Project Threads Design

Group 24

Name	Autograder Login	Email
Sydney Tsai	student134	sydneytsai@gmail.com
Anusha Muley	student143	anusha.muley@berkeley.edu
Cindy Yang	student258	cindycy@berkeley.edu
Qi Li Yang	student85	qiyang2@berkeley.edu

Efficient Alarm Clock

Data Structures and Functions

- list of threads that are currently sleeping
- lock on the list so we only yield one at a time

Locations:
`src/devices/timer.c`
`src/threads/thread.h`

in `src/threads/thread.h`

```
// keep track of how many ticks till wake for each thread
struct thread {
  ...
  int_64 ticks_till_wake; // at how many ticks since start the thread needs to wake
}

// also use from thread.h:
void thread_block(void);
void thread_unblock(struct thread*);
struct thread* thread_current(void);
```

in `src/devices/timer.c`

```
struct list sleeping_threads; // keep track of all the queued lists currently asleep

// references the elem list_elem parameter of struct thread

struct lock sleeping_threads_lock; // lock to access and pop elements off the sleeping threads list
```

Algorithms

- be able to block the current thread and allow another thread to run (thread_yield allows the scheduler to schedule the same thread over and over again even when not enough time has passed for them to wake up)
- make sure to turn on interrupts
- In timer.c `timer_init()` add sleeping threads list initialization
- In thread.c `init_thread()` add ticks_till_wake initialization?
- In `thread_sleep()`

```
    - set ticks_till_wake = thread_current()->ticks_till_wake //rem

    - lock_aquire(&sleeping_threads_lock);

    - list_insert(&sleeping_threads, thread_current→list_elem)// order based on ticks_till_wake

    - lock_release(&sleeping_threads_lock)

    - thread_block();
```

- In timer.c `timer_interrupt()`

```
while ticks == sleeping_threads_ticks[0]->ticks_till_wake:
// pop multiple if alarm for same tick

    - lock_aquire(&sleeping_threads_lock);

    - thread = list_pop_front(&sleeping_threads); // list ordered

    - thread_unblock(thread_id);

    - lock_release(&sleeping_threads_lock);

// keep original yield logic below
```

Synchronization

We are synchronizing the timer_interrupt which could mistakenly pop multiple elements off if we don't synchronize the access to the list. We have a lock that must be acquired before we insert and release elements from the list. Interrupts should also be disabled while we are modifying the list to ensure synchronization.

Rationale

We chose to keep an ordered list of the sleeping thread so we don't have to parse it every time we want to pop off an element. We have a lock over that list for synchronization. We initialize the sleeping threads list when we create the timer with `timer_init()` where we also initialize the lock over the list. We are keeping track of each threads specific wake time by including a `ticks_to_wake` parameter in the thread structure that is initialized with `init_thread()` so that each thread can then be added to the sleeping threads list during `thread_sleep()`. We chose to also use the thread block/unblock and remove the `thread_yield()` method to ensure we only wake a thread when it needs to run to get rid of busy waiting and not reawakening threads before they need to be woken up.

Strict Priority Scheduler

Data Structures and Functions

```
struct process {
  ...
  list all_threads; // list of all the threads per process
  int time_quant; // for our round robin, but we don't really
need to worry about this


}
```

Add a linked list of all threads (list contains struct threads) running this process so that we can traverse this list to get priorities and determine which thread should run next. Add a `int time_quant` variable so that threads with the same priority can be executed in a round robin fashion.

```
struct lock {
  ...
  list waiting_threads;
}
```

Add a linked list of threads that are waiting on this lock, so that we can traverse their priorities when we are donating priorities.

```
static struct thread* next_thread_to_run(void) {
  struct thread for max_thread
  for each thread in allelems {
      if thread has higher priority than max_thread:
        save max_thread as current thread
  }
  return max_thread
}
```

Modify `next_thread_to_run` so we traverse the linked list in the PCB and return the thread with the highest effective/temp priority.

```
int thread_get_priority(void) { //take donations into account
  if current thread has an effective/temp priority, return tha
t value
  else return the priority int
}
```

Modify the `thread_get_priority()` function so that we return the effective/temp priority of the thread (when a donation occurs) rather than just the donation stored in the priority variable from the thread struct.

```
static struct thread* thread_schedule_priority(void)
```

New function that returns the next thread struct that should be run based on priority values. This function will be added to the `scheduler_func* scheduler_jump_table`. We will also need our `enum sched_policy active_sched_policy` to be set to this new priority scheduler function.

```
struct thread{
```

```
  ...
  int effective_priority; //effective priority
  int base_priority;
  // we can also just keep track of the specific lock that we
are blocked on
  //list of locks we have
  //reference to lock we are blocked on
  list donated_priority_to; // owner and iterate on waiters
  list got_priority_from;
}
```

Add a temp_priority variable in the thread struct so that we can keep track of donated priorities (this is separate from the `int priority` which serves as the base priority that is already in struct thread). We also need a list of threads that we have donated priority to so that we can compute effective/temp priority in nested donations. Lastly, we will need a list of threads we have received priority from.

```
struct lock{
  struct list_elem waiting_threads; // so we can keep a list p
er thread of all the locks held
}
```

```
void thread_set_priority(int new_priority)
```

Set the current thread's priority to the new priority (make sure to update the temporary priority as well). Can only act on current thread. If the current running thread no longer has the highest priority, yield it.

- have interrupts disabled in this section

```
void lock_acquire(lock_t* lock)
```

Modify lock_acquire so lower priority threads have to wait on the lock if a higher priority thread needs it (Priority is based on effective/temp priority). If there is a thread that is currently holding onto this lock with a lower effective priority, we will need to donate our priority to them. Once we have received the lock, make sure to update the structs with the correct/updated data.

```
void lock_release(struct lock* lock)
```

Once the lock has been released, we need to update the waiting threads list by removing this thread. We will also need to revert the effective priority if a priority was donated to this thread.

- will also have to yield if something else has a higher priority

```
void donate_priority(struct thread* donor_thread, struct threa
d* receiver_thread)
```

So high priority threads are able to donate their priority to lower threads that are holding onto a resource that the high priority thread needs.

```
void sema_up(struct semaphore* sema)
```

We will need to modify the `sema_up` function so that we can implement priority scheduling on threads that are waiting on a semaphore with a value less than 1 (Priority is based on effective/temp priority).

```
cond_signal(struct condition* cond, struct lock* lock UNUSED)
```

We will need to modify the `cond_signal` function so that we can implement priority scheduling on threads waiting for a signal to wake up (Priority is based on effective/temp priority).

```
tid_t thread_create(const char* name, int priority, thread_fun
c* function, void* aux)
```

We will need to modify `thread_create` so that newly created threads with higher priorities cause a preemption on the current running thread.

Algorithms

```
static struct thread* next_thread_to_run(void) {
  traverse all_threads list in pcb and return the thread with
the highest priority
}
after getting the next thread, we should also call thread_yiel
d to pass the current thread and the new thread we want to run
```

Traverse run queue in PCB to find thread with highest temporary/effective priority and return that struct thread. This will be the thread that the scheduler chooses to

run.

```
void lock_acquire(lock_t* lock)
- if lock not init = ret error
- if lock == 1
  - thread_block(thread_current())
  - add thread to waiting list (should be def in priority sect
ion)
- else if lock == 0
  - if waiting queue empty set lock to 1 and return 1
  - else thread_block(thread_current()) and
```

if current thread that is calling `lock_acquire` is also the thread that has highest temporary or actual priority in `waiting_threads` (found in struct lock), then allow current thread to acquire lock else thread will have to wait on lock
we should also disable interrupts in this area, so that it becomes a critical section

```
void sema_up(struct semaphore* sema)
```

instead of calling `thread_unblock` on the waiter thread that is at the front of the `sema→waiters` list, we want to loop through `sema->waiters` and find the thread with the highest temporary priority. Then, we will remove this waiter thread from the list and call `thread_unblock` on it. If this newly unblocked thread has higher priority than the current running thread, we must yield the current running thread.

```
cond_signal(struct condition* cond, struct lock* lock UNUSED)
```

instead of calling `sema_up` on the waiter thread that is at the front of the `cond→waiters` list, we want to loop through `cond→waiters` and find the thread with the highest temporary priority. Then, we will remove this waiter thread from the list and call `sema_up` on it. If this newly unblocked thread has higher priority than the current running thread, we must yield the current running thread.

```
void donate_priority(struct donor_thread*, struct receiver_thr
ead*)
{
  set priority of receiver_thread to donor_thread priority
```

```
  join on receiver_thread (so we know the receiver thread is d
one running its job)
  revert the priority of the donor_thread
}
```

Function for threads to donate their priority (set the temporary/effective priority). We need to make sure that we take into consideration nested priority donation by traversing the list of threads it donated priority to.

```
struct thread* get_lowest_thread(list waiting_threads) {
  traverse through the list of waiting_threads
    find the lowest priority of the waiting_threads
  return lowest priority thread
}
```

Function to return the lowest priority thread

```
tid_t thread_create(const char* name, int priority, thread_fun
c* function, void* aux)
```

For `thread_create`, we will need to include a priority check (for the effective priority of the current running thread and this new priority of the created thread) so that if this newly created thread has a higher priority, the current running thread is preempted.

Synchronization

We are modifying the lock struct and lock_acquire function to allow for synchronization of threads with varying priorities.

We are initializing a time_quant integer to allow for round robin scheduling. If threads with the same priorities want to run, they can only run for the specified amount of time designated in the time_quant.

We don’t need synchronization with the new changes to the synchronization variables and also the scheduler. This is because only when the interrupts are disabled, can the sema waiting list (sema->waiters) and the list of waiting threads be edited. In addition, we will be locking shared resources such as the condition waiting lists.

- interrupts are disabled for scheduler, so we don't need to synchronize

Rationale

We decided to initialize a list of all threads waiting on a lock so it is easier for us to perform nested priority donation. Using this list alongside our `get_lowest_thread function`, we are able to easily find the lowest priority thread that holds a resource that will eventually allow our highest priority thread to run.

User Threads

Data Structures and Functions

Locations: lib/user/pthread.h

```
struct thread {
...
  if THREADS: // if the project is threads?
    uint8_t* stack_lower;      /* Saved stack pointer to lower boundary. */
  int effective_priority; //effective priority
  int base_priority;
  keep a reference to a page;
  list donated_priority_to;
  list got_priority_from;
}

struct process {
...
//user level locks and semas
  list all_locks; //list of all the locks and the respective lock_t so that we can use the char to index
```

```
  list all_semas; //list of all semaphores and respective user sema_t chars
  list thread_data; //shared data between threads for joining and exit status
  uint8_t* lower_bound; //keep track of the lower bound of the stack

  struct lock lock; //for thread operations
}

//shared data to keep track of exit statuses for joining threads
struct shared_data {
  tid_t tid;
  struct semaphore sema_exit;
  int exit status;
  struct list_elem elem;
}
```

wrapper functions

that will call corresponding syscalls

```
pthread_create
pthread_execute
pthread_exit
pthread_join
```

In src/userprog/syscall.c we need to add to the syscall handler to handle:

```
tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg)
void sys_pthread_exit()
tid_t sys_pthread_join(tid_t tid)
tid_t pthread_execute(stub_fun, pthread_fun, void*)
bool lock_init(lock_t* lock)
```

```
void lock_acquire(lock_t* lock)
void lock_release(lock_t* lock)
bool sema_init(sema_t* sema, int val)
void sema_down(sema_t* sema)
void sema_up(sema_t* sema)
tid_t get_tid(void)
```

Other functions we are using:

```
is_trap_from_userspace(struct intr_frame* frame)
process_activate()
thread_exit()
start_pthread()
thread_block()
install_new_page()
```

Algorithms

pthread syscalls

```
tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, cons
t void* arg) {
  return syscall3(SYS_PT_CREATE, sfun, tfun, arg);
}
```

To create a thread, we must initialize the thread struct and allocate space for it. We initialize all elements of the struct, activate the process and call pthread execute on Finally, we will return the tid variable for the newly created process thread.

In `pthread_create sycall`:

```
  - set tid = allocate_tid()
  - copy the pointer to the processes pcb and set the thread p
cb equal to it
  - initialize the thread struct
  - allocate a new stack in the processes space below the curr
ent lower bound and
```

```
    decrement the lower bound
  - decrement stack, set a new lower stack pointer, palloc a n
ew page using
  install_new_page()
  - set priority to default (for both priority and effective p
riority)
  - add to priority scheduling threads list
  - create shared data struct, initialize variables, add to li
st in PCB
  - process_activate() to activate the process
  - thread_block() default
  - call pthread_execute(arg)
  - return the tid of this newly created thread
```

pthread_execute (called by pthread_create)

```
  - set all the arguments to create an execute struct to pass
to start_pthread
start_pthread():
  similar to start process, start and execute the function in
this new thread.
```

```
void sys_pthread_exit(void) {
  syscall0(SYS_PT_EXIT);
  NOT_REACHED();
}
```

In pthread exit syscall:

```
check thread_current() == cur->pcb->main_thread
if main thread:
  - call pthread_exit_main
else
```

```
  - deallocate stack by incrementing stack pointer and free pa
ge in process, remove
    from all_threads list in the PCB
  - ~~release locks for the thread~~
  - up its own exit semaphore and set exit status to true //do
this earlier?
    - find this in the list of shared data structs in the PCB
  - remove this thread from the list of all threads
  - call thread_exit()
```

pthread_exit_main():

```
  - up exit semaphore here too
- release locks (the locks for each user's threads are in a li
st in the pcb)
  - pthread_join on all the threads in the process (going thro
ugh the all threads list for the proc)
  - call process_exit w exit_code = 0
```

The sys_pthread_exit function will set the enum thread_status of the current thread to THREAD_DYING. Its stack will be deallocated and exit status and semaphore will be updated so that it can join. Moreover, this function will check if the calling user thread is the main thread. If so, all threads in the process specific list_elem allelem should be joined and exited.

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

Loop through linked list of the thread shared data in the PCB to find the corresponding shared data of the thread with the tid given. Block on the thread's exit semaphore to wait for the thread's exit status. There will be no blocking if the thread given has already terminated. If the exit status of the thread is 1, remove the thread's shared data struct from the list in the PCB and return its tid. Otherwise return TID_ERROR.

User-level synchronization syscalls

Locks & Semaphores:

We will create syscalls for each of the user-level functions, and inside each syscall we will call the low-level functions that were previously defined. We will also have corresponding user level locks/semaphores to low level lock/semaphores by mapping them with a char.

```
bool lock_init(lock_t* lock) // initialize new user created lo
ck
- malloc low level lock that will correspond to user lock and
assign it a char
- call a syscall to the low level void lock_init(struct lock
*);
- set lock to char to 'index' our new low level lock struct it
is referring to
- add lock to our list of locks in PCB
- return true if success, false if malloc/lock init fail
```

```
bool lock_acquire(lock_t* lock)
- check if the lock exists for this char pointer 'index' to th
e low level lock and that lock is valid
- call a syscall to the low level void lock_aquire(struct lock
*);
- return true if success, false if not valid/couldnt access ch
ar
```

```
bool lock_release(lock_t* lock)
- check if the lock exists for this char pointer 'index' to th
e low level lock and that lock is valid
- call a syscall to the low level void lock_release(struct loc
k*);
- return true if success, false if not valid/couldnt access ch
ar
```

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

```
- malloc struct semaphore to correspond to sema and assign it a char
- call a syscall to the low level void sema_init(struct semaphore* sema, unsigned value) setting the value to val
- set sema to char to 'index' our new low level sema struct it is referring to
- add sema to the list of semaphores in PCB
- return true if success, false if malloc or sema_init fails
```

```
bool sema_down(sema_t* sema)
- find low level semaphore corresponding to this char sema if it exists
- check to see if bool sema_try_down(struct semaphore* sema)
  - if not, thread_block
- else, call the syscall to the low level void sema_down(struct sempahore*)
- return false if sema is null or sema_down fails. else, return true
```

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

Same logic as sema_down but we call low level `void sema_up(struct semaphore*)` instead
after retrieving the low level semaphore that sema corresponds to.
Return false if sema is null, otherwise return true.

```
tid_t get_tid(void)
```

Return current thread's tid from its TCB

Changes to Process Control Syscalls:

1. Exec
   a. our current design already ensures there is only a single thread of control when exec-ing a new process
2. Wait

a. our current design already ensures the thread that is waiting is the only thread of the process waiting when calling process_wait

3. Exit

```
  exit(n)
loop through all the threads in the process
  - release locks (the locks for each user's threads are in a
list in the pcb)
  - signal all threads to wakeup and exit
if thread is a main thread:
  return 0
otherwise return n
```

To handle the case where a thread (not necessarily the main thread) calls process_exit(), we need to go through the exit process and make sure that when any threads in the process wake up they are immediately killed. To do this we copy the list of tids from the pcb and if the interrupt handler sees a trap from userspace (check the is_trap_from_userspace) if the thread id is in the list of the threads to kills, we dont allow it to return to userspace and we kill it while in kernel mode and deallocate the necessary data.

Synchronization

- We added a semaphore for each thread to keep track of its exit status, so that when threads join, the calling thread blocks on the semaphore of the corresponding thread. It can then receive the exit status and act accordingly.
- under user level synchronization, we have shared resources through the list of all locks and semaphores. we access these resources on the higher level when we need to update the locks or semaphores.
- We added a lock in the PCB for thread operations
- We use this lock to lock all thread operations in the kernel if needed and also when threads need to modify shared data structures. We will also make sure to lock actions regarding the file descriptors when opening, closing files, or removing files. Similarly, we will use the lock to create critical sections when writing into and out of buffers.

Rationale

pthread functions:

We added syscalls to handle the higher level user initiated functions: pthread_create, pthread_execute, pthread_exit, and pthread_join. The user thread will call the higher level functions and then trap into the kernel when the syscalls are called. A “user thread” and “kernel thread” are essentially the same thread but with different permissions when needed. This ensures the 1:1 mapping of a user to kernel thread.

//designate main thread to be the one that always exits the process

For exit and join:
Originally we thought we could put the semaphore to be used for joining in the TCB but if the thread has already exited then we wouldn’t be able to access it anymore. So we would need a list of the semaphores in the PCB, and essentially we ended up adding a shared data structure similar to what we did for processes. We then added the exit status so that we know whether the thread is valid to join on, to account for the case that it does not exit.

For user locks and semaphores:
We created a mapping of the user level locks and semaphores to the low level struct locks and semaphores in the kernel just as how file descriptors are mapped to files. We can then call a syscall that will call the kernel lock/semaphore functions. This allows us to use the low level synchronization that is already implemented but protect the kernel from the user, since the user only has access to the user lock_t, sema_t chars.

Concept check

1. Page containing its stack and TCB should be freed when the parent terminates because threads share memory resources (if a thread frees its memory too early, it can affect other threads)
2. kernel stack of the thread that is running
3. The first thread acquires lockA → the second thread acquires lockB → the first thread now needs to acquire lockB to continue running and the second thread

needs to acquire lockA to continue running (This scenario assumes that there is no way for a thread to backtrack or recover).
    a. No thread can continue running because there are no available resources
4. The problem occurs when the thread stack is freed. For instance, if the thread that was killed had any shared synchronization variables, the act of freeing the stack would also free these shared variables, which will prevent threads to use these variables in the future.
5. Let's say we have 3 threads A, B, and C. A has the highest priority, B has the second highest, and C has the lowest priority. Also, thread A needs to wait on thread B who is holding a resource that thread A needs (so thread A donates its priority to thread B). When sema_up is called on a semaphore that was originally at a value of 0, if donated priority is not taken into consideration, the sema_up will signal thread A to run. However, this will lead to a deadlock when A needs to acquire the resource that thread B is holding (since A has now decreased the semaphore back down to 0 for a scheduling workflow implementation of semaphores).
    a. If base threads of high priority and low priority are waiting on a semaphore, but the low priority thread has a higher effective priority, then the semaphore will actually unblock the lower priority thread (that had the higher base priority)

https://docs.google.com/document/d/1d_OqFyakVoQQCfzzffroxIdb-iFXdDTQE1V5hIXhg4l/edit

our notes