Lab 1 Introduction

Peng Zijun

Content

- Thread Control Block
- Switching Thread
- Create a new Thread
- Synchronization
- Lab1 Tasks

Thread Control Block (TCB)

- This struct represent a thread in kernel or a process in user space.
- Every struct Tread occupies the beginning of its own page of memory.
- The rest of the page is used for the thread's stack.

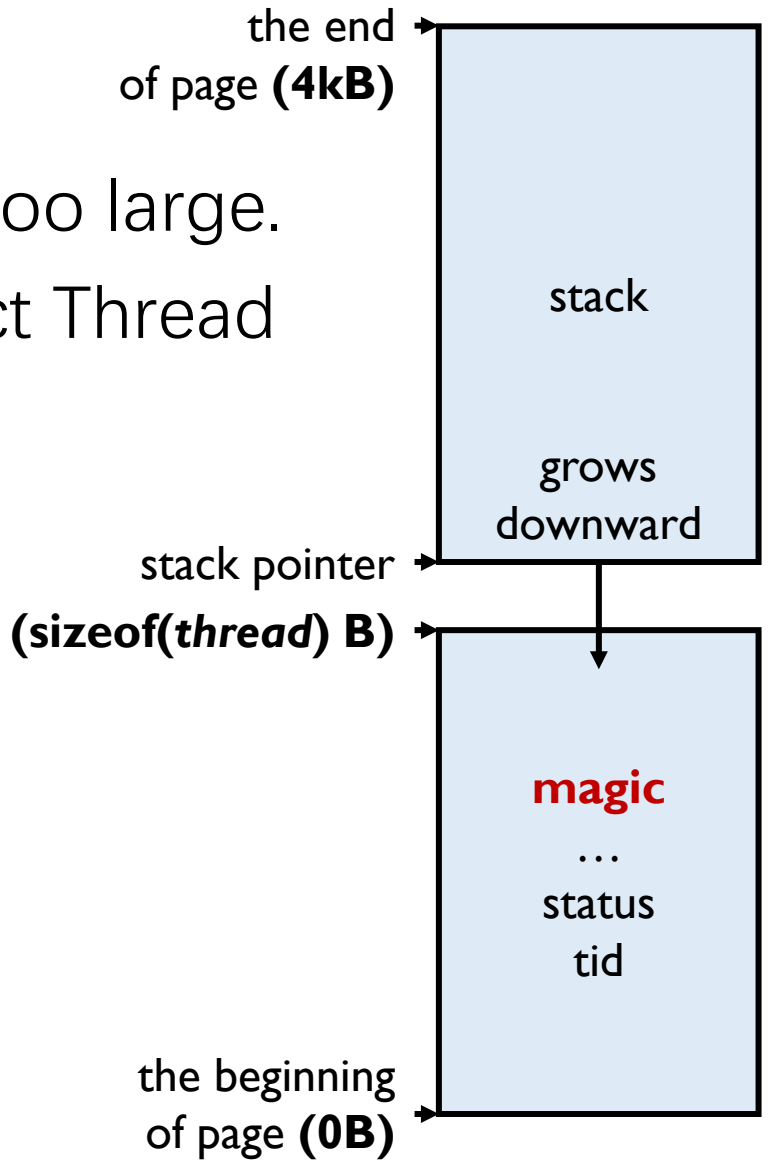
```rust
/// All data of a kernel thread
#[repr(C)]

pub struct Thread {
    tid: isize,
    name: &'static str,
    stack: usize,
    status: Mutex<Status>,
    context: Mutex<Context>,
    pub priority: AtomicU32,
    pub userproc: Option<UserProc>,
    pub pagetable: Option<Mutex<PageTable>>,
}
```

Thread Control Block (TCB)

- Struct Thread must not be allowed to grow too large.
- There is a magic number right after the struct Thread to detect stack overflow.

```rust
pub fn overflow(&self) -> bool {
    unsafe { (self.stack as *const usize).read() != MAGIC }
}
```



Struct Thread

- Please read src/thread/impl.rs & src/thread.rs

```rust
/// Get the current running thread
pub fn current() -> Arc<Thread> {
    Manager::get().current.lock().clone()
}

/// Yield the control to another thread (if there
pub fn schedule() {
    Manager::get().schedule()
}
```

```rust
/// Mark the current thread as [`Blocked`](Status::Blocked
/// yield the control to another thread
pub fn block() {
    let current: Arc<Thread> = current();
    current.set_status(Status::Blocked);

    #[cfg(feature = "debug")]
    kprintln!("[THREAD] Block {:?}", current);

    schedule();
}
```

```rust
/// Wake up a previously blocked thread, mark it as [`Ready`](Status::Ready),
/// and register it into the scheduler.
pub fn wake_up(thread: Arc<Thread>) {
    assert_eq!(thread.status(), Status::Blocked);
    thread.set_status(Status::Ready);

    #[cfg(feature = "debug")]
    kprintln!("[THREAD] Wake up {:?}", thread);

    Manager::get().scheduler.lock().register(thread);
}

/// (Lab1) Sets the current thread's priority to a given value
pub fn set_priority(_priority: u32) {}

/// (Lab1) Returns the current thread's effective priority.
pub fn get_priority() -> u32 {
    0
}
```

Struct *thread*

- some handy interfaces(Pintos)
  - thread/thread.h

```
extern bool thread_mlfqs;

void thread_init (void);
void thread_start (void);

void thread_tick (void);
void thread_print_stats (void);

typedef void thread_func (void *aux);
tid_t thread_create (const char *name, int priority, thread_func *, void *);

void thread_block (void);
void thread_unblock (struct thread *);

struct thread *thread_current (void);
tid_t thread_tid (void);
const char *thread_name (void);

void thread_exit (void) NO_RETURN;
void thread_yield (void);
```

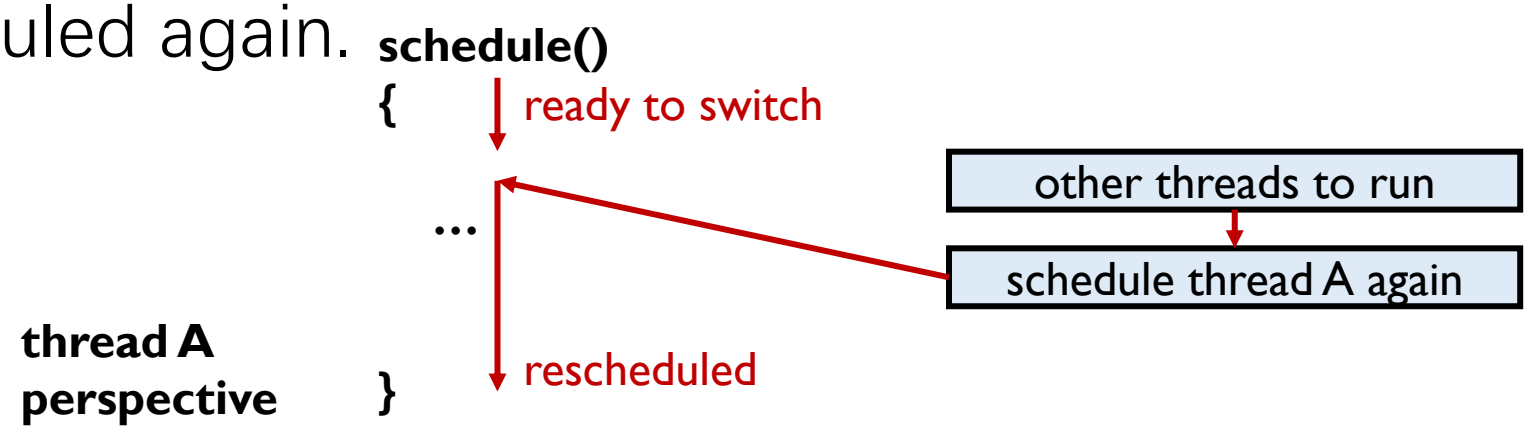
```
/** Performs some operation on thread t, given auxiliary data AUX. */
typedef void thread_action_func (struct thread *t, void *aux);
void thread_foreach (thread_action_func *, void *);

int thread_get_priority (void);
void thread_set_priority (int);

int thread_get_nice (void);
void thread_set_nice (int);
int thread_get_recent_cpu (void);
int thread_get_load_avg (void);
```

Switching Thread

- In Tacos, read **src/thread/manager.rs** and **src/thread/switch.rs**.
- When `schedule()` is called, kernel find a next thread using scheduler and call `switch()`.
- `switch()` use inline assembly to save the context, that is the value of all the registers and other useful information.
- When the thread is return from `schedule()`, that means the thread is scheduled again.



1. Close interrupt
2. Get the context struct of two thread
3. Call switch
4. Reset interrupt

```rust
pub fn schedule(&self) {
    let old: bool = interrupt::set(level: false);

    let next: Option<Arc<Thread>> = self.scheduler.lock().schedule();

    // Make sure there's at least one thread runnable.
    assert!(
        self.current.lock().status() == Status::Running || next.is_some(),
        "no thread is ready"
    );
    assert!(!self.current.lock().overflow(), "Current thread has overflowed its stack.");

    if let Some(next: Arc<Thread>) = next {
        assert_eq!(next.status(), Status::Ready);
        assert!(!next.overflow(), "Next thread has overflowed its stack.");
        next.set_status(Status::Running);

        // Update the current thread to the next running thread
        let previous: Arc<Thread> = mem::replace(dest: self.current.lock().deref_mut(), src: next);
        // Retrieve the raw pointers of two threads' context
        let old_ctx: *mut Context = previous.context();
        let new_ctx: *mut Context = self.current.lock().context();

        // WARNING: This function call may not return, so don't expect any value to be dropped.

        unsafe { switch::switch(previous: Arc::into_raw(this: previous).cast(), old_ctx, new_ctx)

    }

    interrupt::set(level: old);
} fn schedule
```

- Store the old context and load the new context.
- And call `schedule_tail_wrapper()`

```
global_asm! {r#"
    .section .text
        .globl switch
    switch:
        sd ra, 0x0(a1)
        ld ra, 0x0(a2)
        sd sp, 0x8(a1)
        ld sp, 0x8(a2)
        sd  s0, 0x10(a1)
        ld  s0, 0x10(a2)
        sd  s1, 0x18(a1)
        ld  s1, 0x18(a2)
        sd  s2, 0x20(a1)
        ld  s2, 0x20(a2)
        sd  s3, 0x28(a1)
        ld  s3, 0x28(a2)
        sd  s4, 0x30(a1)
        ld  s4, 0x30(a2)
        sd  s5, 0x38(a1)
        ld  s5, 0x38(a2)
        sd  s6, 0x40(a1)
        ld  s6, 0x40(a2)
        sd  s7, 0x48(a1)
        ld  s7, 0x48(a2)
        sd  s8, 0x50(a1)
        ld  s8, 0x50(a2)
        sd  s9, 0x58(a1)
        ld  s9, 0x58(a2)
        sd s10, 0x60(a1)
        ld s10, 0x60(a2)
        sd s11, 0x68(a1)
        ld s11, 0x68(a2)

        j schedule_tail_wrapper
"#} global_asm!
```

- `schedule_tail_wrapper` called `schedule_tail`.
- This function do some necessary staff after switch.
- Note that this function is still the previous thread.
- But when it return, the pc will jump to register `ra`, and that is the thread we want to switch to.

```rust
pub fn schedule_tail(&self, previous: Arc<Thread>) {
    assert!(!interrupt::get());

    #[cfg(feature = "debug")]
    kprintln!("[THREAD] switch to {:?}", *self.current.lock());

    match previous.status() {
        Status::Dying => {
            // A thread's resources should be released at this point
            self.all.lock().retain(|t: &Arc<Thread>| t.id() != previous.id());
        }
        Status::Running => {
            previous.set_status(Status::Ready);
            self.scheduler.lock().register(thread: previous);
        }
        Status::Blocked => {}
        Status::Ready => unreachable!(),
    }

    if let Some(pt: &Mutex<PageTable, Intr>) = self.current.lock().pagetable.as_ref() {
        pt.lock().activate();
    } else {
        KernelPgTable::get().activate();
    }
} fn schedule_tail
```

Create a new thread

- What does spawn do?

1. Call `build`, which create a new thread control block (with necessary setups, including create a new pagetable…). Kernel thread has no new pagetable though.
2. Register the new thread to the thread manager.
3. Next time when schedule() is called, the new thread has the chance to run.

```rust
pub fn spawn(self) -> Arc<Thread> {
    let new_thread: Arc<Thread> = self.build();

    #[cfg(feature = "debug")]
    kprintln!("[THREAD] create {:?}", new_thread);

    Manager::get().register(new_thread.clone());

    // Off you go
    new_thread
}
```

The first thread

- Tacos create the first thread in src/manager.rs:get(). The thread is created lazy statically, that is the first thread will not be created util the Manager::get() is called.

```
pub fn get() -> &'static Self {
    static TMANAGER: Lazy<Manager> = Lazy::new(|| {
        let initial: Arc<Thread> = Arc::new(data: Thread::new(
            name: "initial",
            stack: bootstack as usize,
            priority: PRI_DEFAULT,
            entry: 0,
            userproc: None,
            pagetable: None,
        ));
        unsafe { (bootstack as *mut usize).write(val: MAGIC) };
        initial.set_status(Status::Running);

        let manager: Manager = Manager {
            scheduler: Mutex::new(Scheduler::default()),
            all: Mutex::new(Vec::from([initial.clone()])),
            current: Mutex::new(initial),
        };

        let idle: Arc<Thread> = Builder::new(function: || loop {
            schedule()
        }) Builder
        .name("Idle") Builder
        .priority(PRI_MIN) Builder
        .build();
        manager.register(thread: idle);
```

Initialization of the first thread

- The first thread is initialized in ***thread_init***

```
/* Set up a thread structure for the running thread. */
initial_thread = running_thread ();
init_thread (initial_thread, "main", PRI_DEFAULT);
```

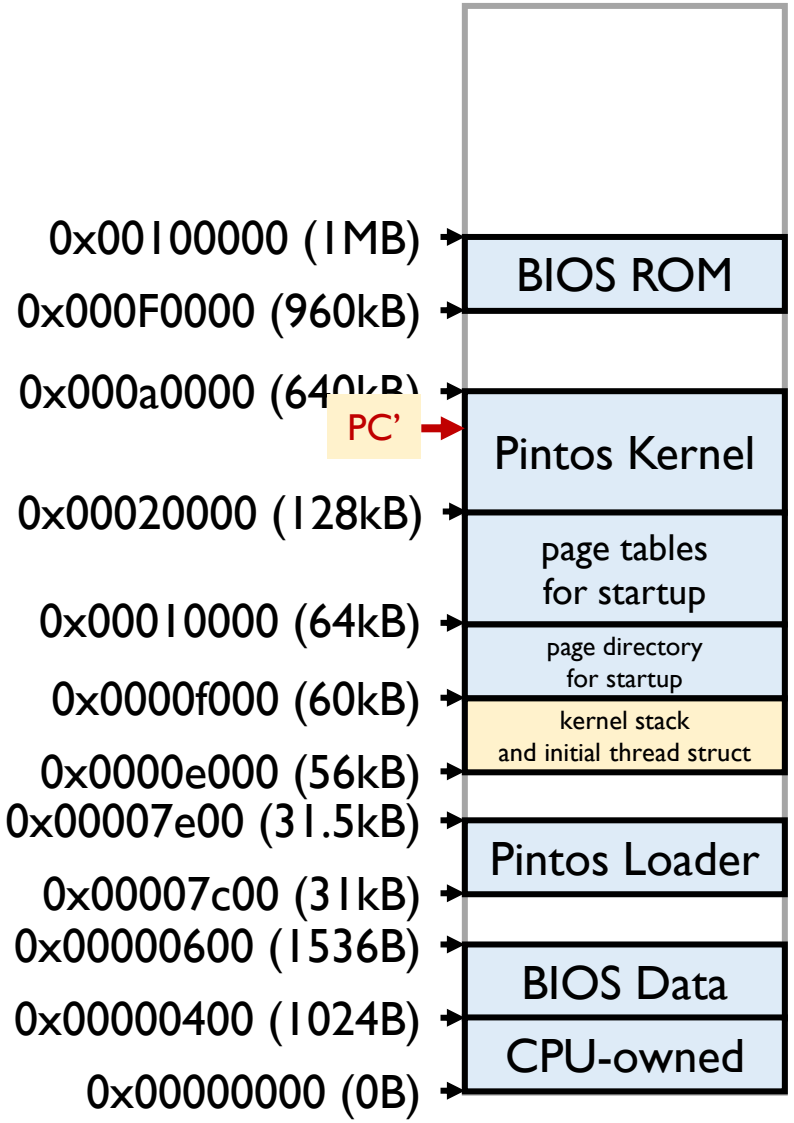
- The struct address of the first thread depends on %esp, which is initialized in *loader.S*

```
/** Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;

  /* Copy the CPU's stack pointer into `esp', and then round that
     down to the start of a page.  Because `struct thread' is
     always at the beginning of a page and the stack pointer is
     somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
```

```
# Set up segment registers.
# Set stack to grow downward from 60 kB (after boot, the kernel
# continues to use this stack for its initial thread).

    sub %ax, %ax
    mov %ax, %ds
    mov %ax, %ss
    mov $0xf000, %esp
```

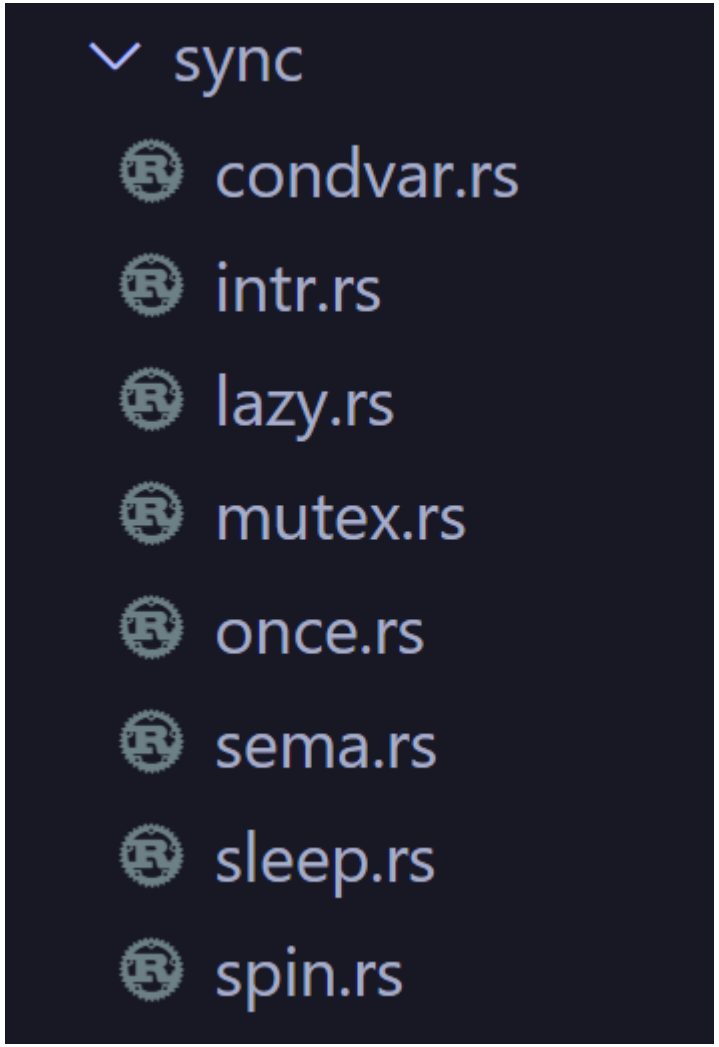


Brief introduction to RAII

- RAII(**R**esource **A**cquisition **I**s **I**nitialization) is an important technology for modern OOP.
- The resource is automatically initialized when we get it the first time, and automatically dropped when we finish our last access.
- Benefit is obvious.
- Need some tricks or language support.
- The first thread in Tacos, is an example. There are more examples of RAII in Tacos.

Synchronization (Tacos)

- Code in src/sync
- Locks:
  - Interrupt lock
  - Sleep lock
  - Spin lock
- Condition variable
- Semephore
- Mutex: Based on semaphore



List in Pintos

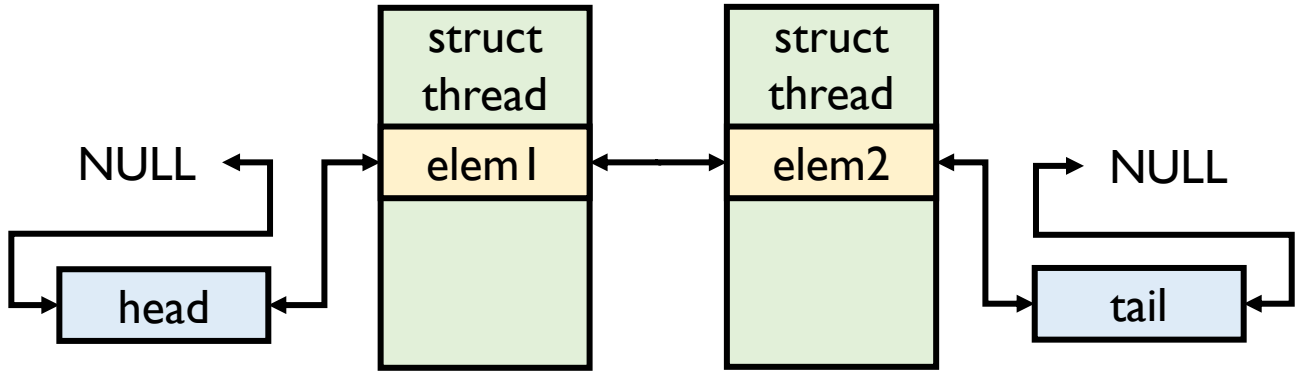
- ***doubly linked list***
- **an empty list**

```
/** List element. */
struct list_elem
  {
    struct list_elem *prev;     /**< Previous list element. */
    struct list_elem *next;     /**< Next list element. */
  };
```

```
/** List. */
struct list
  {
    struct list_elem head;      /**< List head. */
    struct list_elem tail;      /**< List tail. */
  };
```

- **a list with 2 elements**

- **a list of complex struct**
  - embed a *list_elem* in the struct
  - manipulate the inner *list_elem*



List in Pintos

- **example: ready_list**

```
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct thread, elem);
}
```

- ***list_entry***: convert pointer to LIST_ELEM into the pointer to the struct containing it

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /*< Thread identifier. */
    enum thread_status status;          /*< Thread state. */
    char name[16];                      /*< Name (for debugging purposes). */
    uint8_t *stack;                     /*< Saved stack pointer. */
    int priority;                       /*< Priority. */
    struct list_elem allelem;           /*< List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /*< List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /*< Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /*< Detects stack overflow. */
  };
```

```
/** List of processes in THREAD_READY state, that is, processes
   that are ready to run but not actually running. */
static struct list ready_list;
```

```
#define list_entry(LIST_ELEM, STRUCT, MEMBER)           \
        ((STRUCT *) ((uint8_t *) &(LIST_ELEM)->next     \
                     - offsetof (STRUCT, MEMBER.next)))
```

List in Pintos

- **some handy interfaces**
  - *lib/kernel/list.h*

```
/** List traversal. */
struct list_elem *list_begin (struct list *);
struct list_elem *list_next (struct list_elem *);
struct list_elem *list_end (struct list *);

struct list_elem *list_rbegin (struct list *);
struct list_elem *list_prev (struct list_elem *);
struct list_elem *list_rend (struct list *);

struct list_elem *list_head (struct list *);
struct list_elem *list_tail (struct list *);

/** List insertion. */
void list_insert (struct list_elem *, struct list_elem *);
void list_splice (struct list_elem *before,
                  struct list_elem *first, struct list_elem *last);
void list_push_front (struct list *, struct list_elem *);
void list_push_back (struct list *, struct list_elem *);

/** List removal. */
struct list_elem *list_remove (struct list_elem *);
struct list_elem *list_pop_front (struct list *);
struct list_elem *list_pop_back (struct list *);

/** List elements. */
struct list_elem *list_front (struct list *);
struct list_elem *list_back (struct list *);
```

```
/** List properties. */
size_t list_size (struct list *);
bool list_empty (struct list *);

/** Miscellaneous. */
void list_reverse (struct list *);

/** Compares the value of two list elements A and B, given
   auxiliary data AUX.  Returns true if A is less than B, or
   false if A is greater than or equal to B. */
typedef bool list_less_func (const struct list_elem *a,
                             const struct list_elem *b,
                             void *aux);

/** Operations on lists with ordered elements. */
void list_sort (struct list *,
                list_less_func *, void *aux);
void list_insert_ordered (struct list *, struct list_elem *,
                          list_less_func *, void *aux);
void list_unique (struct list *, struct list *duplicates,
                  list_less_func *, void *aux);

/** Max and min. */
struct list_elem *list_max (struct list *, list_less_func *, void *aux);
struct list_elem *list_min (struct list *, list_less_func *, void *aux);
```

Lab1 Tasks

1. Alarm clock: sleep using spinning is inefficient, so try to implement a better sleep().
2. Priority scheduling:
    1. Implement a strict prority scheduling algorithm
    2. Implement function to get and set prority
    3. Implement fifo between threads with same prority
    4. Implement prority donation **only** for locks.
3. **Important: For Tacos, please synchronization your tacos with the github repo.**

Some hints

- Read docs and code before every thing.
- Do your design before writing.
- Test driven developing
- Debug by using gdb or printing logs.

- I try to let opencode(deepseek reasoner) solve Tacos lab1 by itself. After using 20M+ tokens, it still can’t pass all the tests.
- Maybe it is struggling with Rust. I don’t know.
- Another possible reason is that most of the model has a dramatic regression with more than 100K context.