Project 2:
CUinSPACE Simulated Flight

***Note:** This assignment is not affiliated with the actual CU InSpace student rocket engineering design team at Carleton, but you can learn more about them **here**!*

Overview

In this project, you will be writing a Makefile and working with large amounts of provided code to get practice working with a simulation and implementing dynamic memory and multi-threading. Most of the simulation logic is provided, and you will mainly be focusing your attention on reading and understanding the code, dynamically allocating memory, freeing the dynamically allocated memory, and then reworking the provided code to enable the simulation to run using multiple threads. If you wish, you can approach the project by getting multi-threading implemented right away, though this could complicate the dynamic memory debugging.

Learning Outcomes

By the end of this assessment, you will have assessed the following… Can you:

- **(Chapter 4.1, 4.2)** Write and use a `Makefile` to compile separate source files into object files and link them into a final executable?
- **(Chapter 3.4)** Allocate and deallocate collection structures using `malloc`, `calloc`, and `free()`?
- **(Chapter 3.3, 3.4, 3.5)** Work with and correctly free dynamic memory which is pointed to by multiple pointers?
- **(Chapter 5.1, 5.4)** Set up multiple threads with `pthread` and synchronize data shared between them using `sem_t` semaphores?

If so, that means you are very likely on track! Make sure to consider these outcomes while you are working to help guide your understanding of different requirements.

(Optional) Context

Hey there! We've made some great strides in our rocket competitions, and your work setting up our data in the previous project was very helpful! We were able to adapt your code and expand it out to work with our Arduino setup. Very exciting. We want to start laying some groundwork for more advanced simulations. Our system is simple: Individual rocket systems consume resource and produce resources. They store their produced resources, and other systems can take those and consume them to produce more! When they have too much to store or too little to produce, they give the "manager" and event, and the manager will speed up or slow down production of systems to balance things out.

A junior developer already set up the basic control flow based on our requirements, but unfortunately, they know how to **use** C, but don't know much about what's happening in the memory. This means their dynamic memory allocation was… Well, we erased it all, and when we asked for a multi-threaded simulation, they said, "One thread is a multiple of every number!". They even scribbled a long shell command on a piece of paper as their "build instructions"… We know you've got the skills to fix this!

You've got this!

Instructions

This project is broken into separate modules, each forming different components of the rubric. Some parts may be completed out of order, but all functionality must be executed and shown to be working correctly to receive full marks.

Avoid wasting memory where possible; for example, do not make unnecessary copies of structures passed in via pointers unless required by the function.

Use appropriate syntax and practices, such as using `->` to access fields in a structure pointer. You must identify and use functions that have already been written wherever possible for full design marks.

In this project, you will work on the following parts and utilize the following skills (may not be exhaustive):

1. Makefiles
    a. Creating a Makefile to build the project
2. Resources and Systems (Dynamic Arrays)
    a. Allocating and freeing dynamic memory for structures.
    b. Initializing structures by passing pointers to functions.
    c. Appending data to and resizing dynamic arrays with malloc, calloc, and free.
3. Event Queue (Linked Lists)
    a. Implementing a linked list to queue events.
4. Multi-Threading Overhaul
    a. Creating and managing threads for systems.
5. Synchronization
    a. Ensuring proper synchronization between threads using semaphores (`sem_t`)

Provided Materials and Important Information

You have been provided with a header file (e.g., `defs.h`) and partial implementations of most source files. The logic is organized to function well with either a multi-threaded or single-threaded implementation, although you will need to modify some of the provided code in order to support more than a single thread.

The functions have all been thoroughly documented to provide you with lots of information about how they should function. **Note:** These are using the a common format for function documentation that makes it easier for IDEs like VS Code to provide information about functions when they are hovered over with a mouse (when C/C++ Extensions are installed).

The simulation involves multiple systems that consume and produce resources, generate events, and are managed by a central manager.

Each `System` will run its own thread, simulating the consumption and production of resources, and generating events when necessary.

The `Manager` will run in its own thread and will handle events that are queued in a Linked List by modifying the speed that each system produces resources, and will terminate the simulation if it detects that the maximum distance has been reached or if the oxygen system runs out of resources.

The basic flow of the simulation is described in the provided **Sample I/O Video posted to Brightspace**.

Part 1: Makefiles

First begin by improving your build process for the project by creating a `Makefile`. The `Makefile` must satisfy the following requirements:

1. It is named `Makefile` and calling `make` in the directory successfully creates an executable file that can be run.
2. Each source file is compiled into a single object file in the directory that `make` is called in.
3. Makefile does not use generic / wildcard targets; each file is explicitly specified (simple macros acceptable)
4. When `make` is run, only the modified C files must be recompiled.
5. It is highly recommended that you build using `-g -Wall -Wextra -pthread` options;
    5.1. `-g` provides additional debugging information
    5.2. `-Wall -Wextra` provide warnings that can help you catch potentially major issues early
    5.3. `-pthread` will link the `pthread` library and include options helpful for debugging
6. Include a target `clean` which will remove all object files and executables when run using `make clean`

At this point, the provided code should be able to be built using `make`, but will cause an error almost immediately upon executing because none of the structures have been correctly created at this stage.

Part 2: Resources and Systems (Dynamic Arrays)

In this section, you will write code to manage resources and systems, including dynamic memory allocation and initialization.

Testing at this stage is difficult, as a lot of functionality must be operational to get the simulation going. You may wish to test your functions individually to make sure they are performing as expected.

This section ignores any multi-threading requirements, and they may need to be revised later. Review the full specification before beginning to determine if you would like to add any additional elements while working on these to support threading early.

1. `void resource_create(Resource **resource, const char *name, int amount, int max_capacity);` creates a new `Resource` struct.
    1.1. Allocate memory for a new `Resource`.
    1.2. Dynamically allocate memory for the resource's `name` field and copy the `name` argument into this memory.
    1.3. Initialize each of the fields using the arguments that were provided.
2. `void resource_destroy(Resource *resource);` destroys a `Resource` struct.
    2.1. Free any dynamically allocated memory associated with the `resource` argument.
3. `void resource_array_init(ResourceArray *array);` initializes an empty `ResourceArray` with an initial capacity of 1.
    3.1. Dynamically allocate memory for the array of `Resource*` pointers, capacity 1.
    3.2. Initialize any data required for the array fields.
4. `void resource_array_clean(ResourceArray *array);` cleans up the `ResourceArray` by destroying all resources and freeing memory.
    4.1. Iterate through the array and appropriately destroy the memory pointed to by the members of the array.
    4.2. Free any other memory associated with the array.

5. `void resource_array_add(ResourceArray *array, Resource *resource);` adds a `Resource` to the `ResourceArray`, resizing if necessary.
    5.1. Add the `Resource` to the end of the array.
    5.2. If there is insufficient capacity for a new element, double the size of the reserved array.
    5.3. Ensure that existing elements are copied to the new array when resizing.
    5.4. Remember: **realloc() is NOT permitted.** Use techniques described in class.
6. `void system_create(System **system, const char *name, ResourceAmount consumed, ResourceAmount produced, int processing_time, EventQueue *event_queue);` creates a new `System` object.
    6.1. Allocate memory for a new `System`.
    6.2. Dynamically allocate memory for the resource's `name` field and copy the `name` argument into this memory.
    6.3. Initialize any data required for the system's fields, setting the initial status to `STANDARD`.
    6.4. Set the initial `amount_stored` to zero.
7. `void system_destroy(System *system);` destroys a `System` struct.
    7.1. Free any dynamically allocated memory associated with the `system` argument.
8. Implement `system_array_init`, `system_array_clean`, and `system_array_add` functions following the same patterns as with the resource array functions.
9. `void manager_clean(Manager *manager);` uses existing functions to clean all of the memory of the simulation; i.e., the fields of the manager.

STOP! At this point, you should test your code by initializing resources and systems and ensuring that memory is properly allocated and freed without leaks. Use tools like `valgrind` to check for memory leaks. The current implementation will run in an infinite loop, so you may wish to add a counter or similar to the loop in `main()` so that it breaks after, for example, 5 or 10 iterations. There should be no memory leaks at this stage. **This is a good time to package your project and submit it for part marks and to ensure you have something submitted before the deadline!**

Part 3: Event Queue (Linked Lists)

In this section, you will implement an event queue to handle events generated by systems, focusing on dynamic memory management and synchronization. The `EventQueue` structure is a linked list which maintains a `head`, but no `tail`.

Each `Event` structure has a `priority` value. New events should be added to the `EventQueue` by priority, with the oldest events of the same priority level coming first, and events are popped from the front (oldest, highest priority events).

1. `void event_queue_init(EventQueue *queue);` initializes the `head` of the `EventQueue` to `NULL`.
2. `void event_queue_clean(EventQueue *queue);` free any memory associated with the `queue`.
3. `void event_queue_push(EventQueue *queue, const Event *event);` pushes an `Event` onto the `EventQueue` in priority order, ensuring the oldest events of the same priority level come first.
    3.1. Iterate over each `EventNode` in the `queue` until finding an `Event` of lower priority
    3.2. Dynamically allocate a new `EventNode` to store the `event` and insert it at the new position
    3.3. **Example:** If a priority 3 event is being added to a `queue` containing four priority 3 events and two priority 2 events, the new event will be inserted in between the priority 3 and priority 2 events.
4. `int event_queue_pop(EventQueue *queue, Event *event);` pops an `Event` from the front of the `EventQueue` and returns a non-zero value if an event was successfully popped or zero otherwise.

4.1. If there are any events in the `queue`, remove the event at the `head` of the queue and copy its `event` data into the `*event` argument. Return a non-zero value (e.g., 1);
4.2. If there were no events in the `queue`, return 0.

STOP! Run `valgrind` with your program and make sure that there are no memory leaks. The program should now correctly execute a full simulation up until the end and exit correctly when the oxygen reaches 0 (unlikely while single-threaded) or distance reaches maximum. **This is a great place to submit for part marks before continuing!**

Part 4: Multi-Threading Overhaul

You are strongly advised to back up your work before attempting to implement multi-threading so that you can submit a fully functional version of the project before starting with threads.

In this section, you will implement the required modifications to the program to support multi-threading. Advice for debugging multi-threaded code can be found in the Assignment 6 overview video on Brightspace. You will be creating one thread for each system and one thread for the manager.

Note that a few "stale" data races are acceptable; this means that if the thread is looking at a variable that is modified partway through and the value changed, but it doesn't really affect the correctness of the program, it is an acceptable data race. Any data races caused by `display_simulation_state()` are acceptable as they do not affect the overall correctness, and reading the system's `state` is acceptable, as only the `manager` thread will be modifying these.

You will be partly required to work out which pieces of code require synchronization and precisely how to implements it. You must make use of `sem_t` for your mutex.

1. `defs.h`: Add two new function forward declarations: One for the `system` threads and one for the `manager` thread, with the appropriate signature for a threaded function.
2. Create thread functions and update `main()`. The project was designed so that the `system_run()` and `manager_run()` functions can handle most of the behaviour of the simulation, but you will still need a thread function with the appropriate signature to pass to `pthread_create()`.
    2.1. In `system.c`, implement your `system_thread()` function.
        2.1.a. The thread function must only expect to be passed a single `System*` and pass this into `system_run()`.
        2.1.b. Call `system_run()` in a loop which exits when the system's `status` field is `TERMINATE`.
    2.2. In `manager.c`, implement your `manager_thread()` function.
        2.2.a. The thread function must only expect to be passed a single `Manager*` and pass this into `manager_run()`.
        2.2.b. Call `manager_run()` in a loop which exits when the manager's `simulation_running` variable is zero.
    2.3. Update `main()` to create your threads and wait for the threads to return:
        2.3.a. Create one `pthread_t` for the manager and one `pthread_t` for each system in the `manager.system_array`.
        2.3.b. Call `pthread_join()` to wait for all threads to complete execution before resuming.

At this point, your code should be able to run with multiple threads, but *poorly* because synchronization is not implemented to protect the shared data. It may successfully run a few times, but may fail due to

segmentation faults or hang without being able to complete, for example, so now we will add synchronization.

You may wish to submit at this point for part marks, but you are encouraged to try to get synchronization functioning before this.

Part 5: Synchronization

Here we will create some semaphores for data that must be protected and use calls to `sem_wait()` and `sem_post()` to protect the data.

1. Add `sem_t` semaphores as fields of structures in `defs.h`:
    1.1. `Resource` structures can have their amount modified at any time by multiple threads,
    1.2. The `EventQueue` is pointed to by every system and the manager, so it can certainly be modified at any time by multiple threads.
2. Initialize the semaphores in `resource_create()` and `event_queue_init()` with an initial value of `1`.
3. Clean up the semaphores in the relevant `destroy` and `clean` functions using `sem_destroy()`
4. Protect the `EventQueue`: The event queue will be modified in a number of locations, so we will add synchronization to the functions `event_queue_push()` and `event_queue_pop()`
    4.1. `event_queue_push()`: The integrity of the whole list is important for this function, so it should be protected the entire time that it is being traversed using the mutex semaphore that you added to the `EventQueue`.
    4.2. `event_queue_pop()`: Use the semaphore to prevent modification to the queue while it is being updated/traversed. Remember to `sem_post` the semaphore in the `EventQueue` in all cases.
5. Next, update the functions in `system.c` to protect the data in a `Resource` when it is being modified.
    5.1. `system_convert()`: When resource amounts are being modified, ensure that the resource cannot be modified by any other thread at the same time by using the system's mutex semaphore field that you added previously.
    5.2. `system_store_resources()`: It is important that the amount of space is correctly calculated and is not modified while the resource amounts are being modified. Similar to `system_convert()`, add calls to `sem_wait()` and `sem_post()` to protect the data.

At this point, you should be able to run `valgrind` and receive no memory errors, perform many runs of the program without any issues, and follow the steps in the Assignment 6 overview video which provides assistance with debugging multi-threaded programs to see only a few data races that can be traced back to acceptably stale values.

Packaging and Submission

Always make sure to **download and test your submission** yourself with plenty of time before the submission period to make changes.

Your submission must meet the following requirements for full marks:

- All files must be included in a `.zip` file which can be decompressed in the Linux terminal and include `.zip` at the end of the filename,
- Do **not** include compiled code (object files or executables) or any other unnecessary files or folders (e.g., `.git` folders or rough work),
- Include a `README.md` file which includes:
  - Instructions to compile your program,
  - Citations of any sources you used to assist you during this project,
  - The names of everyone involved in the creation of this project,
- The project **must compile and execute** to receive the vast majority of marks. Code which does not compile and execute will face **major penalties**. If you have partially completed code, make sure that it is not executed, and if it causes issues in compilation, that it is commented out with a description of the issue you are encountering in the comments.

Failure to follow these guidelines can lead to major penalties and even a **zero** if the project can not be opened and reviewed.

Grading

This project will be graded partially using an autograder but primarily by manual review of teaching assistants.

An approximate grading breakdown will be provided shortly after the release of the specification alongside the autograder.

Part 1: Autograder

(Coming Soon) The autograder will perform some evaluation of the correctness of individual functions to ensure they follow the correct logic. Expect similar tests to Assignments 4, 5, and 6, where dynamic memory is evaluated on a per-function basis as well as checks for data races. A few data races will be automatically excluded (namely, those involved with `display_simulation_state` and reading the `status` of a system).

- **Makefile:** A Makefile is present.
- **Dynamic Memory Allocation:**
  - Dynamic memory is correctly allocated and deallocated, in the correct functions, for: New `EventNode`s, new `Resource`s and `Resource` names, new `System`s and `System` names, new arrays of `System`s and `Resource`s when initializing and resizing `System` and `Resource` arrays.
- **Correctness:**
  - `Resource`s, `System`s, and their respective arrays, `EventQueues` are initialized with the correct values
  - Events are pushed to and popped from the correct location in the `EventQueue`
  - `Resource` and `System` are resized correctly (doubled, without use of `realloc`)
- **Multithreading:**
  - Correct number of threads are created,
  - Only acceptable data races are reported (minimal ones as described above)

Part 2: TA Review

- **Makefile**: Makefile correctly compiles the program into separate object files and includes a `clean` target which correctly removes all object files and executables.
- **Execution and Usability**: Program executes multiple times without issues.
  - No compilation errors
  - No compilation warnings
  - No memory leaks or memory warnings
- **Approach**:
  - Correctly using appropriate techniques discussed in class
  - Uses specified algorithms where specified
  - Appropriate use of provided functions and calling existing functions instead of duplicating code
- **Code Design and Packaging**:
  - Including README
  - Avoiding magic numbers and using provided definitions where applicable
  - Documenting newly created functions with input/output parameters and return values,
  - Clear and consistent whitespace and readable variable names
  - Consistent use of `->` operator where appropriate
  - Does not create temporary structures to avoid using pointer syntax
- **Deductions:** Additional deductions may apply for unanticipated violations of the specification or course code design guidelines.
  - Use of global variables
  - Including additional header files not required by the assignment (exception: `stdbool.h` and `stdint.h` are permitted)
  - Use of `realloc()`: While this is an excellent tool to use in your regular programming, we want to evaluate your understanding of memory and use of `malloc()` and/or `calloc()` with `free()` helps us evaluate this more clearly
  - Modifying provided code where it is not specified

Important Reminders

Remember to review all policies in the course outline.

1. Remember that technical issues at the deadline are **NOT** grounds for additional accommodations, submit early, make backups. If you cannot submit to the course website on time, send your code via email and ensure it was received. We **must** have evidence of submission prior to the deadline.
2. Remember that extenuating circumstances for extensions do **NOT** include travel, starting too late, having too much work in different courses, or other circumstances within your control.
3. Remember that a short-term accommodation is five days or less, available only for extenuating circumstances.
4. Note that a long-term accommodation at the end-of-term can be very stressful, as it will lead to greater uncertainty of your grade and put much more weight on other materials.
5. Consider reviewing the relevant course notes before jumping into a section of work to have a more grounded foundation before started.
6. Autograder marks are **subject to change** if any elements are found to be incomplete or misaligned with the project specification. It is your responsibility to review the requirements, seek clarification on the forums, and review any clarifications in announcements ahead of the deadline.

Appendix A: Pairs and Teamwork

You **must** register your pair if you wish to work in pairs.

If you work in pairs, you are permitted to submit together or individually. You may wish to submit individually if you do not want to commit to consistently working together, but still want to discuss the approach with another student without committing misconduct.

Note that discussing your problem solving or code with a student you are not properly paired with constitutes **academic misconduct**, and any highly similar code between students not in a group together may be forwarded to the Dean's Office for investigation.

Part 1: Registering Your Group

For your group to be official, you must do two things:

1. Join the same group on Brightspace:
    1.1. Go to "Tools > Groups" and select "Project #2 Groups"
    1.2. Join the same group number as your partner
    1.3. This will grant you access to a private discussion forum for your group and a group locker where you can store files.
    1.4. A guide is available here: https://carleton.ca/brightspace/students/viewing-groups-and-using-groups-locker/
2. Submit as a group on Gradescope, **if** you are planning to submit together:
    2.1. When you submit your project, and you should try to submit early as a test, select "Group Members" or "View" or "Edit" Group in the outline area of the submission. A guide is available here: https://guides.gradescope.com/hc/en-us/articles/21863861823373-Adding-Group-Members-to-a-Submission

It is your responsibility to register your group on Brightspace before the deadline (March 26th at 23:59).

If you would like to find a partner member, you can post to the "Project 2: Looking for Partner" topic on Brightspace. Unfortunately, random partners will not be assigned at this time, but you can look for a partner via the forums.

If you are collaborating with a partner but you do not wish to submit with each other, simply discuss the project together and compare, this is acceptable **if and only if** you register as a group on Brightspace before the deadline. Collaboration with a partner not registered on Brightspace before the deadline and not submitting together is **not** acceptable and is considered misconduct.

Part 2: Setting Group Expectations

- Compare your schedules at the beginning of the project to identify when you can work together,
  - Identify times that you can meet and schedule multiple times per week to collaborate on the project,
  - Identify any times when you will be away for several days/
- Do **NOT** plan to take a "divide-and-conquer" approach to projects where each student takes on a different component and merges together your work:
  - This can lead to lower final exam scores by not getting necessary experience with the material,
  - The merging process can be very challenging, particularly with C code,
  - You can learn significantly more by co-working together on code and discussing it together,
- You are **strongly** recommended to write a small team contract for your partnership to hold each other accountable:
  - Who is responsible for submitting the final project submission?
  - What dates will we meet?
  - About how many hours per work session/overall do we each intend to commit?
  - What is our target grade / overall expectations for the project?
  - How will we share files and collaborate?
  - Will we meet online or in-person?
  - Where are we going to hold discussions?
  - Under what circumstances will we be forced to work separately? E.g., missing a team meeting, missing deadlines, not responding for X days in a row

Part 3: Pair Programming

You should be taking a "pair programming" approach with teamwork. Pair programming is a standard practice in industry, it can help learning, and can lead to better code. Pair programming is often misunderstood, but here is a general description and some advice for effective pair programming:

Pair Programming has two roles: The "Driver" and the "Navigator"

- The Driver is in front of the keyboard and is typing the code.
  - **Talking:** The Driver should be talking out loud about what they are coding, what they are confused about, and what they are planning to do next.
  - **Action:** The Driver is writing code and focusing on the current task. They should ask questions to the navigator if anything is unclear or if they need something to be reviewed or looked up.
- The Navigator is beside the Driver, and is focusing on reading code and getting ahead of problems.
  - **Talking:** The Navigator should be answering questions, checking in if they get confused by the Driver's approach or if they aren't sure what the Driver is doing.
  - **Action:** The Navigator typically balances a few tasks, but their main goal is to help maintain the Driver's coding flow and make sure the code being written is high quality.
    - **Review Specification:** The Navigator can keep the specification open for the current problem being solved or the next problem they will be tackling and keeping the requirements in mind. They can consider edge cases; "What if the pointer is null?"
    - **Review Code:** The Navigator can watch the code being written and try to consider if there are better approaches or issues that might arise. The Navigator should try to do this problem solving without interrupting the Driver until they have something important.
    - **Review Documentation:** If there are C functions or syntax that will be needed that might be confusing (e.g., `string.h` functions), they can get the documentation ready to clarify for the Driver.
    - **Review Memory:** The Navigator can try to focus on the memory side of things by considering what pointers are pointing to, what structure fields are, and how the data should update.

It is important to recognize that **pair programming must not be silent**. You are working together to provide a commentary of your thought process and help overcome problems. You do not want to be distracting, but you also want to make sure you are both aware of what is happening.

Change roles frequently. It is typical to swap roles every 15-30 minutes to keep fresh and make sure both programmers are comfortable with what is being worked on. A common structure is to work for about 20 minutes, take 5 minutes to discuss how things are going and plan what's next, then swap roles and continue. If you notice that things are getting quiet, or that you are getting stuck, or that no code has been written for a while, this is usually a good indicator that a swap is needed.

Pair Programming should **not** look like one student completing the project silently while another one watches. Both programmers are vital, active participants in the process. By having a second pair of eyes look up documentation, read the code to catch bugs and typos before they become problems, you should save yourself time debugging later and missing important pieces of the specification.

Part 4: Planning

It can be very helpful as a team, **and** as an individual, to set up a plan. Break the project into manageable tasks, estimate about how long each task will take (approximately), double (or triple) those estimates for safety.

Then for each task, write out a checklist of requirements until you would consider that task “Done”. For example:

```
01  Task, 30 minutes: Get the menu printing placeholder error messages for each menu option.
02  - [ ] Add a definition for ERR_NOT_IMPLEMENTED
03  - [ ] Write empty functions for every function that return ERR_NOT_IMPLEMENTED
04  - [ ] Write a while loop and if statements in main() that prints the menu calls the
placeholder functions
05  - [ ] Write comments / write code for any other functionality we'd need?
```

Try to order the tasks in the order you think you will need to tackle them. Once you have this task list, it can become **much** easier to work through your project and feel confident about finishing each task. You can also set clear goals for where you want to wrap up in a work session. It also assists in pair programming situations, where the navigator can keep the current and next tasks in mind and verify that all requirements are met as you work.

Documentation is another way to assist in planning. For each function, before you even start writing code, consider writing the function comments. For example, consider all of the error codes that a function might return, what the parameters are expected to look like, and a brief description of what the function is trying to accomplish. This way, you don’t have to keep swapping out to refer to other descriptions each time you start work on a new function.

Part 5: Planning: 1% Side Quest Mark

Each project has an opportunity to receive 1% in side quest marks for submitting a plan for your project prior to March 21st at 23:59.

It can be very helpful as a team, **and** as an individual, to set up a plan. Break the project into manageable tasks, estimate about how long each task will take (approximately), double/triple those estimates for safety.

Then for each task, write out a checklist of requirements until you would consider that task "Done". \

```
Task, 30 minutes: Get the menu printing placeholder error messages for each menu option.
- [ ] Add a definition for ERR_NOT_IMPLEMENTED
- [ ] Write empty functions for every function that return ERR_NOT_IMPLEMENTED
- [ ] Write a while loop and if statements in main() that prints the menu calls the
placeholder functions
- [ ] Write comments / write code for any other functionality we'd need?
```

Try to order the tasks in the order you think you will need to tackle them. Once you have this task list, it can become **much** easier to work through your project and feel confident about finishing each task. You can also set clear goals for where you want to wrap up in a work session. It also assists in pair programming situations, where the navigator can keep the current and next tasks in mind and verify that all requirements are met as you work.

Documentation is another way to assist in planning. For each function, before you even start writing code, consider writing the function comments. For example, consider all of the error codes that a function might return, what the parameters are expected to look like, and a brief description of what the function is trying to accomplish. This way, you don't have to keep swapping out to refer to other descriptions each time you start work on a new function.

Side Quest Submission: In the private side quest forum, each individual on the project should submit their plan prior to March 21st at 23:59. The plan should at minimum include:

- A high-level task list of features to complete in approximate order. Time estimates and requirements checklists are not required, but encouraged to get practice and help you prepare.
- A description of the shared data and thoughts around the need for semaphores and threading (bullet list, description, some thoughts on your approach to synchronization)

Part 6: Version Control: 1% Side Quest Mark

Each project has an opportunity to receive 1% in side quest marks for working with a Version Control system during the project. NOTE: This is increased from the 0.5% mentioned in the course outline.

Version control is used across the industry and likely in any co-op position you might find yourself in. One of the most popular VC systems is `git`, which was created to help manage the creation of Linux. VC allows you to track changes to files during a project so that you can view changes from one version to the next or revert back to previous versions of a file. It is frequently used to support hosting backups of code online (e.g., GitHub) and collaborate with other programmers. Link: Interactive Git Tutorial, Link: Git EBook

Side Quest Submission: In the private side quest forum, each individual on the project should submit a screenshot or multiple screenshots which showcases that a version control system such as Git was used consistently throughout the project; e.g., showing a screenshot of a git history with many commits.