Operating Systems:
Final Project Specification

This project is intended to integrate many aspects of OS design and implementation, from containers, to scheduling, to synchronization, to memory management, and file systems. You are to implement this in the xv6 OS. You will implement the core servers, logic, and coordination facilities for a container environment in xv6! This is a large and hard project and is scoped around four team members. If you choose a smaller team, you're still responsible for the entire project, so I suggest you form your team soon. The *most important* thing you can do to ensure you finish on time is to **start early**. Like now. I'm not kidding. If you don't start for 2 weeks, you're *significantly* decreasing your chances of success. *To the keyboard!*

Final goal:
Container Management and Communication

The final goal of the project is to implement the core facilities for containers in xv6. Recall that containers are based on the *partitioning* of system namespaces to make it so that different subsets of processes cannot address the same resources, thus providing some isolation. Containers additionally attempt to use kernel facilities to prevent starvation between different containers. You'll implement a subset of the functionality provided by exceedingly common software like Docker.

Note that many of you will be using containers in the future for development, so a secondary goal of this assignment is that you can *put on your resume that you implemented a container infrastructure for xv6 UNIX*.

The containers will

- Execute a container that only has access to a subset of the file system. For example, if normally, the file system includes `/ls`, `/cat`, etc… However, if you want to run a container, you might want it to be able to access only a subset of the file system. We will simplify this by restricting a container to accessing files in a subtree of the file system (i.e. only files and directories in `/evil_users_container/*`). We will call this the *container's file system root*.
- Only be able to use a restricted number of system resources. While Docker can limited the CPU, memory, and I/O rates of access for each container, we'll simplify in xv6, and simply make sure that a given container can only create a specified maximum number of

processes, and can only use locks, condition variables, and shared memory regions that are used only within the container. The process restriction prevents a container from using a "fork bomb" to starve the rest of the system from creating processes, and the resource restrictions limit the access of the container to shared abstractions (locks, cvs, shared memory).

To get a perspective on containers, see one of the original papers on it! Students who did this assignment in the past have said that they wished they had read this before they did the project.

The containers infrastructure you'll implement includes the ability to:

- Containers must be managed by a *container manager*. This is a single process in the system that receives requests to create new containers, and interact with the new container via standard input and output.
- Each container will be specified using a simple json format. This specification will include
    a. The path to the container's file system root. For example, if this is set to `/evil_users_container/`, then the processes in that container will see their root (/) as containing the contents of that path. If `/evil_users_container/file1` and `/evil_users_container/file2` exist, programs within the container will see `/file1` and `/file2` when they run `ls /.`[1]
    b. The maximum number processes that can be created in the container.
    c. The initial program in the container's file system root[2] to run when the container is booted up.
- Booting up a container will run its initial program, which can only access files within its file system root, and only (itself or its process tree) create the maximum number of processes (minus one since the initial program is already running).

The container manager (CM) is a program you'll write and run in xv6. It will receive requests from other programs to boot up and run different containers. It will require a new set of system calls to change the root file system path accessible to the current process (similar to chroot), and restrict the number of processes that can be created by a process and its descendants. As such, it will use the new container-specific system calls you're adding. The container manager will need to receive requests from other processes to create new containers, which will require efficient communication.

To allow the user to interact with the CM, you'll provide a `dockv6` (a play on "docker") program that communicates and sends commands to the CM.

---

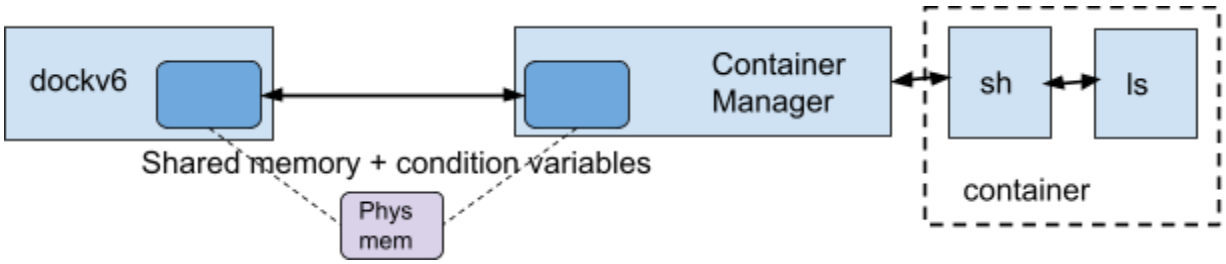
[1] Note that the `ls` program must also be present in the file system for you to be able to run `./ls`. This might get very annoying, so it might be worth writing a simple program to copy the programs into each container's directory when you boot up. Some details: to *test* your system, this means that you'll need a second directory that includes some programs to execute. I suggest that you write a program to make a directory (to use for this testing), and to copy some programs (e.g. `ls`) into it. Otherwise, you'll need to manually create the directory, and copy files into it every single time you want to test the system. This is not efficient.

[2] Again, you must be careful that the initial program is correctly placed in the container's file system, or it won't find the executable.

A hypothetical session in xv6 using containers:

```
$ ls /
cat
c1/
echo
…
$ ls /c1/
cat
ls
helloworld.txt
sh
$ cm &
Container Manager starting…
CM awaiting requests…
$ cat container_spec.json
{ “init”: “/sh”, “fs”: “/c1/”, “nproc”: 4 }
$ dockv6 create container_spec.json
CM creating container with init = sh, root fs = /c1/, and max num processes = 4
$ dockv6 io
// now we’re in the container’s sh process! The following input is being sent to the
// container, and the output from the container is being sent to dockv6, which is
then  // printing it out.
$ ls /
cat
ls
helloworld.txt
sh
$ cat helloworld.txt
I’m allllliiiiiveeee!
```

A diagram of the above system follows that depicts the system after the ls command has been executed. You’ll see in the coming sections that the shared memory and condition variables are specified.



Specification and Implementation Plan

There are four main *modules* to this project.

1. The Container Manager (CM) which uses the kernel API and your container-specific extensions to the kernel, which creates and manages new containers. The dockv6 command interacts with the CM. Eventually, the CM will likely be implemented as multiple processes so that its implementation can be concurrent (interacting with the terminal, and with the container).
2. Shared memory support for communication between the CM, and the processes that are using the CM's container support (e.g. `dockv6`). This shared memory region is used to both pass requests to the CM, and monitor the container's execution. This will require kernel hacking to add shared memory.
3. The synchronization code necessary to provide synchronization on that shared buffer. This will include both (blocking) mutexes (i.e. from hw4), and the ability to block waiting for an event (i.e. a new client request) -- a function normally provided by condition variables. This will require kernel hacking to implement mutexes for user-level (as you've already implemented), and to add logic for condition variables.
4. Scheduler policy additions that add multiple priorities, and schedule the highest priority thread at any point in time. This will include system calls for setting the priority of various threads and processes. The CM process should always execute with high priority as it is providing a service to other processes in the system.

There are varying levels of support for each of these components, and you should be able to develop each relatively independently, so if one person does not meet their commitment, the others should be able to still make progress. Level 1 is intended to be very basic functionality. Leveling up from there strengthens your project. Only a project at "project level 5" will receive full credit. Each level often, but doesn't always assume the functionality from the previous levels (i.e. you often can't get to level 2 without achieving level 1).

A note on group work and collaboration: I designed these modules to be independent, but that *does not mean* that they are of equal difficulty. Do not plan on simply assigning one to each team member, and when one of the modules is complete assume that team member is "done". After a module is done, you must help the other members on the other modules. Each team member must stay up-to-date with each other teammate. You should use github to coordinate your code modifications. In fact, I **highly** suggest that once each of the implementations get to Level 1, that you start integrating them together. Once integrated, you can keep leveling up in each module. Module 4 (scheduler) is the easiest, and the other 3 should be split initially between teammates.

Educational Objectives

Through the course of this assignment, you should get experience applying the following areas you've learned in class:

- Communication with a bounded buffer
- Shared memory
- Synchronization and condition variables
- Threads and processes
- Microkernels vs. monolithic systems (the CM is similar to a microkernel service)
- Scheduling
- File systems
- Threading

Module #1: Container Manager & Kernel Container Support

The CM has three main functionalities:

- *Container creation*: When the CM determines it should make a container, it will
    a. Create a set of file descriptors that will be pipes to emulate the stdin, stdout, and stderr file descriptors (numbers 0, 1, and 2, respectively). For normal processes these are usually just the console (see init.c for how these are created), but for our container, we want any reads from stdin, and writes to stdout/stderr to go to the container manager, and it will transfer them to clients (e.g. `dockv6`).
    b. Fork a new process to contain the initial container execution.
    c. Within that process, use new system calls to change the root to the container's file system root, and constrain the number of processes in the container.
    d. Execute the container's initial program by calling exec on it.

  At this point, the initial program in the container will essentially execute "normally", but will only see the files/directories within the container file system root, only be able to call fork (or have descendents that call fork) and create a limited number of processes, and all stdin/out/err goes through the CM (so that it can be appropriately multiplexed or sent to/from a client).
- *Communication with the client*: The CM does not create containers without prompting from other processes. The `dockv6` program is an example client program that communicates with the CM to 1. Send over the container specification (in json format), 2. Send and receive a stream of bytes
- Parsing the container specification: The container specification is a simple json file. An example specification is { "init": "/sh", "fs": "/c1/", "nproc": 4 }. I strongly recommend that you use the jsmn library to parse this. You'll have to make minor modifications to the library since xv6 doesn't provide stddev.h. I suggest you get your jsmn parsing code working in Linux, then port it to xv6.

The new kernel API to constrain the file system namespace, and number of processes follows:

- `int cm_create_and_enter(void)` - This creates a new container, and enters into it. The current process that calls this function will go from not being part of a container (thus using global system namespaces) to being in a container, thus having potentially restricted namespaces. The namespaces for locks, cvs, and shared memory created from this point on are *local to this container.* One container can try and create a lock called "locky_mc_lockface" and it will find its lock[3]. A different container trying to find the same lock will actually create a different lock (as there are per-container namespaces). Once this process is terminated the container is destroyed. Define your own appropriate error conditions and values.
- `int cm_setroot(char *path, int path_len)` - Change the root directory for the current process to the given path (the `path` string has length `path_len`). The path has to be absolute (i.e. its first character must be `/`). Any future file system system calls (e.g.

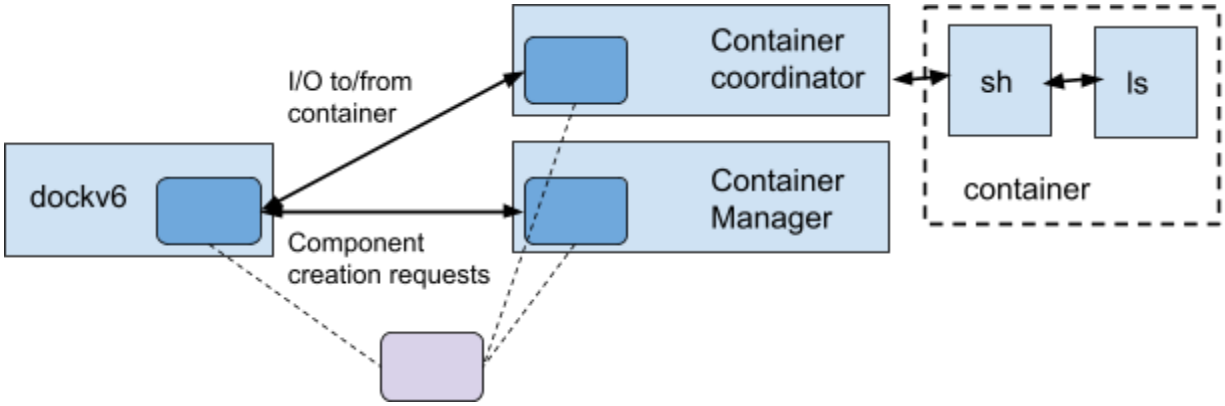
[3] You'll see later that locks and shared memory have textual names so that they can be shared between different processes that don't have parent/child relationships.

open) will progress treating that path as the root. You *must* prevent the use of "`..`" to go "under" the root. This can only be called when the current process is executing in a new container (i.e. when `cm_create_and_enter` has been called by this process, or one of its parents). Should switch the current process' current working directory to this new root. Returns an error appropriately. You should specify your own error conditions well. An example of what you'd expect with this function is shown in the command line trace above, but I'll add to it here:

```
… // continuing from above
$ cd ..
cat
ls
helloworld.txt
sh
// Notice that this behaves the same as if you tried to "cd .." in "/" in
normal // xv6: just interpret a ".." in "/" as staying in "/". Importantly,
the ".." does // not let us "escape" the container to the real "/".
```

- `int cm_maxproc(int nproc)` - This sets the maximum number of processes (including this process) that can exist in a process tree of descendents rooted at the current process. If a process has already used this function, it or its descendents, cannot use it again (if they do, it returns an error). Additionally, for simplicity, you can *assume* that the process that calls this system call will not terminate until all of their children have exited. This can only be called when the current process is executing in a new container. Returns an error code appropriately.

The communication between CM and clients will use a shared memory communication channel protected with mutexes and condition variables. The CM will provide this channel of communication, and will use it to 1. Receive the string of json that encodes the container specification, 2. Receive a stream of bytes from the client to the CM that the CM will use to pass to the stdin of the container, and 3. Send a stream of bytes from the container's stdout to the client, and 4. Send a stream of bytes from the container's stderr to the client. However, your *initial* implementation (before the shared memory is available) can use pipes to coordinate between `dockv6` and the `cm`.



This diagram demonstrates the way that I suggest you implement the system. The Container Manager forks off a new Container coordinator process for each container. It talks to the container (via file descriptors 0 and 1), and uses the shared memory to pass input/output to dockv6.

The different levels of completion for this module are:

- Level 0: Implement the logic to send to the CM the container specification. Parse and print out the configuration correctly.
- Level 1: Implement and test the system call to change the root in the fs of a process. You can test this outside of the context of the CM. Make sure to thoroughly test this including proper handling of "..", and use of the system call within a process tree that is already working in a restricted root environment.
- Level 2: Implement and test the system call for setting the maximum number of processes for a subtree of the process hierarchy. Test this completely.
- Level 3: Integrate the use of these two system calls into the CM along with the container specification to implement the resource/namespace constraining functionality of the CM. This should run the containers init program.
- Level 4: Integrate the stdin/stdout/stderr routing through the CM from the container to the client and vice-versa. This will likely require using *multiple* processes so that different processes can block waiting for input from outside the container to send to the stdin within the container, and to block waiting for input from the stdout of the container to output onto the terminal. This should be possible with multiple containers created by the same CM.

Extra credit:

- Implement your CM to use a number of threads per container instead of multiple processes.

Independent testing: To avoid needing to wait for the other modules to be completed before you can implement your CM server, you can communicate through pipes with a client process. Later, you can integrate with the shared memory, and synchronization and use that form of IPC

instead. Note that when you do that, you'll likely want to have a mutex/condition variable protecting each direction of communication (request and response).

This module requires the most code, but most of it is user-level programming. The kernel-level programming is in the new system calls. Integration of all components will involve this module. This is best for the team member who is more comfortable with xv6 user-level programming, and are willing to dive into the xv6 user APIs (e.g. reading sh.c, etc…), but is also willing to dive a little into the FS in the kernel.

Module #2: Shared Memory

Shared memory is often used for IPC between processes. It is more efficient than message passing (e.g. pipes) as you can often have fewer `memcpy` operations (e.g. zero or one vs. two). This module will add shared memory to your project. Xv6 does not support shared memory, so this will require understanding the xv6 mechanisms for mapping memory into processes, and how that interacts with calls to `sbrk` (extending the amount of memory accessible by a process), and `fork/exec/exit`. The API of system calls that you'll add to xv6 includes:

- `char *shm_get(char *name)` - Map a 4096 page into the calling process's virtual address space. For the simplest implementation, it should likely map at the address that a sbrk would normally use next. A subsequent call to `shm_get` or `sbrk` should still work (i.e. it can't overlap with the region a previous call returned). Return `NULL` if `name` is `NULL` or if the memory cannot be mapped. The `name` identifies *which* shared memory region we want to map in. If there are no shared memory regions, and a process calls `s = shm_get("ggez");`, then the system will allocate a page (e.g. with `kalloc()`), and map it into the process. If another process then makes the call with the *same name*, then it will map in the *same page*, thus creating shared memory.

  Note that this API is created to allow you to call this function *once*, get the memory mapped in, then to just use the shared memory from that point onward. Thus, the client, and the server, communicating via shared memory, should each only call this function once. This is essential to get the performance we'd like by avoiding system calls.
- `int shm_rem(char *name)` - This removes a mapping of the shared page from the calling process's virtual address space. If there are no more references to the shared page, then it is deallocated, and any subsequent mappings will return a new page. This returns -1 if the mapping with the given name doesn't exist, or if `name` is `NULL`.

Tracking the shared memory regions should be performed in two structures in the code (similar to what you've seen previously with locks).

1. You can create a global kernel structure that tracks the page that is used as the shared memory, the name associated with it (which can be looked up on `shm_get`), and a "reference count" which tracks how many processes have the shared page mapped into them. Once the reference count is zero, you can deallocate the page. You will support a maximum of `SHM_MAXNUM` shared memory pages in the system, thus your global structure will likely be an array of this tracking information. Feel free to place this in `vm.c`.
2. Each process must track which shared memory regions (by name) it has mapped inside of itself, and at which virtual address they are mapped. This is required so that you can support shm_rem, and so that when the process forks, exits, or execs, you can know which shared memory regions are getting more, or fewer references.

You will have to read through a fair amount of the code in `vm.c` and `proc.c` to understand how to implement all of this, and a good place to start is with sbrk.

The different levels of completion for this module are:

- Level 0: System calls and kernel data-structure to track `SHM_MAXNUM` shared memory regions (tracking processes, and virtual memory addresses the regions are mapped into).
- Level 1: Implement the system calls to share memory between two processes, including mapping in the actual shared memory.
- Level 2: Maintain the shared memory region across forks of the process.
- Level 3: Thoroughly test that your shared memory region is properly maintained, or deallocated, depending on which processes fork, exit, and exec. This will require reference counting on the shared memory region, and you only deallocate the memory for it when all threads accessing it `exit` or unmap it.

Extra Credit:

- Ensure that mutexes and shared memory cannot be accessed either across containers, or from the normal xv6 world and a container. Trying to get an existing name from outside of the current container will generate a **new** shared memory region (or mutex).

Independent testing: You should be able to test this implementation independently from the other modules by simply using multiple processes to create the shared memory region, and test that it is functional. You do *not* need to create specific test files, but you will be graded by giving a demo of the functionality. Please provide a *convincing* set of tests that demonstrate your levels.

This module requires investigating parts of the kernel that you're least familiar with, but is likely not much code. The team member most comfortable with the kernel as a whole should dive into this.

Module #3: Synchronization

Module 2 provides shared memory between processes. Now that we have shared memory, we must synchronize access to it! We want to provide a means for passing data to and from the CM, and for synchronizing the return value being passed to the client. Xv6 does not provide any memory sharing between user-level processes, so your job is to implement a set of mutex system calls, and to enable processes to block waiting for requests/responses, you'll also provide condition variable system calls. The prototype of these follows:

- `int mutex_create(char *name);` - return a `muxid` which is an integer identifier for a mutex (analogous to a file descriptor which is an integer that identifies a file), or return -1 if it cannot allocate the mutex.
- `void mutex_delete(int muxid);` - deallocate the mutex if we are the last process referencing (i.e. all other processes that previously `mutex_created` it have `mutex_deleted` it, or have `exited`/`execed`).
- `void mutex_lock(int muxid);` - Lock a mutex identified by the mutex id. If the mutex is not owned by another thread, then the calling thread now owns it, and has mutual exclusion until the mutex is unlocked. If the mutex is already owned, then we must block, awaiting the other process to release the mutex.
- `void mutex_unlock(int muxid);` - Unlock a mutex that was previously locked by us. If other threads have since tried to take the mutex, and are blocked awaiting entry, we should wake one of them up to go into the critical section.

- `void cv_wait(int muxid);` - The condition variable API is vastly simplified compared to the pthread version. We assume that each mutex can have a *single* condition associated with it. This works well if we just want the CM waiting on the condition that a client sends a request, and that clients will block waiting on the condition that the CM replies to them. Because of this, we need only specify the mutex id that the condition variable is associated with, and not which condition variable to use.

  As with pthread condition variables, the calling thread will release the mutex, and *block* waiting on the condition variable (unless it has already been signaled). Note that you *must* implement blocking for the mutex separate from blocking for the condition variable or you are likely to break mutual exclusion. Once the blocked thread is signaled, it will awaken, and eventually re-take the mutex (when the scheduler determines it should execute, and the mutex is available).

  Please note that you are to implement condition variables as a user-level API (i.e. a set of functions that can be called from user-level), implement their logic in the kernel, and use them in your IPC with the CM. You don't need to *use* condition variables to implement your kernel implementation of mutexes.
- `void cv_signal(int muxid);` - wake up any threads that are blocked on the condition variable associated with the mutex identified by its mutex id. This doesn't alter the

mutex at all, and only wakes up any blocked threads. You can implement this such that it wakes all threads, or just one.

This will, again, require some data-structures in the kernel:

- A global structure which is the array of mutexes that can be created in the system. These are similar to those for shared memory, but there can be `MUX_MAXNUM` number of them, maximum. I would suggest that each of these tracks the mutex name, and whatever you need to track for the blocked threads on the mutexes and the condition variable associated with the mutex.
- An array per process that is indexed by the mutex id, with each entry simply containing a reference to the global mutex. This enables different processes to have different mutex ids for the same mutex as they are looked up in per-process data-structures. It also allows you to "clean up" the mutexes if the process exits.

You can **not** use spinning techniques to provide synchronization, and must focus instead on blocking threads that are waiting for an occupied critical section, or blocking waiting for some condition to be true. You also cannot simply use `yield` threads to stop them from executing. You must block and wakeup threads (i.e. make them ineligible for scheduling).

Note that the in-kernel implementation of locks (see the `acquire` and `release` functions) uses spinlocks for cross-CPU synchronization, and simply disable interrupts during the critical section. You *can* use these to protect your own data-structures, but you *cannot* simply call these for your mutexes. This would result in user-level executing with interrupts disabled which is not safe or secure. You *must* use blocking for your mutex and condition variables.

The different levels of completion for this module are:

- Level 0: Implement the system calls to allocate and free mutexes, along with the kernel data-structures to track the mutexes.
- Level 1: Logic in the kernel to block processes locking a mutex that is already locked (i.e. that has an already occupied critical section), and to wake up a thread blocked on a mutex for the same reason. Mutual exclusion is required. This is very similar to the previous homework.
- Level 2: Implement condition variables that inter-operate properly with the mutexes from the previous level.
- Level 3: Handle the case where a process terminates accidently (faults). Release any mutexes it holds, and ensure that it is no longer blocked on conditions.
- Level 4: Implement your mutexes in a manner where if a critical section is *not* owned by another thread, and a thread is attempting to lock it, it will avoid making a system call similar to futexes as spelled out above.

Extra Credit:

- Once you have everything else working, you can level up your implementation to the max level. The highest you can level up to, is to implement a simple version of Linux

futexes (google it). The idea is that when a mutex is uncontended (no thread is in the critical section), you should be able to use simple compare and swap instructions at user-level to set a bit in a word in the shared memory denoting that the lock is now taken. Another thread attempting the same will see the bit set, will set another bit showing that the mutex is “contended”, and will instead make a system call to the kernel to block on the mutex. When the mutex owner releases the mutex (unlocks it), it will use compare and swap to unset the taken bit, and when it notices that the contended bit is set, it will make a system call to wake the threads waiting to enter the critical section. In this way, when a thread simply locks and unlocks a lock, it will do so using only a couple of compare and swap, atomic instructions and will completely avoid making system calls. Only when a thread contends a taken lock will system calls be made.

- A naive implementation of this (as spelled out here) is quite good. But there are some race conditions that you’ll need to solve to get a 100% solution. If you implement an xv6 equivalent of futexes, you can augment the mutex API (likely the create function) to include the address of the lock in the shared memory.

Independent testing: The mutex API is specially formulated to not require shared memory. Mutexes are simply associated with a name, so any process can get one, and lock/unlock it. This means that two processes that don’t even share memory can be used to test the mutual exclusion of the mutex. The same applies to the condition variables.

This module is best approached by the team member who is most comfortable with mutexes (esp. blocking). It is, in many ways, the most familiar, but includes a fair amount of kernel programming.

Module #4: Scheduling

Add fixed priority-based scheduling to xv6. This module is relatively straightforward, and is what it sounds like. The aim is to add O(1) priority-based scheduling to the kernel, based on the O(1) priority queue data-structure (remember HW1!). Priorities span from `0` to `PRIO_MAX`, where `0` is the highest priority. All processes start executing at the priority of their parent, and the init process should start at `0`. This requires a new system call:

- `int prio_set(int pid, int priority);` - This attempts to set the priority of the process identified by `pid` to `priority`. However, this will successfully set the priority and return success (`0`) only if the current process is in the ancestry of the process with process id `pid`, or if `pid` identifies the current process. In either of those cases, the current process cannot raise the priority (numerically lower numbers) higher than the current process's priority. In all other cases, return an error (`-1`).

The different levels of completion for this module are:

- Level 0: Implement the system call, and set the processes' priority.
- Level 1: Proper error checking for all edge cases for the system call.
- Level 2: Properly schedule using an O(1) scheduling queue.
- Level 3: Integration of the scheduling into the CM. The CM should be at the highest priority in the system, and all other applications should be at lower priority.
- Level 4: Extensive testing of proper scheduling. This will take some creativity, and require that you write a set of processes to show that the proper ones are running when they should.

This module is relatively simple compared to the others, thus the team member assigned this task will also act as the **Testing and Integration Lead**. This means they are responsible for helping *all* other team members generate test cases to help evaluate their code, and documenting those tests in the Report described below. They also should act as a central source of information to help coordinate the other modules and integrate them.

Overall Project

The levels for the overall project are:

- Level 0: Level 1 in four modules.
- Level 1: Level 2 in four modules.
- Level 2: Level 3 in four modules. Two of the modules must be integrated together. You must have tests to evaluate this integration.
- Level 3: All four modules must be integrated together. You must have tests to evaluate this integration.
- Level 4: Highest level minus 1 in two modules, and highest level in two.
- Level *Super Saiyan*: Highest level in all modules.

Group work failure cases: If one teammate does not pull their weight, then you should continue leveling up all other modules as far as they can go. Once a teammate falls behind, it is on *them* to catch up. First, I hope that you've notified me through the team check-ins (below). Second, you should come up with ways to proceed with the project *minus* that teammate's contributions. That means "simulating" their contribution. If the teammate doing locks and condition variables drops the ball, you should be able to proceed using pipes for communication. Same with shared memory. If the CM teammate drops the ball, you can proceed by creating a set of processes that *act* like the CM (including all communication, input/output, etc…) that use the other features. I hope the point is clear: if a teammate (or multiple teammates) fail, the project will proceed *without* them. Grades will be assigned individually in such cases.

Code Organization, Tests, and Submission

You must fill out the form that tells your professor who is in your group, and what their github names are. You should use the github classroom link to create your team's repo. Your submission will simply be the code in your repo at the time of the deadline. This is the code that you'll demo. Please don't check in binary files (executables, .o files, etc…) to your repo. At the demo, you'll be expected to say which level you made it to in each of the four modules, and to adequately demo that functionality. Have a document that simply spells out your levels. "Almost" doesn't count, and I expect tests that validate the levels that you claim. Your demo (thus your submitted code) must be "clean" in that the output is clear, absent of debugging output, and functional. You must create a set of programs that concretely show off your submission, and you're expected to convince me that you implemented your stated levels in each module in **3 minutes each**. This will require organization and preparation, so please spend the necessary time to plan your demo. I will cut you off at 5 minutes, and if you haven't convinced me that you have achieved a given level, you won't get credit. I suggest focusing on your test cases.

Your final code should be "clean". This means no print statements for debugging, no chunks of commented out code, proper indentation, and an effort to simplify the structure of your code and remove all redundancy. Your grade will be decreased if you don't produce clean code.

You **must** test your code thoroughly. I've had the following exchange many times. Student: "We're mostly at level 3". Me: "What does 'mostly' mean?" Student: "The code is there, and seems to work." Me: "No. Untested code doesn't work." There is a deep misunderstanding here about how you must test your own code. The first exchange shows a lack of understanding about what it takes to get code working. In systems, you often spend 6x more time debugging than coding. Without thorough testing, you *should have very little confidence that anything works*. Generally, "untested code" might as well be "no code". You get **zero** credit for untested code.

Another common exchange follows. Me: "How did you test that you achieved level x?" Student: "We integrated everything together, and it works." This exchange demonstrates a lack of understanding of how complicated system execution is. You cannot expect that a single usage pattern of mutexes and condition variables tests their implementation sufficiently. You *must* write tests specifically to evaluate all of the different conditions (contention, no-contention, etc…). I expect to see tests to specifically evaluate the different paths of your implementation.

Extra Credit: You will get extra credit for doing individual features and bug fixes as (many) separate commits, and for doing pull requests and code-reviews. You can read more about how this is done via:

- https://www.atlassian.com/git/tutorials/comparing-workflows#feature-branch-workflow

- http://www2.seas.gwu.edu/~gparmer/resources/2017-08-07-Trello-And-Github.html (see the “Github” section)

If you did thorough code reviews, and properly-sized commits, please (quickly) demonstrate as such during your demo. **Note:** a penalty will be charged if your repository does not accurately reflect which students did which work (i.e., be sure git is configured to know your name and make sure that you are creating multiple commits).

Final Report

You must include a `REPORT.md` file in your repo. First, it must include a section that lists the levels you achieved in each module, and in the overall project (fill in the `[X]` boxes). Second, fill out a list of work attribution. For each level, for each module/project, please add a prioritized list of team members who contributed to having leveled that up. This should match up with the commits of the `WORKPLAN.md` Third, it must include documentation on how you evaluated each level to convince me that you thoroughly tested your implementation. This might include pointing me to different programs you wrote, or different functions you wrote to do the tests.

Deadline Milestones

The Deadlines for each of these is available in the class resources:

- **Group Selection.** Please fill out the provided form with your group name and members.
- **Group Responsibility Specification.** Add a file `WORKASSIGN.md` into your repo. Please fill out the provided form spelling out which team member is responsible for which modules. I expect the majority of the check-ins for that module to be made by the corresponding team member. If you work “together”, then appropriately switch off who commits.
- **A Work Plan.** You must give a **detailed** specification (e.g. including links to code where modifications are necessary) of the timeline for each module in a `WORKPLAN.md` file. Each module should have a bulleted list with at least one entry per week. Each entry must include concrete features that are to be completed. “Planning” and “looking at” are not valid actions, only code that will result in commits.

  The team member assigned a module is responsible for their module’s timeline. Integration and testing must be assigned 2 weeks in the schedule at the end, and the timeline must be agreed to by all team members. You can also (and should) do integration work before the last two weeks.

  Every week, you **must** add a link to the corresponding commit that completes any of the work items.

  As you get further in the project, you’ll likely realize that you forgot to list some work items. You must update the work plan to include these as you discover them.

- **Code Deadline.** Final commit due. Please fill out the `REPORT.md` as described above.
- **Demo.** You will have around 15 minutes to walk me through what works in your implementation, and what doesn't. If you do it right, that will just be a quick walk-through of the tests for your implementation. You **must** practice this, and use your time wisely. Using this time effectively will have a huge impact on your grade. The details about the demo will be shared later.

Academic Honesty

You are allowed to collaborate freely within your group. In fact you must. The professor will set up a group repository that you can use. That repository *must* not be visible to people outside of your group, and you are *not* allowed to collaborate with others outside of your group. You can discuss the assignment specification, and you can discuss general OS topics, but you cannot discuss your implementation or code.

Even if you receive help from me or the TA, you **must be able to explain your code.** If you don't understand what your own code is doing, and can't explain it, then *you don't get credit for it*.

IMPORTANT: The above policy applies even after you complete this course – you **may never** post your solution to this project publicly! If you need a way to share a repo with a potential employer, check out tools like GitFront. In most cases though, a good description on your resume is enough!