Project Threads Design

Group 107 (replace X with your group number)

Name	Autograder Login	Email
Yuanxiu Wang	Student364	iamwangyuanxiu@berkeley.edu
Qirui Ruan	Student153	ruanqirui@berkeley.edu
Marvin Hsin	Student243	bdfb1997@berkeley.edu
Tianxiang Liu	student156	ltx@berkeley.edu

Efficient Alarm Clock

Data Structures and Functions

```
/* New structures */
struct list sleep_threads;
struct thread {
  int64_t wake_time;
};


/* Modification */
static void init_thread(struct thread *t, const char *name, in
t priority){
  t->wake_time = -1;
}
void thread_init (void){
  list_init(&sleep_threads);
}
```

1. Maintain a global list that keeps track of sleeping threads.
2. Add new member that stores the wake up time of a thread in struct thread.

Algorithms

The main task is to improve the efficiency of timer_sleep so that we won't waste CPU cycles.

```
void timer_sleep(int64_t ticks) {
  disable interrupt;
  calculate wake up time and update;
  block the thread;
  insert into sleep_threads by wake up time;
  enable interrupt
}

/* helpers */
void thread_block(void);
static bool sleep_less_than(const struct list_elem *x, const struct list_elem *y, void *aux);
void list_insert_ordered(struct list*, struct list_elem*, list_less_func*, void* aux);
```

In order to really improve its performance, we need to actually put threads into sleep rather than yielding. We introduced new member wake_time to represent when the thread will wake up. By creating a list to store all sleeping threads, we have all sleeping threads with their different wake_time. After blocking the thread, importantly we need to insert it in the order that least wake_up time first so that we can pop the thread in-order as timer ticks in the future. To do so, we need a comparison function **sleep_less_than** to pass in as an argument to **list_insert_ordered**. Last but not least, we should disable interrupt when operating global list to prevent from data race.

```
static void timer_interrupt(struct intr_frame* args UNUSED) {
  thread_tick();
  disable interrupt;
  while (sleeping threads is not empty) {
    pop a sleeping thread from the list
```

```
    if (ticks reaches thread's wake up time) {
      unblock the thread
    } else {
      insert back into sleep_threads;
      break;
    }
  }
  enable interrupt
}

/* helpers */
void thread_unblock(struct t*);
struct list_elem* list_pop_front(struct list*);
```

timer_interrupt checks if there is any thread waking up when timer ticks. Firstly, we tick the timer. Then, we check if there is any sleeping threads in the list. If there is one, retrieve it and see if it's the wake time. At this point, we do not need to worry about which thread to pop because they are inserted in timer order. If it's the wake time, unblock the thread. Otherwise, we don't do anything. The same as **timer_sleep**, interrupt should be disabled while we operate the global list.

Synchronization

The synchronization issue is dealt by enabling and disabling interrupt when adjusting sleep_threads. We need to make sure that the list insertion and popping is an atomic operation so that we don't run into data race.

Rationale

Initially, we think the synchronization could be solved using a lock, but we discover that disabling interrupt is a simpler way to reach the same goal. Through blocking and unblocking, the threads go idle, so we do not waste CPU cycles to schedule them and keep yielding them.

Strict Priority Scheduler

Data Structures and Functions

thread.h

```
//Modification
struct thread {...
  struct list_elem elem;     /* List element. */
  bool lock_init;
  int lock_index;
  int priority;
  struct lock* held_locks[30];   /* Held locks. */
  struct lock* waiting_lock; /* Lock this thread is waiting o
n. */
  static struct list effective_priority_list; /* Stores thread
s that donated to this thread. Has thread, priority, lock. All
ows effective priority to be calculated on the fly. */
  static struct list donated_list; /* Stores {threads, lock} t
his thread has donated to. */
};
```

thread.c

```
int get_effective_priority(struct thread t); /* new func to ge
t thread's effective priority, iterates through effective_prio
rity_list and returns max(t->priority, max effective_priority_
list priority) */


tid_t thread_create(const char* name, int priority, thread_fun
c* function, void* aux)
// We want to add thread_yield() here


void thread_set_priority(int new_priority)
// We want to add thread_yield() here


int thread_get_priority(void)
```

```
void thread_exit(void)//kill threads
static void init_thread(struct thread* t, const char* name, in
t priority)//init elems


static struct thread* thread_schedule_prio(void)
/* logic for iterating through fifo_ready_list and choosing hi
ghest effective priority thread in this function. */


static struct thread* thread_schedule_fair(void)


static void thread_enqueue(struct thread* t) /* Use this to pu
sh threads that are ready onto fifo_ready_list */


//New Implementation
  struct thread* get_thread(tid_t tid) //helper, return the co
rresponding thread
```

synch.c

```
void sema_up(struct semaphore* sema) // We want to add thread_
yield() here
```

- We use a ready list like fifo_ready_list as a queue
  - thread_enqueue() pushes threads that are ready to run onto this list
  - the list will be sorted by their effective priority. Using thread_get_priority and thread_set_priority, we can use these priority values.
- Since we want to keep track of both effective and base priorities, we will be adding 2 new data members to the thread struct: static struct list effective_priority_list and static struct list donated_list. effective_priority_list stores {threads, priority, locks} of the threads that donated to this current thread to allow us to calculate the effective priority on the fly, while donated_list stores {threads, locks} of the threads that this thread has donated to.

Algorithms

static void thread_enqueue(struct thread* t)

- Add thread to our fifo_ready_list according to sched_policy

int get_effective_priority(struct thread t);

- new func to get thread's effective priority, iterates through effective_priority_list and returns max(t->priority, max effective_priority_list priority)

static struct thread* thread_schedule_prio(void)

- iterates through fifo_ready_list and returns the thread with the highest effective priority
- uses int get_effective_priority(t) to calculate its effective priority
- if there's nothing, then return idle thread by default

thread_create

- Synchronize child thread with parent thread's exec_info_struct
- Store init struct into stack frame
- Yields to the new thread if it has higher priority than the currently running thread

thread_set_priority

- Set the base priority value of the thread
- if it called thread_set_priority with a low value or it released a lock, it must immediately yield the CPU to the highest-priority thread.

thread_get_priority

- gets the base priority value of the thread

lock_acquire

- if lock holder effective priority < thread trying to acquire lock
    - set lock holder's priority = current thread's priority

lock_release

- Pop entry with highest effective priority from the effective priority list unless effective priority = base
- We want to allow thread to keep running after it gives up the lock. Need to set priority back down if required.

sema_down

- ensure semaphore's waiting list is in decreasing priority
- use list_insert_ordered

condition variables

- cond_wait: use list_insert_ordered
- cond_signal: wake up thread with highest priority

Priority Donation

- get the effective priority by calling `get_effective_priority`
- compare the current with the running one
- if the running thread has a lower-priority,
  - want to temporarily raise its priority to the current one's priority
  - call `lock_acquire`
  - call `lock_release`
  - make sure donation is undo when a lock is released
- If a thread call `thread_set_priority` with a low value or it released a lock
  - call `thread_yield` to yield the CPU to the highest-priority thread if exists
- else if the running thread has a higher-priority,
  - wait
- If a thread acquires multiple locks, it lowers its effective priority as it releases these locks with the help of `get_effective_priority` and `lock_release`
- For multiple sources donation,
  - use a semaphore initialized to 1 to ensure there is a single owner of the resource that a thread can donate to
  - call `sema_up` and `sema_down`
- For nested/recursive donation,
  - use *effective_priority_list* stored info and `list_sort`
  - make sure updating the changes and donations

Synchronization

By using the effective_priority_list, we make sure that the scheduler always run the one with the highest effective priority. And by having the priority donation system, it prevents the priority inversion problem does not occur. We acquire the locks and ensure that it is not being held by another thread before we run the new thread. If it is being held, we set the current lock holder's priority to new thread's.

Rationale

To take care of the edge cases of priority inversion such as multiple sources donating and nested donation, we store a list of the threads that donate to the current one in effective_priority_list so that the effective priority can be calculated on the fly. Majority of the implementations and functions were already provided in the skeleton, and are pretty self explanatory → just have to follow the comments. We do need to add some data to the thread structs. Since sorting the fifo_ready_list is time consuming, we instead have the effective priority be calculated on the fly when it is necessary.

User Threads

Data Structures and Functions

```
tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg);//syscall.c
bool setup_thread(void (**eip)(void) UNUSED, void** esp UNUSED, thread_node *node);
tid_t pthread_execute(stub_fun sf UNUSED, pthread_fun tf UNUSED, void* arg UNUSED);
static void start_pthread(void* exec_ UNUSED);

void sys_pthread_exit(void) NO RETURN;//syscall.c
void pthread_exit(void);
void pthread_exit_main(void);

tid_t sys_pthread_join(tid_t tid);//syscall.c
tid_t pthread_join(tid_t tid UNUSED);

tid_t get_tid(void);

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);

static void syscall_handler(struct intr_frame* f);/*get the new syscall args and push the return values into the functions above*/

/*struct changes*/
struct process {
  ...
  struct list usrsemaphores; /*keeps the list of all user semaphores in the pcb*/
  struct list usrlocks; /*keeps the list of all user locks in the pcb*/
  struct list usrthreads; /*keeps the list of all user threads in the pcb*/

  /*single process locks that protects information from editing by multiple threads*/
  /*when changing information in the specified lists, obtain these locks*/

  struct lock childlock;/*protects the list of children*/
  struct lock semalock; /*protects the list of semaphores*/
  struct lock locklock;/*protects the list of locks*/
  struct lock threadlock;/*protects the list of threads*/
  struct lock pagelock; /*protects the pagedir*/
};
```

```
/*acts as nodes within the list of user locks in a single process*/
struct lock_node {
  struct lock_t* usrlock;
  struct lock* kernellock;
  struct list_elem lockellem;
}

/*acts as nodes within the list of user semaphores in a single process*/
struct sema_node {
  struct sema_t* usrsema;
  struct semaphore* kernelsema;
  struct list_elem semaellem;
}

/*acts as nodes within the list of user threads in a single process*/
struct thread_node {
  tid_t tid; /*the tid of the user thread*/
  void *esp;/*the saved user stack page pointer of the user thread*/
  struct semaphore status;/* sema = 0: thread alive; sema = 1: thread terminated*/
  bool waited; /*whether the thread is being waited upon*/
  struct list_elem userellem; /*belongs to the list of the user threads in the pcb*/
};
```

```
/*passes information when creating threads, gets passed into t
he start_pthread as a pointer argument*/
struct thread_exec {
  stub_fun sfun;//stub function to run
  pthread_fun tf;//function to execute
  void* arg;//pointer arguments
  struct process* pcb;//current pcb
  struct semaphore load_done;//whether loading the function is
done
  bool load_success;//whether the load to the function is succ
essfull
};

/*Fixing old syscall implementations*/

static void start_process(void* exec_); /*initialize the lock
and the list of threads when initializing the pcb*/

/*acquire locks when trying to change information in the pcb*/
pid_t process_execute(const char* file_name);
int process_wait(pid_t child_pid);

void process_exit(void); /*fix it so that it would terminates
all user threads in the process*/

tid_t thread_exit_specified(tid_t tid); /*terminates the threa
d being specified. If the thread being specified is the curren
t thread, return tid_error */
```

Algorithms

```
tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, cons
t void* arg) ;
```

Calls **pthread_execute** with the same arguments.

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

1. initialize a struct thread_exec, and put the arguments to the function into the struct.
2. put the current thread's pcb into the thread_exec struct.
3. initialize the semaphore in thread_exec to:
   sema_init(&thread_exec→load_done, 0);
4. give an arbitrary name to the created thread.
5. call **thread_create**(name, PRI_DEFAULT, start_pthread, thread_exec*) and retrieve the tid returned by it.
6. **sema_down**(&thread_exec→load_done) to wait for the thread to load.
7. after that, check whether **load_success** is true. If it's true, return the tid returned by the thread_create function. Else, return tid_error.

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

1. cast the pointer passed in as a thread_exec pointer.
2. point the pcb pointer in the thread_exec to the current thread.
3. allocate the new **thread_node** and put:
   thread_node→tid = thread_current()→tid;
   sema_init(&thread_node→status, 0);
   thread_node→waited = false;
4. if allocation fails, put **load_success = false** and
   sema_up(&thread_exec→load_done) before calling thread_exit()
5. initialize the intr_frame as in the start_process function.
6. call the setup_thread function on initialized intr_frame eip and esp, and the thread_node pointer, and return whether the load was successful.
7. if successful, put locks around changing the pcb, and push the thread_node into the list of the thread's pcb.
8. Then, notify by putting **load_success = true** and
   sema_up(&thread_exec→load_done).
9. if not sucessful, free the thread_node pointer and put **load_success = false.** Then,
   sema_up(&thread_exec→load_done) before calling thread_exit().

10. start the user process by jumping to the user frame as in the start_process function.

```
bool setup_thread(void (**eip)(void) UNUSED, void** esp UNUSE
D, thread_node *node);
```

1. retrieve a new user page by calling **palloc_get_page**(PAL_USER | PAL_ZERO) in palloc.c.
2. acquire the pagedir lock before changing the page directory.
3. iterate from the PHY_BASE with step size equal to page size, to retrieve the top page is not mapped into the page directory. Map that page to the new page by calling pagedir_get_page in pagedir.c repeatedly until the function returns NULL.
4. After that, map the new page to the top page by calling pagedir_set_page in pagedir.c. If the function fails, release the lock, free the new page, then return false to indicate load failure.
5. else, release the lock, save the stack page pointer owned by the thread to the esp variable in the thread_node.
6. use **memcpy** to push the two pointers first arg then tf to the location below the new page one by one. Then, push a null pointer as a return address. Finally, set esp to be the new page address + pgsize - (sizeof(tf) + sizeof(args) + sizeof(return pointer))
7. set eip to be the pointer of the stub function.
8. return true to indicate load success.

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

1. check if the current thread is the main thread. If not, call pthread_exit(void).
2. else, call pthread_main_exit(void).

```
void pthread_exit(void);
```

Find its own thread node from the list of threads. Acquire the pagedir lock. Free its own user stack by calling **palloc_free_page** on its stack page pointed by the esp pointer in thread node**.** Release the lock. Then, **sema_up**(&thread_node→status). Finally, simply call **thread_exit()**.

```
void pthread_exit_main(void);
```

This is being called by the main thread.

Acquire the process thread lock. iterate through the list of threads in the pcb. For each node in the list of threads, call **sys_pthread_join(tid**) on it. Release the lock. Then, call **process_exit().**

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

calls pthread_join using the same arguments.

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

1. put thread locks when iterating through the list of threads.
2. iterate through the list of threads and find the given thread_node. If the tid is not found, return tid_error.
3. if the thread node is found, check the wait boolean on the thread node. If it's true, it means that another thread is already waiting on the thread. release the lock and return tid_error.
4. set the wait boolean to true. release the lock. then, **sema_down**(&thread_node→status) to wait for the thread to terminate.
5. afterwards, remove the thread_node from the list and free it.
6. return the given tid.

```
tid_t get_tid(void);
```

returns thread_current()→tid to retreive the current thread's tid.

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

1. create a lock struct in the kernel context and initialize the kernel lock.
2. allocate a lock_node struct and put the user lock_t pointer and the kernel lock pointer into the struct.
3. if allocation fails, return false.
4. put the lock process lock around changing the pcb. then, push the lock_node struct into the pcb list.
5. return true.

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

1. acquire the lock process lock, then look up the lock_node struct in the list of user locks.
2. if there is no such struct in the list, release the lock and return false.
3. else, release the process lock and call **lock_acquire** on the mapped kernel lock.

4. then, return true.

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

1. acquire the lock process lock, then look up the lock_node struct in the list of user locks.
2. if there is no such struct in the list, release the lock and return false.
3. else, release the process lock and call **lock_release** on the mapped kernel lock.
4. then, return true.

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

1. create a semaphore struct in the kernel context and initialize the kernel semaphore using the specified value.
2. allocate a semaphore_node struct and put the user sema_t pointer and the kernel semaphore pointer into the struct.
3. if allocation fails, return false.
4. put semaphore locks around changing the pcb. then, push the semaphore_node struct into the pcb list.
5. return true.

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

1. acquire a semaphore process lock, then look up the semaphore_node struct in the list of user semaphores.
2. if there is no such struct in the list, release the lock and return false.
3. else, release the process lock and call **sema_down** on the mapped kernel semaphore.
4. then, return true.

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

1. acquire a semaphore process lock, then look up the semaphore_node struct in the list of user semaphores.
2. if there is no such struct in the list, release the lock and return false.
3. else, release the process lock and call **sema_up** on the mapped kernel semaphore.
4. then, return true.

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

Add initialization to the new process’s lists.

list_init(&t->pcb->threads)
list_init(&t->pcb->locks)
list_init(&t→pcb→semaphores)
initialize all the locks that protects the different lists.
allocate a new thread_node struct for the main thread and push it into the thread list. map the main process's esp pointer in the thread_node as null, since calling exit on the main thread simply exits the program eventually.

```
pid_t process_execute(const char* file_name);
```

Add child locks when pushing the child information to the thread's pcb.

```
int process_wait(pid_t child_pid);
```

Add the child process lock around iterating through the child list and removing the child node, so that when multiple threads within the same process try to access the same pcb's data, no conflicts are reached.

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

Acquire the pcb's thread lock when trying to terminate all the threads in the pcb so that pcb data is preserved.
Add a loop to iterate through the list of the threads and terminate them.
For each thread_node, check value of the status semaphore by calling **sema_try_down**. If it fails, that means the semaphore value is 0 and the thread is still running. Exit the specified thread by calling **thread_exit_specified(tid)** on the tid.
Then, remove the thread_node from the list and free it.
This ensures that the current thread is the only thread running in the process.
After that, release the lock before freeing other structures in the pcb.
Additionally, free all the lock_node structs in the pcb. The user stacks of the other threads would be freed when destroying the pcb's pagedir.

```
tid_t thread_exit_specified(tid_t tid);
```

1. save the old interrupt status when calling **intr_disable()** to disable interrupt.
2. check if the thread being specified is the current thread by comparing tids. If they are equal, enable interrupt by calling **intr_set_level(old_level)** and return tid_error.
3. If not, iterate through the list of all threads **all_list** to find the thread with the specified tid.

4. remove the specified thread from the **all_list**.
5. set the specified thread's state to THREAD_DYING.
6. free the kernel stack of the thread by calling **palloc_free_page** on itself.
7. restore the interrupt status by calling **intr_set_level(old_level)**.
8. return tid from the function.

Synchronization

The resources inside the pcb are being shared across threads, because all threads within one process owns the same pcb. Thus, locks in charge of different lists are added in the process struct so that each change to the pcb would be protected.

a. The **process_execute, process_wait** and the **thread_join** functions access one set of read and write of the pcb's data to retrieve running information of the target process and thread. Locks need to be applied so that no conflicts can occur due to synchronization within threads that own the same process.
b. The **process_exit** function would terminate all threads before exiting the process. By adding a lock when terminating all threads within the process, this preserves the thread list to not get changed while iterating through it and killing all threads. After all threads are killed, it is okay to free the pcb structrue without using locks.
c. Since multiple threads would be able to change the same page directory in pcb, when creating a new thread and mapping its stack, acquire the page directory lock so that no conflicts are reached.
d. Currently, file operations still run within the global lock. No synchronization is dealt there.

While executing the function to kill the specified thread, it's crucial to disable interrupts so that the status of the target thread gets set properly and entirely.

Rationale

Another approach I thought of maintaining the list of threads was to add another list element in the thread struct. However, it would be very hard for the threads to maintain their execution information after they terminated. Thus, another struct called thread_node needs to be built externally for tracking the execution state of threads.

Concept check

1. The stack of the threads is freed after scheduling and after the new thread was already set to running. This ensures that the scheduling process still ran by the old thread goes well before switching to the new thread.
2. The ThreadTick function executes in the kernel stack of the running thread that was being called upon regularly, which could be any threads, including the idle thread.
3. ThreadA acquires the lockA and then threadB acquires the lockB. Later, threadA tried to acquire the lockB, but put to sleep. ThreadB tried to acquire the lockA and also put to sleep. In this case, they become deadlock.
4. ThreadB might be owning a lock or have downed a semaphore before being killed by ThreadA. The former creates a deadlock. If ThreadA tries to own the lock or down the semaphore afterwards, it would get stuck.
5. Implement two functions FunA and FunB with a global variable const.
   a. FunA tries to acquire a lock, then tries to down a semaphore, then changes the global variable const before upping the semaphore, releasing the lock and exiting.
   b. FunB ups the semaphore, then goes into a circular wait loop to wait for the global variable to get changed.

   Initialize the locks and semaphores and the global variable in the main function. First, create ThreadC with a low priority that runs FunA. Second, create a ThreadA with a high priority that runs FunA. Finally, create ThreadB with middle priority that runs FunB to so that it waits for either ThreadA or ThreadB to finish. In the case of priority donation being implemented in semaphores, the test would always terminate without getting stuck.

   However, when semaphore does not support priority donation, there would be a case when the program gets stuck. If ThreadC acquires the lock first with ThreadA waiting for it, it would never get to run because ThreadB always has a higher priority. The program would get stuck in the circular wait loop in ThreadB.