In this assignment, we are digging into the first weeds of the operating system structure! In particular, we are going to spend our time building the main backbone of the operating system: the processes and threads subsystem. As a result of completing this assignment, you will have a system that can launch and switch between any statically provided "user mode" program via its main function.

This document is organized into two parts. The first part overviews the holistic system, and the second part gets into the implementation details that you will need to successfully complete this assignment.

Collaboration Policy

This assignment is intended to be implemented as a *pair-programming* assignment! That means that you and your partner are working on the assignment **together**.

After the initial programming phase of the assignment, you and your partner will participate in a *code review* of another group's submission. Be sure to review the code reviewing standards from Homework 0 submit your code review on Gradescope!

Submission

You are asked to submit three deliverables for this assignment: ① the programming component, ② your code review for your partnered group, and ③ a reflection to the review of your code. In addition, in-class participation is required on **September 30th** to both give and receive feedback for full credit.

Getting Started

This assignment is designed to run on the course virtual machine (e.g., `ssh stum2025@itbdcv-lnx04p.campus.pomona.edu`). The stencil for assignments in this class assume particular compiler versions and processor features, so using the VM will make your life easier.

Download the source for this assignment to get started! You can find all of the tasks that you will need to implement by running "`grep "TODO: implement me!" -r .`" from the "shell-stencil" repository. In the repository, you will find a `Makefile` to help you build the various sources for each part of this assignment.

Deadlines

Section	Deadline	Submission URL
Programming Component	Sept 28 (EOD)	HW2 Programming
Code Review	Sept 30 (before class)	HW2 Code Review
Code Review Reflection	Oct 2 (EOD)	HW2 Code Review Reflection
Post-Review Resubmission	Oct 2 (EOD)	HW2 Programming (late due date)

1 Before You Get Started

This assignment has a lot of different components associated with it. Your job will be much easier with a solid understanding of the system to be implemented. As such, **I highly recommend that you go through this document before you start programming**. Make sure that you and your partner(s) feel comfortable with the relationship between different components.

1.1 Helper "Classes"

Key to this assignment will be to use pre-packaged data structures that come with your "OS kernel." Much like the `string` library that you implemented as part of your shell, the processes subsystem uses several libraries that must be natively implemented with the operating system kernel (the OS is the one who implements the dynamic linking mechanism!). In this assignment, you are exposed to two additional libraries `util/list.{h,c}` and `util/spinlock.{h,c}` that are provided for you. These libraries are largely based on the prebuilt definitions used in Linux[1].

To provide a reference as to the interface for each of these libraries, you may find the following examples useful to you in your development of the processes subsystem.

```
// construct the list
list_init(&proc_list);

// add element to list
proc_t example_proc;
list_link_init(&example_proc.p_list_link);
list_insert(&proc_list, example_proc.p_list_link);

// remove elements from the proc_list by unlinking them individually
for (list_link_t *link = list_remove_front(&proc_list);
        link != NULL; link = list_remove_front(&proc_list)) {
    // if we want to access the parent
    proc_t *parent = (proc_t *) link->parent;

    // do stuff!
}
```

```
// construct the lock
spinlock_init(&proc_list_lock);

// acquire lock
spinlock_lock(&proc_list_lock);

// do synchronized stuff!

// release lock
spinlock_unlock(&proc_list_lock);
```

[1] As an example, see `https://github.com/torvalds/linux/blob/master/include/linux/list.h`

2 CS134 Processes Overview

In this assignment, we are working from the process struct defined in `include/proc.h`. This struct is loosely based on an early version of Linux (referred to as System V). It is replicated below for your convenience.

```
/* Process descriptor */
typedef struct proc
{
    pid_t p_pid;                 // Process ID
    char p_name[MAX_STRING_LEN]; // Process name

    list_t p_threads;            // Threads list
    spinlock_t p_threads_lock;   // lock for threads list
    list_t p_children;           // Children list
    spinlock_t p_children_lock;  // lock for children list
    struct proc *p_pproc;        // Parent process

    list_link_t p_list_link;     // Link of list of all processes
    list_link_t p_child_link;    // Link on parent's list of children

    long p_status;               // Exit status
    proc_state_t p_state;        // Process state
} proc_t;
```

2.1 Within a Process

Threads Recall that, for any running process, there must be one or more threads associated with it to implement the execution. Because the "process" is a structure of the operating system, any operating system maintenance of the threads (recall: referee, illusionist, glue) must be implemented with some way of it is the job of the process to maintain the execution of its running threads. This means that, in the process structure itself, there needs to be a way to track these threads in a way that is accessible to the OS kernel.

In your implementation, you may choose to solely deploy single-threaded instances of your OS kernel. However, if it was the case that your OS could run on separate cores concurrently, it would be possible for the same process to be running multiple threads on separate cores concurrently. If this were to happen, it would be entirely possible for each thread to both spawn new threads (e.g., `pthread_create`) at the same time. To avoid a data race, the process struct also maintains a spinlock (implemented in `util/spinlock.{h,c}`) for these threads to synchronize. Recall, the OS cannot import any libraries, so this library needs to be implemented in the OS kernel itself.

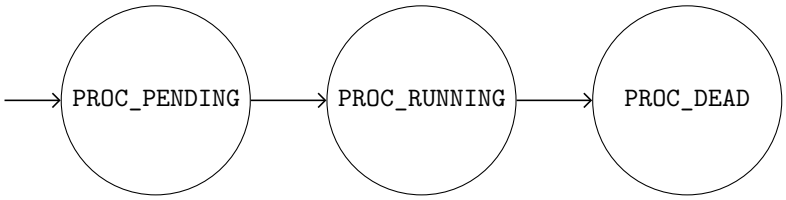
Child Processes Much like the threads of a process, any process and/or thread may choose to make a call to `fork` which will spawn a new process. As such, our process struct needs to maintain a reference to each of these newly created processes for when they finish executing. This is generally done in the `p_children` field of the proc struct. Also like with the threads, the child processes may synchronize with a spinlock particularly designated for modifying the `p_children` list.

Seeing as any process[2] will need be a child of some other process, the process struct needs to have some way of allocating an entry in the parent process' `p_children` list. It does so using the the `p_child_link` attribute, which should be linked into the parent's `p_children` list.

[2] Aside from the `idleproc`.

Parent Processes From this process structure, we can learn several key features that every process should have in our system. For one, every process struct has a `p_pproc` attribute. This means that processes are naturally attached to a parent process by virtue of maintaining a reference to this process. Recall from CS105 that the parent process of any process is the process that created it. This means that the `p_pproc` field for all processes (see footnote 1) should be the process that creates it.

Process State In this processes subsystem, processes may be in one of three states: `PROC_PENDING`, `PROC_RUNNING`, or `PROC_DEAD`. In general, the state machine for a process lifetime is relatively simple in this system. It follows the state machine depicted below:



What can we learn about process lifetimes from this state machine? Unlike more complex systems, we can expect that, while processes may be created at any point, the only possible transition that can occur from `PROC_RUNNING` is to the `PROC_DEAD` state. This implies that processes, once running, **cannot be preempted** (for simplicity of design). They will run to their completion before the processes subsystem starts executing another process.

2.2 The Relationship Between Processes and `kthreads`

As alluded to above, processes are running instances of programs. However, processes themselves are not the "executable primitives." This is instead implemented by "kernel threads" or `kthreads`. A `kthread` is an object associated with a process that maintains the stack, return value, errno, and context associated with a thread. That is, when a process "starts running", the execution that starts running is the context associated with the kthread associated mapped to the main thread of the process using the stack, errno, and return values in the `kthread` struct.

2.3 The Relationship Between `kthreads` and Contexts

A context describes the state of all of the registers associated with a currently running thread. Every `kthread` has an associated context, which has an associated struct in which the state (e.g., registers, stack size, etc...) can be stored. In order to "make a context active" or "perform a context switch", the OS kernel needs to take the registers, stack size, etc... associated with that context (and, in turn, that thread) and load it into the actual hardware registers for execution. Switching out a context implies taking the values from the hardware registers and saving them into the associated fields of the struct to remain accessible for later execution.

2.4 The Processes Subsystem

Outside of the definition of what the `proc_t` struct looks like, the processes subsystem of the operating system needs to have some mechanism to track and maintain its processes to implement the various interfaces into this subsystem. For instance, system calls like `fork` will need to create processes and `waitpid` will need to detect that a process' execution has terminated.

To implement these functions, the attributes of the subsystem are described as such:

```c
// from include/proc.h
/*******************************
 * Process Subsystem Attributes *
 *******************************/

// global curproc variable to track the current process set in globals.h

// maintain the active processes
list_t proc_list;
pid_t next_pid;
spinlock_t proc_list_lock;

// idleproc is stored globally to the process subsystem
// rather than in the process list as it exists per core
proc_t idleproc;

// pointer to the init process
proc_t *proc_initproc;

// allocator for process descriptors
slab_allocator_t proc_allocator;

// the currently running process (declared globally in src/proc.c)
proc_t *curproc;

// the currently running thread (declared globally in src/kthread.c)
kthread_t *curthr;

// the "context" for BIOS/simulator main function (defined in include/context.h) for
    clean exit
context_t bios_ctx;
```

Some key things to note about these declarations:

- All active processes in the system are maintained in the `proc_list`;
- When creating a new process, it must be assigned a *Process ID* (PID) that is unique to the system. This value is a monotonically increasing;
- Recall that, to boot the system, the bootloader is not running in a "process" per se. However, seeing as all processes need a parent process, the `idleproc` serves as a default parent for the `proc_initproc`. The `idleproc` should never finish as the system should shutdown at the completion of the `proc_initproc`;
- To create new objects in the processes subsystem (and all other subsystems), we need to reserve space from the memory device (managed by the "memory management subsystem" — not yet implemented!) from the `proc_allocator` object.

3 Your (Programming) Task

In general, your job is to implement all functions with the signature `TODO: implement me!` (you can find them by running `grep "TODO: implement me!" -r .`). There are 14 such functions for you to implement in this assignment:

- `proc_init` (src/proc.c),
- `proc_idleproc_init` (src/proc.c),
- `proc_create` (src/proc.c),
- `proc_destroy` (src/proc.c),
- `proc_cleanup` (src/proc.c),
- `proc_thread_exiting` (src/proc.c),
- `proc_kill` (src/proc.c),
- `proc_kill_all` (src/proc.c),
- `kthread_init` (src/kthread.c),
- `kthread_create` (src/kthread.c),
- `kthread_clone` (src/kthread.c),
- `kthread_destroy` (src/kthread.c),
- `kthread_cancel` (src/kthread.c),
- `kthread_exit` (src/kthread.c)

The general structure of the assignment is that you can find the formal definition of how these functions should be defined in the header file (found at `include/{proc.h, kthread.h}`) with implementation hints to be found in the associated source. As with any assignment, you can always get further hints in office hours if you are stumped or any component is unclear.

4 Tests and Code Review Notes

When performing your code review, be sure to examine the `test` directory to find the starter for two tests.

4.1 Boot

The first test, called `boot_test.c`, emulates the process of booting a device and launching into the processes subsystem. This first creates a bootloader context that executes the `idleproc` and sets the `bios_ctx` to the main function of this test. A functional test results in a clean exit from this program after context switching into the bootloader and from the bootloader into the initproc.

Your task. During the code review portion of this assignment, your task is to add various sanity checks to this main function during/after the main functional body executes. Be sure to add various checks to confirm that the state of the processes subsystem is what you would expect at each of these points.

4.2 Fork

The second test, called `fork_test.c`, is intended to model the behavior of the `fork` system call. It is currently unimplemented.

Your task. To run this test, you must implement the intended behavior of both `fork` and `wait` by implementing the `initproc_run` function. Rather than merely returning `NULL`, this function should instead create a new process that executes the `childproc_run` function.

4.3 Additional Test

You and your partner should come up with one more test concerning the processes subsystem! The test may choose to implement a "boot-like" procedure or something else. You are encouraged to express how this additional test exercises some other component of the functionality of the subsystem. You may find it useful to consider the system calls that invoke or relate to processes.

5 (Optional) Extensions

5.1 Multithreaded User Programs

Thus far, all tests have run a single `kthread` per process. Consider what it would look like to write a multithreaded program. What would need to be extended from the current process structs? What additional bugs could arise?

5.2 Implementing Preemption

Suppose we wanted to implement a more robust scheduling system for the processes subsystem. What additional states would be needed for the processes? What about the threads? If we implemented preemption, this would allow for a more complex scheduling system!