Race conditions:

```
struct list *list = NULL;

void insert (int data) {
   struct list *l;
   l = malloc (sizeof (struct list));
   l->data = data;
   l->next = list;
   list = l;
}
```

Multiple Processors:

```
Thread 1:               Thread 2:
l->next = NULL;         l->next = NULL;
list = l;               list = l;
```

Single Processor:

```
Thread 1:
l->next = NULL;
schedule();
Thread 2:
l->next = NULL;
list = l;
schedule ();
Thread 1:
list = l;  /* eventually list has only one element */
```

How to ensure atomicity on a Single Processor:

```
void insert (int data) {
   struct list *l;
   l = malloc (sizeof (struct list));
   l->data = data;
   old_level = disable_interrupts ();  /* scheduler cannot run when interrupts are disabled*/
   l->next = list;
   list = l;
   intr_set_level (old_level);
}
```

Scheduler:

The scheduler job is to multiplex a single CPU among multiple threads. One of the widely used scheduling policy is round robin scheduling. In the round-robin scheduling, the scheduler maintains a FIFO list of all the threads (also called ready_list). After a constant time-slice (say 10 ms), it pops a thread from the front of the list and adds the current thread to the end of the list.

Let us see how pintos implements the schedule routine. Pintos maintains a “struct thread” data structure for every thread. The scheduler maintains a ready_list of all the threads that can be scheduled. On every timer interrupt, the interrupt handler first checks if the time-slice has been expired. If yes, it yields the current thread (see timer_interrupt routine in pintos).

```
timer_interrupt ();
thread_yield ();
```

The thread_yield() routine does the following task:

1. disables interrupts to avoid race condition during the manipulation of the ready list and the entire schedule () operation
2. adds the current thread to the end of the ready list,
3. pops the next thread to run from the front of the ready list (schedule () in Pintos)
4. saves all the registers on the stack (switch_threads() in Pintos)
5. saves the stack pointer of the current thread (switch_threads())
6. restores the stack pointer of the target thread (switch_threads())
7. restores registers of the target thread from the stack (switch_threads())

Threads can voluntarily yield. For example, a sleep can be implemented as:

```
tick = get_cpu_tick();
while (get_cpu_tick() – tick < 10) {
   thread_yield ();
}
```

It is not a good idea to disable interrupts for a longer duration. Unnecessarily blocking other threads is not suitable for an interactive application. A thread cannot do IO with interrupts are disabled (most of the devices require interrupts to be enabled). Usually, interrupt handlers are very small, don’t do any IO and executed with interrupt disabled.

To handle large critical section or cases when a thread wants to interact with other threads a different data structure is used which is called semaphore. For example:

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

Here parent thread wants to wait until child thread is scheduled.

```
struct semaphore {
  int value;
  struct list waiters;
};
```

```
/* sema_init initializes the value to val. val must me a non-negative number.*/

sema_init (struct semaphore *s, int val) {
     s->value = val;
}
```

A thread waits in a wait_list until the semaphore value is zero. Semaphore value can never be negative.

```
sema_down (struct semaphore *s) {
    old_level = interrupts_disable ();
    while (s->value == 0) {
        list_insert (&s->waiters, thread_current());
        schedule();
     }
     s->value--;
     interrupt_set_label (old_level);
}
```

sema_up wakes up a waiting thread by adding them to the ready list and also increments the semaphore value.

```
sema_up () {
   old_level = interrupts_disable ();
    if (!list_empty (&sema->waiters)) {
        thread = list_pop (&s->waiters);
        add thread to ready_list
     }
     s->value++;
     interrupt_set_label (old_level);
}
```

A lock can be implemented by initializing a semaphore to 1.

```
struct lock {
    struct semaphore s;
};

lock_init (struct lock *l) {
    sema_init (&l->s, 1);
}

lock_acquire (struct lock *l) {
    sema_down (&l->s);
}

lock_release (struct lock *l) {
   sema_up (&l->s);
}
```

Condition variables

```
char buf[BUF_SIZE];     /* Buffer. */
size_t n = 0;           /* 0 <= n <= BUF_SIZE: # of characters in buffer. */
size_t head = 0;        /* buf index of next char to write (mod BUF_SIZE). */
size_t tail = 0;        /* buf index of next char to read (mod BUF_SIZE). */
struct lock lock;       /* lock. */
struct condition not_empty; /* Signaled when the buffer is not empty. */
struct condition not_full; /* Signaled when the buffer is not full. */
```

...initialize the locks and condition variables...

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

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

Consider the above example: The "put" routine adds a character to a buffer. The "get" routine consumes a character from the buffer. The buffer size if fixed. If the buffer is full, the "put" routine waits until the "get" routine consumes a character. Similarly, if the buffer is empty, the "get" routine waits for the "put" routine to add a character to the buffer. The buffer is implemented as a fixed size circular queue. Notice, as we have discussed earlier that spinning is not a good solution for all operations. To handle this case, we have a different abstraction called cond_wait and cond_signal. The cond_wait wait until a given condition is true, and cond_signal wakes up a thread which is waiting on a given condition. The above code can be written as:

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

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

Notice, that the above code is still incorrect because multiple threads are accessing shared data structure ("n"). To prevent this, we can use locks.

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

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

But we still have a problem. Because a thread may be waiting on some condition while acquiring the lock, nobody will be able to signal that thread. This scenario is also called deadlock. To avoid the deadlock, we can release and acquire locks as below.

```
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;     /* Add ch to buf. */
  n++;
  cond_signal (&not_empty);
  lock_release (&lock);
}
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];    /* Get ch from buf. */
  n--;
  cond_signal (&not_full);
  lock_release (&lock);
}
```

Even this code is not correct. Consider a scenario when the "get" routine found that n is zero and it released the lock. Let us say at this point, "put" was scheduled, which added a character to the buffer and waked up a thread which was waiting on the condition "not_empty" (currently no one because "get" hasn't called cond_wait yet!). After that, "get" was scheduled and it called "cond_wait". Notice that at this point, even if there is some data in the buffer to consume, the get routine is still waiting because it missed a wakeup call from the "put" routine.

The lost wakeup problem exists because the release of lock and waiting on the condition is not atomic. To prevent this cond_wait also takes "lock" as input and ensures the atomicity of lock release and conditional wait. The final code is given below:

```
void put (char ch) {
  lock_acquire (&lock);
  while (n == BUF_SIZE)            /* Can't add to buf as long as it's full. */
      cond_wait (&not_full, &lock);
  buf[head++ % BUF_SIZE] = ch;     /* Add ch to buf. */
  n++;
  cond_signal (&not_empty); /* buf can't be empty anymore. */
  lock_release (&lock);
}

char get (void) {
  char ch;
  lock_acquire (&lock);
  while (n == 0)                 /* Can't read buf as long as it's empty. */
      cond_wait (&not_empty, &lock);
  ch = buf[tail++ % BUF_SIZE];    /* Get ch from buf. */
  n--;
  cond_signal (&not_full); /* buf can't be full anymore. */
  lock_release (&lock);
}
```