CS 367 Project 1 - Fall 2022:
Avan CPU Scheduler
Due: Friday, September 9th 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 finish writing functions for the **Avan CPU Scheduler**. Finish the function definitions in **src/avan_sched.c** to implement a CPU scheduler for the TRILBY Virtual Machine (VM).

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)*
TRILBY-VM is a lightweight process-level virtual machine that allows users to interlace Linux programs with custom execution techniques. These machines are 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 idea is to pick a process and run it for a very, very short amount of time. Then, you can put it back in a **ready queue** (linked list) and then pick another from that queue to run. As long as you let each process run for a tiny amount of time, and keep swapping them out, then it seems like your Operating System is running many different programs at the same time! This is the main idea of **multitasking.** You will be writing the **Avan CPU Scheduler.**

<u>Project 1 is to finish functions for the Avan Scheduler, to choose which process to run next.</u>

Table of Contents:

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 feature to the project.

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

Your code (**avan_sched.c**) works with pre-written files to implement several of these operations. You will be maintaining three singly linked lists (**Ready Queue, Suspended Queue, and Terminated Queue**). These structs are defined in **inc/avan_sched.h**.

You can add additional helper functions, but you cannot change how we compile it.

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

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

```
avan_header_t *avan_create();
```

- Creates the Avan Header struct, which has pointers to all three Linked List Queues.

```
int avan_insert(avan_header_t *header, process_node_t *process);
```

- Inserts the Process node in ascending PID order in of the Ready Queue Linked List.

```
int avan_suspend(avan_header_t *header, pid_t pid);
```

- Remove a Process from the Ready Queue and Add it (in PID Order) to the Suspended Queue

```
int avan_resume(avan_header_t *header, pid_t pid);
```

- Remove a given Process from the Suspended Queue and Add (in PID Order) to the Ready Queue

```
int avan_quit(avan_header_t *header, process_node_t *process, int exit_code);
```

- Inserts the Process (in PID Order) to Terminated Queue

```
int avan_terminate(avan_header_t *header, pid_t pid, int exit_code);
```

- Remove a Process from the Ready or Suspended Queues and Add (in PID Order) to Terminated Queue

```
process_node_t *avan_new_process(char *command, pid_t pid, int priority, int critical);
```

- Create a new Process node from the arguments and return it.

```
process_node_t *avan_select(avan_header_t *header);
```

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

```
int avan_get_size(queue_header_t *ll);
```

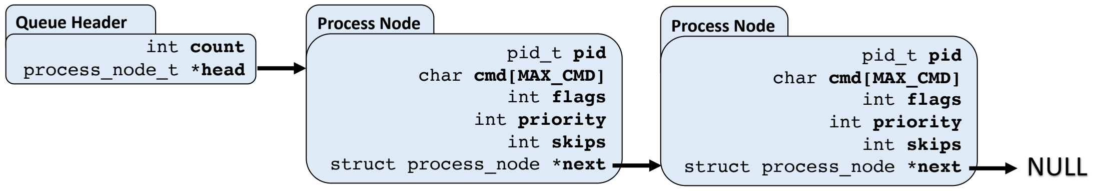
- Return the number of Nodes in the given Queue Linked List

```
void avan_cleanup(avan_header_t *header);
```

- Free the Avan Header, the Three Linked List Queues, and all of their Process Nodes.
- TRILBY-VM will run with no Memory Leaks

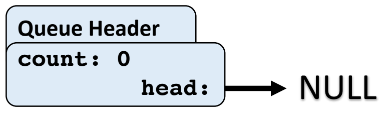
TRILBY-VM (this is the virtual machine that calls your functions and is already written for you)

The VM runs on top of Linux and implements custom **multitasking** of processes. The basic idea is that every time you start running a new process, what's really happening is that you're adding that process information to a node in a **ready queue**, shown below.

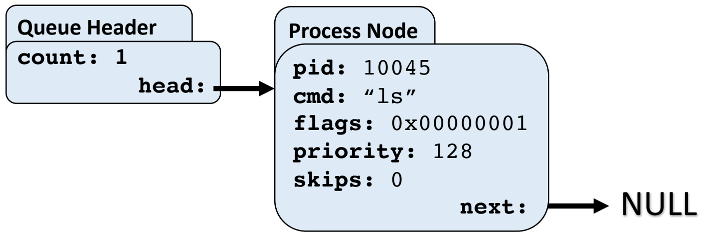


Each Linked List starts from a Queue Header (queue_header_t *) node. These nodes have a count that you can maintain and a pointer to the first Process Node in your linked list. (No dummy nodes!) For the **Ready** Queue, when a new process is created, you will add it to in Ascending PID order to the Ready Queue Linked List.

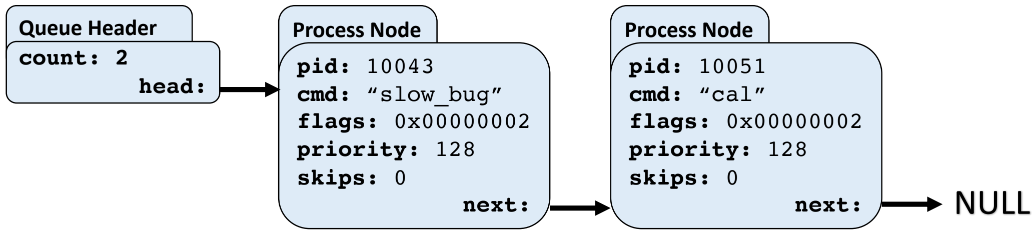
For example, here is a set of depictions of starting with an empty Ready Queue.



Next, start up process "ls" with Process ID (PID) 10045, which is added as the only node to start.



Then start process "slow_bug" with PID 10043 and add it to the front to keep in PID order.



Your code won't need to worry about starting any processes; that's all done by the TRILBY-VM, but your code will be called to create the process nodes and add them to the queue.

TRILBY-VM will use your Ready Queue to keep track of all of the processes that are ready to run. When TRILBY-VM needs the next process to execute on the CPU, it will call one of your functions (avan_select), which will remove the best node and return it.

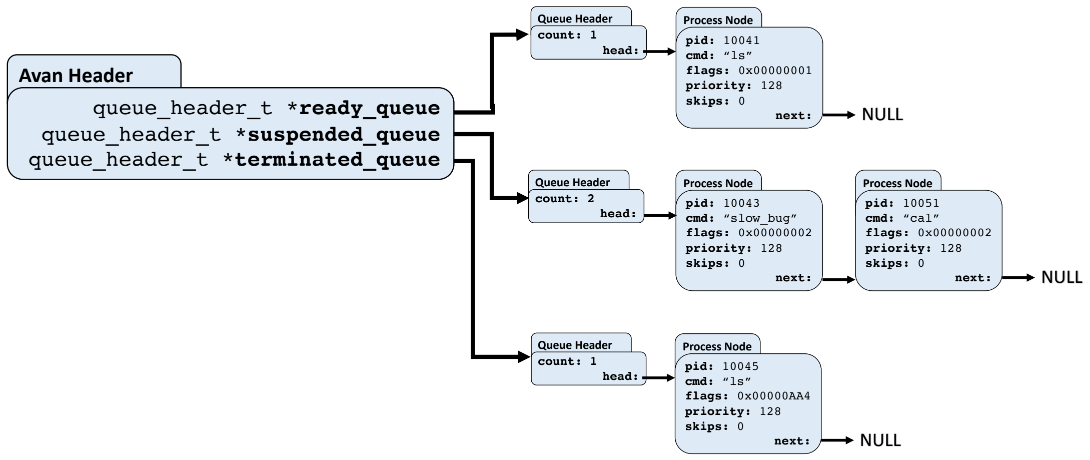
Using a timer, TRILBY-VM will run the process you returned to it for a set amount of time (by default, this is 250ms). It will then return that process to you by either calling your avan_insert function again to put it back in ascending PID Order in the Ready Queue, meaning it's still not finished and is ready to run again, or it will call another one of your functions to put it into the **Terminated Queue**, which tracks all terminated processes. These do not run again.

When TRILBY-VM 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, this will run each of them for a small amount of time, then return them go to the next one, in a big cycle, until all have ended up in your Terminated Queue.

During this operation, the user may also choose to **suspend** a process. This doesn't kill the process, it just moves it to a third queue, called the **Suspended Queue**. When a process is in the suspended queue, it doesn't get chosen to run on the CPU. Later, a user can **resume** a process, which will call your function to move that process node from the suspended queue back to the ready queue, where it can run again in the normal cycle.

In order to help you manage all of these three Linked Lists (queues), you have another struct that contains pointers to them all. (This also means you won't need dummy nodes.)

Here's a picture of what this looks like.



Your functions will create the Avan Header, 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

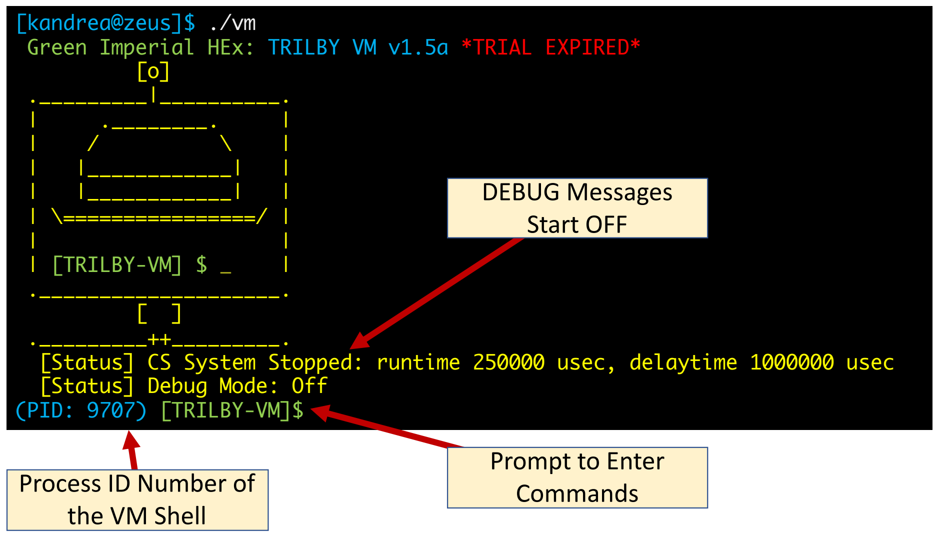
2.1 Source Code Files

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 of 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). 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 **avan_sched.c**

Your code (**avan_sched.c**) consists of 10 functions that will be called from TRILBY-VM'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**, **suspended queue,** and a **terminated queue**). The structs for these lists are all defined in **inc/avan_sched.h**

2.2 TRILBY-VM Details

TRILBY-VM is not a simulator, it provides a shell that you can use to run normal non-interactive Linux commands like **ls**, **cal**, and **clear**. It can deal with arguments (like **ls -al** or **wc foo.txt**) but it cannot deal with any commands with an interactive interface – you cannot use it run **vim**, for example.



```
[kandrea@zeus]$ ./vm
 Green Imperial HEx: TRILBY VM v1.5a *TRIAL EXPIRED*
          [o]
 .________|__________.
 |                    |
 |     .________.     |
 |    /          \    |
 |   |____________|   |
 |   |____________|   |
 | \================/ |
 |                    |
 | [TRILBY-VM] $ _    |
 .____________________.
          [  ]
 .__________++_________.
  [Status] CS System Stopped: runtime 250000 usec, delaytime 1000000 usec
  [Status] Debug Mode: Off
(PID: 9707) [TRILBY-VM]$
```

For more information on TRILBY-VM, including on how to Compile it (make) , its built-in commands, how to run it with programs, and sample outputs, see **P1_Trilby_Manual.pdf**.

2.3 Process PID and Priorities

Avan Schedule works with Processes using a few different key 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 underlying operating system. This is how we identify and work with processes.

Each Process also has a **Priority Level**. As this is a Linux system, the smaller the Priority Level value, the higher the priority of that process. For this system, 1 is the highest priority and 255 is the lowest. **The default priority level for new processes is 128**.

The Priority Level is used to determine which processes should run first. The highest priority process (lowest value) would always be picked when we go to choose a new process to run. This, however, leads to a major problem in CS called **starvation**, where a higher priority process may prevent low priority processes from running for a very long time.

To fix this, TRILBY uses a solution called **aging**. To implement this, each process also tracks how many times it was skipped over during selection. What happens is that normally you pick the highest priority process to run and return that. When you do that, you set its skipped count to 0, because it was just picked to run, and then you go through every process in the Ready Queue and increment their skipped values by 1, because they were all just skipped.

When we go to pick the next process to run, we always go to pick the one with the highest priority, however, if we ever see a process that has been skipped too much (we define this in Section 4), then we'll pick that instead, so it gets a little bit of run time.

The result of this is that you will see high priority processes running a lot, and lower priority processes running a little bit every once in a while.

There is one more concept that TRILBY has for processes: Critical Run. If a Process is started with the Critical option, then it should always be picked to run immediately, even if there are higher priority processes or starving processes.

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

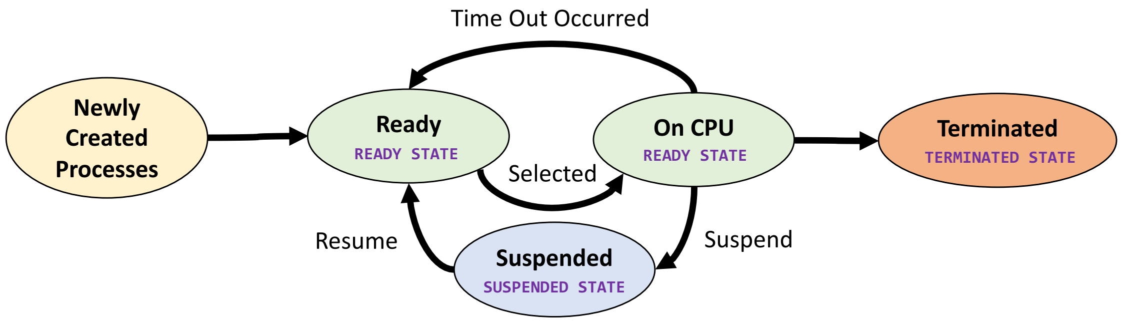
2.4 Process States

When you are working with Process Nodes (process_node_t structures), you will see that there is one field in the struct definition called **flags**. This is an int (32-bit signed 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 flags, 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 the Ready Queue, where it will be in the **Ready State**, indicating it's ready to run whenever it gets scheduled. When the process running on the CPU either finishes, or has been running too long, it'll be switched out and put in PID Order in the Ready Queue, so it'll run again later. Then the Scheduler will pick the next Process from the **Ready Queue** and put it on the CPU to run for its little piece of time.

If someone chooses to suspend a Process, it will be moved to a **Suspended Queue**, and it will be changed to the **Suspended State**. When it gets resumed, it'll be moved back to the **Ready Queue** and put back into the **Ready State**. When a Process finishes running, it'll be moved into the **Terminated Queue**, which keeps a list of all of the terminated processes. Anything in the Terminated Queue will be in a **Terminated** (known as a **Zombie**) **State**.

Every process can be in exactly one of three possible states in the VM 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 `avan_suspend()` function is called on it, then you will change it to the Suspended State and move to the Suspended Queue.

2.5 Process Node Flags

The `Process Node` (process_node_t) struct maintains their current state using the `flags` member. This 32-bit int contains pieces of information that have been combined together using bitwise operations. **You must use bitwise operators for these flags.** (| & ^ >> <<)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	exit_code																												C	Z	T	R
Bit Numbers	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

**The numbers at the bottom of the diagram represent which bit in the int it's referring to.*

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

State Bits: Bits 0-2 represent the current State of the Process. (1-bit values)

R = `READY_STATE` T = `SUSPENDED_STATE` Z = `TERMINATED_STATE`

Critical Bit: Bit 3 is a flag representing if the process was set with Critical Run. (-c)

C = `CRITICAL_FLAG`

Bits 4-31 are the lower 28 bits of the Exit Code when the process finished running on the CPU.

Example: Process in the Terminated Queue, Terminated State, run as Critical, with exit_code of 6

```
Bin:  0000 0000 0000 0000 0000 0000 0110  1100

Hex:     0    0    0    0    0    0    6     C

            So, flags = 0x0000006C
```

(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 (avan_sched.h)

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

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

```c
/* Avan Header Struct Definition */
typedef struct avan_header {
  queue_header_t *ready_queue;          // Linked List of Processes ready to Run on CPU
  queue_header_t *suspended_queue;      // Linked List of Processes suspended
  queue_header_t *terminated_queue;     // Linked List of Terminated Processes
} avan_header_t;
```

**Note that by convention in C, custom types usually have a _t suffix, as we have here.*

Avan Header contains pointers to three `queue_header_t` type structs, called `ready_queue`, `suspended_queue`, and `terminated_queue`.
Each of these three linked lists must be allocated and initialized as well.

Queue Header Struct

A `Queue Header` struct contains a pointer to a `Process 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.

```c
/* Queue Header Definition */
typedef struct queue_header {
  int count;     // How many items are in this linked list?
  process_node_t *head; // Points to the FIRST node of Linked List.  No Dummy Nodes.
} queue_header_t;
```

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[MAX_CMD];  // Name of the Process being run
int flags;          // Contains the current State of Process AND the Exit Code (set by OS)
int priority;       // The Priority Level of the Process (from 1 - 255)
int skips;          // The number of times this process was not selected
struct process_node *next; // Pointer to the next Process Node in a Linked List
} process_node_t;
```

Each process has a character array (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 (unsigned int) called pid.

You will also have a 32-bit int flags, which contains several pieces of data that are combined together using bitwise operations, as specified in Section 2.5.

3 Building and Running the TRILBY-VM (./vm)

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:handout$ 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 **avan_sched.c**, which is the only file you will be modifying and submitting. You will also have very useful header files (**avan_sched.h**) along with other files for the simulator itself in the **inc/** directory.

3.1 Building TRILBY-VM

To build TRILBY-VM, run the make command:

```
kandrea@zeus-1:handout$ make
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -c -o obj/vm.o src/vm.c
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -c -o obj/vm_cs.o src/vm_cs.c
gcc -std=gnu99  -Og -Wall -Werror -Wno-error=unused-variable -Wno-error=unused-function -
pthread -I./inc -L./obj -g -c -o obj/vm_shell.o src/vm_shell.c
...
```

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

3.2 Running TRILBY-VM

To start TRILBY-VM, run **./vm**

Instructions for Running the TRILBY-VM are in the P1_Trilby_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.

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

4 Implementation Details

You will complete all of the functions in **avan_sched.c**. You may create any additional functions that you like, but you cannot modify any other files; you only submit **avan_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. If you are receiving a pointer, **always make sure that pointer is not NULL**, unless you are expecting it to be NULL. Failing to check the validity of inputs can, and often will, result in SEGFAULTS.

If there is an error on any step (eg. invalid input or malloc returning NULL), then you should return an error.

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

avan_header_t *avan_create();

Creates a new Avan Scheduler Header with initial values.

- Create a Avan Header (avan_header_t) struct and initialize it.
  - All allocations within this struct will be dynamic, using **malloc** or **calloc**.
- Take note that Header contains pointers to Queue (queue_header_t) structs, which themselves also need to be allocated and initialized!
  - Each of the three Queue structs contain a pointer to the head of a singly linked list, which must be initialized to NULL.
  - Each Queue struct also contains a count of the number of items in its Linked List, which must be initialized to 0.

On any errors, return NULL, otherwise return a pointer to your new Avan Header.

int avan_insert(avan_header_t *header, process_node_t *process);

Adds a Process Node into the Ready Queue.

- Set the flags member of the Process (process_node_t) to the Ready State.
    - Set the Ready State bit to a 1 and the other two State bits to 0.
    - Only one of the three State bits should be set (1) at any given time.
- Add the Process Node (process_node_t) struct to Ready Queue in Ascending PID Order.
    - The head should point to the Process struct with the lowest PID.
    - If the head pointer is NULL, then this will be your first Process, and the Ready Queue's head pointer will point to this process.

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 avan_suspend(avan_header_t *header, pid_t pid);

Moves a Process from the Ready Queue, by pid, to the Suspended Queue.

- Find the Process with matching pid in your Ready 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 struct you found in the linked list.
- Set the flags member of the process to the Suspended State.
    - Set the Suspended State bit to a 1 and the other two State bits to 0.
    - Only one of the three State bits should be set (1) at any given time.
- Insert that Process into the Suspended Queue in ascending PID order in the Linked List.

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

int avan_resume(avan_header_t *header, pid_t pid);

Moves a Process from the Suspended Queue back to the Ready Queue

- Find the Process with matching pid in your 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 struct you found in the linked list.
- Set the flags member of the process to the Ready State.
    - Set the Ready State bit to a 1 and the other two State bits to 0.
    - Only one of the three State bits should be set (1) at any given time.
- Insert that Process into the Ready Queue in Ascending PID Order
    - (eg. resume PID 1, you might have ready_queue->head->1->2->3->NULL)

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

int avan_quit(avan_header_t *header, process_node_t *node, int exit_code);

Inserts the Process into the Terminated Queue and returns its Exit Code.

- Set the flags member of the process to the Terminated State.
  - Set the Terminated State bit to a 1 and the other two State bits to 0.
  - Only one of the three State bits should be set (1) at any given time.
- Set the flags bits used for exit_code to the value of the exit_code passed in.
  - You will set the lower 28 bits of the passed in exit_code as the upper 28 bits of your flags member of the Process Node.
- Insert that Process into the Terminated Queue in ascending PID order.

Return the 0 on success or -1 on any error. (**This is different from avan_terminate's return)**

int avan_terminate(avan_header_t *header, pid_t pid, int exit_code);

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

- Find the Process with matching pid in your Ready or Suspended Queues.
- 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 struct you found in the linked list.
- Set the flags member of the process to the Terminated State.
  - Set the Terminated State bit to a 1 and the other two State bits to 0.
  - Only one of the three State bits should be set (1) at any given time.
- Set the flags bits used for exit_code to the value of the exit_code passed in.
  - You will set the lower 28 bits of the passed in exit_code as the upper 28 bits of your flags member of the Process Node.
- Insert that Process in ascending PID order in the Terminated Queue.

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

process_node_t *avan_new_process(char *cmd, pid_t pid, int priority, int critical);

Create a new Process struct and initialize its members.

- Create a new Process Node (process_node_t) and initialize it with these values.
  - Initialize the state bits of flags to be in the Ready State.
  - Initialize the priority member to the value passed in for priority.
  - Initialize the Critical bit of flags to be 0 if critical is false (0) or 1 if true.
  - Initialize the upper 28-bits of flags to be all 0s.
  - Initialize the skips member to 0.
  - String Copy (**strncpy** for safety!) cmd into your struct.
- Return a pointer to your new process.

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

process_node_t *avan_select(avan_header_t *header);

Choose the next process to run using the given algorithm, then return a pointer to it.

- Algorithm to find the Best Node:
    - Iterate through the nodes in the Ready Queue
    - If any process has the Critical flag, the first Critical one found is the Best.
        - You can stop iterating immediately after finding the first Critical process.
    - If no processes have the Critical flag, but at least one process is Starving (skips >= MAX_SKIPS), then the Starving process with the lowest PID is Best.
    - If none are Starving, then the process with the lowest Priority Number is best.
        - If you have a tie, then the one with the lowest PID is Best.
- Once you find the Best, remove that process from the Ready Queue.
- Then set the Best process' number of skips to 0 (it was just picked)
- Then set the Best process' next to NULL.
- Finally, increment skips member for every remaining process in the Ready Queue.

Return a pointer to the Best process on success, or NULL on any errors or on an empty list.

int avan_get_size(queue_header_t *ll);

Returns the size of the given Queue

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

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

void avan_cleanup(avan_header_t *header);

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

- Free all Nodes from each Queue in the Avan Header.
- Free all Queues from the Avan Header
- Free the Avan Header

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 C Review section on Blackboard on Bitwise Operations. This project involves some basic use of bitwise operations to set and clear flags and to extract data from flags. You will use bitwise operations more extensively for the next project.

In our model, your scheduler uses an algorithm called **Priority Scheduling** to select the process from the Ready Queue. Each process will then run for a small period of time and then get returned to the Ready Queue if not finished, or to the Terminated Queue if it finished in that time period. ***(You'll study this problem later in CS471)***

5.1 Memory Checking

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

```
kandrea@zeus-2:handout$ valgrind ./vm
```

You are looking for a line that says:

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

If you find any leaks in Valgrind, 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: avan_insert (avan_sched.c:89)
==539428==    by 0x401075: cs_thread (vm_cs.c:104)
```

This shows there was an error caused by my avan_insert code on line 89. We won't be grading on any of these Errors (like Invalid Read), only the Heap Blocks were Freed message.

One note on Valgrind with TRILBY-VM is that is the vm is killed when a process is active, you will see memory leaks from the process itself that's just been killed. This is fine. We'll only check for memory leaks when the vm properly exits with no processes running.

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

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 your code without running the TRILBY-VM, we do have a special **src/test_avan_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 Blackboard. **Make sure to put your G# and name as a commented line in the beginning of your program. Note that the only file to submit is avan_sched.c**

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

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

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

Your grade will be determined as follows:

- **20 points - Code & Comments.** Be sure to document your design clearly in your code comments. This score will be based on reading your source code.
  - **Rubric Breakdown for the 20 points: (Posted on Blackboard 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.
    - **Correctness:** You need to check pointers before using them, initialize all 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 flags member (eg. using &, |, <<, >>, ~, ^) as needed to work with the bits. Hardcoding flags with large if/else branches is not going to work for this project. You should be setting/clearing individual bits of the flags without affecting the others. (Hint: We will be reviewing this in the second Recitation and there is a Slide set on Bitwise operations on Blackboard)
- **80 points - Automated Testing (Unit Testing).** We will be building your code using your submitted avan_sched.c and our Makefile and running unit tests on your functions.
  - It must compile cleanly to earn any points.
  - Each function will be tested independently for correctness by script.
  - Partial credit is possible.
  - **Border cases and error cases will be checked!**
  - 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

- v1.2 - Release Version
- v1.2a - Fix for avan_quit's return type.
  - Updated avan_quit's description and details in Section 4 to have it return 0 on success.
    - This matches the comments in avan_sched.h
  - Removed references to slow_hat and replaced them with slow_bug
    - The slow_bug program was the replacement for slow_hat
- V1.2b – Fix for avan_new_process to give the initializer for skips.
  - In avan_new_process, skips should be initialized to 0.
- V1.2c – Made the avan_terminate summary to match the implementation details in Section 4.
  - In avan_terminate's detailed instructions in Section 4, it says to remove the Process from either Ready or Suspended Queues.
  - Fixed the one-line quick summary to be more clear about this on page 2.