CS 367 Project 1 - Spring 2023:
On-Program (OP) CPU Scheduler
Due: Friday, February 10th 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 **On-Program (OP) CPU Scheduler**. Finish the functions in **src/op_sched.c** to implement a CPU scheduler for our 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 (a program being run) and run it for a very, very short amount of time on the CPU. Then, you can put it back in a **ready queue** (linked list) and 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 completing the **OP CPU Scheduler.**

Project 1 is to finish functions for the OP Scheduler, to choose which process to run next.

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/op_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 state.

Your code (**op_sched.c**) works with pre-written files to implement several of these operations. You will be maintaining three singly linked lists (**Ready Queue – High Priority, Ready Queue – Low Priority, and Defunct Queue**). These structs are defined in **inc/op_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 op_sched.c to do operations.

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

```
Op_schedule_s *op_create();
```

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

```
Op_process_s *op_new_process(char *command, pid_t pid, int is_low, int is_critical);
```

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

```
int op_add(Op_schedule_s *schedule, Op_process_s *process);
```

- Inserts the Process node at the end of **the appropriate Ready Queue** Linked List.

```
int op_get_count(Op_queue_s *queue);
```

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

```
Op_process_s *op_select_high(Op_schedule_s *schedule);
```

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

```
Op_process_s *op_select_low(Op_schedule_s *schedule);
```

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

```
int op_promote_processes(Op_schedule_s *schedule);
```

- Ages processes in the **Ready Queue - Low**, then Moves Starving Processes to the **Ready Queue - High**.

```
int op_exited(Op_schedule_s *schedule, Op_process_s *process, int exit_code);
```

- Inserts the Process node at the end of the **Defunct Queue**

```
int op_terminated(Op_schedule_s *schedule, pid_t pid, int exit_code);
```

- Remove a Process from **Ready Queue - High** or **Ready Queue - Low** and Add to End of **Defunct Queue**

```
void op_deallocate(Op_schedule_s *schedule);
```

- Free the OP Schedule, the Three Linked List Queues, and all of their Process Nodes.

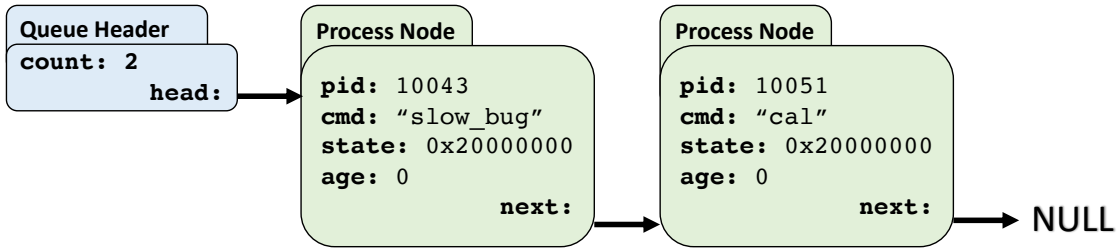
TRILBY-VM should run with no Memory Leaks

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, 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 - Low**, based on its priority level. 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. This is the default priority level for all processes in our system. A second queue is called the **Ready Queue – Low** priority, which is where low-priority processes go to wait for their turn. A third queue, the **Defunct Queue**, is where all processes that have exited normally (or have been terminated by the OS) are stored.

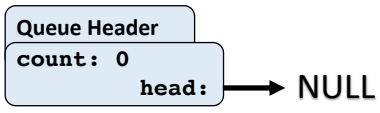
A sample of one of these linked list Queues is shown below.



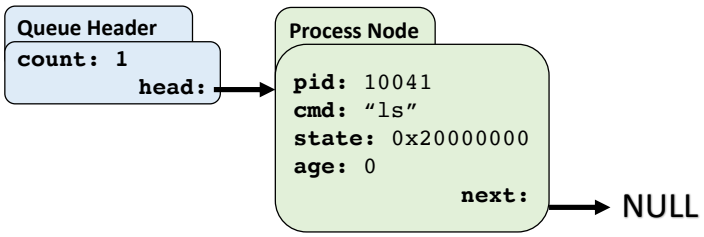
Each Linked List starts from a Queue Header (Op_queue_s *) node. These nodes are used to hold the head pointer for each linked list. This means that if the head of the linked list is inside of this queue header, which is passed into many of your functions. This way you can change the head pointer without 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 **add them to the end of the Linked List**.

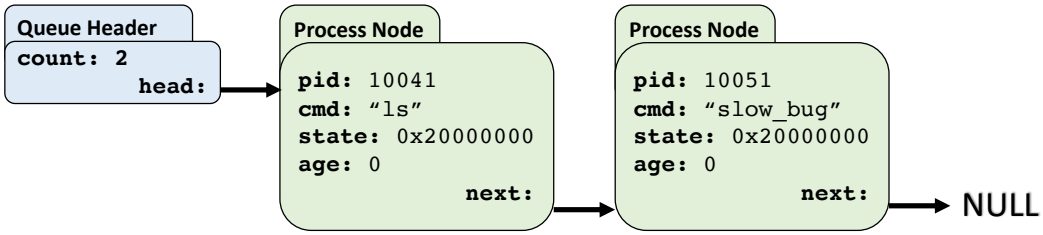
For example, here is a set of diagrams of starting with an empty **Ready Queue – High** Priority.



Next, we have a process "ls" with Process ID (PID) 10041, which is added as the only node.



Then we start process "slow_bug" 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 the TRILBY-VM, but your code will be called to create the process nodes and manage them in the Linked Lists.

TRILBY-VM 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. TRILBY-VM will always try and choose a process from this Linked List first to run. If this queue is ever empty, then TRILBY-VM will try to select a process from the **Ready Queue – Low** Priority linked list. So, these processes in the low priority list will run much less often.

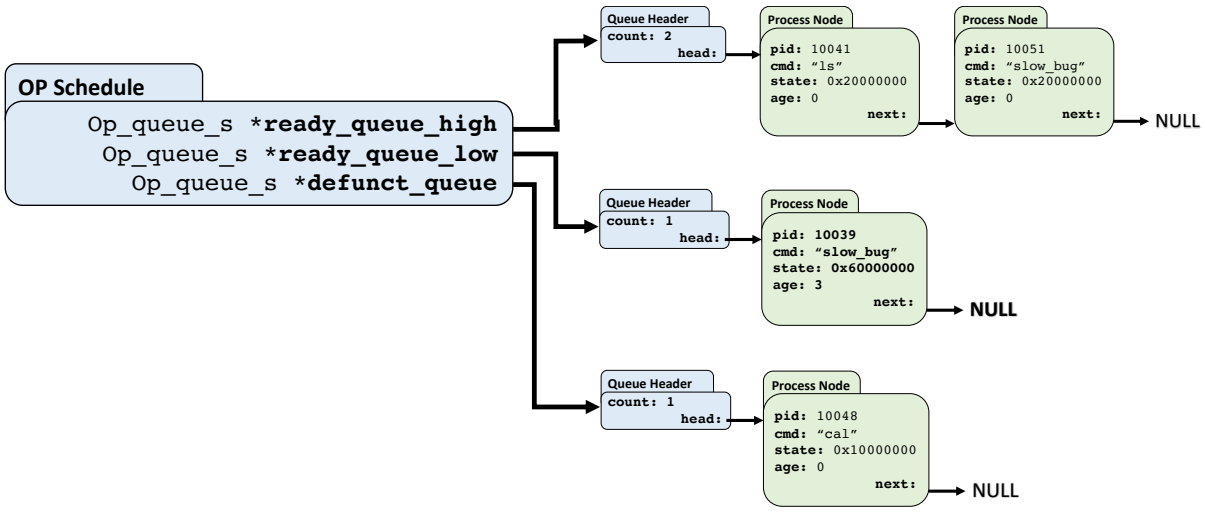
When TRILBY-VM needs a next process to execute on the CPU, it will call your **op_select** function, which removes the best node (details in Section 4) and returns a pointer to it.

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 **op_add** function to put it back in the end of the proper Ready Queue so it can resume running later, or, if it already finished, TRILBY will call your **op_exited** function to put it into the **Defunct Queue**, which tracks all finished or 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 Defunct Queue.

To help you manage all of these three Linked Lists (Queues), you have another struct, **Op_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 OP 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

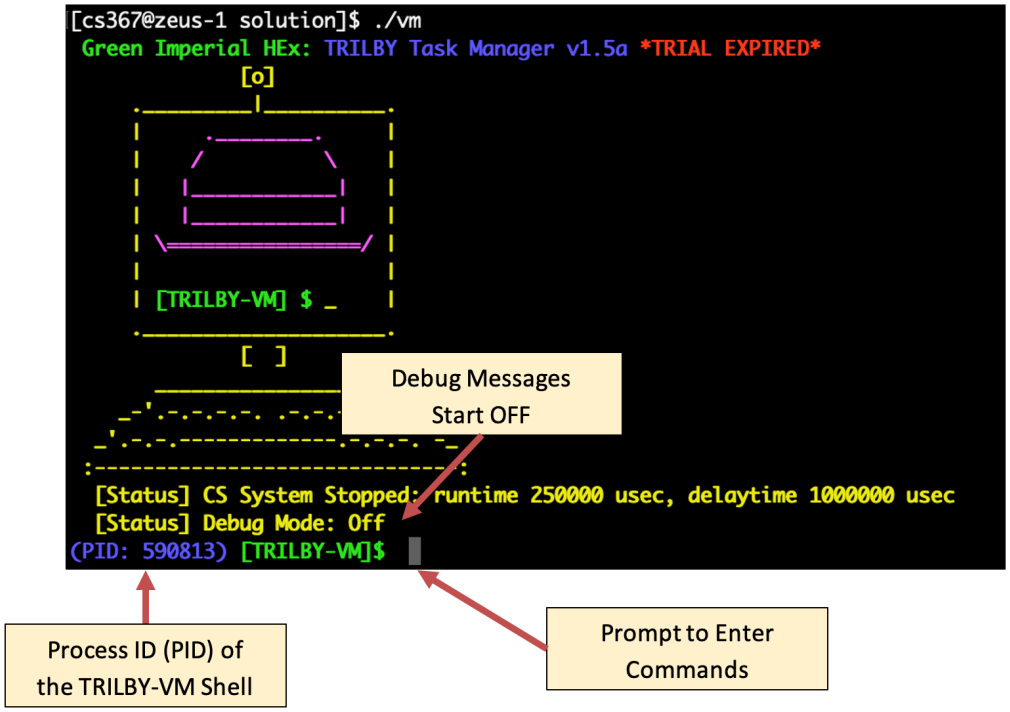
2.1 Source Code Files

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). 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 **op_sched.c**

Your code (**op_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 – High** priority, **Ready Queue – Low** priority, and a **Defunct Queue**). The structs for these lists are all defined in **inc/op_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** 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 **vim**, for example.



For more information on TRILBY-VM, including on how to Compile it with **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

OP Schedule 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 underlying 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. In reality, it's just a typedef name for an **int**. So, whenever you see pid_t, you can treat it the same as an **int**.

Each Process also has a **Priority** of either **High** or **Low**. All processes have a **default priority of High**, so we don't have anything special to designate them. For Low priority processes, we will need to set a special bit in the **state** of your node, which is just an **unsigned int**. This bit (bit 30) will be a 1 if the process is Low priority or 0 if it is High priority. High Priority processes get added (in **op_add**) to the

Ready Queue – High priority every time. Low Priority processes get added (in **op_add**) to the **Ready Queue – Low** priority linked list instead.

The Priority Level is used to determine which processes should run first. If there is any process node in the **Ready Queue – High** linked list, they will always be selected before any process in the **Ready Queue – Low** linked list. This, however, leads to a major problem in CS called **starvation**, where a higher priority process may prevent low priority processes from running.

To fix this, TRILBY uses a solution called **aging**. To implement this, we also track how long a process has been waiting in the **Ready Queue – Low** linked list. TRILBY-VM will call your **op_promote_processes** function after a process is selected. In this function, you will increment the age of all processes in the **Ready Queue – Low** linked list, and then if any of them are above or equal to **MAX_AGE** (a provided constant equal to 5), you will move them into the **Ready Queue – High** linked list. This temporarily gives them higher priority for one run on the CPU to let them get picked faster. Once they run, all processes go back to their normal priority level ready queues.

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.

Normally you will always pick the process at the front of a queue to run next, however, there is one more concept that TRILBY has for processes: **Critical** priority level. If a Process is started with the **Critical** option, then it should always be picked to run immediately, even if there are other processes in front of it in the same queue. Only high priority processes can be marked **Critical**.

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

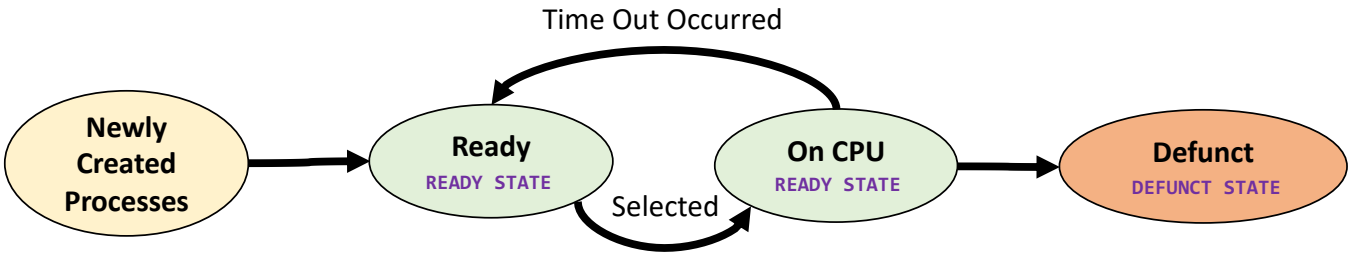
2.4 Process States

When you are working with Process Nodes (Op_process_s structures), you will see that there is one field in the struct definition called **state**. This is an **unsigned int** (32-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**, 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 back at the end of the appropriate Ready Queue, so it can run again later. Then the Scheduler will pick the next Process from a **Ready Queue** and put it on the CPU to run for its little piece of time.

When a Process finishes running, it'll be moved into the **Defunct Queue**, which keeps a list of all the terminated processes. Anything in the Defunct Queue will be in a **Defunct** (known as a **Zombie**) **State**.

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



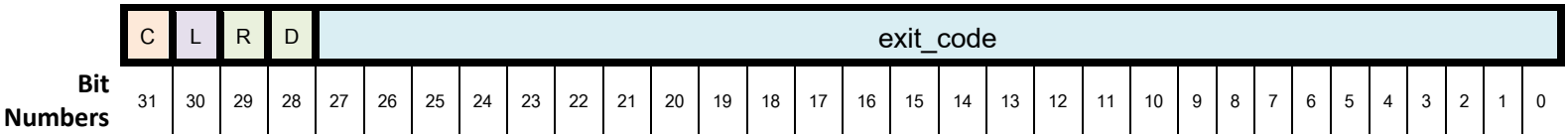
This shows the states that each Process can be in and when they change. As an example, if a Process is in one of the Ready Queues in the **Ready State** and your `op_terminated()` function is called on it, then you will change it to the **Defunct State** and move to the **Defunct Queue**.

2.5 Process Node State

The `Process Node` (Op_process_s) struct maintains their current state using the `state` member. This 32-bit unsigned int contains information that have been combined together using bitwise operations.

You must use bitwise operators to set this state.

`|` `&` `^` `>>` `<<`



**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 28-29 represent the current State of the Process. (1-bit values)

R = `READY STATE` D = `DEFUNCT STATE`

Low-Priority Bit: Bit 30 is a flag representing if the process was set with Low Priority. (-l)

L = `LOW-PRIORITY FLAG`

Critical Bit: Bit 31 is a flag representing if the process was set with Critical Run. (-c)

C = `CRITICAL FLAG`

Note: Critical Processes will never be Low-Priority.

Bits 0-27 are the lower 28 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 2^{28} -1)

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

Bin: 1001 0000 0000 0000 0000 0000 0000 0110

Hex: 9 0 0 0 0 0 0 6

So, state = 0x90000006

(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 (op_sched.h)

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

The overall struct of type **Op_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.

```
/* OP Schedule Struct Definition */
typedef struct op_schedule {
  Op_queue_s *ready_queue_high;  // Linked List of Processes ready to Run on CPU (High Priority)
  Op_queue_s *ready_queue_low;   // Linked List of Processes ready to Run on CPU (Low Priority)
  Op_queue_s *defunct_queue;     // Linked List of Defunct Processes
} Op_schedule_s;
```

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

OP Schedule contains pointers to three **Op_queue_s** type structs, called **ready_queue_high**, **ready_queue_low**, and **defunct_queue**. Each of these three linked lists must be allocated and initialized as well. (Remember to initialize ALL values!)

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.

```
/* Queue Header Definition */
typedef struct queue_header {
  int count;              // How many items are in this linked list?
  Op_process_s *head;     // Points to the FIRST node of Linked List.  No Dummy Nodes.
} Op_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 int state; // Contains the current State of Process, Priority Flag, the Exit Code
int age;            // How long this has been in the Ready Queue – Low Priority since last run
struct process_node *next; // Pointer to the next Process Node in a Linked List
} Op_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`**.

You will also have a 32-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 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 **op_sched.c**, which is the only file you will be modifying and submitting. You will also have very useful header files (**op_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 **op_sched.c**. You may create any additional functions that you like, but you cannot modify any other files; you only submit **op_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, as specified in the following Function API References.

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

This section specifies exactly what each of your 10 functions needs to do.

Op_schedule_s *op_create();

Creates a new OP Scheduler Header with initial values.

- Create an OP Schedule (Op_schedule_s) struct and initialize it.
  - All allocations within this struct must be dynamic, using **malloc** or **calloc**.
- The Header contains pointers to Queue (Op_queue_s) 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 OP Schedule.

Op_process_s *op_new_process(char *command, pid_t pid, int is_low, int is_critical);

Create a new Process struct and initialize its members.

- Create a new Process Node (Op_process_s) and initialize it with these values.
  - Set the state member to the Ready State.
    - Set the Ready State bit to a 1 and Defunct State to a 0.
    - Only one of the two State bits should be set (1) at any given time.
  - Set the Low-Priority bit of state to be 0 if is_low is false (0) or 1 if true.
  - Set the Critical bit of state to be 0 if is_critical is false (0) or 1 if true.
  - Initialize the lower 28-bits of state to be all 0s.
  - Initialize the age member 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 your new process.

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

int op_add(Op_schedule_s *schedule, Op_process_s *process);

Adds a Process Node into the Appropriate Ready Queue based on its Low Priority Flag.

- Set the state member of the Process (Op_process_s) to the Ready State.
  - Set the Ready State bit to a 1 and Defunct State to a 0.
    - Only one of the two State bits should be set (1) at any given time.
  - Make sure to set this without changing the Critical or Low Bits.
- Add the Process Node (Op_process_s) struct to the end of the appropriate Ready Queue
  - If the Low Priority Bit in the state is a 0, insert to end of the Ready Queue – High
  - If the Low Priority Bit in the state is a 1, insert to end of the Ready Queue – Low
- 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.

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 op_get_count(Op_queue_s *queue);

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.

Op_process_s *op_select_high(Op_schedule_s *schedule);

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

- Algorithm to find the best process to choose:
  - Choose the front process from the Ready Queue – High Priority List.
  - Then, iterate the Ready Queue – High Priority List
    - If any process has Critical flag, choose the first Critical process instead.
    - You can stop iterating immediately after finding the first Critical process.
  - If there are no processes in the Ready Queue – High Priority List, return NULL.
- Once you have Chosen a process, remove that process from the Linked List it is in.
  - Remember to update the pointers!
- Then set the chosen process' age to 0 (it was just picked)
- Then set the chosen process' next to NULL.
- Finally, return a pointer to that **same** process you just removed from the linked list.

Return a pointer to chosen process on success, or NULL on any errors or the Queue was empty.

Op_process_s *op_select_low(Op_schedule_s *schedule);

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

- Algorithm to find the best process to choose:
  - Choose the front process from the Ready Queue – Low Priority List.
  - If there are no processes in the Ready Queue – Low List, just return NULL.
  - *(Note: You will not have any processes that are both Critical and Low)*
- Once you have Chosen a process, remove that process from the Linked List it is in.
  - Remember to update the pointers!
- Then set the chosen process' age to 0 (it was just picked)
- Then set the chosen process' next to NULL.
- Finally, return a pointer to that **same** process you just removed from the linked list.

Return a pointer to the chosen process on success, or NULL on any errors or if List was empty.

int op_promote_processes(Op_schedule_s *schedule);

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

- Increment the age field for every process in the Ready Queue – Low linked list.
- After incrementing the age field, if any process has age >= MAX_AGE
  - Remove it from this linked list properly.
  - Set this process' age to 0 (it was just promoted)
  - Set the chosen process' next to NULL.
  - Add it to the end of the Ready Queue – High linked list.

Note: MAX_AGE must be used as a defined constant. It's value is set to 5.

Return 0 on success or -1 on any errors.

int op_exited(Op_schedule_s *schedule, Op_process_s *process, int exit_code);

Inserts the given Process into the Defunct Queue and return 0 on success.

- Set the state member of the Process (Op_process_s) to the Defunct State.
  - Set the Defunct State bit to a 1 and Ready State to a 0.
    - Only one of the two State bits should be set (1) at any given time.
  - Make sure to set this without changing the Critical or Low Bits.
- Set the state 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 lower 28 bits of your state member of the Process Node with bitwise operators.
- Insert that Process to the end of the Defunct Queue.

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

int op_terminated(Op_schedule_s *schedule, pid_t pid, int exit_code);

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

- Find the Process with matching pid **in either Ready Queue - High** or **Ready Queue – Low**
  - 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 state member of the Process (Op_process_s) to the Defunct State.
    - Set the Defunct State bit to a 1 and Ready State to a 0.
      - Only one of the two State bits should be set (1) at any given time.
    - Make sure to set this without changing the Critical or Low Bits.
  - Set the state 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 lower 28 bits of your state member of the Process Node.
  - Insert that Process to the end of the Defunct Queue.
- If the Process with pid is not in either of the Queues, return -1 instead.

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

void op_deallocate(Op_schedule_s *schedule);

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

- Free all Nodes from each Queue in the OP Schedule.
  - Remember to free the cmd String!
- Free all Queues from the OP Schedule
- Free the OP 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 Blackboard 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 Defunct 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 **valgrind** *(GDB and Valgrind refresher Videos are on Blackboard)*

```
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: op_add (op_sched.c:89)
==539428==    by 0x401075: cs_thread (vm_cs.c:104)
```

This shows there was an error caused by my **op_add** 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 op_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/involving the TRILBY-VM**, we do have a special **src/test_op_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 op_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 use late tokens automatically); and submitting the correct version 48 hours after the due date 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 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. You can only modify op_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 op_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 - Release Version
- v1.01 – Updated the member name used on page 12 from cmd to command.
  - This now correctly states you need to strncpy from the command argument to the cmd member in your struct. It previously had mentioned cmd on the wrong side.
- v1.02 – Updated op_new_process to initialize the **pid** member from the **pid** argument.