For this function, you can assume exit_code is valid and >= 0 and <= 255 when passed in.

- Set the Terminated State bit of the **state** member of the process to a 1.
  - Only one of the four State bits should be set (1) at any given time.
  - This means Ready, Running, and Suspended bits should be 0s.
  - Make sure to set this without changing the Critical bit.
- Set the `state` bits used for `exit_code` (bits 0-10) to the value of the exit_code passed in.
  - You will set the lower 11 bits of the passed in exit_code as the lower 11 bits of your state member of the Process Node with bitwise operators.
- Finally, insert the same process at the end of the Terminated Queue.

Return the 0 on success or -1 on any error.

int egul_killed(Egul_schedule_s *schedule, pid_t pid, int exit_code)

Move a process from either the Ready Queue or Suspended Queue into the Terminated Queue.

- *This is called when a process that is in one of your queues is terminated by the user.*

Note: For this function, you can assume exit_code is valid and >= 0 when passed in.

- Find the Process with matching pid **in either Ready Queue** or **Suspended Queue**
  - Remove that Process from the Queue if found and set its next pointer to NULL.
    - Do not free, delete, or clone this! You will be working with a pointer to the **same struct** you found in the linked list.
- Set the Terminated State bit of the **state** member of that process to a 1.
  - Only one of the four State bits should be set (1) at any given time.
  - This means Ready, Running, and Suspended bits should be 0s.
  - Make sure to set this without changing the Critical bit.
- Set the `state` bits used for `exit_code`(bits 0 – 10) to the value of the exit_code passed in.
  - You will set the lower 11 bits of the passed in exit_code as the lower 11 bits of your state member of the Process Node with bitwise operators.
- Finally, insert the same process at the end of the Terminated Queue.

Return the 0 on success or -1 on any error or if the PID is not found.

int egul_get_ec(Egul_process_s *process)

Extracts and returns the exit_code bits from the state member of the process.

- If the process passed in is in the Terminated State…
  - Extract the exit_code bits (bits 0 – 10) from the `state` member of the process.
  - Return an int with the same value as the exit_code bits that were just extracted.

Return the exit_code value from the process on success or -1 on any error or if the Process passed in is not in the Terminated State.

void egul_deallocate(Egul_schedule_s *schedule);

Cleans up the Egul system, freeing all memory that had been allocated in your code.

- Free all Nodes from each Queue in the Egul Schedule.
  - Remember to free the **cmd** Strings!
- Free all Queues from the Egul Schedule
- Free the Egul Schedule

There is nothing to return from this void function.

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 or CS222 at GMU) in C programming. Chapter 2.1-2.3 of our textbook also describes the use of Bitwise operations in Systems Programming, and the material for the first week of instruction in this course will also be helpful when working with bitmasks and operations.

You will use bitwise operations more extensively for the next project, so it's good practice here.

For practical context, in our model, your scheduler uses an algorithm called **Priority Scheduling** with **Starvation Protection** to select the process to run on the CPU next. Each process will run for a small period of time, called a quantum, and then get returned to the appropriate Ready Queue if not finished, or to the Terminated Queue if it finished in that time period.

(This is actually a real OS CPU Scheduling Algorithm that you'll study later in CS 471)

5.1 Memory Checking

To check for memory leaks, use **valgrind** *(GDB and Valgrind refresher Videos are on Canvas)*

```
kandrea@zeus-1:handout$ valgrind ./strawhat
```

You are looking for a line that says:

All heap blocks were free -- no leaks are possible

```
(PID: 406455) [StrawHat]$  quit
...
   [Status] Deallocating all Processes.
 ==406455==
==406455== HEAP SUMMARY:
==406455==     in use at exit: 0 bytes in 0 blocks
==406455==   total heap usage: 14 allocs, 14 frees, 4,784 bytes allocated
==406455==
==406455== All heap blocks were freed -- no leaks are possible
==406455==
...
```

Important Note: If you had processes still running inside of StrawHat, then Valgrind will report on ALL of them! This means you might see other programs that have leaked memory, even though StrawHat was clean. So, how can we tell which lines refer to StrawHat and which are for the other processes?

On the left of the "All heap blocks were freed" line above, you can see ==406455==. This is the PID of the process it's reporting on. Now, look at the top of that same box and you can see the StrawHat prompt, which says "(PID: 406455) [StrawHat]". That tells you this is the same process with no leaks.

So, whenever you're testing with Valgrind, make sure to only look at lines starting with the same PID that you can see on your StrawHat prompt lines. If the PID is different, then it's not any worry for you.

If you do find any leaks in Valgrind for your StrawHat process, you can use the "--leak-check=full" option to get more information. If you find any memory errors, you can look at them to get more information about what lead to them. You should see the line of your code near the top of each error:

```
==539428== Invalid read of size 8
==539428==    at 0x4020FB: egul_insert (egul_sched.c:89)
==539428==    by 0x401075: cs_thread (strawhat_cs.c:104)
```

This shows there was an error caused by your `egul_insert` code on line 89 of your `egul_sched.c` file. We won't be grading on any of these Errors (like Invalid Read), only the Heap Blocks were Freed message, however, these usually indicate a problem that you likely want to fix.

Again, if you exit StrawHat when it has non-terminated processes in it, you may see memory leaks from those other processes. This is fine.

We'll only check for memory leaks when StrawHat properly exits with no processes running.

6 Testing your Code

There is one other option that we have in this project. This is not necessary to use at all, however, if you want to test just your code **without running/involving StrawHat**, we do have a special `src/test_egul_sched.c` source file that has a main you can use to call your functions with whatever arguments you want and you can then look at the outputs to test your functions in isolation first.

The reference for testing your code in this way is in the **P1_Self_Testing.pdf** document.

7 Submitting and Grading

Submit this assignment electronically on Canvas. **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 `egul_sched.c`**

Do NOT make any .tar or .zip files. You should submit just egul_sched.c as a C file in Canvas.

You can make multiple submissions; but we test and grade ONLY the latest version that you submit.

Important: **Make sure to submit the correct version of your file on Canvas!** Submitting the correct version late will incur a late penalty (and use late tokens automatically); and submitting the correct version 48 hours after the due date OR a submitting the wrong file will not bring any credit.

Your code must compile to receive any points from testing.

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.
    - **Rubric Breakdown for the 20 points: (Posted on Canvas as well)**
        - **Comments:** Add enough comments to help us understand 'why' you wrote code. There is no fixed number of comments needed, but the TAs should be able to understand what your code is doing from the comments at a big level. If it's easy to follow, then you're doing fine.

        - **Organization**: You may make as many helper functions, #define constants, or any other organization helpers you like. We have no rules on how long a function can be at max, but very large functions are hard to read, hard to follow, hard to debug, and hard to test. You may not need helper functions based on your design, or they may be very helpful indeed. Make sure your code is easy to follow, and you'll do fine. **You can only modify egul_sched.c though.**

        - **Correctness:** You need to check pointers before using them to ensure they are not NULL, initialize all local variables on creation, and generally make sure that you're not letting errors flow through your code.

        - **Required Components:** These points will be for specifically using bitwise operations when working with your state member (eg. using &, |, <<, >>, ~, ^) as needed to work with the bits. Hardcoding state with large if/else branches is not going to work for this project. You should be setting/clearing individual bits of the state without affecting the others. This also requires you to use Linked Lists.

- **80 points - Automated Testing (Unit Testing).** We will be building your code using only your submitted `egul_sched.c` and running unit tests on your functions.
    - It must compile cleanly on Zeus to earn any points.
    - Each function will be tested independently for correctness by our scripts.
        - Partial credit is possible.
    - **Border cases and error cases will be checked!**
    - Only legal and valid commands will be tested.
        - ie. Only commands that run processes will be checked, since your code had nothing to do with the shell or CS Engine part of this code.

8 Document Changelog

- v1.0 (build 4a99cc46)
  - Release Version
- v1.1 (build 4a99cc46)
  - On Pages 2 and 14, changed the type of the data parameter to **Egul_create_data_s ***
    - It was incorrectly listed in the initial version due to a typo.