Project Threads Design

Group 48

Name	Autograder Login	Email
Ben Plate	student219	bplate@berkeley.edu
Jeffrey Liang	student50	jeffreyliang@berkeley.edu
Teddy Cho	student139	teddycho@berkeley.edu
Theophilus Pedapolu	student270	theopedapolu@berkeley.edu

Efficient Alarm Clock

Data Structures and Functions

We add `static struct list sleeping_threads` to `devices/timer.c` which stores instances of `struct sleeping_thread`, defined as follows.

```
struct sleeping_thread {
  uint64_t wake_ticks;
  struct semaphore wake_wait;
  struct list_elem elem;
};
```

We modify the following functions.

- `void timer_init(void)` in `devices/timer.c`
- `void timer_sleep(int64_t ticks)` in `devices/timer.c`
- `static intr_handler_func timer_interrupt` in `devices/timer.c`

We add the following function.

- `static list_less_func sleeping_thread_less` in `devices/timer.c`, a comparison function for the `sleeping_threads` list

Algorithms

We modify `timer_sleep(int64_t ticks)` to remove the current busy-waiting implementation.

- Disable interrupts using `intr_disable`, storing the old `intr_level` as `old_level`.
- Create a new `sleeping_thread` struct with `wake_ticks` set to `timer_ticks() + ticks` and `wake_wait` initialized to 0.
- Add the new struct to the `sleeping_threads` list using `list_insert_ordered` with the newly-added function `sleeping_threads`, defined by the natural ordering of `wake_ticks`.
- Call `sema_down` on `wake_wait` (`sema_down` may be called with interrupts disabled).
- Restore the old `intr_level` with `intr_set_level(old_level)`.

We make the following additions to `timer_interrupt(struct intr_frame args)`.

- Iterate through all the `sleeping_thread` structs in `sleeping_threads` and compare the value of `wake_ticks` to `timer_ticks()`.
  - If `wake_ticks` is less than or equal to `timer_ticks()`, remove the `sleeping_thread` from the list, call `sema_up` on `wake_ticks`, free the `sleeping_thread`, and continue iterating.
  - If `wake_ticks` is greater than `timer_ticks()`, break from the loop. We can do this because the list is sorted and we will not find any more threads that should be put back on the ready queue.

Synchronization

The `sleeping_threads` list is modified by both the `timer_interrupt` function and the `timer_sleep` function. Interrupts are already disabled in `timer_interrupt`, and we disable interrupts in `timer_sleep` to prevent any race conditions on `sleeping_threads`.

Rationale

Instead of busy waiting in `timer_sleep`, we keep track of the threads that should be put on the ready queue at a given tick. To do this, we use the `sleeping_thread` data structure in a list that we check at every timer interrupt, when `ticks` is incremented. Because we insert into the `sleeping_threads` list in order by `wake_ticks`, `timer_sleep` will have $O(n)$ runtime, where n is the number of sleeping threads. However, this is still significantly more efficient than busy-waiting.

Strict Priority Scheduler

Data Structures and Functions

We add `static struct list prio_ready_lists[PRI_MAX - PRI_MIN + 1]` to `threads/thread.c`. This is an array of lists, indexed by priority, containing the threads in `THREAD_READY` state for each effective priority.

We add the following fields to the TCB.

```
struct thread {
  /* Existing fields omitted */
  int effective_priority;    /* Effective priority of this thread */
  struct list locks_held;    /* List of locks held by this thread */
  struct lock* lock_waiting; /* Pointer the lock this thready may be waiting on */
};
```

To support the list `locks_held`, we add a `struct list_elem` to `struct lock`.

```
struct lock {
  /* Existing fields omitted */
  struct list_elem elem; /* List element for TCB's locks_held list */
};
```

To support ordering by priority for monitors, we add a pointer to a TCB to `struct semaphore_elem` in `threads/synch.c`.

```
struct semaphore_elem {
  /* Existing fields omitted */
  struct thread* thread; /* Pointer to the thread awoken by up'ing this semaphore */
```

```
};
```

We modify the following functions.

- `void thread_init(void)` in `threads/thread.c`
- `static void init_thread(struct thread*, const char* name, int priority)` in `threads/thread.c`
- `static struct thread* thread_schedule_prio(void)` in `threads/thread.c`
- `static void thread_enqueue(struct thread*)` in `threads/thread.c`
- `int thread_get_priority(void)` in `threads/thread.c`
- `void thread_set_priority(int)` in `threads/thread.c`
- `void sema_up(struct semaphore*)` in `threads/synch.c`
- `void lock_acquire(struct lock*)` in `threads/synch.c`
- `void lock_release(struct lock*)` in `threads/synch.c`
- `void cond_wait(struct condition*, struct lock*)` in `threads/synch.c`
- `void cond_signal(struct condition*, struct lock*)` in `threads/synch.c`

We add the following functions.

- `void update_effective_priority(void)` in `threads/thread.c` recomputes the effective priority of the current thread and yields the CPU if the thread no longer has the highest priority.
- `list_less_func thread_priority_greater` in `threads/thread.h` is a comparison function for `struct thread`'s that will sort them by effective priority in descending order.
- `static void donate_priority(void)` in `threads/synch.c` recursively donates priority to any thread blocking the current thread.
- `static list_less_func semaphore_elem_greater` in `threads/synch.c` is a comparison function for `struct semaphore_elem`'s that will sort them by the effective priority of their corresponding threads in descending order.

Algorithms

`thread_schedule_prio` should return the highest priority thread in `THREAD_READY` state, with ties broken by FIFO.

- Iterate through the array `prio_ready_list` in reverse order from index `PRI_MAX - PRI_MIN` to index `0`.
  - For the first nonempty list, pop its first element and return a pointer to the corresponding thread.

- If all lists are empty, return `idle_thread`.

For `thread_enqueue`, if `active_sched_policy` is `SCHED_PRIO`, we append the given thread to the end of the run queue corresponding to its `effective_priority`.

- Assert that `t->effective_priority >= PRI_MIN` and `t->effective_priority <= PRI_MAX`.
- Append the given thread to the end of `prio_ready_lists[t->effective_priority - PRI_MIN]`

`thread_get_priority` should be modified to return `effective_priority` instead of (base) `priority`.

`thread_set_priority` should be updated to yield the CPU if the priority is lowered such that the running thread no longer has the highest priority.

- In addition to setting `thread_current()->priority` to `new_priority`, call the helper function `update_effective_priority` (described below).

We add the helper function `static void update_effective_priority(void)`. This function recalculates the current thread's effective priority using the effective priorities of threads that it's blocking. If the thread no longer has the highest priority after updating its effective priority, it yields the CPU.

- Disable interrupts, saving the old interrupt level.
- Save the existing value of `thread_current()->effective_priority` as `old_effective_priority`.
- Iterate through the list `thread_current()->locks_held`
    - Get the effective priority of the highest priority thread blocked by `lock`. To do this, we call `list_max` on `lock->semaphore.waiters` with `thread_priority_greater`.
- Set `thread_current()->effective_priority` to the maximum of `thread_current()->priority` and the highest effective priority found among the threads blocked by the current thread.
- Iterate through `prio_ready_lists` from index `old_effective_priority - PRI_MIN` to index `t->effective_priority + 1`.
    - If one of these ready lists is nonempty, there exists a thread in state `THREAD_READY` with higher priority, so we yield the CPU by calling `thread_yield`.

- Restore the old interrupt level.

`sema_up` should be modified to unblock the highest priority thread in its `waiters` list. Specifically, the call to `list_pop_front` should be replaced with a call to `list_max` with `thread_priority_greater` and a call to `list_remove`.

`lock_acquire` should be modified to perform priority donation in the event that a thread with priority lower than that of the current thread holds the lock.

- Disable interrupts, saving the old interrupt level.
- Before the call to `sema_down`
  - Set `thread_current()->lock_waiting` to `lock`.
  - Call the helper function `donate_priority` (described below).
- After the call to `sema_down`
  - Set `thread_current()->lock_waiting` to `NULL`.
  - Append `lock` to `thread_current()->locks_held` using `list_push_back`.
- Restore the old interrupt level.

We add the helper function `static void donate_priority(void)` to perform priority donation.

- Disable interrupts, saving the old interrupt level.
- Define the variable `struct thread* receiver` to be `t->lock_waiting->holder`.
- Loop while `receiver` is not `NULL` and `receiver->effective_priority < t->effective_priority`.
  - Set `receiver->effective_priority` to `t->effective_priority`.
  - Set `receiver` to `receiver->lock_waiting->holder`.
- Restore the old interrupt level.

`lock_release` should be modified to undo priority donation and yield the CPU if the current thread no longer has the highest effective priority.

- Disable interrupts, saving the old interrupt level.
- After the call to `sema_up`
  - Remove `lock` from `thread_current()->locks_held` using `list_remove`
  - Call `update_effective_priority`.
- Restore the old interrupt level.

`cond_wait` should be updated to store a pointer to the current thread in `struct semaphore_elem`.

`cond_signal` should signal the highest priority waiting thread. Specifically, the call to `list_pop_front` should be replaced with a call to `list_max` with `semaphore_elem_greater` and a call to `list_remove`.

Synchronization

We consider synchronization for the following shared resources.

- `prio_ready_lists` is accessed only while interrupts are disabled (in `thread_enqueue` and `thread_schedule_prio`). No additional strategy is required.
- `struct thread`
    - `priority` is accessed only by the current thread (in `thread_set_priority`), so no synchronization strategy is required.
    - `effective_priority` is accessed by `thread_enqueue` and `thread_get_priority` and modified by `update_effective_priority` and `donate_priority`. Interrupts are disabled in `thread_enqueue`, so we do likewise in `update_effective_priority` and `donate_priority`. It's not necessary to disable interrupts in `thread_get_priority` because it simply returns the current value.
    - `locks_held` is accessed by `update_effective_priority` and modified by `lock_acquire` and `lock_release`. These functions can be called only on the current thread, to which `locks_held` belongs, so no synchronization is necessary.
    - `lock_waiting` is modified in `lock_acquire` and accessed in `donate_priority`. We disable interrupts in both.
- `struct semaphore`
    - `waiters` is accessed in `update_effective_priority` and modified in `sema_up`. Interrupts are disabled and reenabled in `sema_up`, so we do likewise in `update_effective_priority`.
- `struct lock`
    - `holder` is accessed in `donate_priority` and modified in `lock_acquire` and `lock_release`. We can't use a lock in our implementation of a lock, so we disable and reenable interrupts in `lock_acquire` and `lock_release`, and likewise in `donate_priority`.
- `struct condition`
    - `waiters` is modified by `cond_wait` and `cond_signal`. The monitor's lock is held during this process, so no additional synchronization strategy is required.

Although disabling interrupts in `donate_priority` and `update_effective_priority` is a coarse synchronization strategy, both functions access resources that are protected by disabling interrupts because these resources are used in the implementation of locks and semaphores. Furthermore, if a thread running `update_effective_priority` is interrupted, during which a higher priority thread becomes blocked by a lock held by the original thread, the new effective priority computed may be incorrect.

Rationale

Because the number of priorities is held constant (between `PRI_MIN` and `PRI_MAX` inclusive) and relatively small, we felt that the best way to keep track of threads and their priorities in the scheduler is using `PRI_MAX - PRI_MIN + 1` queues, one for each priority. This means that adding and removing threads to and from the scheduler's ready queue is an $O(1)$ operation for any priority. For updating a thread's priority, we have to find it in the list corresponding to its current priority, remove it, and push it the end of the list corresponding to its new priority. The latter operations are $O(1)$, but the first is $O(n)$, meaning that updating a thread's priority overall is $O(n)$ with respect to the number of threads that share its current priority. Finally, using a queue as our data structure inherently implements the FCFS ordering we desire for threads sharing a priority.

The use of the `locks_held` list and `lock_waiting` pointer in the TCB gives rise to a tree structure with the root being an active thread and the descendants being the locks (and by extension waiting threads). This is because although a thread can have multiple locks with other threads waiting on those locks, a thread can only wait on a single lock held by a single thread. This lets us implement priority donation by traversing up the tree through the `lock_waiting` and `lock_waiting->holder`, setting the effective priority as we go if the donator's priority is greater than the effective priority as we traverse. It also means that if we reach a thread with a greater effective priority than the donator's priority, we can stop the traversal as every ancestor thread will necessarily have a priority greater than or equal to the current thread. When a thread releases a resource, we simply recalculate the effective priority through the locks it still owns and its base priority. By exploiting the tree structure that our design naturally gives rise to, we are able to efficiently manage locks held by threads and prevent priority inversion.

User Threads

Data Structures and Functions

Create the following struct in `process.c`:

```
struct pthread_args {
  stub_fun sf;
  pthread_fun tf;
  void *arg;
  struct semaphore create_wait; // Used to wait for thread cre
ation in pthread_create
  bool success;
}
```

Create the following struct in `process.h`:

```
struct user_thread {
  tid_t tid;              // Same as tid of corresponding kern
el thread
  uint8_t* stack;         // Thread's userspace stack
  struct list_elem elem;
  struct semaphore join_wait;  // Used to block joined threads
  bool exited;  // True if this user thread has exited
  bool waited;  // True if this user thread has been joined on
}
```

Add the following fields to the PCB:

```
struct process {
  /* Owned by process.c. */
  ...
  struct list all_locks;     // Process-level list of user_loc
ks
  struct list all_semaphores; // Process-level list of semapho
res
```

```
  struct list user_threads;   // List of all user threads crea
ted under this process
  struct lock user_list_lock;
  struct lock locks_list_lock;
  struct lock semaphores_list_lock;
  struct lock join_lock;          // Lock to call pthread_join
};
```

Create the following struct in `process.h` :

```
struct user_lock {
  struct lock kernel_lock;
  struct list_elem elem;
  lock_t lockID;  // Identifier for the lock
}
```

Create the following struct in `process.h`

```
struct user_semaphore {
  struct semaphore kernel_semaphore;
  struct list_elem elem;
  sema_t semaID;  // Identifier for the semaphore
}
```

Algorithms

For reference, we use `[]` to represent the name of the data structure that was created, i.e. in `struct structure something [name]` name refers to the struct `something`

sys_pthread_create

Modify the signature of `setup_thread` in `process.c` to `setup_thread(struct *pthread_args args, void (**eip)(void), void** esp)`

And do the following in the function:

- Call `process_activate()`

- Set `*eip` to `args->sf`, the stub function
- Set `*esp` to a malloced block of `PGSIZE` bytes. Return false if malloc fails
- Push the pointers `args->arg` and `args->tf` in that order onto the stack pointed to by `esp`
- Push a "fake" null return address onto the stack
- Return true if the above has been completed

Modify `pthread_execute` to do the following:

- Malloc a new `pthread_args` struct called `arguments` setting the parameters `sf, tf, args` to the fields in this struct. Initialize the semaphore `create_wait` to 0
- Call `thread_create("stub", PRI_DEFAULT, start_pthread, arugments)`
- Call `sema_down(&arugments->create_wait)`
- If `&arguments->success` is false, free `arguments` and return `TID_ERROR`. Otherwise, free `arguments` and return the tid returned by `thread_create`

Modify `start_pthread` to do the following:

- Unpack the `void *exec_` parameter into a `struct *pthread_args` `[args]`
- Malloc a new `struct user_thread` with fields `tid` initialized to the `thread_current()->tid`, `join_wait` initialized to 0, `waited` initialized to false, and `exited` initialized to false.
- If malloc fails, set `args->success` to false and return
- Use `list_push_back` to add the new user_thread struct to the current process's `user_threads` list, acquiring and releasing the current PCB's `user_list_lock` as necessary.
- Create an `intr_frame` [`if_`] and set its flags similar to how they are set in `start_process`. Then call `setup_thread(args, &if_.eip, &if_.esp)`. If `setup_thread` returns false, set `args->success` to false and return, otherwise set `args->success` to true, set the previously created `struct user_thread` 's `stack` field to `if_.esp` and call `sema_up(args->create_wait)`
- As in `start_process`, use `asm volatile` to change the `esp` to `&if_` and jump to `intr_exit` to simulate a return from an interrupt

In the `syscall_pt_create_handler` in `syscall.c`, validate the arguments (check they are in user memory and there are 3 pointers in `args`) then call `pthread_execute(args[0], args[1], args[2])` and place the returned `tid_t` in `*eax`

sys_pthread_exit
Get the current thread * `t` via `thread_current()`. Iterate through the `user_threads` list in `t->pcb` and find the corresponding `*user_thread` struct `[ut]` so that `ut->tid == t->tid`

We define an active user thread as a thread with `exited == false` in its user_thread struct.
If the current thread is the main thread, i.e. `is_main_thread(t, t->pcb)` is true, call `pthread_exit_main()` in `process.c` which will do the following:

- Set `ut->exited` to true
- Call `sema_up(ut->join_wait)`
- For every active user_thread `user` in `t->pcb->user_threads` except the main_thread, call `sys_pthread_join(user->tid)`
- Free `ut->stack`
- Call `process_exit(0)`

If the current thread is not the main thread, call `pthread_exit()` in `process.c` which will do the following:

- Set `ut->exited` to true
- Call `sema_up(ut->join_wait)`
- Free `ut->stack`
- Call `thread_exit()`

sys_pthread_join
Acquire `join_lock` in the current PCB
Iterate through the `user_threads` list in the current process, i.e. `thread_current()->pcb->user_threads` and find the `*user_thread` struct `[ut]` with `ut->tid == tid`, the user_thread we want to join on (`tid` is the parameter passed in to sys_pthread_join)

If no `ut` could be found or `ut->waited == true`

- Return `TID_ERROR`

If `ut->exited == true`

- Return `ut->tid`

Otherwise:

- Set `ut->waited` to true
- Call `sema_down(ut->join_wait)`
- Return `tid`

Release `join_lock` before each return

lock_init

Malloc a new `struct *user_lock` `[u_lock]` and set `u_lock->lockID` = `*lock`, the parameter passed in. Return false if malloc fails. Otherwise, call the kernel-level function `lock_init(&(u_lock->kernel_lock))` and use `list_push_back` to add `u_lock` to the `all_locks` list in the current PCB, acquiring and releasing the `locks_list_lock` as necessary. Return true once the above is completed.

lock_acquire

Iterate through the current PCB's `all_locks` list until a `user_lock` with `lockID` == `*lock` (the parameter passed in) is found

- If no such `user_lock` is found, return false
- If a `user_lock` is found (call it `u_lock`) and `u_lock->kernel_lock->holder == thread_current()`, i.e. the current thread already holds the lock, return false
- Otherwise, call `lock_acquire(&(u_lock->kernel_lock))` and return true

lock_release

Iterate through the current PCB's `all_locks` list until a `user_lock` with `lockID` == `*lock` is found

- If no such `user_lock` is found, return false
- If a `user_lock` is found (call it `u_lock`) and `u_lock->kernel_lock->holder != thread_current()`, i.e. the current thread does not hold the lock, return false
- Otherwise, call `lock_release(&(u_lock->kernel_lock))` and return true

sema_init

Malloc a new `struct *user_semaphore` `[u_sem]` and set `u_sem->semaID` = `*sema`, the parameter passed in. Return false if malloc fails. Otherwise, call the kernel-level function `sema_init(&(u_sem->kernel_semaphore), val)` and use `list_push_back` to add `u_sem` to the `all_semaphores` list in the current PCB, acquiring and releasing the `semaphores_list_lock` as necessary. Return true once the above is completed.

sema_down
Iterate through the current PCB's `all_semaphores` list until a `user_semaphore` with `semaID` == `*sema` is found

- If no such `user_semaphore` is found, return false
- If a `user_semaphore` is found (call it `u_sem`), call `sema_down(&(u_sem->kernel_semaphore))` and return true

sema_up
Iterate through the current PCB's `all_semaphores` list until a `user_semaphore` with `semaID` == `*sema` is found

- If no such `user_semaphore` is found, return false
- If a `user_semaphore` is found (call it `u_sem`), call `sema_up(&(u_sem->kernel_semaphore))` and return true

get_tid
return `thread_current()->tid`

Modifications to process control syscalls:
exec
Modify `process_execute` in `process.c` to :

- Initialize the lists `all_locks`, `all_semaphores`, and `user_threads` in the new PCB that is being created. Also call `lock_init` on `user_list_lock` , `locks_list_lock`, `semaphores_list_lock`, and `join_lock`
- Malloc a new `user_thread` struct with `tid` initialized to `thread_current()->tid`, `join_wait` initialized to 0, and `exited` and `waited` initialized to false. Add this `user_thread` to the `user_threads` list of the new PCB

wait
No changes need to be made to `process_wait` because our existing implementation already ensures that only the thread that calls the `wait` syscall is blocked. The other user threads will continue running.

exit
Add the following logic at the end of `intr_handler` in `interrupt.c`

- If the current PCB's `exit_status` has the field `exited` set to true
  - Use the function `is_trap_from_userspace` to check if we are going from user mode to kernel mode. If true, call `pthread_exit()`. Note that we call

`pthread_exit()` and not `pthread_exit_main()`, even on the main thread so no joining will take place.

Modify `thread_exit` in `thread.c` as follows:

- At the beginning of the function, iterate through the `locks_held` list of the current thread and release every lock

Modify `process_exit` in `process.c` as follows:

- At the beginning of the function, set the current PCB's `exited` field to true
- Iterate through the `user_threads` list of the current PCB and find the corresponding `user_thread` struct for the current thread. Once found, call `sema_up` on the `join_wait` semaphore in this struct
- Iterate through the `user_threads` list of the current PCB and call `sys_pthread_join` on every active user_thread in this list
- When freeing the process resources, iterate through the lists `all_locks`, `all_semaphores`, and `user_threads` in the current PCB and free every element
- At the end, change `thread_exit()` to `pthread_exit()` so the current thread will also free it's userspace stack before killing the process

Synchronization

We need synchronization for the following lists in the process struct:

- `all_locks` because multiple user threads may try to initialize and add a new lock to the `all_locks` list at the same time. We use `locks_list_lock` to ensure that only one thread can modify this list at a time.
- `all_semaphores` because of the same reason as `all_locks`; multiple threads may try to initialize a new semaphore at the same time. We use `semaphores_list_lock` to ensure only one thread modifies this list at a time.
- `user_threads` because multiple user threads may call `sys_pthread_create` simultaneously and try to add their new `user_thread` struct to the `user_threads` list in the PCB at the same time. We use `user_list_lock` to ensure only one thread can add to this list at a time

We also need synchronization for the `pthread_join` function because multiple threads may attempt to join on the same user thread. In this case, one of the calling user threads may be blocked on `sema_down` forever, which we don't want. We define a process-level `join_lock` so that, for any process, at most one thread can call the `pthread_join` function at a time. Further join calls on the same user thread will not work since the thread has already been joined.

We don’t need additional synchronization for the following:

- `pthread_create` because, other than `user_threads`, all resources are modified independently by the calling thread
- `pthread_exit` because the thread only interacts with its own `user_thread` struct to set fields and free the stack. It does not modify resources used by any other thread
- `lock_acquire`, `lock_release`, `sema_up`, `sema_down` because they only read from the `all_locks` and `all_semaphores` lists and do not modify them

Rationale

The main idea behind our design was to maintain a 1-1 mapping between kernel threads and `user_thread` structs via the `user_threads` list in the PCB. Then, we can interface with this list whenever there are any syscalls involving user threads. We thought about using the existing `thread` struct and its fields to write the user thread syscalls, but this would not have worked since a user thread can both wait on a child process while joining another thread. Hence, we need different semaphores and boolean flags for user threads. For user thread syscalls, we tried to mirror the process control syscalls. For `pthread_create` we wrote the helper functions in a similar way to the helper functions for process `exec` but loading the `stub` function instead of an executable. For `pthread_join` we used a semaphore to block the calling thread, similar to the semaphore we used in the `wait` syscall. And for `pthread_exit` we simply free the userspace stack allocated to this user thread and call the corresponding kernel `thread_exit` function. In modifying the `exit` syscall, we considered different approaches such as having the calling user thread modify the status of every other user thread in the program to `THREAD_DYING` or modifying the `thread_exit` function to forcibly remove every other user thread. However, these approaches would have caused more problems so we ultimately decided on a mechanism that checks if a user thread is supposed to be killed on entry into the interrupt handler, then have that user thread kill itself.

For the user-level synchronization primitives, we maintained a mapping for each user-level primitive to a kernel-level structure. Hence, the user locks and semaphores simply act as identifiers to actual kernel locks and semaphores, which we store as lists in the PCB. We found this to be the cleanest approach since we don't have to modify the existing char primitive for `lock_t` and `sema_t` it's easy to reason about the user-level synchronization syscalls.

Concept check

1. When a kernel thread calls `thread_exit`, the page containing its stack and TCB is freed in `thread_switch_tail`, which is run by the next thread that is scheduled. We can't do this in `thread_exit` because it is run by the exiting thread, which depends on its own stack and TCB.
2. The `thread_tick` function is executed in the kernel stack of the thread that is currently running. If that thread's time slice expires, it will switch to the next thread to be scheduled.
3. The scheduler ordering A, B, A, B will cause a deadlock. In this order, it is possible that thread A will acquire `lockA`, thread B will acquire `lockB`, thread A will wait for `lockB`, and thread B will wait for `lockA`. This is circular waiting, and neither thread will make progress.
4. Thread A taking thread B off the ready queue and freeing its thread stack may leak resources. Other memory allocated on the heap by Thread B is not freed, and my cause performance issues as the system continues to run. Also, Thread B may have held locks that will never be released, leaving any other threads waiting on those locks in a blocked state.
5. To show that priority donation does or doesn't work, we will need at least 3 threads, one high priority thread that will wait on a low priority thread (through a lock, for example), and a medium priority thread. The low and medium priority threads will both down a semaphore that will be upped by the main thread. If priority donation works, then the low priority thread will run first, as its effective priority is now that of the high priority thread that is waiting on it. If the bug in `sema_up` prevents priority donation, the semaphore will incorrectly only consider the base priority and begin running the medium priority thread instead.
   To demonstrate this, we'll initialize a semaphore to 0 as well as a lock in the main thread. Then, we'll create the low priority thread (still higher than the main thread's priority) and have it acquire the lock and down the semaphore. The main thread will then continue running and create the high priority thread, which will

attempt to acquire the lock currently held by the low priority thread. If priority donation works, then the low priority thread should have a high effective priority. We go back to the main thread again and create the medium priority thread, which downs the semaphore and returns us again to the main thread. Finally, we'll up the semaphore; our expected output will be that the low priority thread runs (which we can check with a `msg` call), but the actual output with the described bug is that the medium priority thread will run first.