Project Threads Design

Group 36

Name	Autograder Login	Email
Pritish Parmar	student251	pritishparmar@berkeley.edu
Siddharth Shashi	student63	sshashi@berkeley.edu
Aman Doshi	student457	aman.doshi@berkeley.edu
Matthew Nguyen	student161	matthew.t.nguyen@berkeley.edu

Efficient Alarm Clock

Data Structures and Functions

Sleeping threads will be stored in a global Pintos list. List elements identify a thread and the thread tick time to wake up the thread.

```
struct list sleeping_list; // List of sleeping threads
struct sleeping_list_elem {
  struct thread *t; // Pointer to sleeping thread
  unsigned int wakeup_time; // Wake up tick time of sleeping thread
  struct list_elem elem; // List element of sleeping_list
};
```

These are helper functions in timer.c are used to implement the efficient alarm clock.

```
bool sleeping_list_less(const struct list_elem *a, const struct list_elem *b, void *aux);
void timer_wakeup(void); // Wakeup sleeping threads after timer expires
```

Algorithms

<u>Data Structure Initialization</u>

1. In thread.c/thread_init(), initialize the sleeping_list Pintos list.

<u>Helper Function: sleeping_list_less()</u>

The sleeping_list_less() function takes as arguments pointers to two list_elem structs (stored in sleeping_list_elem structs), and returns true if the first sleeping_list_elem's wakeup_time is less than that of the second sleeping_list_elem, and false otherwise.

<u>Timer Sleep</u>

1. In timer.c/timer_sleep(), if ticks <= 0, return (do nothing). Otherwise, a new sleeping_list_elem struct will be malloced, where thread **t** is set as the thread_current() and wakeup_time is set as (timer_ticks() + ticks).
2. Next, interrupts are disabled by calling intr_disable(), to ensure synchronization of sleeping_list. Then, sleeping_list_elem is inserted into the sleeping_list by calling list_insert_ordered() with sleeping_list_less() helper function. This ensures the sleeping_list stays in sorted ascending order of wakeup tick times.

3. Then, the thread is put to sleep with a call to thread_block().

Timer Interrupt Handler

1. In timer.c/timer_interrupt(), call a new helper function called wakeup_timer().
2. In timer.c/wakeup_timer(), get and store OS ticks by calling timer_ticks(). If the sleeping_list is empty, return. Otherwise, get the first element of the sleeping_list and check if current tick count >= sleeping_list_elem's wakeup_time. If so, pop the sleeping_list_elem off the list, call thread_unblock on the corresponding thread, then free the struct. Repeat this process until current tick count < sleeping_list_elem's wakeup_time or the sleeping_list is empty.

Synchronization

The sleeping_list Pintos list can be read and written to by multiple threads concurrently. It is accessed in timer.c/timer_sleep() and timer.c/wakeup_timer().

1. In timer_sleep(), synchronization is ensured by disabling interrupts, meaning only 1 thread can enter critical sections. Locks cannot be used as wakeup_timer() is in an external interrupt context, meaning the interrupt thread cannot go to sleep.
   a. Disabling interrupts can limit thread concurrency. However, interrupts are only disabled for a short time, so the impact will be minimal.
   b. Additionally, no memory costs incurred for using interrupts as a synchronization tool.
2. In wakeup_timer(), interrupts are disabled when this function is called from timer_interrupt(), so no additional synchronization is needed.

Rationale

- Our proposed solution is simple to understand and requires little code to implement. The use of helper functions makes the design flexible and extendable for additional features. The short comings of the design is the reliance of interrupts for synchronization, but this is unavoidable due to external interrupt handlers being unable to sleep.
- Using list_insert_ordered(), the insertion time into sleeping_list takes O(N), where N is the number of threads in sleeping_list. Waking up threads takes O(M), where M is the number of threads that must be woken up, since only the first M+1 threads need to be checked for wakeup. An alternate solution considered was inserting threads arbitrarily in sleeping_list, and searching the entire list when waking up threads. Waking threads increases to O(N) with this approach, so we decided against it.
- Since interrupts are disabled when threads are being woken, the threads can be woken in any order, regardless of their priority.

Strict Priority Scheduler

Data Structures and Functions

In thread.c, the prio_ready_list is a global array of Pintos lists, where index k stores a Pintos list of threads of priority k.

```
struct list prio_ready_list[64];
```

The following code shows modifications to existing structs.

```
struct thread { // thread.h
  struct list held_lock_list; // List of a thread's held locks
```

```
  struct lock *waiting_on; // Set if thread is sleeping on a lock acquire
  int priority; // Removed attribute
  int base_prio; // Base priority of a thread
  int donated_prio; // Highest donated priority of a thread, -1 if no donations
  // --- other attributes ---
};
struct semaphore { // synch.h
  struct lock *lock; // Pointer to lock containing the semaphore (set if inside lo
ck)
  // --- other attributes ---
};
struct lock { // synch.h
  struct list_elem elem; // List element of thread's held_lock_list
  // --- other attributes ---
};
struct semaphore_elem { // synch.c
  struct thread *t; // Pointer to waiting thread in a condition variable
  // --- other attributes ---
};
```

Inside thread.c file, helper functions used to implement the Strict Priority Scheduler.

```
int get_effective_prio(struct thread *t); // Return t's effective priority
int set_donated_prio(struct thread *t, int new_prio); // Set t's donated priority
void update_thread_prio(struct thread *t, int old_prio); // Update prio_ready_list
[]
int get_max_ready_prio(void); // Get max priority from prio_ready_list
```

Algorithms

Initialization of Data Structures

1. In thread.c/thread_init(), initialize all Pintos lists in the prio_ready_list[ ] array.
2. In sync.c/thread_create(), initialize a thread's held_lock_list, set waiting_on to NULL, set base_prio to priority argument, and set donated_prio to -1.
3. In synch.c/sema_init(), set sema→lock pointer to NULL.
4. In synch.c/lock_init(), set lock→sema.lock pointer to address of current lock.
5. In synch.c/cond_wait(), set semaphore_elem's thread pointer to the current thread.

Thread Priority: Helper Functions

$$\text{donated\_priority} = \begin{cases} \max_{lock \in held\_lock\_list} \max_{waiting\_thread \in lock} \text{effective\_priority}(\text{waiting\_thread}) & \text{if waiting threads} \\ -1 & \text{otherwise} \end{cases}$$

$$\text{effective\_priority} = \max(\text{donated\_priority}, \text{base\_priority})$$

Note: in this design document, “priority” and “effective priority” are equivalent ideas.

1. In thread.c/get_max_ready_prio(), return the max priority of a ready thread in prio_ready_list[ ] by searching for non-empty lists, highest to lowest priority. Return -1 if no ready threads.
2. In thread.c/get_effective_prio(thread), return max of thread’s base priority and donated priority. Donated priority is not computed. Instead, the donated_prio value is used from thread struct.
3. In thread.c/thread_get_priority(), call and return get_effective_prio() on the current thread. Inside function call, disable and reset interrupt state for synchronization of entire function contents.
4. In thread.c/update_thread_prio(thread, old_prio), get the thread’s priority. If old_prio does not equal the current priority, update the thread’s location in prio_ready_list[ ]. Check if the current priority < get_max_ready_prio(). If so, call thread_yield().
5. In thread.c/thread_set_priority(new_priority), get current thread’s priority and set base priority to new_priority. Then, call update_thread_prio() on the current thread and its old priority if the thread is READY. Inside function call, **disable** and reset interrupt state for synchronization of entire function contents.
6. In thread.c/set_donated_prio(thread, new_prio), get the thread’s effective priority and set the thread’s donated priority to new_prio. Then, call update_thread_prio() on the thread and its old priority if the thread is READY.
7. In thread.c/get_donated_prio(thread), computes and returns a thread’s donated priority using the function described above. The get_donated_prio() value and thread.donated_prio can be different, if the values are not synced up.

<u>Priority Scheduling</u>

1. In threads.c/thread_enqueue(thread), check if the active scheduling policy is Strict Priority Scheduler. If so, push the thread to the back of the Pintos list prio_ready_list[get_effective_prio(thread)].
2. In threads.c/thread_schedule_prio(), call get_max_ready_prio(). If -1 returned, return the idle thread. Otherwise, pop and return from the front of Pintos list prio_ready_list[max priority].

<u>Thread Creation</u>

At the end of threads.c/thread_create(), check if the new thread has a higher priority than the current thread. If so, call thread_yield to put the current thread to sleep, so the new thread can be scheduled.

<u>Synchronization Primitives: Thread Wakeup</u>

1. Inside synch.c/sema_up(), when interrupts are disabled, if threads are waiting, search sema→waiters list for highest priority thread and thread_unblock() it.
2. Inside synch.c/cond_signal(), if there are threads waiting, search cond→waiters list for highest priority thread and call sema_up() to wake it up.

<u>Lock Acquire: Priority Donation</u>

Inside synch.c/sema_down(), between interrupt disable and semaphore decrement.

1. If semaphore can be decremented or sema→lock == NULL (semaphore not part of lock), ignore following steps.
2. Set current thread’s waiting_on lock pointer to sema→lock.
3. Get semaphore’s lock’s holder thread. If the lock holder == NULL, ignore the following steps.
4. If lock holder thread’s priority >= current thread’s priority, ignore the following steps. Otherwise, set lock holder’s donated priority to current thread’s priority. Use set_donated_prio() and get_effective_prio() functions.
5. If lock holder thread’s waiting_on != NULL, goto step 3 with the waiting_on thread.
6. Block the current thread by calling thread_block(). After waking up, set current_thread’s waiting_on lock pointer to NULL. Add current lock to the thread’s held_lock_list Pintos list.

7. Recompute and set the current thread's donated priority using helper function get_donated_prio() and set_donated_prio().

<u>Lock Release: Priority Donation</u>

Inside synch.c/sema_up(), between disabling interrupts and unblocking a thread.

1. If sema→lock == NULL (semaphore not part of lock), ignore the following steps.
2. Remove current lock from thread's held_lock_list Pintos list.
3. Recompute and set the current thread's donated priority using helper functions get_donated_prio() and set_donated_prio().
4. If thread is no longer highest priority thread, yield

Synchronization

- Base priority, donated priority, priority ready list, waiting on lock, and held lock list are all variables that are shared and can be changed by multiple threads. To prevent data races and ensure synchronization, interrupts must be disabled when accessing these variables. Other synchronization primitives (e.g., locks, semaphores) cannot be used as these variables are being used to implement these primitives (locks, semaphores).
  - The functions sema_up(), sema_down(), thread_get_priority, and thread_set_priority() all disable interrupts, hence are thread safe.
  - The functions thread_enqueue(), thread_schedule_prio(), get_effective_prio(), set_donated_prio(), update_thread_prio(), and get_max_ready_prio() are called from functions where interrupts are disabled, hence are also thread safe.
  - Disabling interrupts serializes the work a CPU can do and limits thread concurrency. However, interrupts are disabled for short quantities of time, and the relevant functions are called infrequently. Hence, the overall impact will be minimum. Moreover, threads will only contend for these shared resources while using primitives, which typically forms a small part of any standard multi-threaded program.
- The semaphore's lock variable requires no synchronization, as it is a read-only variable after the initialization.
- Cond_signal() function is protected by an acquired lock, so its critical section is thread safe.
- In the proposed system, no memory is allocated, so there is no concern about synchronized memory deallocation.

Rationale

- Our proposed solution is easy to conceptualize and understand. A good chunk of code needs to be written, but it is split across many helper functions, which makes it easier to write code and debug. Moreover, helper functions makes the code flexible and adaptable when introducing new features. The short comings of the design is the reliance of interrupts for synchronization, but this is inevitable since the code is used to implement other synchronization primitives.
- For tracking all the READY threads, using an array, where each index corresponds to a single priority, means enqueueing a thread takes O(1) time. Additionally, dequeuing the highest priority thread or changing priority of a READY thread also takes O(1) time. Since the number of priorities is small and constant, the space complexity of this setup is O(N), where N is number of READY threads.
  - Our group originally thought of using an unordered Pintos list to track all READY threads, but the time complexity of dequeuing the highest priority thread would take O(N), as the Pintos list would have to be searched each time.
  - Alternatively, maintaining an ordered Pintos list of READY threads would mean queuing a new thread or changing priority of a READY thread takes O(N) time.

User Threads

Data Structures and Functions

The following code shows modifications to existing structs and attributes of new structs.

```
struct process { // process.h
  struct list user_locks; // Pintos list of user locks
  char next_lock_id; // Next available lock ID [0, 255]
  struct lock user_locks_lock; // Synchronize lock data

  struct list user_semas; // Pintos list of user semaphores
  char next_sema_id; // Next available semaphore ID [0, 255]
  struct lock user_semas_lock; // Synchronize semaphore data

  bool terminated; // Indicate if process is exiting
  struct lock terminated_lock; // Synchronize terminated

  int pthread_num_active; // Number of active pthreads in process
  struct sema pthread_sema; // Main process sleeps until active pthreads == 0
  struct list pthread_list; // List of pthread info structs
  struct lock pthread_lock; // Synchronize pthread data

  bool pthread_main_exit; // Indiciate if main pthread exited

  // --- other attributes ---
};
struct pthread { // thread.h
  tid_t tid; // TID of pthread
  int page_id; // User stack page (0, 1, ... from top of userspace)
  uint8_t* page; // User stack page, used to deallocate user stack
  int ref_cnt; // Reference count of pthread_info, initialized to 2
  bool joined_on; // True if pthread_info is being joined on, false otherwise
  struct lock ref_lock; // Synchronize ref_cnt, joined_on
  struct sema join_sema; // Synchronize pthread join
  struct list_elem elem; // List element of pthread_list
};
struct thread { // thread.h
  struct pthread *pthread; // Thread's pthread struct (if process)
  // --- other attributes ---
}
```

```
struct user_lock { // process.h
  char lock_id;
  struct lock lock;
  struct list_elem elem; // List element of user_locks
};
struct user_sema { // process.h
  char sema_id;
  struct semaphore sema;
  struct list_elem elem; // List element of user_semas
};
```

The following code shows signatures for new helper functions.

```
struct lock *get_user_lock(lock_t lock); // Get the pointer to user lock
struct sema *get_user_sema(sema_t sema); // Get the pointer to user semaphore
struct sema *get_pthread(tid_t tid); // Get pointer to pthread with TID == tid
void pthread_terminate(void); // Clean up child pthread's data
void pthread_terminate_main(bool from_proccess_exit); // Clean up main pthread's d
ata
```

Algorithms

Initialization of Data Structures

1. In process.c/start_process(), in the PCB struct
   a. Initialize Pintos lists user_locks, user_semas, and pthread_list.
   b. Initialize locks user_locks_lock, user_semas_lock, terminated_lock, and pthread_lock.
   c. Initialize semaphores pthread_sema to 0.
   d. Set next_lock_id to 0, next_sema_id to 0, terminated to false, pthread_num_active to 1, pthread_main_exit to false.
2. In process.c/start_process(), malloc a pthread struct. Setup pthread as follows.
   a. Set tid to current thread’s TID, page_id to 0, ref_cnt to 2, joined_on to false.
   b. Initialize ref_lock. Initialize join_sema to 0.
   c. Add pthread to PCB’s pthread_info_list. Store pthread address in thread’s pthread attribute.
3. In process.c/setup_stack, store return value of palloc_get_page() in pthread’s page attribute.

Deallocation of Data Structures

1. In process.c/start_process(), when process allocation fails, free current thread’s pthread before PCB is freed.
2. In process.c/process_exit(), free all pthread’s stored in PCB’s pthread_list before the PCB freed. Also free all the user_lock and user_sema structs in the user_locks list and user_semas list respectively.

System Call: Overview

In project Userprog, we designed the syscall_handler() function to carry out the following tasks.

1. Validate user memory of syscall arguments, then copy as needed to kernel space.
2. Identify system call type with a switch case statement.
3. Call a syscall helper function with validated arguments to execute the respective syscall.

The structure allows easy expansion for additional system calls required for User Threads.
Note: in the following sections, assume code is running inside syscall helper functions, with memory validated arguments.

System Call: PThread Create

```
void syscall_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg);
```

Call pthread_execute(sfun, tfun, arg) and set EAX register to the returned TID.

Helper Function: PThread Execute

```
tid_t pthread_execute(stub_fun sf, pthread_fun tf, void* arg);
```

1. Call thread_create() with priority of the current thread, startup function as start_pthread, and **aux** data as (sf + tf + arg + address of current thread's PCB). Use malloc() and memcpy() to setup the **aux** data. Return thread_create()'s TID.

Helper Function: Start PThread

```
static void start_pthread(void* exec_);
```

1. Extract sf, tf, arg, PCB from exec_, then free exec_. Next, set the current thread's pcb to PCB and call process_activate() to setup the page tables.
2. Malloc space for a pthread struct. Setup struct as follows.
   a. Set tid to current thread's TID, ref_cnt to 2, joined_on to false.
   b. Initialize ref_lock. Initialize join_sema to 0.
   c. Acquire PCB's pthread_lock. Search PCB's pthread_list for an available page ID (starting from 0). Note: the list is ordered by page ID in increasing order. Assign pthread's page_id to the next available page ID.
   d. Insert pthread struct into pthread_list using list_insert_ordered (ordered by increasing page_id). Increment PCB's pthread_num_active.
3. Next, create a struct inter_frame named if_ and call setup_thread(&if_.eip, &if_.esp, tf, sf, arg).
   a. If false is returned, remove pthread from PCB's pthread_list and free it, decrement PCB's pthread_num_active, and release PCB's pthread_lock. Then, call thread_exit() to terminate the thread.
   b. Otherwise, just release PCB's pthread_lock.
4. Use assembly to jump to the intr_exit function to simulate a return from an interrupt, so the user function can be executed in user mode and in user space.

Helper Function: Setup Thread

```
bool setup_thread(void (**eip)(void), void** esp, stub_fun sf, pthread_fun tf, voi
d* arg);
```

1. Set *eip ← &sf. Then, palloc_get_page() a user page and return false if NULL is returned. Otherwise, store return value of palloc_get_page() in current thread's pthread's page attribute.
2. Next, install_page() writable at PHYS_BASE - PGSIZE*(page_id + 1). If install fails, palloc_free_page() the allocated page and return false.
3. Set *esp ← PHYS_BASE - PGSIZE*(page_id) - 0x14, with *esp - 0x4 and *esp - 0x8 as tf and arg function arguments respectively for 16B alignment. Then, return true.

Helper Function: Get Pthread

```
struct pthread *get_pthread(tid_t tid); // return struct
```

1. Acquire PCB's pthread_lock, search through PCB's pthread_list for pthread associated with tid, then release lock. If pthread not found, return NULL. Otherwise, return pointer to the pthread.

System Call: Syscall PThread Exit

```
void syscall_pthread_exit(void); // syscall.c
```

1. If is_main_thread(thread_current()), call pthread_exit_main(). Otherwise, call pthread_exit().

Helper Function: PThread Exit

1. Call palloc_free_page on current_thread()→pthread→page to free the user stack page. Then, call pthread_terminate() helper function (defined below).

Helper Function: PThread Exit Main

```
void pthread_exit_main(void); // process.c
```

1. Set PCB's proc_info's exit_status to 0. Then, call pthread_terminate_main(false) helper function (defined below).

Helper Function: PThread Terminate

```
void pthread_terminate(void); // process.c
```

1. Call sema_up on current thread's pthread's join_sema, allowing another thread to join on it.
2. Acquire pthread's ref_lock, decrement ref_cnt, and release ref_lock. Acquire PCB's pthread_lock. Check if ref_cnt is now 0, and if so, remove pthread from PCB's pthread_list, and free pthread.
3. Decrement PCB's num_active_pthreads, and check if it is 0. If so, call sema_up on PCB's pthread_sema to wake up main (no remaining child processes). Release PCB's pthread_lock.
4. Finally, call thread_exit().

Helper Function: PThread Terminate Main

```
void pthread_terminate_main(bool from_proc_exit)
```

1. Call sema_up() on the main pthread's join_sema, allowing other threads to join on main.
2. Acquire main's pthread's ref_lock, decrement pthread's ref_cnt and release ref_lock.
3. Acquire pthread's ref_lock, decrement ref_cnt, and release ref_lock. Acquire PCB's pthread_lock. Check if ref_cnt is now 0, and if so, remove pthread from PCB's pthread_list, and free pthread. Get and decrement PCB's num_active pthreads and release PCB's pthread_lock.
4. If num_active_pthreads is not 0, call sema_down on PCB's pthread_sema to sleep to join on remaining pthreads.
5. Set PCB's pthread_main_exit to true. If parameter from_proc_exit is not true, call process_exit().

Helper Function: Process Exit

```
void process_exit(void);
```

1. Acquire PCB's terminated_lock, set PCB's terminated to true if it is false, and release the lock.
2. If current_thread is not the main thread, call pthread_termanate(), which does not return. Otherwise, if PCB's pthread_main_exited boolean is false, call pthread_terminate_main(true).
3. Free all the PCB's pthread structs from pthread_list.

Interrupt Handler: Exit Pthreads

```
void intr_handler(struct intr_frame* frame);
```

Before the end of the function

1. Check if is_trap_from_userspace(frame) and PCB's terminated bool is true. If so, call process_exit().

System Call: Exit, Wait, Exec

With the existing code and proposed design, the exit, wait, and exec syscalls require no additional modifications.

Exception: Exit Code
In exception.c/kill function, before process_exit() is called, set the PCB's proc_info's exit code to -1.

System Call: PThread Join

```
void syscall_pthread_join(tid_t tid);
```

1. Call pthread_join(tid), and store returned value in EAX register.

Helper Function: PThread Join

```
tid_t pthread_join(tid_t tid);
```

1. Call get_pthread(tid) to get the thread to join on. If NULL returned, return TID_ERROR.
2. Acquire pthread's ref_lock. If pthread's joined_on is true, release ref_lock and return a TID_ERROR (thread already joined on). Otherwise, set joined_on to true and release ref_lock.
3. Next, we call sema_down() on the pthread's join_sema to join on it.
4. After waking up, acquire pthread's ref_lock and decrement pthread's ref_cnt by 1, then release ref_lock. If ref_cnt is now 0, remove the pthread from PCB's pthread_list and free it.
5. Finally, return pthread's tid.

System Call: Get TID

```
void syscall_get_tid(void);
```

Call thread_current() and store TID of thread struct in the EAX register.

Helper Function: Get User Lock

```
struct lock *get_user_lock(char lock_id);
```

Acquire the current PCB's user_locks_lock, then search user_locks list for lock associated with lock_id. If lock found, return pointer to lock. Otherwise, return NULL. Release the current PCB's user_locks_lock before returning in either case.

System Call: Lock Initialize

```
void syscall_lock_init(lock_t *lock);
```

1. If lock == NULL, set EAX register to false and return. Otherwise, acquire the current PCB's user_locks_lock.
2. If PCB's next_lock_id == 0 and user_locks list is not empty, release user_locks_lock, set EAX register to false, and return. Condition checks if 255 locks have already been created.
3. Malloc space for new user_lock. If malloc fails, release user_semas_lock, set EAX register to false and return. Otherwise, push user_lock to back of the user_locks list. Set user_lock's lock_id to the PCB's next_lock_id, and initialize the lock. Increment the PCB's next_lock_id.
4. Set *lock to the user_lock's lock_id. Release user_locks_lock, set EAX register to false, and return.

System Call: Lock Acquire

```
void syscall_lock_acquire(lock_t lock);
```

1. Get pointer to lock by calling get_user_lock() on user's lock. If lock not found, set EAX register to false and return. If lock_held_by_current_thread() is true, set EAX register to false and return.
2. Call lock_acquire() on the lock, then set EAX register to true and return.

System Call: Lock Release

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

1. Get pointer to lock by calling get_user_lock() on user's lock. If lock not found, set EAX register to false and return. If lock_held_by_current_thread() is false, set EAX register to false and return.
2. Call lock_release() on the lock, then set EAX register to true and return.

Helper Function: Get User Semaphore

```
struct sema* get_user_sema(char sema_id);
```

Acquire the current PCB's user_semas_lock, then search user_semas list for semaphore associated with sema_id. If semaphore found, return pointer to semaphore. Otherwise, return NULL. Release the current PCB's user_semas_lock before returning in either case.

System Call: Semaphore Initialize

```
void syscall_sema_init(sema_t *sema, int val);
```

1. If sema == NULL or val < 0, set EAX register to false and return. Otherwise, acquire the current PCB's user_semas_lock.
2. If PCB's next_sema_id == 0 and user_semas list is not empty, release user_semas_lock, set EAX register to false, and return. Condition checks if 255 semaphores have already been created.
3. Malloc space for new user_sema. If malloc fails, release user_semas_lock, set EAX register to false and return. Otherwise, push user_sema to back of the user_semas list. Set user_sema's sema_id to the PCB's next_sema_id, and initialize the semaphore to val. Increment the PCB's next_sema_id.
4. Set *sema to the user_sema's sema_id. Release user_semas_lock, set EAX register to false, and return.

System Call: Semaphore Down

```
void syscall_sema_down(sema_t *sema);
```

1. Get pointer to semaphore by calling get_user_sema() on user's semaphore. If semaphore not found, set EAX register to false and return.
2. Call sema_down() on the semaphore, then set EAX register to true and return.

System Call: Semaphore Up

```
bool sema_up(sema_t* sema);

void syscall_sema_up(sema_t *sema);
```

1. Get pointer to semaphore by calling get_user_sema() on user's semaphore. If semaphore not found, set EAX register to false and return.
2. Call sema_up() on the semaphore, then set EAX register to true and return.

Synchronization

Synchronization Schemes

- In the PCB struct, reference these following synchronization schemes to prevent data races.
  - The user_locks and next_lock_id are variables shared between pthreads. The lock user_locks_lock is used to synchronize data.
  - The user_semas and next_sema_id are variables shared between pthreads. The lock user_semas_lock is used to synchronize data.
  - The terminated variable is shared between pthreads. The lock terminated_lock is used to synchronize data.
  - The pthread_list and pthread_num_active is shared between pthreads. The lock pthread_lock is used to synchronize data.

- To allow the main thread to join on all other threads, the pthread_sema is used. The semaphore is initialized to 0. Hence, the main thread calls sema_down() and the final exiting thread calls sema_up() to ensure synchronization.
- The pthread_main_exit does not require synchronization, as only the main thread edits the variable. The main thread cannot compete with itself.

- In pthread, the ref_cnt and joined_on variable is shared between pthreads, when joining and freeing the pthread struct. To synchronize joins and memory deallocation, the lock ref_lock is used. The ref_cnt attribute ensures the struct stays allocated until references to it are required. Moreover, ref_cnt ensures only one pthread will free the struct.
- In pthread, the join_sema is used to synchronize join attempts between pthreads. The semaphore is initialized to 0. Hence, the joining thread calls sema_down() and the joined_thread calls sema_up() to ensure synchronization.
- Inside process_exit(), only main thread is running and all other pthreads have exited. Hence, deallocation of memory and accessing previously shared variables does not require synchronization.
- The user_lock and user_sema structs require no synchronization as the contained data is constant after initialization.

Impact of Synchronization

- The synchronization schemes involving locks means only 1 thread can run in critical sections. This limits thread concurrency, but critical sections are kept small to minimize serial computation.

Rationale

- The proposed design is not very easy to conceptualize and understand. Introducing user threads requires lots of synchronization schemes and modifications to the existing code structure. However, new code is abstracted across many helper functions to ensure code modularity and flexibility for additional features. The complexity of synchronization means a considerable quantity of code must be written to implement user threads.
- For user locks and semaphores, 2 different designs were considered.
  a. Locks and semaphores are stored in a Pintos list. The Pintos list uses O(N) space, where N is the number of locks/semaphores. Each time a lock/semaphore is initialized, it is added to the list. This takes O(1) time. Each lock/semaphore operation (acquire, release, up, down) takes O(N) time, where N is the number of locks/semaphores, since the Pintos list must be searched to find the corresponding lock/semaphore.
  b. Locks and semaphores are stored in an array of size 256. Each time a lock/semaphore is initialized, it is inserted into the array. The array takes up O(M) space, where M is the maximum number of locks/semaphores. This takes O(1) time. Each lock/semaphore operation also takes O(1) time, as the array can be indexed in constant time.

  In the end, our group decided to use the first approach. Firstly, the maximum number of locks/semaphores is small (256), hence searching a Pintos list does not take much time. Moreover, the second approach uses considerably more memory for every process. Very few processes will require so many locks and semaphores, hence the first approach is more reasonable.
- For pthreads, our group considered storing the pthread information inside the thread struct instead of a separate pthread struct. However, this approach was decided against due to memory constraints. Every thread struct is stored inside a thread's kernel stack. Hence, it is preferable to keep the size small so usable stack space is largely. Additionally, not every thread is a pthread, so the allocated memory would be wasted. Instead, the approach of mallocing space for separate pthread structs is a more reasonable approach.

Concept check

Question 1

The `thread_exit` function cannot free the current thread's kernel page since `thread_exit`'s stack frame is inside the current thread's kernel page. Instead, `thread_exit` removes the thread from the all_threads list, sets the thread's status to THREAD_DYING, and calls schedule(). Inside schedule(), a call to switch_threads() switches to a different kernel page. Hence, it is now safe to call palloc_free_page() on the dying thread's kernel page. This is done inside thread_switch_tail().

Question 2

The intr_handler() function executes inside the kernel stack of the interrupted thread. Hence, subsequent calls to timer interrupt handler and thread_tick will be inside the kernel stack of the interrupted thread.

Question 3

Thread A runs acquires lock A. Then, thread B runs acquires lock B. Then, thread A tries to acquire lock B and falls asleep. Finally, thread B tries to acquire lock A and falls asleep, ensuing a deadlock situation.

Question 4

If thread B is forcibly killed, it may leave the Kernel in an unsafe state. Consider the following possibilities.

1. Thread B has allocated memory in the kernel heap. After being forcibly killed, this memory remains allocated, resulting in a memory leak.
2. Thread B has acquired a kernel lock. After being forcibly killed, thread B was unable to release the kernel lock, meaning no other thread can acquire the lock (dead-lock situation).
3. Thread B is writing to a file. After being forcibly killed, thread B may have left the file in a corrupted state, which can lead to future issues.

Question 5

1. Consider the following setup:
   a. Resources: 1 semaphore (initialized to 0) and 1 lock.
   b. 4 Threads: A with priority 3, B with priority 2, C with priority 1, D with priority 0.
2. Test description and expected output
   a. Thread C is created and acquires the lock. Then, thread C creates thread A, B, and D.
   b. Thread A runs (highest priority). Thread A tries to acquire the lock, donates its priority to C, and goes to sleep. Thread C has effective priority 3.
   c. Thread C runs (highest priority), downs the semaphore, and goes to sleep.
   d. Thread B runs (highest priority). Thread B downs semaphore and goes to sleep.
   e. Thread D runs (highest priority). Thread D ups semaphore. Thread C has a higher effective priority than thread B, so it should wake up. Thread C runs and exits the process with exit code 0.
3. In the actual implementation, in step e, thread B is woken up first as it has a higher base priority than thread C. When thread B runs, it exits the process with exit code 1.