CS 367 Project 1 - Spring 2025:
Wlf CPU Scheduler
Due: Friday, February 07, 2025 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 **Wlf CPU Scheduler**. Finish the functions in **src/wlf_sched.c** to implement a CPU scheduler for our StrawHat Task Manager.

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

Table of Contents:

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

- StrawHat is a lightweight process-level task manager that allows users to interleave Linux programs with custom execution techniques. A task manager is useful for testing complicated interactions between processes (programs being run) by manually scheduling them in certain orders or allow the user to run processes with custom priority orderings.

So, what is CPU scheduling and how does it work?

- The job of a CPU Scheduler is to decide which processes (programs being run) should be run, and when. The idea is to pick a process to **run** for a very short amount of time on the CPU. Then, put it back in a **Ready Queue** (linked list) and pick another to run until all are finished.
- As long as you let each process run for a tiny amount of time, and keep swapping them, then it seems like your Operating System is running many different programs at the same time! This is the main idea of **multitasking.** For this Project, you will be writing the missing code needed to complete the **Wlf CPU Scheduler.**

Project 1 is to finish functions for the Wlf CPU Scheduler to manage the Process Queues.

Project Overview

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

You will be finishing code in **src/wlf_sched.c** to create, insert, remove, and find process nodes on **three singly linked lists** in C. Each linked list represents a queue to manage processes in this task manager. You will also be using some **bitwise operators** in C to work with process states.

Your code (**wlf_sched.c**) works with pre-written files to implement several operations. You will be maintaining three singly linked lists (**Ready Queue – High Priority, Ready Queue – Normal Priority, and a Terminated Queue**). The structs for all of these are defined in **inc/wlf_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 wlf_sched.c to do operations.

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

```
Wlf_schedule_s *wlf_initialize();
```

- Creates the Wlf Schedule struct and all three singly Linked List Queues.

```
Wlf_process_s *wlf_create(Wlf_create_data_s *data);
```

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

```
int wlf_enqueue(Wlf_schedule_s *schedule, Wlf_process_s *process);
```

- Insert the Process node at the END of the appropriate **Ready Queue** Linked List.

```
int wlf_count(Wlf_queue_s *queue);
```

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

```
Wlf_process_s *wlf_select(Wlf_schedule_s *schedule);
```

- Remove and return the best Process in the appropriate **Ready Queue** Linked List. (Details in Section 4)

```
int wlf_promote(Wlf_schedule_s *schedule);
```

- Increments the age of all Processes in the **Ready Queue – Normal** Linked List, moving Starving to **High**.

```
int wlf_exited(Wlf_schedule_s *schedule, Wlf_process_s *process, int exit_code);
```

- Inserts the Process node in the **Terminated Queue**

```
int wlf_killed(Wlf_schedule_s *schedule, pid_t pid, int exit_code);
```

- Remove a Process from the appropriate **Ready Queue** and Insert to the **Terminated Queue**

```
int wlf_reap(Wlf_schedule_s *schedule, pid_t pid);
```

- Remove the process with given pid from the **Terminated Queue** and returns their exit code.

```
int wlf_get_ec(Wlf_process_s *process);
```

- Returns the value of the exit code for the given process in the **Terminated Queue**.

```
void wlf_cleanup(Wlf_schedule_s *schedule);
```

- Free the Wlf 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)

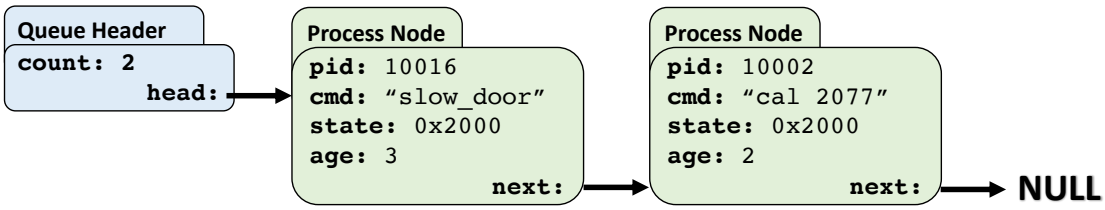
The StrawHat (strawhat) program is written for you but needs your Wlf code to run. In this manner, Wlf (pronounced Wolf) is a library that StrawHat uses to do the process scheduling.

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 give it back to you to put at the end of the queue, so it can continue running later. So, the programs will alternate back and forth until they're all finished.

The task manager runs on Linux and implements custom **multitasking** of 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 either **Ready Queue – High** or **Ready Queue - Normal**, based on its settings. These ready queues are each just **Singly Linked Lists**.

We have one ready queue, **Ready Queue – High** priority where all high-priority processes wait for their turn to run on the CPU. A second queue, which is the default priority queue, is called the **Ready Queue – Normal** priority, which is where non-high-priority processes go to wait for their turn. A third queue, the **Terminated Queue**, is where all processes that have exited normally (or have been terminated by the OS) are stored.

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



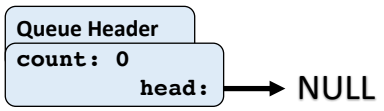
Each Linked List starts from a Queue (Wlf_queue_s *). This struct is used to hold a pointer for the head for each linked list, along with a count value to hold the number of nodes currently in that linked list. This means that the head of the linked list is inside of this queue header, which is passed into many of your functions.

This way you can change the head pointer without having to use double-pointers in C!

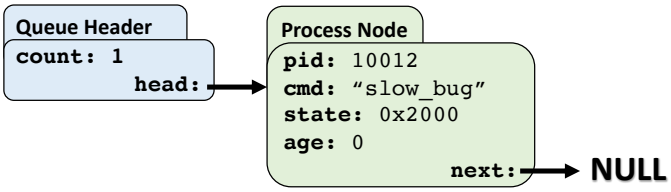
Each **head** pointer will only be pointing to NULL or to a valid node. **You will not use any dummy nodes in this project.**

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 List**.

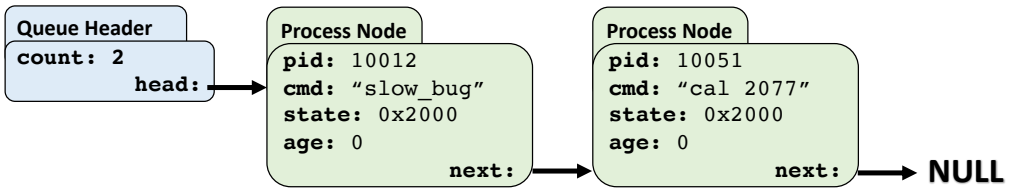
For example, here is a set of diagrams of starting with an empty **Ready Queue – Normal** list.



Next, we have process "slow_bug" with Process ID (PID) 10012, which is added as the only node.



Then we start process "cal 2077" with PID 10051 and add it to the **end** of the Linked 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 two Ready Queues to keep track of all the processes that are ready to run. The **Ready Queue – High** Priority linked list holds all processes ready to run with high priority. StrawHat will always try and select a process from this Linked List first to run. If this queue is ever empty, then StrawHat will try to select a process from the **Ready Queue – Normal** Priority linked list. So, these processes in the normal priority list will run much less often.

When StrawHat needs a next process to execute on the CPU, it will call your **wlf_select** function, which uses your code to identify the best node (details in Section 4) from the queues, removes it from its queue, and returns a pointer to it.

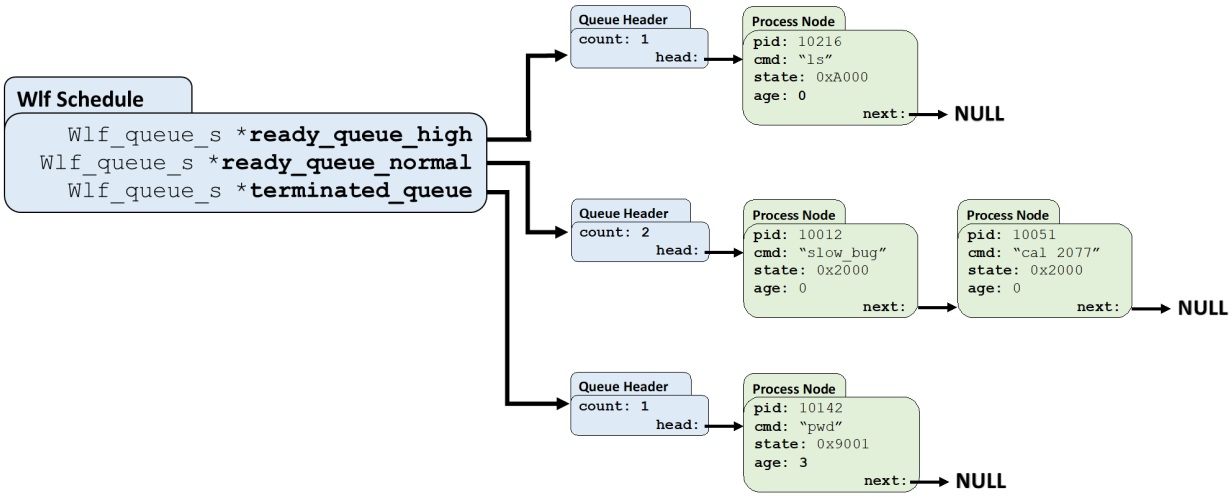
Using a timer, StrawHat will run the process you returned to it for a set amount of time (by default, this is 250ms). It will then return that process to you by either calling your **wlf_enqueue** function to put it back in the end of the proper Ready Queue so it can resume running later; or, if it is already finished, StrawHat will call your **wlf_exited** function to put it into the **Terminated Queue**, which tracks all finished or terminated processes. These do 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, then it will run it for 250ms, then it will call your function to put it back into the Ready Queue so it can run again in the future.

When you have multiple processes running, each of them runs for a small amount of time, then returns them to go to the next one, in a cycle, until all end up in your Terminated Queue.

To help you manage these three Linked Lists (Queues), you have another struct, **Wlf_schedule_s** that contains pointers to them all. (You won't need any dummy nodes at all.)

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



Your functions will create the Wlf Schedule, then create each of the three Queue Headers, and then will be responsible for creating Process Nodes, 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. It's ok, because you will only be writing code with one file in the **src/** directory, called **wlf_sched.c**

Your code (**wlf_sched.c**) consists of 11 functions that will be called from StrawHat's main code. Each function does exactly one job, as is described in Chapter 4 of this document. You are free to make any

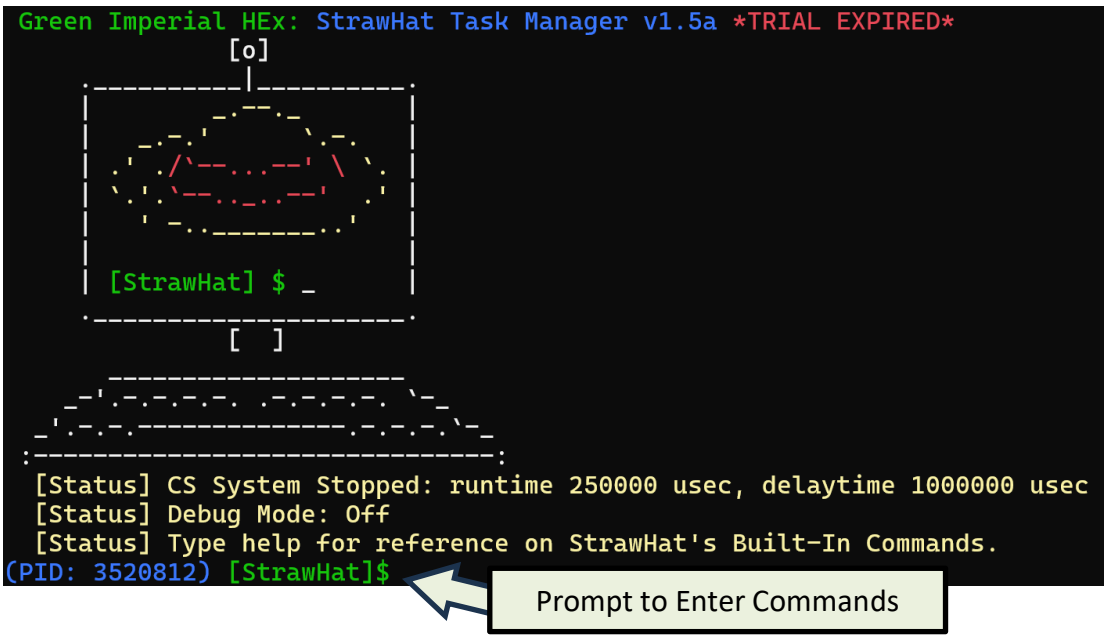
number of helper functions that you want, of course. Your functions will be maintaining all three singly linked lists (**Ready Queue - High**, **Ready Queue - Normal**, and a **Terminated Queue**).

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

Helper Functions are highly encouraged!

2.2 StrawHat Details

StrawHat is not a simulator. It provides a shell that you can use to run normal non-interactive Linux commands like **ls** and **cal**, or any program you want to run locally! 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

The Wlf Scheduler, which you will be writing, works with Processes using a few different concepts. The first is that every Process has its own 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 system custom type. It is just a typedef name for an **int** in Linux. 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 can also be **Normal** or **High** priority. All processes default to normal priority so we don't have anything special to designate them. The Priority Level is used to determine which processes should run first. If you start a process on the command prompt using the **-h** flag (we'll show the commands later), then it will be a high priority process instead.

The normal priority processes will all be enqueued (inserted) at the end of the **Ready Queue – Normal**, which is a singly-linked list where all normal priority processes are kept. Any high priority processes will instead be enqueued at the end of the **Ready Queue – High**, which is just like the other ready queue, but is for high priority processes.

Processes from the **Ready Queue - High** linked list should always be selected (picked) first. Only if there are no high priority processes, will we select a process from the **Ready Queue - Normal**. This, however, leads to a major problem in CS called **starvation**, where a higher priority process may prevent low priority processes from ever running!

To fix this, StrawHat uses a common solution in CS called **aging**. We will track how long a process has been waiting in the **Ready Queue - Normal** linked list. After a process is selected, we call your **wlf_promote** function to increment the age of all remaining processes in the **Ready Queue - Normal**.

To fix this problem of starvation, after we increment the age, we'll **promote** any **starving** process, which is a process that has an age greater than or equal to a **pre-defined constant** called **STARVING_AGE**. So, if there is such process, then after their age is incremented, they will be removed from the **Ready Queue - Normal** linked list and enqueued to the **end** of the **Ready Queue – High** linked list.

Any promoted processes will then get a fair chance to run. This doesn't make them a high priority process; it just lets them live in the **Ready Queue – High** so they can get selected. Once they are returned, they'll go back to their normal **Ready Queue - Normal** linked list, since they got a little chance to run.

Whenever a process is selected to run on the CPU, we'll also be changing its age to 0.

The result of this is that you will see high priority processes running a lot, and low priority processes running a little bit every once in a while, as they each become starving.

There is, however, one more concept that StrawHat has for processes: **Critical** priority level. 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 other processes ahead of it in the linked list.

So, a critical process will always run first, and exclusively, until it exits. The good news is that all Critical processes will only be in the **Ready Queue – High** list. (The -c flag will automatically also add -h)

So, the actual algorithm you'll see in Section 4 when you need to select the best process to run is you'll first look to see if there are any **Critical** processes in the **Ready Queue - High**. If so, you select the first one you find. Otherwise, you'll take the first process in the **Ready Queue - High**. If none exist, then you'll select the first process from the **Ready Queue - Normal**.

Then, after all of that, we'll call your **promote** function, where you'll increment the age of all processes in the **Ready Queue - Normal**, and move any that are starving up to the end of the **Ready Queue – High**.

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

2.4 Process States

When you are working with Process Nodes (**Wlf_process_s** structures), 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, we first need to do 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 here that since processes run for such a tiny amount of time, you have to be lucky to see this state on the schedule.

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

When a Process finishes, it'll be moved into the **Terminated Queue**, which keeps a list of all the killed processes. Anything in the Terminated Queue will be in a **Terminated State (T)**.

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

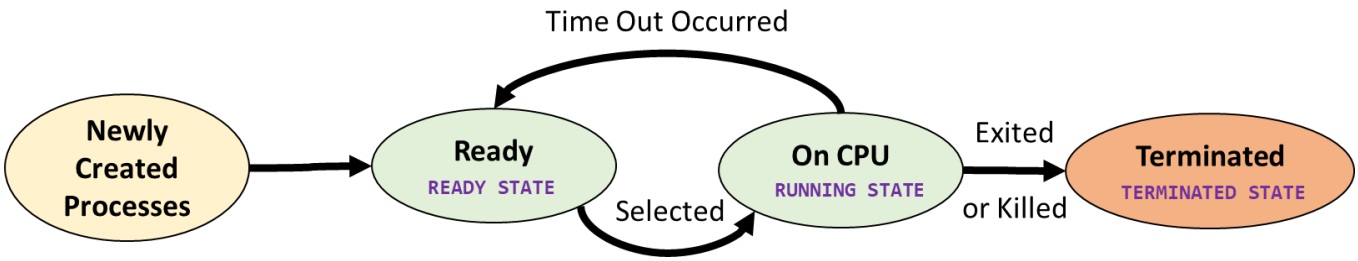


Figure 1: Process State Diagram

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 - High in the **Ready State** and your `wlf_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.

2.5 Process Node State

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

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

&	^	>>	<<

Description	State Flags			Crit	High	Unused (All 0s)			Exit Code (for Terminated Processes)							
Field Name	R	U	T	C	H	0	0	0	exit_code							
Bit number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit Numbers
(Counting from the Right!)

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

All Flags are individual bits inside of the **unsigned short** called **state** in the struct. Each bit will either be 0 or a 1. If the bit is a 1, it represents that Flag is set. If the bit is a 0, it means that Flag is not set.

State Flags: Bits 13-15 represent the current State of the Process. (1-bit values)

R = `READY STATE` U = `RUNNING STATE` T = `TERMINATED STATE`

Critical Flag: Bit 12 is a flag representing if the process was set with Critical Run. (-c)

C = `CRITICAL FLAG`

High Flag: Bit 11 is a flag representing if the process was set with High Priority. (-h)

H = `HIGH FLAG`

Bits 0-7 are the lower 8 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.

Example: Process in the **Terminated State**, run as **Critical** and **High**, with exit_code of **6**

Bin: 0011 1000 0000 0110

Hex: 3 8 0 6

So, `state` = 0x3806

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

2.6 Schedule, Queue, and Process struct overviews (wlf_sched.h)

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

The overall struct of type **`Wlf_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.

```
/* Wlf Schedule Struct Definition */
typedef struct wlf_schedule {
  Wlf_queue_s *ready_queue_high;        // Linked List of High Priority Processes ready to Run
  Wlf_queue_s *ready_queue_normal;      // Linked List of Normal Priority Processes ready to Run
  Wlf_queue_s *terminated_queue;        // Linked List of Terminated Processes
} Wlf_schedule_s;
```

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

Wlf Schedule contains pointers to three **`Wlf_queue_s`** type structs, called **`ready_queue_high`**, **`ready_queue_normal`**, and **`terminated_queue`**. Each of these three linked lists must be allocated and initialized as well. (Remember to initialize ALL values!)

Queue Schedule Struct

A **`Queue Schedule`** struct contains a pointer to an **`Wlf_Process_s Node`** 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 should point to either NULL or the first Process.

```
/* Queue Schedule Definition */
typedef struct queue_header {
  int count;              // How many items are in this linked list?
  Wlf_process_s *head;    // Points to the FIRST node of Linked List.  No Dummy Nodes.
  Wlf_process_s *tail;    // Points to the LAST node of Linked List.  No Dummy Nodes. (Optional)
} Wlf_queue_s;
```

Process Node Struct

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

```
/* Process Struct Definition */
typedef struct process_node {
pid_t pid;                  // PID of the Process you're Tracking
char *cmd;                  // Name of the Process being run
unsigned short state;       // Current State of Process, Critical and High Flags, and the Exit Code
int age;                    // How long this has been in the Ready Queue - Normal since last run
struct process_node *next; // Pointer to the next Process Node in a Linked List
} Wlf_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`**" or "**`slow_countup 20 -h -c`**". Every process on the Operating System also has a unique Process ID (PID), which is stored here as a pid_t (an **int**) called **`pid`**.

You will also have a 16-bit unsigned int **`state`**, which contains several pieces of data that are combined together using bitwise operations, as specified in Section 2.5.

3 Building and Running the StrawHat (./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.tar**, which will create a handout folder on Zeus.

```
kandrea@zeus-1:p1$ tar -xvf project1_handout.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 **wlf_sched.c**, which is the only file you will be modifying and submitting. You will also have very useful header files (**wlf_sched.h**) along with other files for the virtual machine itself in the **inc/** directory.

3.1 Building StrawHat

To build StrawHat, run the make command: *(This will be a lot of output the first time your run it!)*

```
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 a lot of bad practices and makes 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 of the functions in **wlf_sched.c**. You may create any additional functions that you like, but you cannot modify any other files; you only submit **wlf_sched.c**

4.1 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.

Error Checks You Need To Make:

If you are receiving a pointer from an external function, **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 in your company used your function wrong.

If you are passing yourself a pointer to a helper function, it is still a good practice to check for NULL, but it is only required for the 11 functions you are required to write.

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, simply return the error return value for that function.

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

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

`Wlf_schedule_s *wlf_initialize(void);`

Creates a new Wlf Scheduler Schedule with initial values.

- Create an Wlf Schedule (Wlf_schedule_s) struct and initialize it.
  - All allocations within this struct must be dynamic, using **malloc** or **calloc**.
- Allocate and initialize a Wlf_queue_s struct, for each queue pointer in the Schedule.
  - Each of the three Queue structs also contains 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 count for the number of items in its Linked List, which must be initialized to 0.
- Return a pointer to your fully initialized Schedule struct.

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

Wlf_process_s *wlf_create(Wlf_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 is_high;                   // 1 if the process is a High Priority process
  int is_critical;               // 1 if the process is run with critical permissions
  pid_t pid;                     // OS Generated, Guaranteed Unique
} Wlf_create_data_s;
```

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

- Create a new Process Node (Wlf_process_s) and initialize it with these values.
  - Set the Ready State bit of the **state** member to a 1.
    - Only one of the three State flags should be set (1) at any given time.
    - This means Running and Terminated bits should be 0s.
  - Set the Critical bit of **state** to be 0 if **is_critical** is false (0) or 1 if true.
  - Set the High bit of **state** to be 0 if **is_high** is false (0) or 1 if true.
    - Note, also set the High bit to 1 if **is_critical** is true.
    - Critical processes are always High Priority.
  - Set the lower 8-bits of **state** to be all 0s.
  - Initialize **age** to 0.
  - Initialize the **pid** member to the pid argument.
  - Allocate memory (**malloc**) for the **cmd** member to be big enough for command.
  - String Copy (**strncpy** for safety!) command into your struct's **cmd** member.
  - Initialize the **next** member to NULL.

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

int wlf_enqueue(Wlf_schedule_s *schedule, Wlf_process_s *process);

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

- Set the Ready State bit of the **state** member to a 1.
  - Only one of the three State flags should be set (1) at any given time.
  - This means Running and Terminated bits should be 0s.
  - Make sure to set this without changing the Critical or High bits.
- Insert the Process node (Wlf_process_s) to the end of the Appropriate Ready Queue.
  - If the High or Critical bit is 1, insert at the end of Ready Queue – High linked list.
  - Otherwise, insert the at the end of the Ready Queue – Normal linked list.
- In either instance, 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 wlf_count(Wlf_queue_s *queue);

Returns the number of Process nodes currently in the given Queue's Linked List.

- 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.

Wlf_process_s *wlf_select(Wlf_schedule_s *schedule);

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

- **Algorithm to find the best process to choose:**
  - If there are any Processes in the **Ready Queue – High** Linked List, start there:
    - If there are **Critical** processes, the best is the first one in the list.
    - If not, then the best is the first process in the list.
  - Only if there are none in Ready Queue – High, then go to **Ready Queue – Normal**:
    - The best is the first process in the list.
- Once you have found the best process, remove that process from the Linked List it is in.
  - Remember to update the pointers!
- Then set the selected process' **age** to 0 (it was just picked)
- Set the Running State bit of the **state** member to a 1.
  - Only one of the three State flags should be set (1) at any given time.
  - This means Ready and Terminated bits should be 0s.
  - Make sure to set this without changing the Critical or High bits.
- Then set the selected process' **next** to NULL.
- Finally, return a pointer to that **same** process you just removed from the linked list.

Return the pointer to the same selected process or return a NULL if the two Ready queues were both empty or if any Error conditions were found.

int wlf_promote(Wlf_schedule_s *schedule);

Increment age for all Ready Queue – Normal processes and moves starving ones to High Queue

- Increment the age field for every process in the Ready Queue – Normal linked list.
- **After** incrementing the age field, if any process has **age >= STARVING_AGE**
  - Remove it from this linked list properly.
  - Set the chosen process' next to NULL.
  - Add it to the end of the Ready Queue – High linked list.
  - Do not change the **age** when moved to the Ready Queue – High.
  - Do not change any flags or states for is process.

Note: STARVING_AGE is already a defined constant and must be used. Its value is set to 5.

Return a 0 on success, or -1 on any errors.

int wlf_exited(Wlf_schedule_s *schedule, Wlf_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 will fit in 8-bits and >= 0 when passed in.

- Set the Terminated State bit of the Process' **state** member to a 1.
  - Only one of the three State flags should be set (1) at any given time.
  - This means Ready and Running bits should be 0s.
  - Make sure to set this without changing the Critical or High bits.
- Set the state bits used for exit_code to the value of the exit_code passed in.
  - You will set the value of the passed in exit_code as the lower 8 bits of your state member of the Process Node with bitwise operators.
- Do not change the **age** of the process.
- Finally, insert the same process at the end of the Terminated Queue.

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

int wlf_killed(Wlf_schedule_s *schedule, pid_t pid, int exit_code);

Move a process from either of the two Ready Queues into the Terminated Queue.

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

Note: For this function, you can assume exit_code will fit in 8-bits and >= 0 when passed in.

- Find a Process with pid in either **Ready Queue – High** or **Ready Queue – Normal** lists.
  - 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 to a 1.
  - Only one of the three State flags should be set (1) at any given time.
  - This means Ready and Running bits should be 0s.
  - Make sure to set this without changing the Critical or High bits.
- Set the state bits used for exit_code to the value of the exit_code passed in.
  - You will set the value of the passed in exit_code as the lower 8 bits of your state member of the Process Node with bitwise operators.
- Do not change the age of the process.
- Finally, insert the same process to the end of the Terminated Queue.

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

int wlf_reap(Wlf_schedule_s *schedule, pid_t pid);

Remove the process from Terminated Queue and free it, returning its exit code.

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

- If a process with the same **pid** is in Terminated 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 Terminated Queue.
- Get just the exit code from the status member of the Process and save it in a variable.
- Finally, completely free the reaped (removed) process.

Return a the Process' exit code on success, or -1 on any errors or if Process was not found.

int wlf_get_ec(Wlf_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 – 7) 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 wlf_cleanup(Wlf_schedule_s *schedule);

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

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

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 there are slides in the Week 1 of the Course Content and in the C Review section on Canvas on Bitwise 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 **Multi-Level Feedback Queue Scheduling** to select the process to run on the CPU next. Each process will run for a small period of time 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-2:handout$ valgrind ./strawhat
```

You are looking for a line that says:

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

```
(PID: 417541) [StrawHat]$  quit
  [Status] Exiting Program.
  [Status] Cleaning up strawhat environment.
...
==417541==
==417541== All heap blocks were freed -- no leaks are possible
...
```

Important Note: If you had processes still ready or running inside of the 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 ==417541==. 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: 417541) [StrawHat]"`.

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 of 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: wlf_enqueue (wlf_sched.c:89)
==539428==    by 0x401075: cs_thread (vm_cs.c:104)
```

This shows there was an error caused by my **wlf_enqueue** code on line 89 of your **wlf_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-killed 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.

If you see leaks or errors on the "at" line as part of the provided code (not involving wlf_sched.c at all), please let us know on Piazza and we can look into it.

You should also pay attention to the following two other errors from Valgrind:

```
==648450== Conditional jump or move depends on uninitialized value(s)
```

This above line means that you have an uninitialized variable that's being used for some sort of a check! The lines immediately following this will help you locate which variable you forgot to initialize.

```
==648936== Invalid write of size 1
```

This line happens when you overwrite an array or write outside of the memory you allocated. Usually, this happens when you are doing a **strncpy/strcpy** and you forgot to allocate space for the **\0**

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 the StrawHat**, we do have a special **src/test_wlf_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 wlf_sched.c**

Do NOT make any .tar or .zip files. You should submit just wlf_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 wlf_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.
- **80 points - Automated Testing (Unit Testing).** We will be building your code using your submitted wlf_sched.c and our Makefile 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 – Initial Release Version
- v1.1
    - On page 16, fixed the typo in function name wlf_get_ec