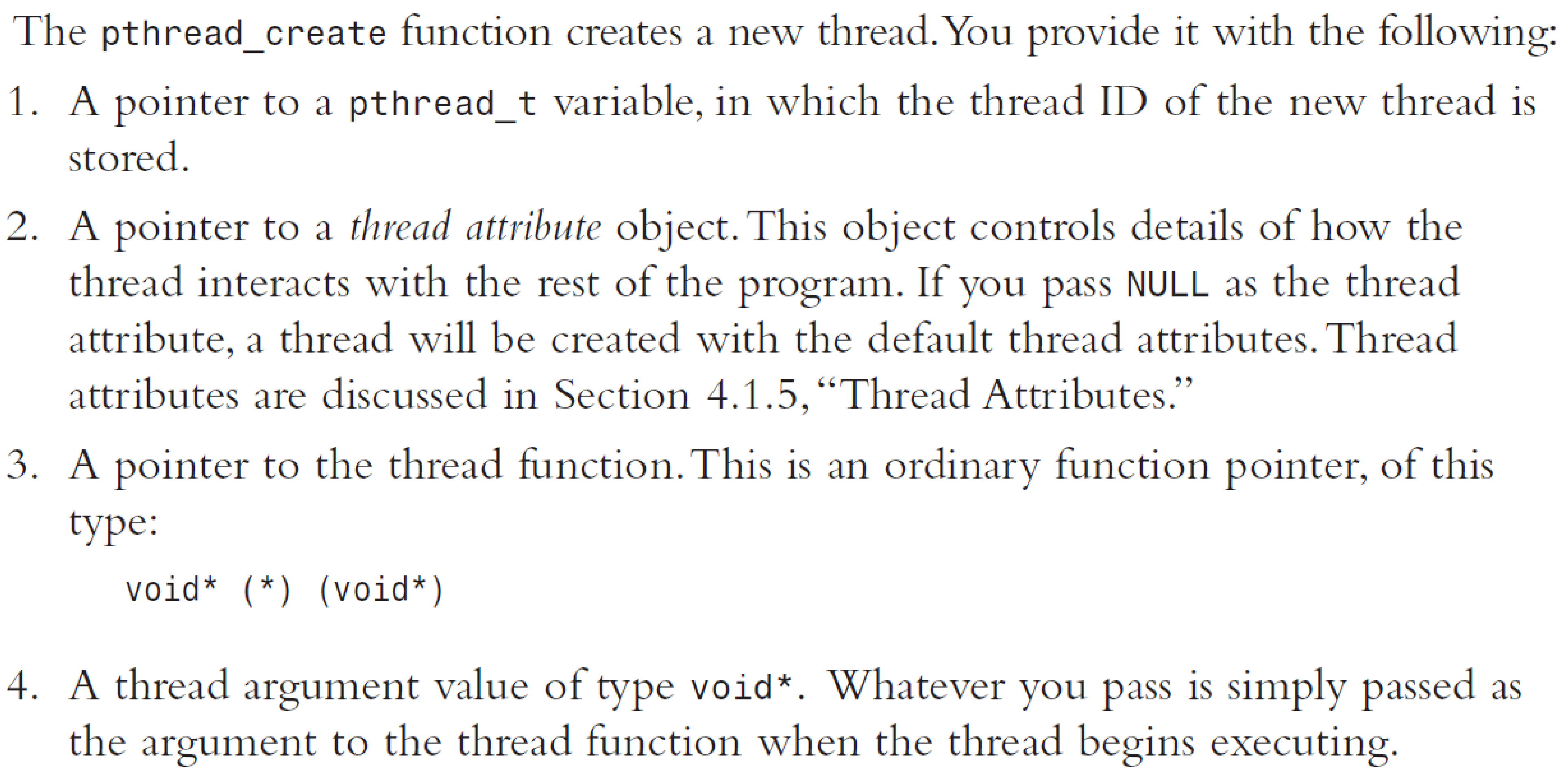
Thread-programming in linux

A short introduction of using posix-threads (pthreads)

Thread creation



The `pthread_create` function creates a new thread. You provide it with the following:

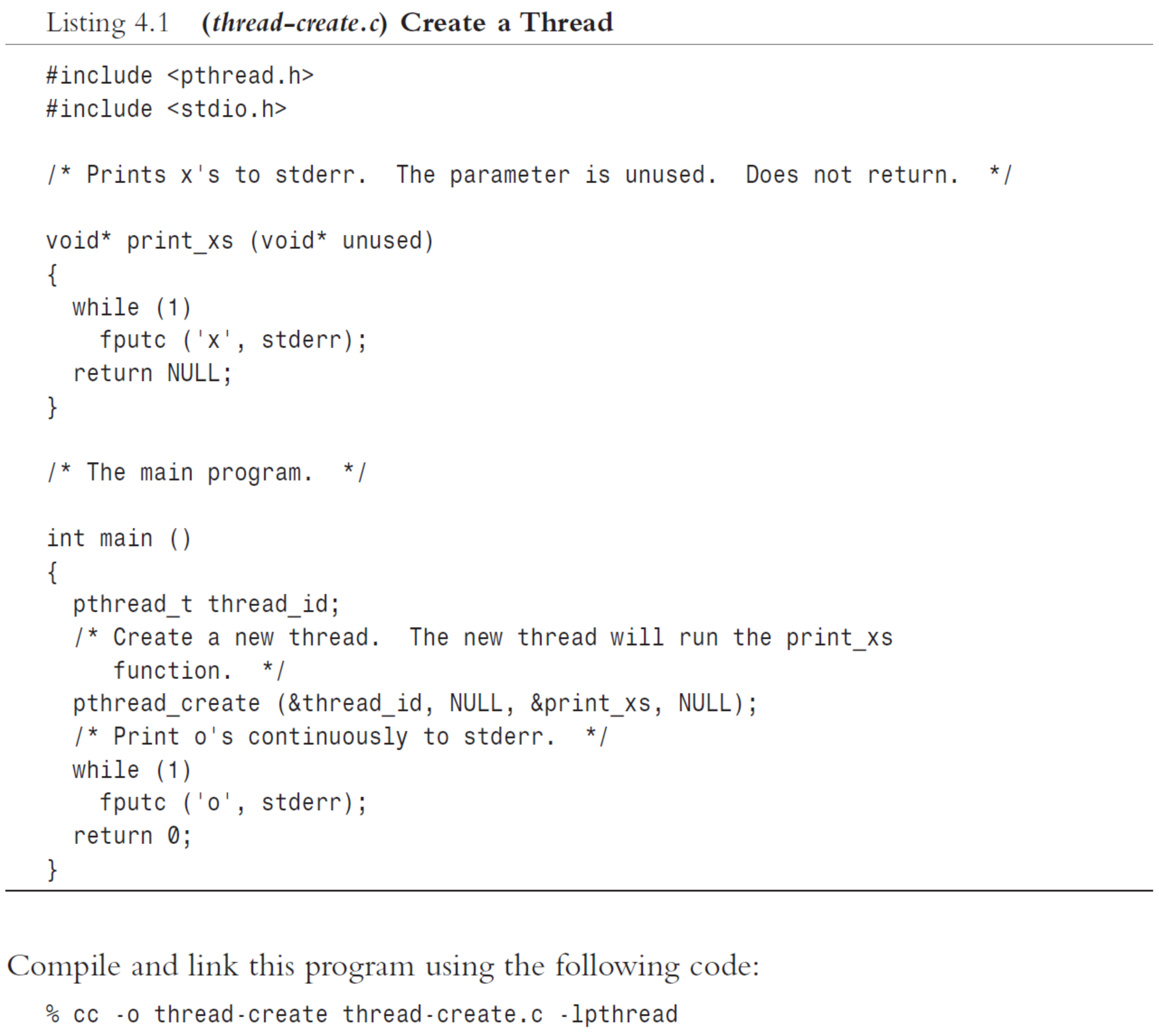
1. A pointer to a `pthread_t` variable, in which the thread ID of the new thread is stored.
2. A pointer to a *thread attribute* object. This object controls details of how the thread interacts with the rest of the program. If you pass `NULL` as the thread attribute, a thread will be created with the default thread attributes. Thread attributes are discussed in Section 4.1.5, "Thread Attributes."
3. A pointer to the thread function. This is an ordinary function pointer, of this type:

   ```
   void* (*) (void*)
   ```

4. A thread argument value of type `void*`. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

```
pthread_create (&thread_id, NULL, &print_xs, NULL);
```

Example



Listing 4.1 ***(thread-create.c)*** **Create a Thread**

```
#include <pthread.h>
#include <stdio.h>

/* Prints x's to stderr.  The parameter is unused.  Does not return.  */

void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}

/* The main program.  */

int main ()
{
  pthread_t thread_id;
  /* Create a new thread.  The new thread will run the print_xs
     function.  */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Print o's continuously to stderr.  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

Compile and link this program using the following code:

```
% cc -o thread-create thread-create.c -lpthread
```

Waiting for threads to die: joining threads

```
/* Make sure the first thread has finished.  */
pthread_join (thread1_id, NULL);
/* Make sure the second thread has finished.  */
pthread_join (thread2_id, NULL);

/* Now we can safely return.  */
return 0;
```

Race conditions

- When the result depends on which thread comes first or last we talk about a race condition

```
int saldo = 100;// must be positive or zero
```

```
T1:

    T1_req = 80;
A1: if ( saldo > T1_req )
A2:     { saldo = saldo – T1_req;}
    else
A3:   { cout <<” request too high”;}
```

```
T2:

    T2_req = 70;
B1: if ( saldo > T2_req )
B2:     { saldo = saldo – T2_req;}
    else
B3:   { cout <<” request too high”;}
```

- Interleavings
  - A1A2B1B3 OK
  - B1B2A1A3 OK
  - A1B1A2B2 not OK saldo ?
  - A1B1B2A2 not OK saldo?

Mutexes for solving race conditions
Mutex: mutual exclusion

- Declaration and initialization of a mutex variable:

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

- Or even simpler:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Locking a mutex (if it is locked the thread will be blocked until the mutex is unlocked):

```
/* Lock the mutex on the job queue.  */
pthread_mutex_lock (&job_queue_mutex);
```

- Unlocking a mutex can only be done by the one who locked it:

```
pthread_mutex_unlock (&job_queue_mutex);
```

Non-bloocking wait for a mutex

```
while ( pthread_mutex_trylock( &my_mutex ) == EBUSY )
{
 // do something else for a while
 }
// now we got the mutex (under normal circumstances)
```

Race condition solved

- When the result depends on which thread comes first or last we talk about a race condition

```
pthread_mutex_t   saldo_mutex = PTHREAD_MUTEX_INITIALIZER;
int saldo = 100;// must be positive or zero
```

```
T1:

    T1_req = 80;
A0: pthread_mutex_lock(&saldo_mutex);
A1: if ( saldo > T1_req )
A2:     { saldo = saldo – T1_req;}
    else
A3:   { cout <<" request too high";}
A4: pthread_mutex_unlock(&saldo_mutex);
```

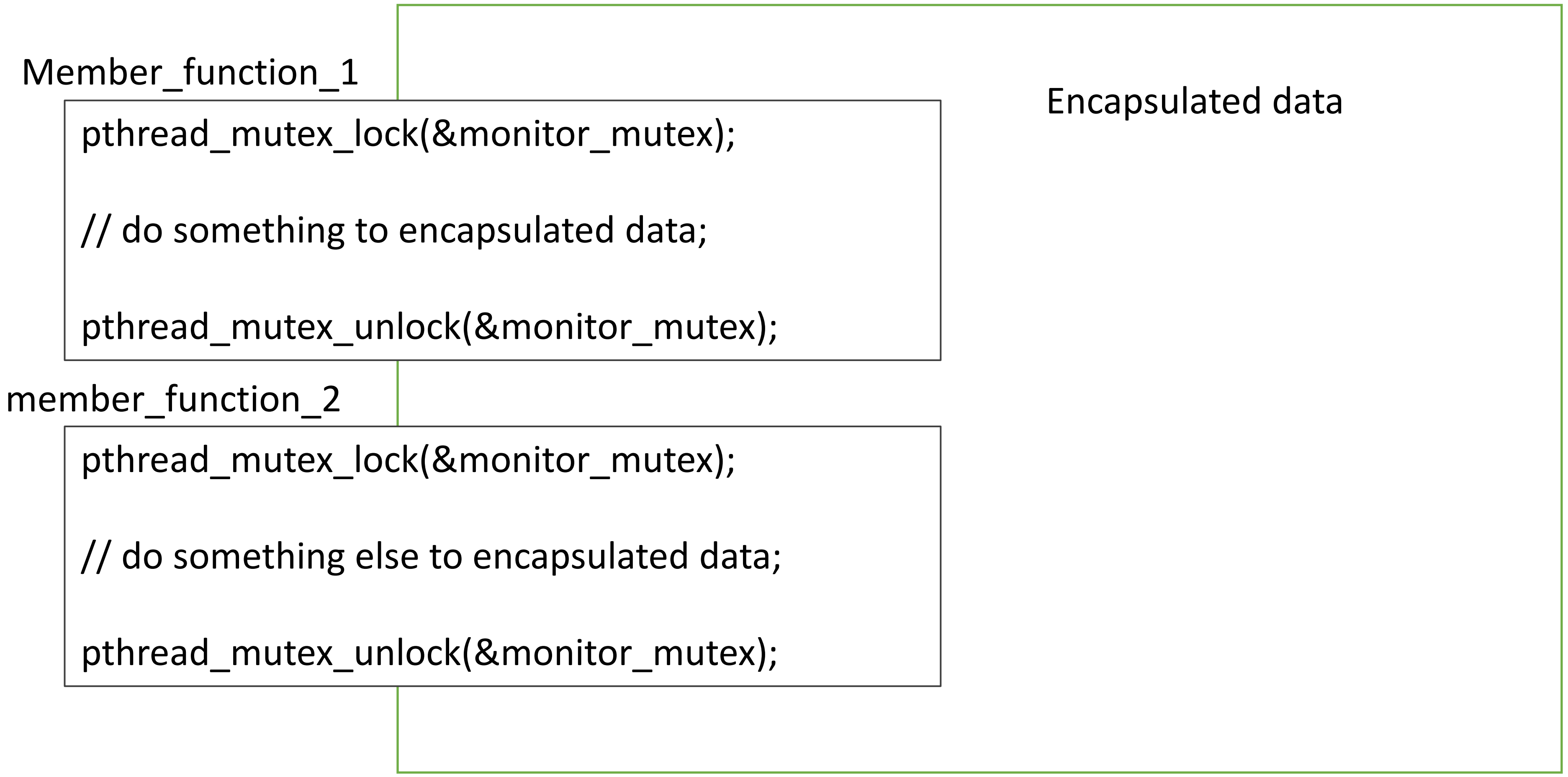
```
T2:

    T2_req = 70;
B0: pthread_mutex_lock(&saldo_mutex);
B1: if ( saldo > T2_req )
B2:     { saldo = saldo – T2_req;}
    else
B3:   { cout <<" request too high";}
B4: pthread_mutex_unlock(&saldo_mutex);
```

- All interleavings OK now
    - A0A1A2A4B0B1B3B4 OK
    - A0A1B0B1A2B2 not possible T1 has the lock on the mutex

Monitor class

- Mutual exclusive access to call one of the member functions.



```
pthread_mutex_lock(&monitor_mutex);

// do something to encapsulated data;

pthread_mutex_unlock(&monitor_mutex);
```

```
pthread_mutex_lock(&monitor_mutex);

// do something else to encapsulated data;

pthread_mutex_unlock(&monitor_mutex);
```

- a much safer approach for shared data !

Semaphores

- For controlling access to several identical resources
- For signalling events to other threads
- A semahore has a counter and two atomic operations wait and post

```
sem_wait:
If ( counter == o )
{ block thread
  counter--;}
else
{ counter --;}
```

```
sem_post:
If ( some thread is blocked at this semaphore )
{ counter++; wake one up}
else
{ counter ++;}
```

Semaphores in <semaphore.h>

- Declaring a semaphore variable:

```
/* A semaphore counting the number of jobs in the queue.  */
sem_t job_queue_count;
```

- Initialization of a semaphore variable: 2.parameter 0 and 3.parameter initial value of counter:

```
/* Initialize the semaphore which counts jobs in the queue.  Its
   initial value should be zero.  */
sem_init (&job_queue_count, 0, 0);
```

- Waiting at a semaphore variable for one resource:

```
/* Wait on the job queue semaphore.  If its value is positive,
   indicating that the queue is not empty, decrement the count by
   1.  If the queue is empty, block until a new job is enqueued.  */
sem_wait (&job_queue_count);
```

- Releasing a resource:

```
/* Post to the semaphore to indicate that another job is available.  If
   threads are blocked, waiting on the semaphore, one will become
   unblocked so it can process the job.  */
sem_post (&job_queue_count);
```

The full example code

Listing 4.12 (*job-queue3.c*) **Job Queue Controlled by a Semaphore**

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>

struct job {
  /* Link field for linked list.  */
  struct job* next;

  /* Other fields describing work to be done... */
};

/* A linked list of pending jobs.  */
struct job* job_queue;

/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
```

continued

```
/* A semaphore counting the number of jobs in the queue.  */
sem_t job_queue_count;

/* Perform one-time initialization of the job queue.  */

void initialize_job_queue ()
{
  /* The queue is initially empty.  */
  job_queue = NULL;
  /* Initialize the semaphore which counts jobs in the queue.  Its
     initial value should be zero.  */
  sem_init (&job_queue_count, 0, 0);
}
```

continued

```
/* Process queued jobs until the queue is empty.  */

void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;

    /* Wait on the job queue semaphore.  If its value is positive,
       indicating that the queue is not empty, decrement the count by
       1.  If the queue is empty, block until a new job is enqueued.  */
    sem_wait (&job_queue_count);

    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty.  Get
       the next available job.  */
    next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Unlock the mutex on the job queue because we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);

    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
```

continued

```
 /* Add a new job to the front of the job queue.  */

 void enqueue_job (/* Pass job-specific data here...  */)
 {
   struct job* new_job;


  /* Allocate a new job object.  */
  new_job = (struct job*) malloc (sizeof (struct job));
  /* Set the other fields of the job struct here...  */

  /* Lock the mutex on the job queue before accessing it.  */
  pthread_mutex_lock (&job_queue_mutex);
  /* Place the new job at the head of the queue.  */
  new_job->next = job_queue;
  job_queue = new_job;

  /* Post to the semaphore to indicate that another job is available.  If
     threads are blocked, waiting on the semaphore, one will become
     unblocked so it can process the job.  */
  sem_post (&job_queue_count);

  /* Unlock the job queue mutex.  */
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Signalling events using semaphores

- Declare an ”event” semaphore:

```
sem_t event_sem;
```

- Initialize the counter of ”event”semaphore counter to zero:

```
sem_init(&event_sem, 0, 0);
```

- Receiver of the event notification waits for the event like this:

```
sem_wait(&event_sem);
```

- Sender of the event notification signals the event like this:

```
sem_post(&event_sem);
```

4.4.6 Condition variables for ”blocking” inside critical sections

- A condition variable is represented by an instance of pthread_cond_t. Remember that each condition variable should be accompanied by a mutex.
- These are the functions that manipulate condition variables:
  - **pthread_cond_init** initializes a condition variable. The first argument is a pointer to a pthread_cond_t instance. The second argument, a pointer to a condition variable attribute object, is ignored under GNU/Linux.
  - **pthread_cond_signal** signals a condition variable. A single thread that is blocked on the condition variable will be unblocked. If no other thread is blocked on the condition variable, the signal is ignored. The argument is a pointer to the pthread_cond_t instance.
  - A similar call, **pthread_cond_broadcast**, unblocks all threads that are blocked on the condition variable, instead of just one.
  - **pthread_cond_wait** blocks the calling thread until the condition variable is signaled. The argument is a pointer to the pthread_cond_t instance. The second argument is a pointer to the pthread_mutex_t mutex instance.
    - When pthread_cond_wait is called, the mutex must already be locked by the calling thread. That function atomically unlocks the mutex and blocks on the condition variable. When the condition variable is signaled and the calling thread unblocks, pthread_cond_wait automatically reacquires a lock on the mutex.

Applying condition variables

- When you have to wait inside a critical region

```
pthread_mutex_t   saldo_mutex = PTHREAD_MUTEX_INITIALIZER;
int saldo = 10;// must be positive or zero
```

T1:

```
T1_req = 80;
pthread_mutex_lock(&saldo_mutex);
  while ~~if~~ ( saldo < T1_req )
    { pthread_cond_wait(&sufficient_saldo_cond, &saldo_mutex);  }
    saldo = saldo – T1_req;
pthread_mutex_unlock(&saldo_mutex);
```

T2:

```
pthread_mutex_lock(&saldo_mutex);
   saldo = saldo + 100;
   pthread_cond_signal(&sufficient_saldo_cond );
pthread_mutex_unlock(&saldo_mutex);
```

- Having a ”while” instead of ”if ” is vital ! Why ?
- Name a condition variable as the condition to wait for.

Listing 4.14 (condvar.c) Control a Thread Using a Condition Variable I

```
#include <pthread.h>
int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag ()
{
/* Initialize the mutex and condition variable.  */
pthread_mutex_init (&thread_flag_mutex, NULL);
pthread_cond_init (&thread_flag_cv, NULL);
/* Initialize the flag value.  */
thread_flag = 0;
}
```

Listing 4.14 (condvar.c) Control a Thread Using a Condition Variable II

```
void* thread_function (void* thread_arg)
{
/* Loop infinitely.  */
while (1) {
/* Lock the mutex before accessing the flag value.  */
pthread_mutex_lock (&thread_flag_mutex);
while (!thread_flag)
{
/* The flag is clear.  Wait for a signal on the condition variable, indicating that the
flag value has changed.  When the signal arrives and this thread unblocks, loop and
check the flag again.  */
 pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
/* When we’ve gotten here, we know the flag must be set. Unlock the mutex.*/
 pthread_mutex_unlock (&thread_flag_mutex);
/* Do some work.  */
 do_work ();
}
return NULL;
}
```

Listing 4.14 (condvar.c) Control a Thread Using a Condition Variable III

```
/* Sets the value of the thread flag to FLAG_VALUE.  */

void set_thread_flag (int flag_value)

{

/* Lock the mutex before accessing the flag value.  */

pthread_mutex_lock (&thread_flag_mutex);

/* Set the flag value, and then signal in case thread_function is blocked, waiting for
the flag to become set.  However,thread_function can’t actually check the flag until
the mutex is unlocked.  */

thread_flag = flag_value;

pthread_cond_signal (&thread_flag_cv);

/* Unlock the mutex.  */

pthread_mutex_unlock (&thread_flag_mutex);

}
```

Using a condition variable outside a critical section

- A condition variable may also be used without a condition, simply as a mechanism for blocking a thread until another thread "wakes it up."
- A semaphore may also be used for that purpose.
- The principal differences are
  - that a semaphore "remembers" the wake-up call even if no thread was blocked on it at the time, while a condition variable discards the wake-up call unless some thread is actually blocked on it at the time.
  - Also, a semaphore delivers only a single wake-up per post; with **pthread_cond_broadcast**, an arbitrary and unknown number of blocked threads may be awoken at the same time.