ECE 2230: Computer Systems Engineering
Machine Problem 1

Spring 2019

Due: 11:59 pm, Friday, January 25

1 Introduction

The goal of this machine problem is to familiarize you with the basics of maintaining information with dynamic memory allocations and pointers. This assignment provides a simple example of data abstraction for organizing information in a list. We will develop a simple abstraction with a few key interfaces and a simple underlying implementation using sequential arrays. In a subsequent programming assignment, we will expand upon the interfaces and explore alternative implementations. We will refer to this abstract data type (ADT) as a sequential list.

2 Problem Statement

Modern computers from your cellphone to the largest supercomputers are capable of running tens to thousands of processes by sharing the system's resources. These processes can be single threaded in which a single worker executes the code or multi-threaded where portions of the code are partitioned into tasks and are scheduled to run in parallel. Each of these parallel tasks has its associated input data and produces an output. In the context of parallel applications, the output of one task is often the input for another.

In order to run tasks or processes on current systems, they must be scheduled by the operating system or a parallel runtime system for execution on the CPU. Because there are many more tasks and processes than CPU cores, each task and core is only able to run for a short window of time. After this time window expires the task/process is blocked and a new task/process is scheduled to run. Cycling though all the tasks/processes running on the system increases the responsiveness and throughput of the system. Central to scheduling the processes/tasks running on the system is efficient management of the available tasks along with their current executing states. You will learn more information about tasks and processes and scheduling in future ECE courses: ECE 3220, ECE 4730, ECE 4780, etc.

For this project, we will write the ADT to store and operate on an array of tasks in a similar fashion to a parallel runtime system. Each task contains its own input arguments and information about its current state. Each task can exist in one of the following states: QUEUED, BLOCKED, RUNNING, or FINISHED. Figure 1 shows the relation between these states. All tasks start in the QUEUED state. After being scheduled they transition into the RUNNING state. During this state, program code is executed and data is read/written to memory. When the task yields the CPU core on which it is executing to another task, it transitions to the BLOCKED state. When that task is scheduled again it will return to the RUNNING state. Once the task completes, it enters the FINISHED state.

You are to write a C program that must consist of the following two files:

task_list.c – contains the ADT code for our sequential list. The interface functions must be exactly defined as described below.

task_list.h – contains the interface declarations for the ADT (e.g., constants, structure definitions, and prototypes). This file is provided and no changes can be made to it.

We provide a driver to test these codes. **DO NOT** modify the functionality of this file.

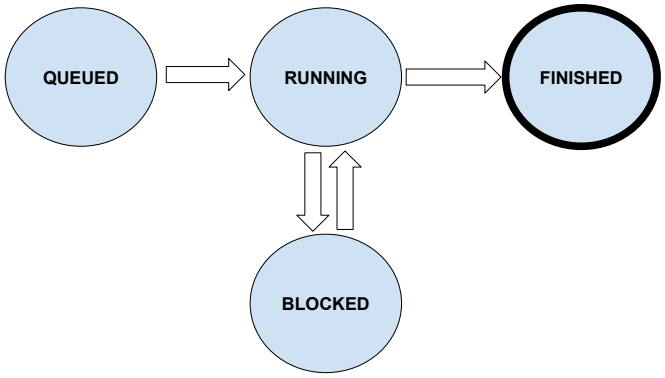

Figure 1: Task state transition diagram for MP1 tasks.

lab1.c – contains the main() function, menu code for handling simple input and output used to test our ADT, and any other functions that are not part of the ADT.

Your program must use an array of pointers to C structures that contain information for each task. The task information that is stored is represented with a C structure. The details are provided in the `task_list.h` file, and no changes to this file are permitted. The data structure for the list is defined as follows:

```
struct task_list_t {
    int task_count;               // current number of records in list
    int list_size;                // size of the list
    struct task_t **task_ptr;     // array of task pointers
};

struct task_t {
    int task_id;                // unique task id
    int priority;               // scheduling priority
    enum state state;           // scheduling state
    double wallclocktime;       // task runtime in seconds
    int nargs;                  // number of input arguments
    int *args;                  // input argument array
    int output;                 // task result
};
```

The sequential list ADT must have the following interface:

```
struct  task_list_t *task_list_construct(int size);
void    task_list_destruct(struct task_list_t *);
int     task_list_number_entries(struct task_list_t *);
int     task_list_add(struct task_list_t *, struct task_t *);
struct  task_t *task_list_access(struct task_list_t *, int idx);
struct  task_t *task_list_remove(struct task_list_t *, int idx);
int     task_list_lookup_first_priority(struct task_list_t *, int priority);
int     task_list_lookup_id(struct task_list_t *, int id);
struct  task_t *task_list_access_priority(struct task_list_t *, int priority);
struct  task_t *task_list_remove_priority(struct task_list_t *, int priority);
struct  task_t *task_list_access_id(struct task_list_t *, int id);
struct  task_t *task_list_remove_id(struct task_list_t *, int id);
```

```
int     task_list_determine_runable(struct task_list_t *, int nargs, int*args);
void    task_list_set_state(struct task_list_t *, int id,  enum state state);
struct  task_list_t* task_list_remove_finished(struct task_list_t *);
struct  task_t* task_list_schedule(struct task_list_t *, int priority, int id);
```

task_list_construct should return a pointer to the header block for the data structure. The data structure includes an array of pointers where the size of the array is equal to the value passed in to the function (the size is a command line parameter as shown in `lab1.c`. See `list_size` in `lab1.c`). Each element in the array is defined as a pointer to a structure of type `task_t`. Each pointer in the array should be initialized to `NULL`. The caller should eventually free the task list with task_list_destruct().

task_list_destruct should free the array of type `task_t*` (do not delete the tasks as they will be freed in main since it allocated them), and finally free the memory block of type `task_list_t`.

task_list_number_entries returns the current length of the task list

task_list_add should take a `task_t` memory block (that is already populated with input information) and insert it at the end of the list such that the list is sequential with no empty gaps between entries in the list. That is, the first record must be found at index position 0, the next ordered record at index position 1, etc. The function should return 1 if you inserted the new record into the list, or it should return -1 if the list is full and the insertion fails.

task_list_access should return a pointer to the `task_t` memory block that is found in the list index position specified as a parameter. If the index is out of bounds or if no task record is found at the index position, the function returns `NULL` and the list is not modified. The caller must not free the returned task.

task_list_remove should remove the memory block at the given index from the list and return the pointer to the memory block. The resulting list should still be sequential with no gaps between entries in the list. If the index given to the remove function is not valid, the function returns `NULL`. The caller may free the returned task list.

task_list_lookup_first_priority should find the `task_t` memory block at the lowest index in the list that matches the specified `priority` and return the index position of the record within the list. If the `task_id` is not found, then return -1.

task_list_lookup_id should find the `task_t` memory block in the list that matches the specified `task_id` and return the index position of the record within the list. If the `task_id` is not found, then return -1.

task_list_access_priority should find the first `task_t` memory block in the list that matches the specified `priority` and return a pointer to the record. If the `priority` is not found, then return `NULL`.

task_list_remove_priority should remove the first memory block from the list with a matching `priority` and return the pointer to the memory block. The resulting list should still be sequential with no gaps between entries in the list. If the `priority` given to the remove function is not valid or no entry exists, the function returns `NULL` and the list is not modified.

task_list_access_id should find the first `task_t` memory block in the list that matches the specified `task_id` and return a pointer to the record. If the `task_id` is not found, then return `NULL`.

task_list_remove_id should remove the first memory block from the list with a matching `task_id` and return the pointer to the memory block. The resulting list should still be sequential with no gaps between entries in the list. If the `task_id` given to the remove function is not valid or no entry exists, the function returns `NULL` and the list is not modified.

task_list_determine_runable should return the index of the first runable task. A task is runable if it is either QUEUED or BLOCKED and all the task's arguments and the supplied arguments agree. If no task is runable then return -1.

task_list_set_state sets the `task_id` task to the current state only if and only if that is a valid state (see Figure 1).

task_list_remove_finished searches the task list for all tasks in the FINISHED state and removes them. All removed tasks are combined into a new `task_list_t` and returned. The resulting lists should still be sequential with no gaps between entries in the list. If no tasks are finished then the function returns a new empty task list. The caller is responsible for freeing the returned task list.

task_list_schedule identifies a task to run based on `priority` and `task_id` and sets the state to RUNNING. If task_id is set and valid, always schedule that task. If this is not true, schedule the first task whose priority equals the given priority. On success return the task that was scheduled. Otherwise return NULL. Note: FINISHED and RUNNING tasks can not be scheduled.

The driver file `lab1.c` provides the framework for input and output and testing the sequential list of task list information. The code reads from standard input the commands listed below. The driver contains prints to standard output, and will be used when grading your code. The driver code in `lab1.c` calls functions found in task_list.c. Based on the return information you will call the appropriate print statements. **DO NOT INCLUDE `printf` statements in your final submission of `task_list.c`.** Doing so may impact grading as we look at the accuracy of standard output for our test cases. Four example input files and the exact output that is required are provided. Do not submit your code if your program cannot produce an exact match to the given output files. If your code is not a perfect match, you must contact the instructor or TA and fix your code. These are the input commands:

INSERT
FIND id
REMOVE id
UPDATE id state
PRINT
STATS
SCHEDULE id priority
DETERMINE
CLEAN
QUIT

The INSERT command allocates a dynamic memory block for the `task_t` structure using `malloc()` and then calls the `fill_task_record` function to prompt for each field of the record. After all the information is collected, an attempt to add the record to the list is made. The `task_list_add` function can insert or reject the record, and prints a corresponding output message. The FIND command prints the information for the task record for which the `task_id` matches. The REMOVE command removes the first entry of the list that matches the provided task_id, but also removes the record from the list (and frees the memory for the record). The UPDATE command updates the `state` of the `task_id` task (if possible). The PRINT command prints each record if there are one or more records in the list. The STATS command prints the number of records in the list. The SCHEDULE command identifies a task to schedule and sets that task's `state` to RUNNING. The DETERMINE command prompts the user for input to create input arguments that are used to determine if a task is runable. The CLEAN command removes all FINISHED tasks from the task list. Finally, the QUIT command frees all the dynamic memory and ends the program.

3 Notes

1. The `task_list_*` function prototypes defined above must be listed in `task_list.h` and the corresponding functions must be found in the `task_list.c` file. Code in `lab1.c` calls functions defined in `task_list.c` only if its prototype is listed in `task_list.h`. You can also add other "private"

functions to `task_list.c`, however, these private functions can only be called from within other functions in `task_list.c`. The prototypes for your private functions **cannot** be listed in `task_list.h`. Note we are using the principle of information hiding: code in `lab1.c` does not "see" any of the details of the data structure used in `task_list.c`. The only information that `lab1.c` has about the task list data structure is found in `task_list.h` (and any "private" functions you add to `task_list.c` are not available to `lab1.c`). The fact that `task_list.c` uses an array of pointers is unimportant to `lab1.c`, and if we redesign the data structure no changes are required in `lab1.c` (including PRINT).

2. Several of the functions you are asked to write will need similar operations. To reduce the volume of code you will have to write, leverage functions already designed or your custom private functions. Moreover, reusing existing debugged code helps to reduce the likelihood of introducing new bugs into your program.

3. Recall that you compile your code using: `make`
   You can pipe my example test scripts as input using <. Collect output in a file using > For example, to run do `./lab1 10 < testinput.txt > testoutput.txt` The code you submit must compile using the `-Wall` flag and no compiler errors or warnings should be printed. All students must verify that their code compiles with no warnings and runs correctly on the CES lab machines (*.ces.clemson.edu). An example `testinput.txt` and `expectedoutput.txt` files are provided. When you run your code on the `testinput.txt` file, your output must be identical to the file `expectedoutput.txt`. You can verify this using `diff testoutput.txt expectedoutput.txt`
   However, **the tests in `testinput.txt` are incomplete!** You must develop more thorough tests (e.g., attempt to delete from an empty list, or insert into a list that is already full). If you have access to a graphical display you can use `meld` in place of `diff`.

4. Be sure that your program does not have any memory leaks. That is, all dynamically allocated memory must be freed before the program ends.
   We will test for memory leaks with `valgrind`. You execute valgrind using `valgrind --leak-check=yes ./lab1 10 < testinput.txt`
   The last line of output from valgrind must be: `ERROR SUMMARY: 0 errors from 0 contexts (suppressed: x from y)` You can ignore the values x and y because suppressed errors are not important and are hidden from you. In addition the summary of the memory heap must show: `All heap blocks were freed -- no leaks are possible`

5. Compress .c and .h files into a ZIP file, and submit the ZIP file to Canvas by the deadline. **Your last submission is the one that will be graded.**

Work must be completed by each individual student. See the course syllabus for additional policies.