# Project Threads Design

---

# Efficient Alarm Clock

## Data Structures and Functions

```
struct thread {
  // existing ones
  ...
  // new ones we add
  int64_t wake_up_time; /* Wake up time for sleeping thread */
};
```

In `timer.c`

```
static struct list sleep_list /* Queue of sleeping threads */
```

## Algorithms

```
void timer_sleep(int64_t ticks)
```

- Set thread's `wake_up_time` to `timer_ticks()` + `ticks`
- Disable interrupts with `intr_disable()`, store `old_level`
- Add current thread to `sleep_list`, use `list_insert_ordered` so that the threads are ordered by wake up time in increasing order
- Block/sleep the thread with `thread_block()`
- Call `intr_set_level()` to set interrupt level back to `old_level`

```
static void timer_interrupt(struct intr_frame* args UNUSED)
```

- Check the queue of sleeping threads, check which threads need to be woken up
- If any threads need to be woken up:
  - Disable interrupts with `intr_disable()`
  - Remove current thread from `sleep_list`
  - Unblock/wake up the thread with `thread_unblock()`

- Call `intr_set_level()` to set interrupt level back to `old_level`

Gaurav: You can wake up thread with a higher priority than urself and call interrupt yield on return

```
bool wake_up_time_comparer(struct list_elem* x, struct list_elem* y, void* aux) {

  struct thread* x = list_entry(x, struct thread, elem);
  struct thread* y = list_entry(x, struct thread, elem);

  if (x-> wake_up_time < y-> wake_up_time){
    return true;
  } else {
    return false;
  }
}
```

## Synchronization

We decided to disable interrupt in both functions. For `timer_sleep`, we don't want any race conditions for multiple threads trying to add themselves to the sleep queue. Same goes for `timer_interrupt` since we are removing the list.

## Rationale

We opted against using a semaphore for each thread to track its sleeping status. This is because `sema_up()` and `sema_down()` disable interrupts during the function, and restore the interrupt state before returning. However, because we need to add threads to after `sleep_list` after disabling interrupts and before we block/sleep them, it made more sense to manually disable and restore interrupts and block/unblock threads instead of calling `sema_up()` and `sema_down()` (in which we would need to again manually disable and restore interrupts to add threads to `sleep_list`).

We use `list_insert_ordered` to insert the threads to the sleep queue by increasing order of the wake up time, this ensures when we iterate through the list in `timer_interrupt` we don't have to traverse through the entire list, as soon as a thread's wake up time is greater than `ticks`, break out of the loop.

---

# Strict Priority Scheduler

## Data Structures and Functions

```
struct thread {
  // existing ones
  ...
  // new ones
  int effective_priority; /* the thread's effective priority. changes during priority donation, and all actions comparing priority should use this value. */
  struct list donor_table; /* a list of all other threads that has donated priority to the current thread.  */


  // GAURAV:
  // pointer to who you could be blocked on
};
```

```
struct donor {
  int value;
  thread* donor;
  struct list_elem elem;
};
```

## Algorithms

FOR THREAD.C

```
void thread_init(void)
```

malloc list of donors d list to keep check of threads that has donated priority

```
int thread_get_priority(void)
```

Modify this function so that it returns the effective priority of the thread.

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

disable interrupts
Set the effective and original priority. if a thread's effective priority lowers, call `thread_enqueue` to put it back in the queue and trigger the scheduler again.
enable interrupts

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

In this function, we use `pop_front` on the `prio_ready_list`. since the list is ordered by the threads' effective priority in decreasing order with `list_insert_ordered`, popping the front will give the highest priority thread.
list of each priority value and that gives the ability for o(1) insert. make it a list of length 64 and 65. return idle
TOO SLOW! create a list of

```
static void thread_enqueue(struct thread* t);
```

Add another if clause to here for when the scheduler is set to be `SCHED_PRIO`, and user `list_insert_ordered` to insert the thread by decreasing effective priority.

FOR SYNCH.C

```
void sema_down(struct semaphore* sema);
```

instead of pushing to the back of the waiters list, insert in decreasing order by effective priority. `sema_up` can remain unchanged, as the thread with the highest priority will be in the front of the waiters list.

scan through

```
void cond_wait(struct condition* cond, struct lock* lock);
```

similar to `sema_down`, instead of pushing to the back of the cond's waiters list, insert in decreasing order by effective priority.

```
void lock_acquire(struct lock* lock);
```

Before calling `sema_down`, if locker holder is not null:

- check who has the lock.
- add a lock wait
- if that thread's priority is lower, `lock -> holder -> effective_priority = current_thread-> effective_priority`; also create a new `donor` and insert into the donee's `donors_table` list.

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

- start by going through the list of threads that has donated priority to the current thread (if any), and pick the next highest effective priority level to roll back the effective priority of the current thread.
- remove the released lock on the current thread's `donors_table` list
- hard to tell if any other threads

yield when unblocking a higher prio thread
yield thread create
disable iterrupt when donation is happening

## Synchronization

Most of the operations with semaphores and thread scheduling happening when interrupts are disabled, so we have no other measures synchronization measures. Moreover, since we are basically modifying the logic of synchronization primitives themselves, we can only rely on disabling and enabling interrupts as the way of creating critical sections of code.

## Rationale

We are modifying list insertion logic within the synchronization primitives and adding a prio ready queue, so that all threads inside will be ordered by their effective priority with `list_insert_ordered`. We ensure that we popping from a queue we will always pop the thread with the highest priority thread. Right now list insertion for the synchronization primitives all push things to the end of the waiting queue, we don't want to search through the waiting queue every time we need to find the thread with the highest effective priority to remove, so we use `list_insert_ordered` to achieve this.

Gaurav: in the donor table, add a way to recursive down. make a variable in the donor table to see what thread ur thread is blocked on.

---

# User Threads

## Data Structures and Functions

In `userprog/process.h`

```
struct process {
  ...
  /* For User Threads*/
  struct list join_status_list; /* a list of join statuses of all threads */
  struct lock join_status_list_lock; /* lock to grab to modify join_status_list*/
  struct list lock_list; /* a list of locks associated with this process. */
  struct lock lock_list_lock; /* lock to grab to modify lock_list*/
  struct list sema_list; /* a list of semaphores associated with this process. */
  struct lock sema_list_lock; /* lock to grab to modify sema_list */
}
```

```
struct pass_to_start_pthread {
  struct semaphore sema; /* semaphore shared between pthread_execute and start_pthread */
  stub_fun sf; /* stub function */
  pthread_fun tf; /* the user function to run */
  void* arg; /* the argument to push onto the stack for the user function */
```

```
  struct process* pcb;   /* PCB of the new user thread */
}
```

In `threads/thread.h`

```
struct thread {
  ...
  /* For User Threads */
  struct list_elem process_elem; /* List element for list of threads in pcb. */
  struct join_status* join_status; /* join_status of thread */
  int exit_status;               /* exit status of the current thread */
  uint8_t* upage; /* user virtual memory pointer of thread's page */
};
```

```
struct join_status {
  tid_t tid;
  struct semaphore join_sema;
  struct list_elem elem;
};
```

In `userprog/process.c`, for user level locks

```
struct user_semaphore {
  struct semaphore* kernel_semaphore;
  struct list_elem process_elem; /* List element for process */
  sema_t* sema_id /* User sema's ID */
};

struct user_lock {
  struct lock* kernel_lock;
```

```
  struct list_elem process_elem; /* List element for process
*/
  lock_t* lock_id; /* User lock's ID */
};
```

## Algorithms

In `src/userprog/process.c`

```
bool setup_thread(void (**eip)(void), void** esp, struct pass_
to_start_pthread* exec_)
```

- Call `palloc_get_page` to allocate memory for a page and return its kernel virtual memory pointer (`kpage`)
- Starting at `upage = PHYS_BASE - PGSIZE`, use `install_page` to map the `kpage` to an address in user virtual memory, decrement `upage` every time mapping is unsuccessful
- If page installation is not successful, call `palloc_free_page` to free the page
- push `arg` and `tf` onto the stack.
- Make `esp` point to the `upage + PGSIZE`
- Set `eip` to point to the stub function `sf`.

In `src/userprog/process.c`

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

- Construct the `pass_to_start_thread` struct
- Call `thread_create` with `start_pthread` being the thread function
- Call `sema_down` on the shared semaphore
- return the `tid`, or `TID_ERROR` if thread creation failed

In `src/userprog/process.c`

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

- Unpack the `exec_` to get back all the variables in the `pass_to_start_thread` struct.
- Create a new `intr_frame` object and place it on the stack with `memset`.
- Call `process_activate()`
- Call `setup_thread` with the `eip` and `esp` of the intr_frame, along with the variables unpacked earlier.

- Initialize the pthread's `join_status`, then add the new `join_status` to the `pcb`'s `join_status_list` and assign it to the thread's `join_status`
  - Note that we also initialize the main thread's `join_status` during `start_process`, which is when the thread's `pcb` and `wait_status` are also initialized.
- Call `sema_up` on the shared semaphore to unblock `pthread_execute`
- Use the same assembly call found in `start_process` to start the pthread

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

- If thread is trying to join on itself, return `TID_ERROR`
- Acquire the `pcb`'s `join_status_list_lock`
- Iterate through the `pcb`'s `join_status_list` to find the `join_status` associated with target `tid`
- If target thread has already been joined on, return `TID_ERROR`
- Remove the thread's `join_status` from the `pcb`'s `join_status_list`
- Call `sema_down` on the target's thread's `join_status`'s semaphore to wait on the target thread with tid `tid`
  - After downing the semaphore, remove the target's `join_status` from the `pcb`'s `join_status_list` and free the `join_status`
- Release the `pcb`'s `join_status_list_lock`
- Return the `tid`

```
void pthread_exit(void)
```

- dealloc the page we previously allocated for the thread.
- free the page directory.
- go through the `join_queue` and call `thread_unblock` on all those threads.
- add `ebp` onto the process `available_addr` list
- if main thread is calling this function
  - call `process_exit`
- otherwise call `thread_exit` to activate the scheduler.

```
void pthread_exit_main(void)
```

- go to the pcb of the thread, iterate through the `threads_list`
  - if the thread is blocked, call `thread_unblock` to continue execution of that thread

    - call `pthread_join` on each thread (**question**: would this block access to the pcb? because we'd have to call `thread_block` to block the main thread)
- call `pthread_exit` to exit the thread
- free everything in `process` struct

**pthread syscalls**

```
tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg)
```

- check each of the pointers
- call `pthread_execute` with the same arguments.
- return the tid returned by `pthread_execute`.

```
void sys_pthread_exit(void)
```

- check if the thread is the main thread of the process:
    - if main thread, call `pthread_exit_main`
    - if not, just call `pthread_exit`

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

- call `pthread_join`.

**User-level synchronization syscalls**

Syscall handler will call all of the following:

In `userprog/process.c`:

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

- create a new kernel lock, allocate memory for it using `malloc()`
- create a new user lock, allocate memory for it using `malloc()`
- set user lock's `lock_id` to `*lock`
- add new lock to process's `lock_list` in process
- call kernel lock's `lock_init()`
- return true if successful, return false if unsuccessful

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

- Search for `lock_id` in the `lock_list` struct
- If lock's holder thread is not `null`, then exit the process, return `false`
- Else, call kernel's `lock_acquire`

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

- search for the lock with the corresponding `lock_id` in `lock_list`
- if the `user_lock` struct doesn't exist, return false
- if the kernel's lock holder is not this thread, return false
- otherwise, call `lock_release` on the kernel lock

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

- create a new kernel semaphore, allocate memory for it using `malloc()`
- create a new user semaphore, allocate memory for it using `malloc()`
- set user semaphore's `sema_id` to `*sema`
- add new semaphore to process's `sema_list` in process
- call kernel semaphore's `sema_init`

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

- search for the corresponding semaphore in the `sema_list`
- if can't be found, return `false`
- call `sema_down` on the kernel semaphore
- return `true`

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

- Look for the semaphore inside the sema_list
- if semaphore doesnt exist, return false
- Increase the semaphore value
- return true if succesfull
- false otherwise

### Other

```
tid_t get_tid(void)
```

- return `thread_current()`'s `tid`.

### Process Control Syscalls Modifications

```
void process_exit(void)
```

- call `pthread_exit_main` to ensure all threads run to completion

- **question**: how should the behavior differ when when exit from user space & from kernel space? we want to understand the significance of **`is_trap_from_userspace`**
- all threads can finish running & end, can't just kill them all. all threads will run for a little bit, unblock, store flag die on return, should i be allowed to do operation then should i die
- any pthread can call process exit, caller becomes a main thread, make everyone else call pthread exit

```
pid process_execute(const char* file_name)
```

- check if the thread is the main thread.
- if not, create a new thread with `pthread_execute`
- otherwise, keep the same logic

## Synchronization

We added a lock (`thread_lock` and `process_lock`) to the `thread` and the `process` structs. Whenever another thread is trying to modify any list structs within a thread or a process, they must acquire this lock first to prevent any concurrent modifications made to the list, which may lead to incorrect results.  We also call `thread_block` when joining to ensure that the caller thread will stop execution until the waited on thread finishes running. In order for `pthread_execute` to wait on `start_thread` to finish executing, we pass a semaphore in the `exec_`. `pthread_execute` will call down on that semaphore, which blocks until `start_thread` calls sema up at the end of the function (this process is similar to passing a semaphore between `process_execute` and `start_process`).

## Rationale

We add a `pass_to_start_pthread` to relay information between `pthread_execute` and `start_pthread`, this serves a similar purpose to the struct passed from `process_execute` to `start_process`. we include a semaphore to ensure that the caller thread is blocked until the thread creation process finishes, and we also pass in the stub function to point the eip to the adress of that function, along with tfun and the arguments for tfun, so that they can be pushed onto the stack.

The `available_addr` struct was created to store available addresses for future threads on the user stack. When user threads were freed from the stack, their addresses would be stored in an `available_addr` struct, which would then be added to a process's list of `available_addrs`. With this method, we're able to prevent internal fragmentation by utilizing freed memory from old threads.

The `user_semaphore` and `user_lock` structs were created as user abstractions for kernel semaphores and locks. Every `user_semaphore` maps to a kernel semaphore, and every `user_lock` maps to a kernel lock. Moreover, `user_semaphore` and `user_lock` structs contain a `sema_id` and a `lock_id` for the user to reference.

---

# Concept check

1. We cant just free the memory using palloc_free_page because the esp will still point to the page after the function is called. The freeing of the stack and TCB happens in function `thread_switch_tail` after the thread is safely switched.
2. `thread_tick` executes on the kernel stack.
3. Thread A acquires `lockA`, Thread B acquires `lockB`, then Thread A tries to acquire `lockB`. Thread A waits for Thread B to release `lockB`, but Thread B cannot release `lockB` until it acquires `lockA`, which Thread A is holding.
4. If Thread A and thread B has some shared resource that Thread B owns. When Thread A kills Thread B, a deadlock might occur due to unexpected freed resource that Thread A would be waiting on.
5. Consider three threads: Thread A, Thread B, and Thread C with priorities: 400, 200, 100, respectively. Thread A has the highest priority, Thread C has the lowest priority, and Thread B has the middle priority. Suppose Thread C currently holds a synchronization lock called Lock 1, and it has been preempted. Thread A calls tries to acquire Lock 1, which calls `sema_up`. However, because Thread C currently holds Lock 1, Thread A is blocked until Thread C releases Lock 1. Thread C cannot release Lock 1 because Thread B runs, as Thread B has a higher priority than Thread C. As such, a priority inversion occurs, where Thread A waits on Thread C, which waits on Thread B.
   **Expected Output:**
   Thread C runs (with donated priority) and releases Lock 1.
   Thread A runs after Lock 1 is released.

Thread B runs last.

**Actual Output Without Proper Priority Handling:**

In a system that does not correctly implement priority donation or effective priority handling in semaphores:

Thread B runs after Thread C is preempted because the system does not adjust Thread C's priority based on Thread A's priority waiting on Lock 1.

Thread A remains blocked waiting for Lock 1 because Thread C cannot run to release it, given Thread B has higher priority than Thread C's base priority.

Thread C eventually runs after Thread B, depending on the system's scheduling policy and whether Thread B yields or is preempted.

Thread A finally runs after Thread C releases Lock 1.

Questions: