Xidian University

《操作系统课程设计》

实验2：Add Priority

黄伯虎

School of Computer Science & Technology

任务

- 原始Pintos系统中对于线程的调度，没有考虑优先级问题，采用的是最为简单的FCFS（先来先服务）策略。

- **本实验要求为Pintos建立优先级调度机制，确保任何时刻CPU上运行的都是最高优先级线程。**

Pintos线程优先级如何定义的？

in threads.h

```
/* Thread priorities. */
#define PRI_MIN 0                       /* Lowest priority. */
#define PRI_DEFAULT 31                  /* Default priority. */
#define PRI_MAX 63                      /* Highest priority. */
```

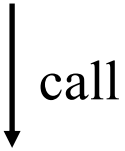
```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved 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. */
```

操作系统中何时会设置/改变优先级

1.线程生成时

in threads.c

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



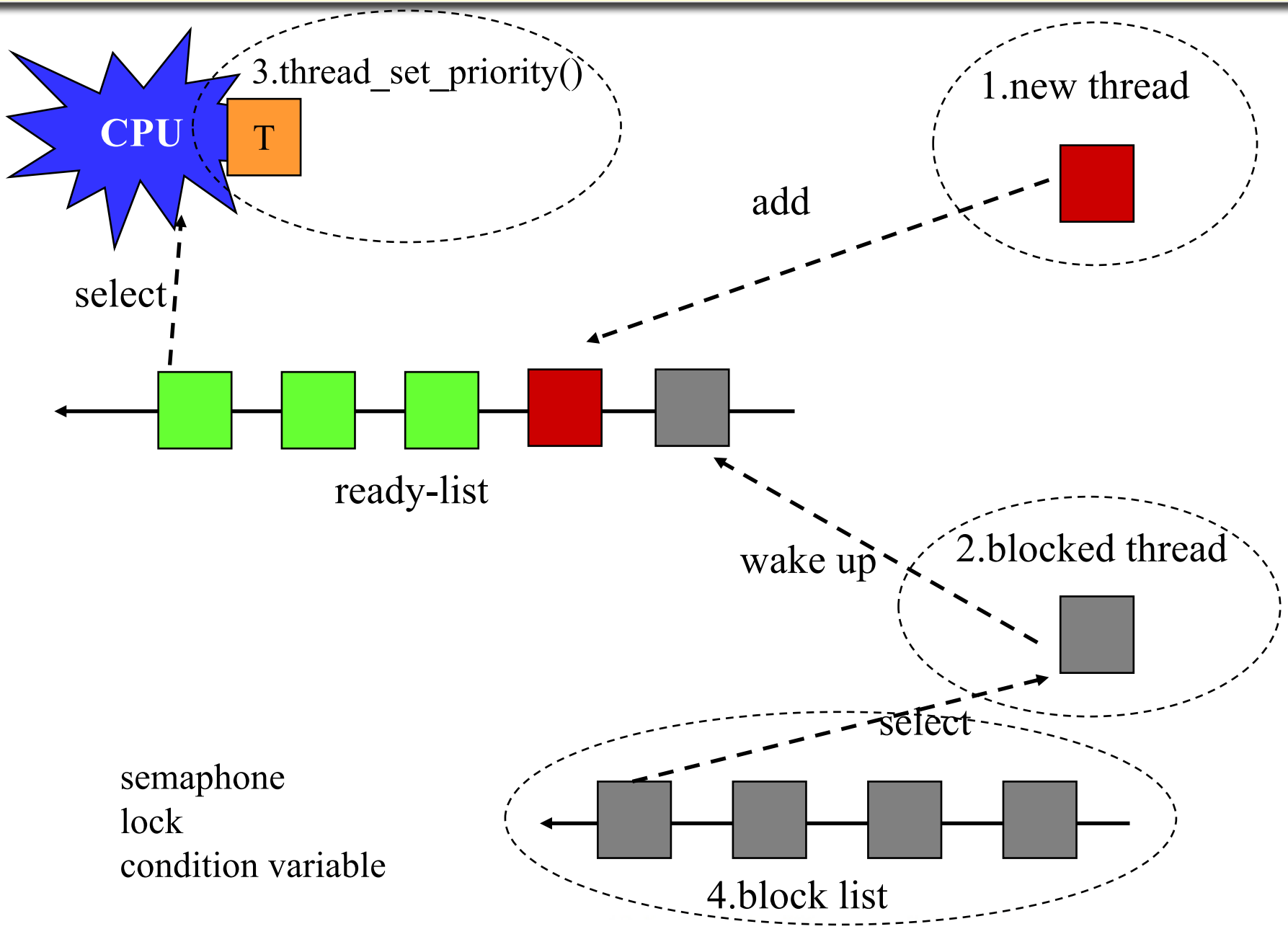

```
init_thread (struct thread *t, const char *name, int priority)
```

2.通过thread_set_priority(函数)改变

in threads.c

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
}
```

何时需要考虑优先级对调度的影响？

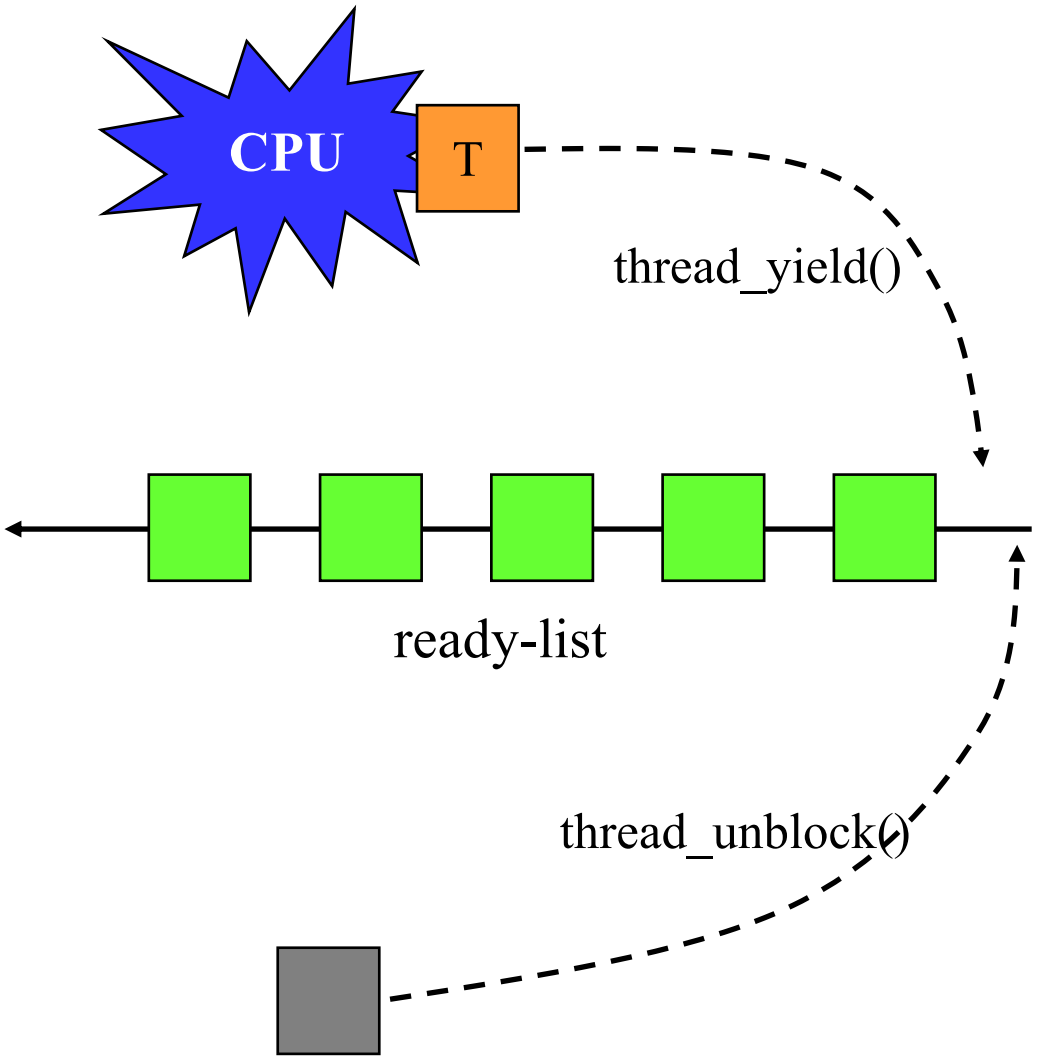


新线程生成时

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

线程生成后，当前线程和ready-list中线程优先级比较，确定当前线程是否让出CPU

ready-list的有序化




```
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);
}
```

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

ready-list的有序化

```
void
list_insert_ordered (struct list *list, struct list_elem *elem,
                     list_less_func *less, void *aux)
{
  struct list_elem *e;

  ASSERT (list != NULL);
  ASSERT (elem != NULL);
  ASSERT (less != NULL);

  for (e = list_begin (list); e != list_end (list); e = list_next (e))
    if (less (elem, e, aux))
      break;
  return list_insert (e, elem);
}
```

调用thread_set_priority()改变线程优先级时

```
void
thread_set_priority (int new_priority)
{
 thread_current ()->priority = new_priority;
}
```

优先级更改后，应当立即比较当前线程和ready-list中线程优先级，确定当前线程是否让出CPU

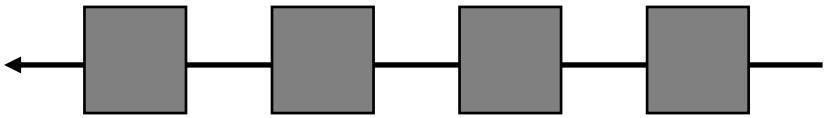
Semaphone

```
/* A counting semaphore. */
struct semaphore
 {
  unsigned value;          /* Current value. */
  struct list waiters;     /* List of waiting threads. */
 };
```

```
void
sema_down (struct semaphore *sema)
{
 enum intr_level old_level;

 ASSERT (sema != NULL);
 ASSERT (!intr_context ());

 old_level = intr_disable ();
 while (sema->value == 0)
   {
    list_push_back (&sema->waiters, &thread_current ()->elem);
    thread_block ();
   }
 sema->value--;
 intr_set_level (old_level);
}
```

```
void
sema_up (struct semaphore *sema)
{
 enum intr_level old_level;

 ASSERT (sema != NULL);

 old_level = intr_disable ();
 if (!list_empty (&sema->waiters))
  thread_unblock (list_entry (list_pop_front (&sema->waiters),
                              struct thread, elem));
 sema->value++;
 intr_set_level (old_level);
}
```

我们的工作：
- 使list有序化
- unblock后进行优先级比较

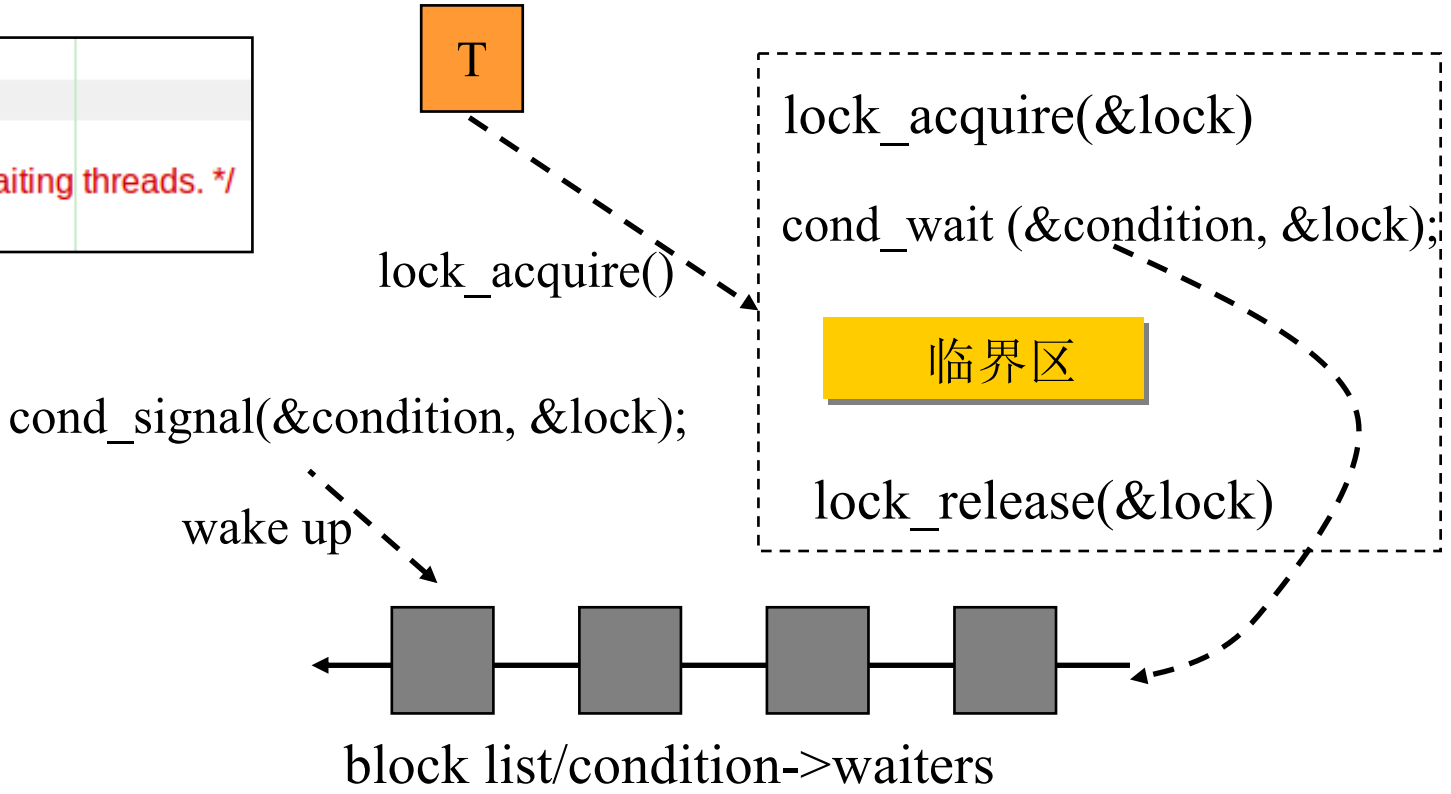


block list/sema->waiters

Lock, Condition varibale

```
/* Lock. */
struct lock
 {
   struct thread *holder;      /* Thread holding lock (for debugging). */
   struct semaphore semaphore; /* Binary semaphore controlling access. */
 };
```

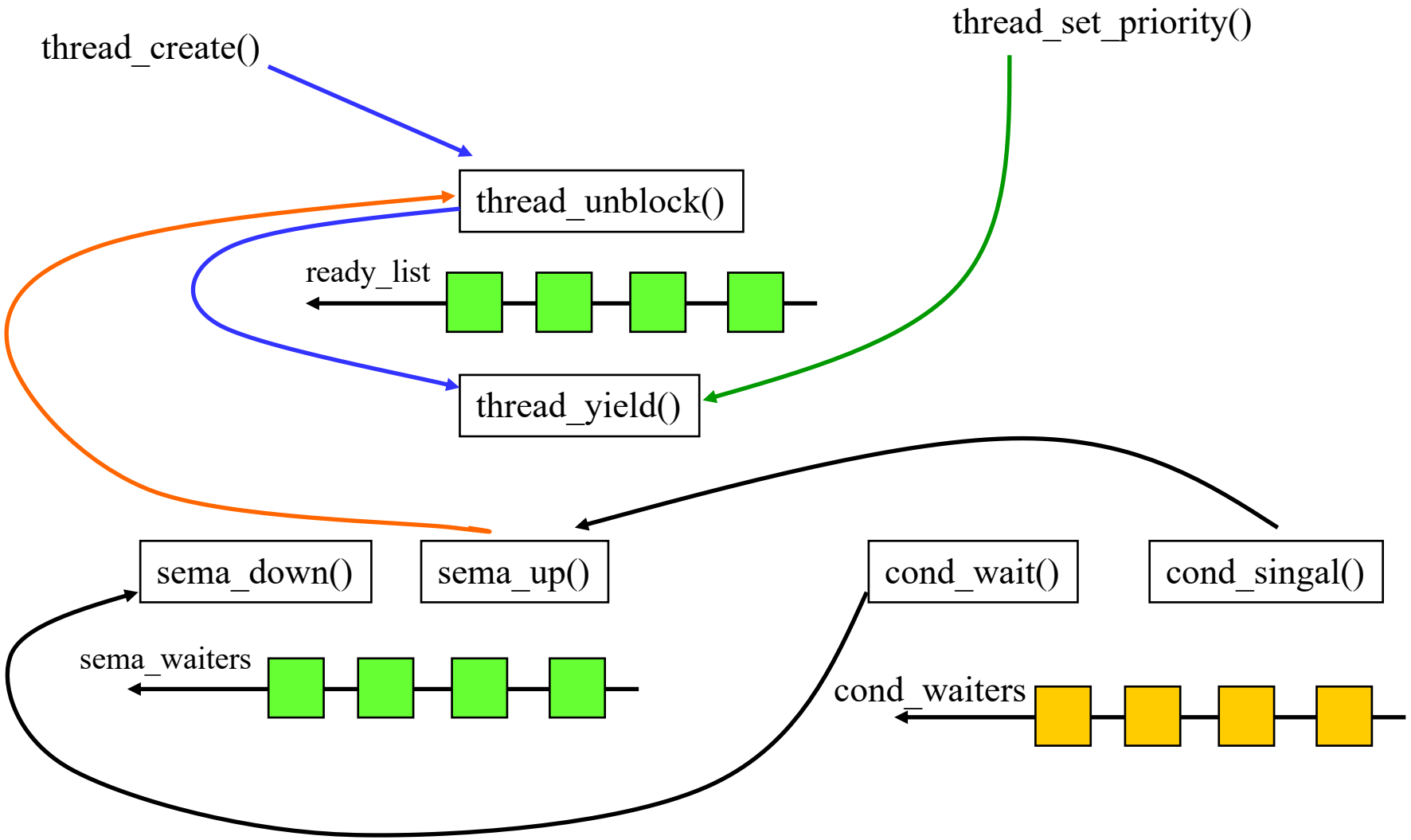
```
/* Condition variable. */
struct condition
 {
   struct list waiters;        /* List of waiting threads. */
 };
```



修改此函数，使cond->waiters队列按优先级有序

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

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  list_push_back (&cond->waiters, &waiter.elem);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

调用关系总结


thread_create()
thread_set_priority()
thread_unblock()
ready_list
thread_yield()
sema_down()
sema_up()
cond_wait()
cond_singal()
sema_waiters
cond_waiters

测试

- 如果你成功完成了这些任务，make check时：
  - **alarm_priority,**
  - **priority-change**
  - **priority-fifo**
  - **priority-preempt**
  - **priority-sema**
  - **priority-condvar**

等6个检测就可以通过。