Threads

```
print_mul (int arr[], int len) {
    int mul = 1, i;
    for (i = 0; i < len; i++)
        mul *= arr[i];
    return mul;
}
print_sum (int arr[], int len) {
    int sum = 0, i;
    for (i = 0; i < len; i++)
        sum += arr[i];
    return sum;
}
```

```
main ()
{
   int *arr1 = malloc (sizeof(int) * 100);
   int arr2[100];
  read_from_file (arr1, 100);
  read_from_file (arr2, 100);
  create_thread (print_mul, arr1, 100);
  print_sum (arr2, 100);
  return 0;
}
```

Schedule

```
struct list *ready_list;
struct thread {
    void *esp;
};
```

- idle_thread

Schedule

```
struct list *ready_list;
struct thread {
    void *esp;
};
```

```
create_thread () {
    struct thread *t = malloc ();
    t->esp = malloc(4096) + 4096;
    enqueue (read_list, t);
}
```

Schedule

```
schedule (struct thread *cur_thread) {
    enqueue (ready_list, cur_thread);
    next_thread = dequeue (ready_list);
    push %eax
    ....
    push %edi
    cur_thread->esp = cur_esp;
    esp = next_thread->esp;
    pop %edi
    ...
    pop %eax
}
```

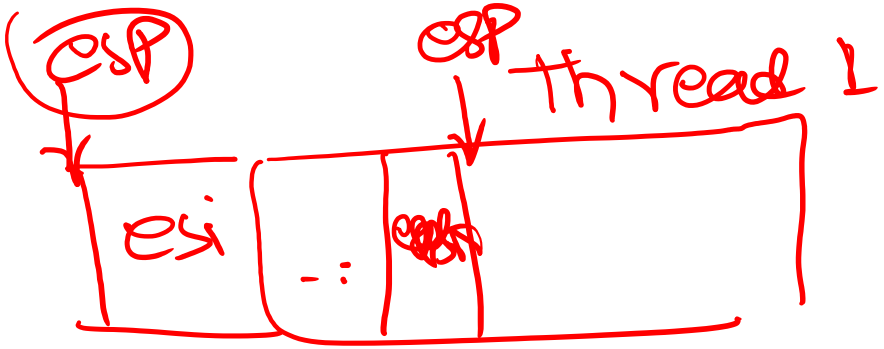
old_

new thread

-1. esp

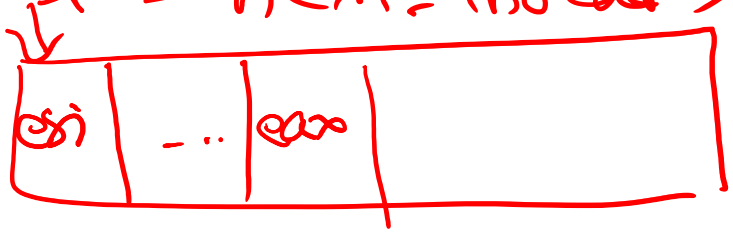
(-). eip

- Why schedule() doesn't save the current program pointer?



cur_thread->esp = esp;

esp = next_thread->esp




```
foo
{
  schedule()
}
```

```
bar
{
  schedule()
}
```

Race condition

```
struct node *head; = NULL;

insert (int val) {
    struct node *n = malloc();
    n->val = val;
    n->next = head;
    head = n;
}
```

Race condition

```
struct node *head = NULL;

insert (int val) {
    struct node *n = malloc();
    n->val = val;
    n->next = head;
    head = n;
}
```

```
thread 1:
n->next = NULL;
schedule ();
thread 2:
n->next = NULL;
head = n;
schedule ();
thread 1:
head = n;
```

Critical section

```
struct node *head = NULL;

insert (int val) {
    struct node *n = malloc();
    n->val = val;
    lock_acquire ();
    n->next = head;
    head = n;
    lock_release ();
}
```

The area between lock_acquire and lock_release is called a critical section.

Only one thread is allowed to execute in a critical section.

Thread 1.
lock_acquire
n->next = NULL;
Schedule
lock_acquire();
schedule().
thread 1: head = n;
lock_release();

schedule();
Thread 2:
n->next = head;
head = n;

Implementation

```
lock_acquire ()
{
  status =  interrupt_disable ();
}
```

```
lock_release (int status)
{
   set_interrupt_status (status);
}
```

insert( )

Implementation

```
struct lock {
    int value;
};

lock_acquire (struct lock *l)
{
    while (l->value == 0) {}
    l->value = 0;
}
```

```
lock_init (struct lock *l)
{
    l->value =1 ;
}

lock_release (struct lock *l)
{
    l->value = 1;
}
```

Thread 1:
l->value == 0
-> schedule()
l->value == 0

Thread 1 :
l->value = 0;
l->value = 1;

Thread 2:
while (l->value == 0);
l->value = 0;

Implementation

```
struct lock {
    int value;
};

Lock_acquire (struct lock *l)
{
    while (l->value == 0) {}
    l->value = 0;
}
```

Race condition in the lock implementation itself

```
Lock_release (struct lock *l)
{
    l->value = 1;
}
```

Implementation

```
struct lock {
    int value;
};

Lock_acquire (struct lock *l)
{
    status = interrupt_disable();
    while (l->value == 0) {}
    l->value = 0;
    set_interrupt_status (status);
}
```

Race condition in the lock implementation itself

```
Lock_release (struct lock *l)
{
    l->value = 1;
}
```

Implementation

struct lock {
    int value;
};

Lock_acquire (struct lock *l)
{
    status = interrupt_disable();
    while (l->value == 0) {}
    l->value = ~~1~~ 0;
    set_interrupt_status (status);
}

Infinite loop:

Lock_release (struct lock *l)
{
    l->value = ~~0~~ 1;
}

Implementation

```
struct lock {
    int value;
};

Lock_acquire (struct lock *l)
{
    status = interrupt_disable();
    while (l->value == 0) {
        schedule ();
    }
    l->value = 1;
    set_interrupt_status (status);
}

Lock_release (struct lock *l)
{
    l->value = 0;
}
```

Implementation

```
struct list *wait_list;

Lock_acquire (struct lock *l)
{
   status = interrupt_disable();
   while (l->value == 0) {
      remove (ready_list, thread_current());
      enqueue (wait_list, thread_current());
      schedule ();
   }
   l->value = 0;
   set_interrupt_status (status);
}
```

```
Lock_release (struct lock *l)
{
   struct thread *t = dequeue (wait_list);
   enqueue (ready_list, t);
   l->value = 1;
}
```

Semaphore

```
struct list *wait_list;

sema_down (struct lock *l)
{
   status = interrupt_disable();
   while (l->value == 0) {
      remove (ready_list, thread_current());
      enqueue (wait_list, thread_current());
      schedule ();
   }
   l->value--;
   set_interrupt_status (status);
}
```

```
sema_up (struct lock *l)
{
   struct thread *t = dequeue (wait_list);
   enqueue (ready_list, t);
   l->value++;
}
```

Semaphore

```
parent () {
  struct semaphore s;
  sema_init (&s, 0);
  create_thread (child, &s);
  sema_down(&s);
}
child (struct semaphore *s) {
  sema_up (s);
}
```

Condition variables

```
char buf[BUF_SIZE];
size_t n = 0;
size_t head = 0, tail = 0;
struct lock lock;
struct condition not_empty, not_full;

void put (char ch) {
  while (n == BUF_SIZE) {}
  buf[head++ % BUF_SIZE] = ch;
  n++;
}
```

```
char get (void) {
  char ch;
  while (n == 0) {}
  ch = buf[tail++ % BUF_SIZE];
  n--;
}
```

```
char buf[BUF_SIZE];
size_t n = 0;
size_t head = 0, tail = 0;
struct lock lock;
struct condition not_empty, not_full;

void put (char ch) {
  while (n == BUF_SIZE)
     cond_wait (&not_full);
  buf[head++ % BUF_SIZE] = ch;
  n++;
  cond_signal (&not_empty);
}
```

```
char get (void) {
  char ch;
  while (n == 0)
     cond_wait (&not_empty);
  ch = buf[tail++ % BUF_SIZE];
  n--;
  cond_signal (&not_full);
}
```

```c
char buf[BUF_SIZE];
size_t n = 0;
size_t head = 0, tail = 0;
struct lock lock;
struct condition not_empty, not_full;

void put (char ch) {
  lock_acquire (&lock);
  while (n == BUF_SIZE)
      cond_wait (&not_full);
  buf[head++ % BUF_SIZE] = ch;
  n++;
  cond_signal (&not_empty);
  lock_release (&lock);
}
```

```c
char get (void) {
  char ch;
  lock_acquire (&lock);
  while (n == 0)
      cond_wait (&not_empty);
  ch = buf[tail++ % BUF_SIZE];
  n--;
  cond_signal (&not_full);
  lock_release (&lock);
}
```

```
char buf[BUF_SIZE];
size_t n = 0;
size_t head = 0, tail = 0;
struct lock lock;
struct condition not_empty, not_full;

void put (char ch) {
  lock_acquire (&lock);
  while (n == BUF_SIZE) {
      lock_release (&lock);
      cond_wait (&not_full);
      lock_acquire (&lock);
}
  buf[head++ % BUF_SIZE] = ch;
  n++;
  cond_signal (&not_empty);
  lock_release (&lock);
}
```

Thread1

```
char get (void) {
  char ch;
  lock_acquire (&lock);
  while (n == 0) {
      lock_release (&lock);
      cond_wait (&not_empty);
      lock_acquire (&lock);
  }
  ch = buf[tail++ % BUF_SIZE];
  n--;
  cond_signal (&not_full);
  lock_release (&lock);
}
```

```c
char buf[BUF_SIZE];
size_t n = 0;
size_t head = 0, tail = 0;
struct lock lock;
struct condition not_empty, not_full;

void put (char ch) {
  lock_acquire (&lock);
  while (n == BUF_SIZE)
        cond_wait (&not_full, &lock);
  buf[head++ % BUF_SIZE] = ch;
  n++;
  cond_signal (&not_empty);
  lock_release (&lock);
}
```

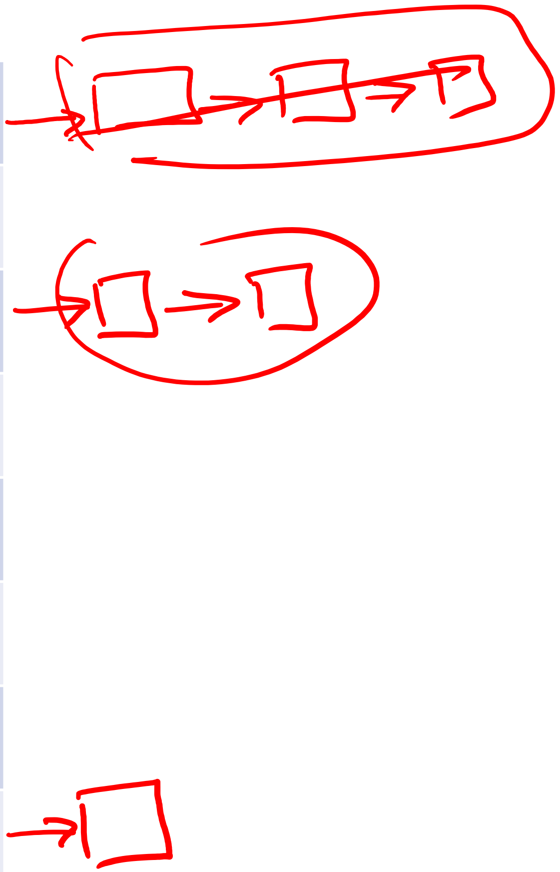
```c
char get (void) {
  char ch;
  lock_acquire (&lock);
  while (n == 0)
        cond_wait (&not_empty, &lock);
  ch = buf[tail++ % BUF_SIZE];
  n--;
  cond_signal (&not_full);
  lock_release (&lock);
}
```

Priority scheduling

- Each thread is created with an initial priority
  - Priority is an integer value
- Let us say the priority range is 0-64
  - A higher value means a high priority
- In round-robin scheduling, the threads are scheduled in a FIFO order
- In priority scheduling threads with the same priority are scheduled in a FIFO order
  - The scheduler always schedules a high priority thread

Priority scheduling

64
63
62
.
.
.
.
0



FIFO

Starvation

- In priority scheduling, a low priority thread may wait infinitely to get scheduled

- To prevent this, an OS periodically (say 15minutes) increase the priority of a low priority thread
  - also called aging

- After a low priority thread (with enhanced priority) gets scheduled the OS sets the priority to its original value

Assignment-1

- Priority scheduling
- Priority donation
- Deadline 9th Feb (Hard deadline, no extensions)
- Design document

Thread1 50

Thread2 40 50

Thread3 30 50

lock_release()

40