Project Threads Design

Group 06

Name	Autograder Login	Email
Seong Hyun Park	student366	spark6015@berkeley.edu
Ronald Arifin	student412	ronaldarifin@berkeley.edu
Noah Ku	student89	noahku@berkeley.edu
Michael Wiradharma	student338	michael.wiradharma@berkeley.edu

---

Efficient Alarm Clock

Data Structures and Functions

timer.c

```
// defines the tick amount per second
#define TIMER_FREQ 100
//This is the funciton that we are currently
void timer_sleep(int64_t ticks)
static void timer_interrupt(struct intr_frame* args UNUSED)
```

thread.c/thread.h

```
struct thread {
  /* Owned by thread.c. */
  ...
  // the number of ticks at which thread can be re-enqueued.
  int64_t wait_remaining;
  ...
```

```
}

// thread block and unblock function to set status as blocked
when waiting
void thread_block(void)
void thread_unblock(struct thread* t)

// current ready list
static struct list fifo_ready_list;
// create waiting threads list
extern static struct list wait_list;
```

interrupt.c

```
enum intr_level intr_disable(void) {
enum intr_level intr_enable(void) {
```

For this segment, we are re-implementing `timer_sleep` and timer`_interrupt` from timer.c. As stated in the project spec, the current implementation of `timer_sleep` reasons for busy-waiting, as the same thread may be re-enqueued repetitively, and thus will constantly run `thread_yield`. Instead, we will use a separate wait_list data structure that stores all the threads currently sleeping. We also modify the thread struct to contain that thread's expected wake-up time in a variable `int64_t wait_remaining`. We are going to use Pintos list to add thread that needs to sleep.

Algorithms

In this section, we will focus our implementation on two functions: `timer_sleep` and `timer_interrupt`.

In timer sleep, we removed the `thread_yield` function call. Instead, we'll call `thread_block` and add the respective thread into the wait_list data structure. We first see if ticks in `timer_sleep` is valid. we are thinking about returning right away in case if its negative or zero because in that case threads do not need to sleep. We now maintain two separate queues of threads, the first of which is the original fifo_ready_list, and the second is our new wait_list. The wait_list will store all the

threads that are currently blocked and put to sleep, while the fifo_ready_list includes all the current threads that are ready to be run. Lastly, upon adding a thread to the wait_list, we'll sort the queue based on their wait_remaining variable. Also before adding we have `intr_disable` and at the end `intr_enable` because thread_block requires to intr to be off. This a small optimization which will be used in `timer_interrupt`.

In timer_interrupt, we will process all the threads that are currently in the wait_list queue. We will perform a loop through all the threads in the queue and check if the thread has slept for enough time, through the wait_remaining variable. If so, we can unblock the thread and place it back on the ready queue such that the kernel can reschedule the thread. As mentioned in the previous paragraph, we've added the sorting optimization such that we won't have to loop through the entire wait_queue at every tick increment. Once the loop reaches a thread that is still sleeping, we can break out of the loop early as the sorted queue ensures all threads after will still be sleeping.

Synchronization

The requirement for synchronization in this section is quite loose in this section. There aren't any critical sections that will require semaphores or mutual exclusion locks. Rather, as mentioned in the docstring of some of the functions, we want to keep interrupts disable when running most of `timer_sleep` and enable at the end of timers_sleep as the contents of the function aren't critical. We simply need to add the thread to the appropriate queue in this function. We should still allow interrupts such that higher priority threads or syscalls may be able to operate normally.

Rationale

The rationale behind this implementation stems from avoiding the busy-waiting of threads by calling thread_yield. We implement a separate wait_queue in order to ensure that there isn't busy waiting, and threads are only queued again once they are done sleeping. This introduces the shortcomings of this solution, as we are placing a heavier workload on the scheduler. The scheduler needs to loop through all waiting threads and check if they can be rescheduled, which is partially alleviated by the sort function optimization in `timer_sleep`.

We also tried to use as many of the existing functions as possible to minimize implementing new features. We are reusing `thread_block` and `thread_unblock` to

signify when threads can be re-queued into the ready queue. In this way, we also re-use the THREAD_BLOCKED status as a method of determining when the thread is asleep.

Strict Priority Scheduler

Data Structures and Functions

thread.h

```
//Strict Priority Scheduler Data Structures
struct thread {
  int priority;
  //NEW CODE
  // An integer that will only be set when the thread is creat
ed
  int promoted_priority;
  // A list of lock structs
  struct list locks_held;
  struct list waiting_locks;
}

extern struct lock *wait_lock;

// Donates priority to all threads that the input thread is wa
iting on
void set_donate_priority(struct thread* start_thread, int valu
e);

// Sets a thread's priority back to promoted_priority
void undo_donate_priority(struct thread* input_thread);
```

thread.c

```
static void thread_enqueue(struct thread* t) {
  // MODIFY: Account for SCHED_PRIO instead of our default SCHED_FIFO
}

// A list specifically used for strict priority scheduling
static struct list priority_ready_list

// Function that helps us donate priorities
void donate_priority(struct thread* lower_prio);
```

synch.h

```
struct lock {
  struct thread* holder;      /* Thread holding lock (for debugging). */
  struct semaphore semaphore; /* Binary semaphore controlling access. */

  //NEW VARIABLES
  struct list waiter_threads;    //Track all threads waiting on lock
  struct list_elem elem;
};

struct semaphore {
   /*...*/
   struct list waiters; // We will be sorting this in this project
};

struct condition {
```

```
   /*...*/
   struct list waiters; // We will be sorting this in this pro
ject
}
```

Algorithms

Thread Selection

The default way to select the next thread to run is to use FIFO. For this task, we will have to start implementing a strict priority scheduler. To do this, we can create and use another list called `priority_ready_list` that is similar to the `fifo_ready_list`. However, insertions will call `list_insert_ordered` instead and will be using another function we will implement for sorting the thread by priority. The sort will be descending. We will be calling `list_insert_ordered` in `thread_enqueue`, and we will be checking to see if the active scheduling policy is `SCHED_PRIO`.

After we have this sorted list, we need to modify `thread_schedule_prio`. We need to return the thread with the highest priority, and we can do this by popping the first element of `priority_ready_list`. In the case that the list is empty, we follow the same structure as the FIFO schedule priority function and return an idling thread.

Priority Preference

When considering priority preference, we need to modify the 3 synchronization variables: locks, semaphores, and conditional variables. For semaphores, we need to sort the `wait_list` based on priority (descending). We would modify `sema_down` such that we insert into the `wait_list` in descending order, so that when we `sema_up`, we will get the highest priority thread.

We also do the same for conditional variables, where we sort each conditional variable's `wait_list` based on priority (descending). We do this so that the top priority semaphores or conditional variables can be easily accessed by popping the first element from these lists, depending on which one is needed. The implementation for priority preference for locks will be explained in Locks section of this design document.

Thread Get and Set Priority

Set Priority
This will override the value of the current thread's "base" priority value to be **new_priority**. If it turns out that the new_priority value is higher than the promoted_priority, then we will call `set_donation_priority` again, but this time we pass in `thread_current()` as our source to propagate down.

After overriding the value, we also need to yield at the end by calling `thread_yield()`. We only do this if the priority was ever changed to be higher than the `promoted-priority`.

Get Priority
This will get the priority of `promoted_priority` because promoted priority will always be **>=** priority. This is because, we have a mechanism to promote the priority of the thread to be higher, but it will never be less than the "default" priority.

Lock Acquiring

Every time we acquire a lock, we know about the previous holder and new holder. If there is a previous holder, then we will need to handle priority donation. To handle this, we will set our **new_holder's→promote_priority** to be the return value of our `set_donation_priority`, which will recursively donate the new_holder priority to threads that are waiting for **current_holder**. We cannot forget to add the **new_holder** into our list of **wait_for_threads** for the particular lock that we are trying to acquire because in this case, there was another thread that owned the lock. Last, we want to block the current thread by calling `thread_block`.

On the other hand, if there is no holder, then we must add 2 things. First, we set **lock→holder** to be the current thread, and second, we add the **current_lock** into the current thread.

Here is some pseudocode for how we will implement lock acquiring:

- previous_holder = lock-holder (may be NULL)
- new_holder = thread_current()
- When we acquire a lock, we check if anyone is a holder.

- **If there is a holder,**
  - priority donation by calling `set_donation_priority`
  - We set **lock→waiter_threads** append **new_holder** using `list_insert_ordered`.
  - thread_current()→**waiting_locks.**append(lock)
  - call **thread_block()**
- **If there is no holder,**
  - holder = thread_current().
  - thread_current→locks_held.append(lock)

Donations

We will be focusing on the function `set_donation_priority`. For donations, we need to consider cases where a thread that has a higher priority is trying to acquire a lock that a thread with a lower priority is holding. Our function will also donate to chains of threads waiting for the same lock as long as the priorities are strictly decreasing. If there was a case where it's not strictly decreasing, it would already be handled by previous `set_donation_priority` calls when a thread was trying to acquire a lock. Let's take a look at an example. We start with Threads A (priority of 100), B (priority of 50), and C (priority of 5). Here is our example below:

A (100) → B (50) → C (5)
becomes
A (100) → B (100) → C (100)

Thread B will first donate to Thread C since B is the first one to acquire that lock that thread C is holding. What happens now is that thread C's priority matches B's, so they both are now 50. Afterwards, Thread A tries to acquire the same lock B and C are working with, so it now waits and donates its priority (100) to both B and C. Now, we end up with all threads having the priority 100.

Here is the pseudocode for how we will implement the iteration through chains of threads:

set_donation_priority(start_thread, value):

- **thread_we_are_on = start_thread**
- **while (thread_we_are_on→wait_for_thread != NULL):**
  - **thread_we_are_on = thread_we_are_on→wait_for_thread**

- **if (thread_we_are_on→promoted_priority > value):**
  - **break**
- **thread_we_are_on→promoted_priority = value**

How this function works is that it will continue to loop through threads by accessing the `waiting_for` thread. It will keep checking to see if there are any chains and stop when we reach the end (this is when `waiting_for` is NULL). The other case that will make the loop end is if the next thread we are waiting on has a priority greater than or equal to our current thread's priority. If this happens, we can stop the loop so it does not accidentally override higher-priority threads.

The way we keep track of restoring a thread's priority is by using the integer called `priority` in the thread struct. This value can be considered to be like a "read-only" integer where we only set the value when the thread is created. When the thread that had its priority raised completes its task fully, the way we can lower it is by setting `promoted_priority` to `priority`. This way, it can be also raised again by other threads if needed and will not affect the overall flow of strict priority scheduling.

Lock Releasing

We will get the current lock holder by calling `lock_held_by_current_thread(lock)`. Then, we will remove the lock from the **locks_held** by the previous_holder. In this case, our current thread no longer keep tracks of this lock since we are releasing it.

To restore priorities, we can use the function `undo_donation_priority` on the thread that releases the lock. What this function does is that it changes the priority of all the threads that is waiting for the remaining locks of this "previous_holder". This is because the remaining locks of "previous_holder" is the locks that are of value to the threads waiting for it. Hence, we should have the this releasing thread to have a priority of MAX(releasing_holder → locks_held → waiting_threads→ priority) where we would need to do some loop inside for each locks and for each waiting threads.

Last, we can call thread_unblock, so that the next scheduled thread can run.

Here is some pseudocode for lock releasing:

- We know that **current_lock_holder** = lock_held_by_current_thread(*lock*)
- We **remove &lock** from **previous_holder→locks_held**
- We will need to update the **current_lock_holder→promoted_priority** to the **MAX(current_lock_holder→locks_held→waiters→priority) [Repeat from section above]**
    - What this means is that we go over all locks held and see who are the waiters, we know that a waiter will promote whoever hold's the lock that is why doing a max will work here.
- **Call** thread_unblock()

Synchronization

The synchronization has mostly been covered in the lock and donation implementations stated above. To summarize, race conditions will be resolved and handled because we are donating priorities when a high-priority thread is trying to access the lock that a low-priority thread has. This forces threads to wait for each other in the correct order and avoids priority inversion. The way we also implemented donation will work for chains of threads as well. As long as the chain has a list of threads that are strictly decreasing in priority as you move through the chain, the highest priority thread will continue to donate all the way through until the end by using the while loop we use in `set_donation_priority`.

On top of locks, our semaphores and conditional variables are sorted based on priority (descending), so the synchronization in this case has also been taken into account. The scheduler will run the next thread in the correct order based on the way we sort the `wait_list` of these structs.

Rationale

For thread selection, we decided to use sorted list inserts since it is a simple way to get the next highest priority. It also takes into account ties, because if you want to use round-robin to get rid of ties, pushing tied high-priority threads will allow us to cycle through those specific high-priority threads before running lower priority threads. All you have to do is to pop one of the highest threads, and inserting it back would put it behind the high-priority threads that it tied priority with.

We also wanted to use a while loop for priority donation to make it easier for us to debug. Although recursion is also a viable option and is one that we may switch to

while we start coding, we believe that both options work and our while loop implementation will have the same result.

For locks, we wanted to take note of the max priority of all waiters (from the list) since it's our way of changing the priority of a thread that is holding multiple locks that multiple threads are waiting on. If we try using other implementations instead of getting the max of all waiters directly, it may be more complicated and our algorithm may be unintentionally stuck from blocked threads.

User Threads

1. user-level synchronization
    a. lock_init,
    b. lock_acquire,
    c. lock_release,
    d. sema_init,
    e. sema_down
    f. sema_up

Data Structures and Functions

```
// in process.c
tid_t pthread_execute(stub_fun, pthread_fun, void*);
static void start_pthread(void* exec_ UNUSED) {}
bool setup_thread(void (**eip)(void) UNUSED, void** esp UNUSE
D) { return false; }
tid_t pthread_join(tid_t);
void pthread_exit(void);
void pthread_exit_main(void);

// process.h list to keep track of all threads. each list elem
is a thread_elem_t
struct process {
```

```
  ...
  struct list thread_list;
  ...
};

typedef struct thread_elem {
  struct thread* thread_tcb;
  struct list_elem elem;
} thread_elem_t;


//User-level synchronization syscalls

typedef char lock_t;
typedef char sema_t;

lock_t lock_t_list[256];
smea_t sema_t_list[256];
bool lock_init(lock_t* lock)
bool lock_acquire(lock_t* lock)
bool lock_release(lock_t* lock)
bool sema_init(sema_t* sema, int val)
bool sema_down(sema_t* sema)
bool sema_up(sema_t* sema)
```

Algorithms

User pthread library functions

```
tid_t pthread_create(stub_fun sfun, pthread_fun tfun, const vo
id* arg)
```

This user syscall will create a new user thread by making calls internally to thread_create, which creates the new kernel thread. This new thread will then be added to the list of all existing user threads in the PCB. (Note: the main thread will not be on this list for simplicity.) We will then make use of the functions `pthread_execute` and `start_pthread`, found at the bottom of `process.c` to load the user program into the thread and continue running the `tfun` function (with arguments `arg`) as specific by the user. This will fix the 1-1 kernel and user thread mapping as mentioned in the spec.

The logic of these two functions will be very similar to that of `process_execute` and `start_process`, which were used to run the initial main user thread. Additionally, we will use `setup_thread` to handle the new thread's stack logic, including handling things such as argument passing. We will follow the spec's implementation of keeping thread 2's stack underneath the first stack for simplicity. The will make use of a constant that determines the size of each thread's stack. This constant will be related to an appropriate amount of pages of memory the stack can fit. This way, calculating boundaries for the stack's frame while validating pointers will be made simpler.

```
void sys_pthread_exit(void) NO RETURN
```

This syscall will cause the thread to immediately. As the function implies, there will be no return value for this function; however, it will manage freeing up existing data on the thread's stack and clearing up memory used for the TCB. The thread's status will also be set to TERMINATED, and stll maintained on the list of user threads in the PCB.

Special logic will occur when the main thread makes this syscall. We apply the same algorithms used to implement pthread_join's logic, which will be furthered discussed in the next section. Furthermore, we will loop through all existing user thread threads and call pthread_join on them.

```
tid_t sys_pthread_join(tid_t tid)
```

We will implement this syscall by throwing the current thread out of the queue and forcing the thread with the given `tid` onto the queue to run. This will be similar to the implementation of alarm clocks above, where the current active thread will remain blocked (also setting its status to BLOCKED) until the new thread returns. This function will then return the `tid` of the thread joined on, or return TID_ERROR if

it fails. Special attention will be made to threads that are already joined on, as to not block the current running thread.

Additionally, if this syscall is called on the main thread, we will allow the main thread to run to completion. The thread that called to join can run even after the main thread calls `pthread_exit`, however, `pthread_exit` won't kill any processes until the calling thread has run to completion. We can simplify this logic by halting the `pthread_exit` syscall by and calling it again once the caller thread is done.

```
tid_t get_tid(void)
```

This is straightforward. The tid of the thread will be stored on its respective TCB, and can be returned from there.

The following section will identify modifications to current existing syscalls:

```
pid_t exec(const char* file)
```

Thread creation for the first time when loading a user program will stay the same. Our current code already creates a new process when exec is called, which contains a single thread of control, as required by the spec. The remaining syscalls will maintain the same functionality, as the user will be able to manage their own pthreads.

```
int wait(pid_t)
```

The wait syscall will need to be modified to ensure that only the user thread which makes the syscall will halt. Our current implementation will force the entire process to pause until the child has finished processing. Revisions will be made to ensure that only the current user thread which made the syscall will be halted. This will be done using a similar process of using sempahore variables and setting the thread's status. Additionally, we will modify the edge case of when there is only one user thread, where we will keep the implementation unchanged.

```
void exit(int status)
```

This sycalls will cause the entire user program to exit immediately. Effectively, we can call `pthread_exit` on all the existing running threads in the PCB. Then we can exit the program using the given status code. `pthread_exit` will manage freeing up each TCB's resources, will the existing implementation will take care of freeing up the PCB's resources.

We will make use of the `is_trap_from_userspace` function to ensure the thread does not exit while in kernel space, as this might cause errors when freeing up user thread memory. Using the assumption provided in the spec, we can expect threads to eventually return to user space, and we can kill the thread then.

In general, we should keep in mind other syscalls which may affect entire processes vs just a specific thread. Like in the case of `wait` above, we want to ensure the syscall won't block the whole process from running. In other cases such as `open`, these should still be called at process-level structures, since the FD table is a shared resource between threads of the same user program.

User library synchronization syscalls

All t**he lock functions should start by verifying lock identifier corresponds to a registered and initialized lock, by consulting the kernel's lock tracking structures.**

```
bool lock_init(lock_t* lock)
```

We need to request to kernel to register the lock. I will be using 256 size of the array to keep track of lock_t. If the lock is successfully registered, the kernel allocates a corresponding synchronization structure to handle lock operations at the kernel level and returns 'true' to indicate successful initialization. There will be an id of lock_t stored each lock_t and can be thought of as a char. With 8 bits (which can represent 256 numbers) we can have a kernel to link each one to the lock. If there is the same char of lock trying to register, we can think of it as undefined behavior.

```
bool lock_acquire(lock_t* lock)
```

If the lock is not initialized, the function should return false. Once this is done, our function needs to request the kernel to acquire the lock for all of threads. This process will be involved with interruption and switch user to kernel. By tracking id, we get the lock and acquire it. The system might need to disable further interrupts temporarily to ensure the atomicity of the lock acquisition process. If this process is done correctly we return true, else false.

```
bool lock_release(lock_t* lock)
```

We should signal the kernal to release the lock in syscall by waking up other theads. We need to handle disabling interrupt to make it atomic. if we are able to realse the lock without error, we return true.

All the sema functions should start by verifying sema identifier corresponds to a registered and initialized lock, by consulting the kernel's lock tracking structures.

```
bool sema_init(sema_t* sema, int val)
```

Then we will register sema into kernel. During registering into kernel, if it was not able to register, the functions needs to return False else true. Also, if it is successfully registered into kernel, we also need to put into the sema_t_list to store the identifier of sema. After successfully register sema, we return true, else false.

```
bool sema_down(sema_t* sema)
```

By tracking identifier in kernel and sema_t_list we should be able to get the value of sema and down the value as needed. if the value is lower than zero, we need to block the current thread to accessing resources. if this process is failed due to uninitialized/unregistered sema, we return false, else ture.

```
bool sema_up(sema_t* sema)
```

Using tracking function, we get sema and based on the sema we get, we simply add 1 to the value. if tracking is failed we return false.

Synchronization

User pthread library

There are a few cases for the synchronization of user pthread library at this level. Firstly, we will use locks when it comes to handling critical data such as managing the thread list in the PCB, or when modifying other aspects of the TCB. Additionally semaphores should also be used when it comes to scheduling user threads or kernel thread interrupts (traps) to ensure data does not get mismanaged. We will also make use of monitors and their conditional variables to in implementing `pthread_join`. This will simply the process of a sleeping thread and prevent busy waiting while the thread remains blocked.

User-level

We need to have a synchronized lock acquisition function because there might be cases where multiple threads are trying to get the lock simultaneously. Releasing a lock will not require synchronization because it won't matter if multiple threads are trying to release the same lock since it won't cause any errors.
During sema_up and sema_down, there needs to be synchronization because there might be a scenario where multiple threads are trying to sema_down, and without proper synchronization, the value might decrease to less than 0, which would be problematic. The same case applies for sema_up as well.

Rationale

There were a lot of design choices made in this section to maintain the flow of syscalls and ensure that all of the existing syscalls made in project 1 will be compatible with multithreaded programs. Our design philosophy for this section was to keep the data structures simple, thus we created a pintos list that contains a list of pointers to all TCBs that the user program has created. All other syscall and multi-threaded logic will be handled in each individual syscall.

Additionally, since the implementation for calling new user threads are very similar to starting a new user program, we heavily relied on existing pintos code and tried to minimize change so as to keep the program simple without over-complicating much of the requirements. This way, we can focus on maintaining the process and thread abstraction between the various user threads, and ensuring that particular syscalls would only affect the relevant threads or process.

We believe that much of the implementation for creating a user pthread library is not complicated. However, we expect the brunt of the work to come from debugging edge cases that we need to consider. For this reason, we have decided to keep the data structures and function as simple as possible, such that much of the changes we make will be reliant on these respective edge cases.

Moreover, for user-level synchronization, we believe that lock_t and sema_t are an id that may guide us to actual lock and semaphore. Using IDs of those, we see the possibility that we can map it to the locks and semaphores to perform acquire/release and up/down.

Concept check

1. This is because we expect the thread to continue running a few more checks before the thread can be properly closed (or “killed”). For example, in the case the thread is the main thread which runs the process, it needs to call “shutdown_power_off”. In the case that it is not the main thread, it needs to schedule the next thread first and called the scheduler’s interrupt before it is killed, otherwise the OS will remain in a limbo state with no thread running. Also, it is possible that the thread needs to clean up some resources. Most importantly, we won’t be able to call palloc_free inside thread_exit as the thread itself is still running. We need to ensure that another thread has replaced it, so that it’s memory is no longer being actively used and can be freed up safely.

2. /* Called by the timer interrupt handler at each timer tick. Thus, this function runs in an external interrupt context. */ Comment under timer_tick and also, we noticed that if we use backtrace while using gdb, it says that the functions is called from `intr_entry`. Therefore, it is in interrupt stack.

3. There is a case where deadlock occurs. If

    - Thread A acquires lock A
    - Thread B acquires lock B
    - Thread A tries to acquire lock B but it is acquired by Thread B so it gets blocked
    - Thread B tries to acquire lock A but it is acquired by Thread B so it gets blocked

    Therefore if we schedule in such a way lock A acquires then B acquires, we can get a deadlock.

4. If Thread A kills Thread B while it is in ready_queue, there might be an issue where ready_queue thinks that there is something to pop from the list. When the queue pops and try to do something with it, because it is killed there might be a dangling reference or kernel panic since it tries to access that something is not there. This may result in resource leaks as well. Killing thread does not remove it from ready_queue. Also, there might be a case where the thread is terminated without proper steps and results in a zombie state.

5. Say T1 acquired a lock called A, and T1000 wants to acquire lock A. T10 will block and promote T1. Next, T100 get's added. We then sema down T1 and T100 on a semaphore of init = 2. We expect T1 to be added to the wait queue of the semaphore first, and then T100. When we cal `sema_up` we expect T100 to run first even though T1 has a higher promoted priority of 1000. In the default code, it would run T1 first because T1 arrived first, but we should run T100 first where the "base" priority is higher than T1 despite the `promoted_priority` of T1 being 1000.