CS531 Project #1: CPU Process Scheduler
Due: Wednesday, Oct 2th at 11:59PM

This is to be an individual effort. No partners.

Before you Start: This project requires a good understanding of basic C programming. In particular, review the text and lectures for Memory and Pointers, Dynamic Memory (malloc/free), Structs and Struct Pointers, and **Linked Lists**. In this project, you will be working with both single and double pointers to structs. The resources on Blackboard, specifically **Linked List Basics** will be helpful.

Overview
For this assignment, you are going to use C to implement a **Process Scheduler** for an Operating System, by implementing a Linked List. In an Operating System (**OS**), each program that is executed is called a **Process** while it is running. Each process has a unique value called a Process ID (PID), which is how the OS tracks and manages each process.

```
kandrea@zeus-1:handout$ ps -e --format="pid comm start cputime" --sort pid
  PID COMMAND          STARTED             TIME
    1 systemd           May 20       00:20:47
    2 kthreadd          May 20       00:00:12
    3 ksoftirqd/0       May 20       00:00:03
    5 kworker/0:0H      May 20       00:00:00
```

In addition to a PID, each process also has a name (shown in the COMMAND column above), a time started, and a time spent on CPU in execution.

On a real OS, there may be hundreds of processes running at once, even though there may only just 1 CPU core. The OS handles this by using a Process **Scheduler**. A Scheduler keeps all of the processes that are waiting to run in a list. When a process finishes executing, or when it has been on the CPU for a certain amount of time, then the OS will perform a **context switch** to take the current process off of the CPU and place it back into the scheduler's list. Any new processes starting would also go into the scheduler to be added to the list as well. The scheduler will then select the next process for the CPU to execute and send that to the processor to run.

Your first project will be to write the functions to implement the **scheduler** for a simulated OS and CPU. For your project, each process will be represented as a **struct** and your schedule list will be a **singly linked list** implemented in C.

Your functions will be called by the OS to **create processes** and fill them in with provided initial values, **insert processes into the list in order**, **select a process to run** and **remove it from the list**, **free processes**, and **provide a count of all of the processes** in the scheduler list.

The algorithms to create a linked list were discussed in class and will be similar, although simpler, than what is discussed in Mastering Algorithms in C, chapter 5.

The struct definitions are provided for you in the file **structs.h**. You will only be working with the one struct of type **Process** for this project. Each process struct has the following member fields:

- char name[255]; // This is your process name. Different processes can have the same name.
- int pid; // The Process ID (unique value) for your process
- int time_remaining; // How much CPU time is needed to finish running this process.
- int time_last_run; // The time value that this process was last run during.

The OS simulator will perform the following operations:

1. Take the currently running process off the CPU and do one of the following:
    - If the process is done, it will call your **schedule_terminate(Process *node)** function.
    - Otherwise, it will call your **schedule_insert(Process **list, Process *node)** function.
2. See if there are any new processes that are starting up.
    - If there is, it will call your **schedule_generate(const char *name, int pid, int time_remaining, int time_last_run)** function to create a new process.
    - It will then call your **schedule_insert(Process **list, Process *node)** function.
3. Get the next process from your list to schedule for running.
    - It will call your **schedule_select(Process **list)** function to get the next process to run.
4. It will then run that selected process for one time unit.
5. Finally, it will go back to step 1 and continue until time ends.

The CPU scheduling algorithm you will be implementing is called **Shortest Remaining Time First**. This means every time you need to select a new process, you will choose the one with the shortest **time_remaining** value. This algorithm has one big problem, which is that if there are enough short processes waiting, any processes with a long remaining time will not be able to run, because there is always a shorter one ready. When this happens, we say that the long process is in a state of **starvation.**

So, the order for selecting a process is as follows:

1. Find the process with the shortest **time_remaining** value.
2. If any process has not run in >= **TIME_STARVATION** from the current time, select that instead.
    - It doesn't matter how long it has been in starvation, only if it is in starvation.

On any ties, always choose the one with the lowest PID value.

Example: Time for this is 8 and the STARVATION_TIME setting is 5. Here, the shortest time remaining is PID: 136, but PID: 422 is starving (8 - 3 >= 5), so it got selected instead.

```
.============
| Starting Time:  8
+------------
| ....
+------------
| In Scheduler:  6 Processes
|       PID:  60 Time Remaining:  6 (Last Run Time:  4) Name: touch
|       PID: 136 Time Remaining:  2 (Last Run Time:  8) Name: mkdir
|       PID: 369 Time Remaining:  9 (Last Run Time:  6) Name: uniq
|       PID: 422 Time Remaining:  7 (Last Run Time:  3) Name: emacs
|       PID: 427 Time Remaining:  6 (Last Run Time:  5) Name: grep
|       PID: 531 Time Remaining:  3 (Last Run Time:  7) Name: emacs
+------------
| Selecting to Run on the CPU
|       PID: 422 Time Remaining:  7 (Last Run Time:  3) Name: emacs
\============
```

Functions to Implement

You will be editing only one file, **schedule.c**, which is the **only file you will be turning in** on blackboard. You have five functions to write in order to complete this project. These functions will be called from the simulated OS to perform the following:

Count the Items in the List

```
int schedule_count(Process *list)
```

- This function should count all of the processes in the linked list called list. The head node of this list is passed in as the one parameter for this function.
- Return the number of processes in the list.

Insert an Existing Node into the List

```
void schedule_insert(Process **list, Process *node)
```

- This function should insert the Process node into your linked list. The head node of the list is a passed in as a Process ** type double-pointer. Be very careful when working with this. The reason this is a double-pointer is because if you need to insert the node as the new head of the list, then you will need to change the value of the list pointer.
- Insert the node into the given list using the following constraints:
    - Insert the given node in order of ascending pid value.
        - Ascending order, such as 1, 2, 3, 30, 42, etc.
- Make sure to handle the case where node should be inserted as the new list head.
- If node is NULL, you should return immediately.

Free an Existing Node not in the List

```
void schedule_terminate(Process *node)
```

- This function should free all memory associated with the given node.

Create a New Node from the Given Parameters

```
Process *schedule_generate(const char *name, int pid, int time_remaining,
                           int time_last_run)
```

- This function should create a new node of type Process * using dynamic memory allocation.
- Initialize all members of the new Process struct to the values provided.
    - Always initialize **next** to NULL
- Since name is a string, you will need to use a string function to copy name into the new struct.
    - Use **strncpy** to perform this copy.
- On any memory errors (malloc) return NULL

Select a Node and Remove it from the List

`Process *schedule_select(Process **list)`

- This function will perform three tasks.
    - First, it will select the next process to run according to a set of rules.
    - Second, it will remove that process from the list and adjust the list if needed.
    - Finally, it will return the selected process.
- **Select the process from the list using the following two rules:**
    - First, select the process with the lowest **time_remaining** value.
        - If there is a tie, choose the process with the lowest **pid** value.
    - Second, if there is any process that has not run in >= **TIME_STARVATION** time, select that process instead of the one with the lowest time_remaining value.
        - You can compute how long it has been since it last run by using the provided clock function to get the current time, **clock_get_time();**
        - If the different from the current time and the time_last_run is >= TIME_STARVATION, then select it instead.
        - If there are multiple processes that are in Starvation, select theone with the lowest pid value.
- Remove the selected process from your list.
    - Make sure to update the list if the one you removed was the beginning of the list.
- Return the selected process.
- If list is empty, return NULL

Starting the Assignment

The starting tar file (project1_handout.tar) has 10 files in it, but you will only be modifying **schedule.c**

- **schedule.c : This is the file you will edit.** It already contains some function signatures for the functions you need to implement. By default they just return immediately; you will need to write all of these functions. You may write other functions as well. Put all of the code for your assignment in this single file.

- **structs.h** : This contains the definition for the **Process** struct that you will be using.

- **Makefile** : This is a makefile that will make and clean your program. When you make it, you will get an executable called **scheduler**

Implementation Notes

- **You will be working with Linked Lists and a combination of single and double-pointers.** Make sure to take time to think about how you want to work with the linked list for each function that you need to implement. Make sure you handle all of the cases that could occur.

Example Run

When you run your program, you can optionally type in a value after the name to randomly generate different data to test your code with. Here is the first few time values of the default run. Your code should match these.

```
kandrea@zeus-1:handout$ ./scheduler 1
.=============
| Starting Time:  1
+------------
| Process Starting
|     PID: 886 Time Remaining:  5 (Last Run Time:  1) Name: cp
+------------
| In Scheduler:  1 Processes
|     PID: 886 Time Remaining:  5 (Last Run Time:  1) Name: cp
+------------
| Selecting to Run on the CPU
|     PID: 886 Time Remaining:  5 (Last Run Time:  1) Name: cp
\=============

.=============
| Starting Time:  2
+------------
| Unloading Process
|     PID: 886 Time Remaining:  4 (Last Run Time:  1) Name: cp
| Process Starting
|     PID: 335 Time Remaining:  2 (Last Run Time:  2) Name: nano
+------------
| In Scheduler:  2 Processes
|     PID: 335 Time Remaining:  2 (Last Run Time:  2) Name: nano
|     PID: 886 Time Remaining:  4 (Last Run Time:  1) Name: cp
+------------
| Selecting to Run on the CPU
|     PID: 335 Time Remaining:  2 (Last Run Time:  2) Name: nano
\=============
```

Notice that when time is 2, you have two processes in the scheduler (PIDs 335 and 886). Since PID 335 has a shorter time_remaining value, it is removed from the list and returned to run on the CPU.

Submitting and Grading

Submit your **schedule.c** on **blackboard** as **schedule.c**. No other naming formats are needed for this. Be sure this file contains everything you need -- **incomplete submissions cannot be graded**.

<u>Make sure to put your name and G# as a commented line in the beginning of your schedule.c file. Also, in the beginning of your program list the known problems with your implementation in a commented section.</u>

All submissions will be **compiled**, **tested, and graded on Zeus!** Make sure you test your code on Zeus before submitting. If your program does not compile on zeus, we cannot grade it. If your program compiles but does not run or generate output on zeus, we cannot grade it.

Questions about the specification should be directed to the CS 531 Piazza forum. However, recall that debugging your program is essentially your responsibility; so please do not post long code segments to Piazza. **Do not post any code on Piazza in a public post.** Always post as a **private** post for code questions. Any general questions about the assignment can be posted publicly.

You **have one late token** that may be used on any project in the course. No late submissions are allowed without using a token, and you only have one token for all three projects. This is meant for personal emergencies, so plan on submitting on time and saving your token for any emergencies throughout the semester.

Your grade will be determined as follows:

- **80 points** - Correctness. This is graded by automated script by checking the results of each of your functions as you complete them. You will get partial credit for most of the functions, so even if something is not working at the end, you may still get credit for any working code.

- **20 points** - Code & comments: Be sure to document your design clearly in your code comments. This score will be based on (subjective) reading of your source code by the GTA. The grader will also evaluate your C programming style according to the below guidelines.

Test your program by running **scheduler** with different values (eg. **scheduler 1**) and looking at the output. Make sure it is selecting the right process for each time stamp. Feel free to add print statements (or use the **gdb** debugger) to make sure you are performing each of the functions properly. It is your responsibility to make sure to test and check your functions.

If your program does not compile, it will get a very low grade (just the code & comments points).

Code Style Guidelines (Will be Part of your Code and Comments Score)

1. No Global Variables may be used. (Global Constants are allowed; eg. const int val = 100;)
2. Always initialize all pointers with a value or to NULL on declaration.
3. Each block of code should increase the indent by 2-4 spaces.
4. Only use one statement per line. (eg. int x = 42; int y = 32; would be wrong)
5. Use Braces { } around all if/if else/else statements, even if they only have one statement.
6. Always check the return value for a call to malloc() to make sure it is not NULL
7. Set pointers to NULL after freeing them.
8. Functions **should** be fairly short (20 lines of code). If you have large functions that perform several operations, you should break them into smaller functions.