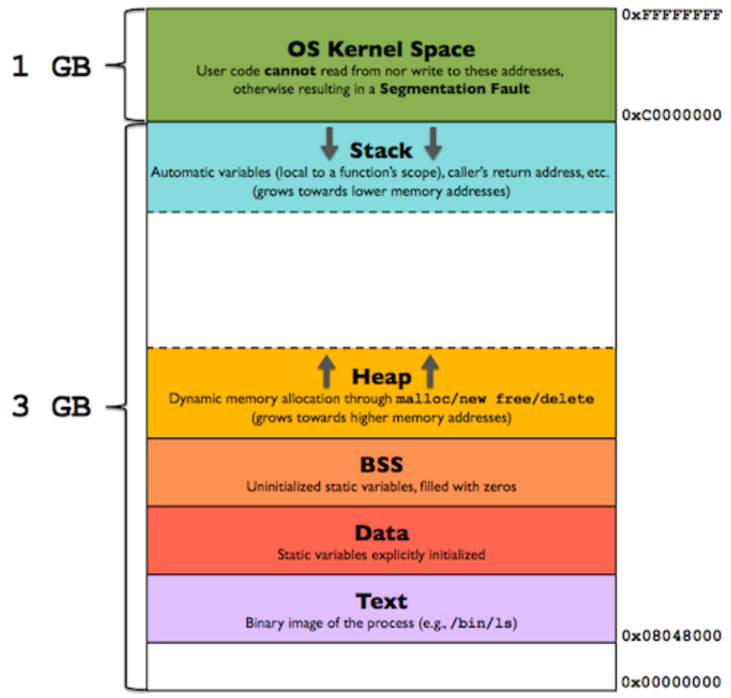
Chapter 8: Threads

Address Spaces

- An address space is the set of addresses in RAM that a process can use.
- Multiple programs in memory need OS to partition the available memory
  - Keep programs from interfering with each other and OS.

Address Spaces

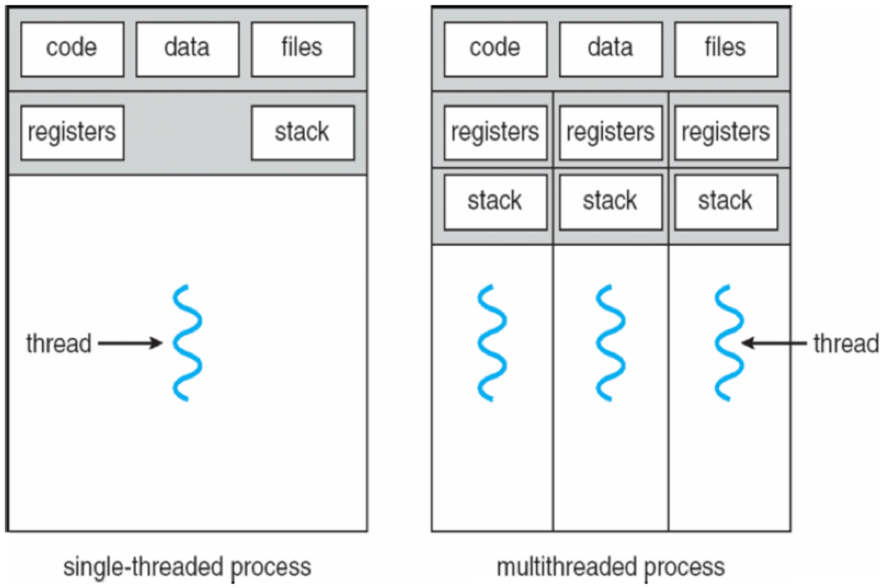


1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

32-bit address = 2^{32} = 4GB of address space

- Kernel gets upper 1 GB

Threads



» In traditional operating systems, each process has an address space and a single line of execution.

» Threads are multiple lines of execution in a shared address space

Thread benefits

» Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:

  – Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.

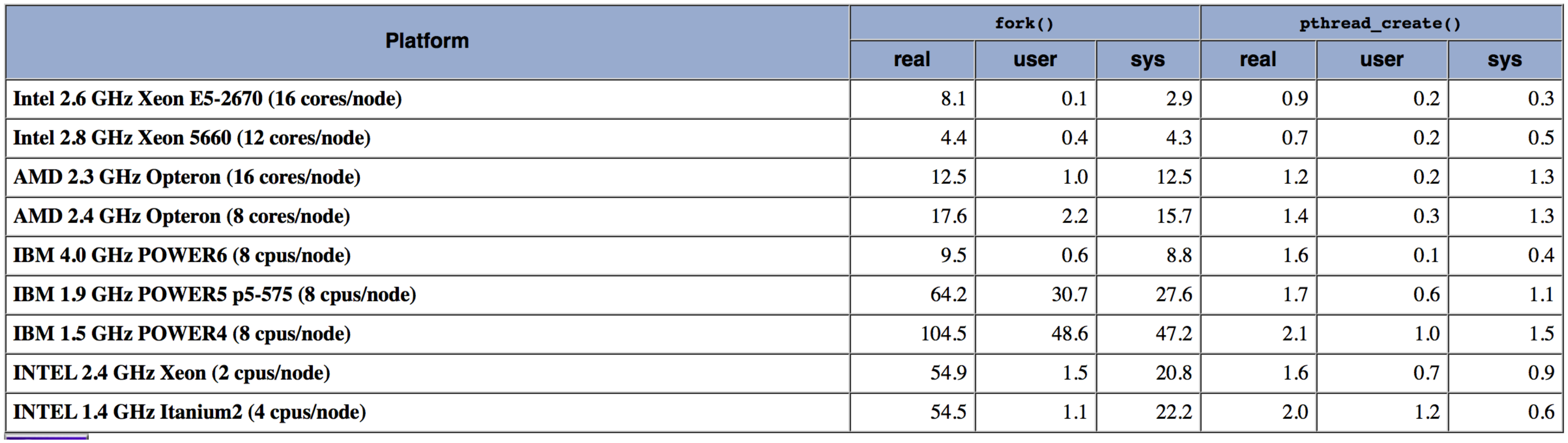
  – Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.

Thread benefits

- Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests
  - A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.

Thread Benefits

» When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.



Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

The Classical Thread Model (1)

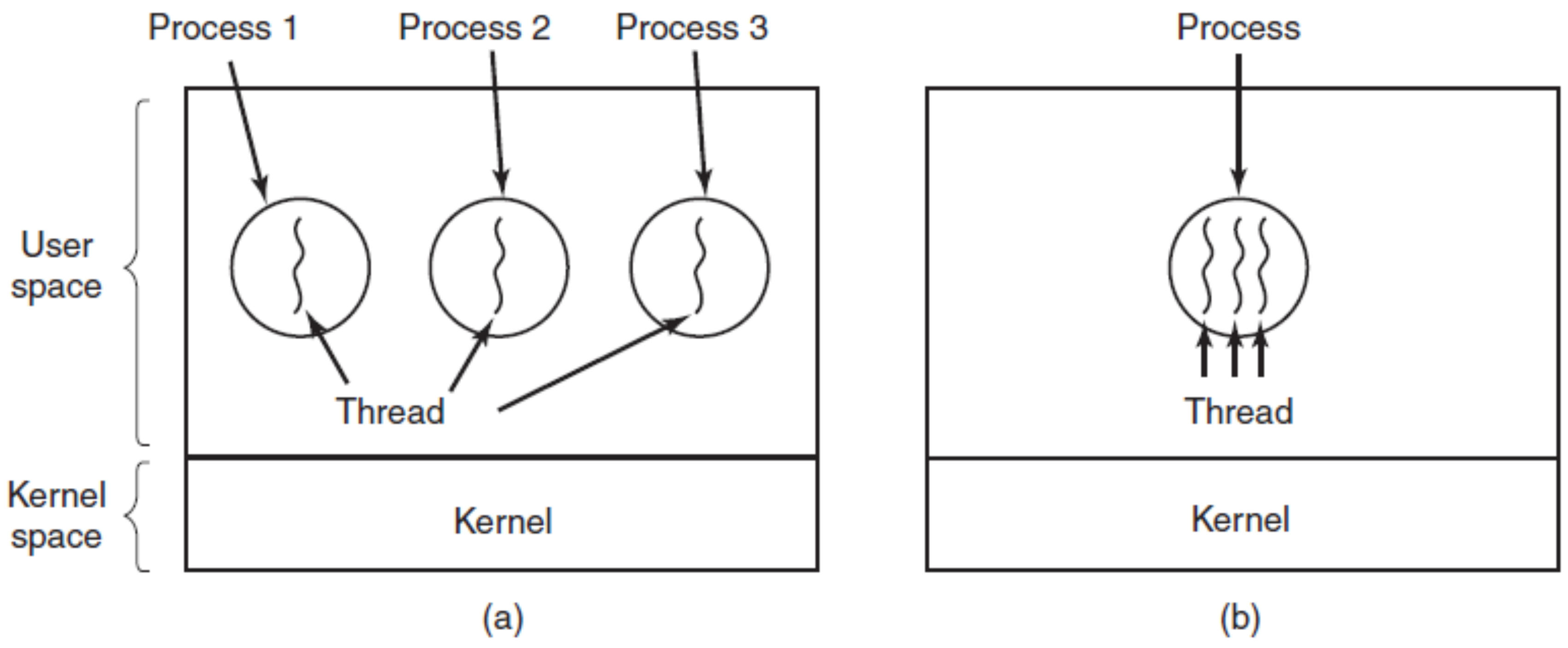


Figure 2-11. (a) Three processes each with one thread.
(b) One process with three threads.

The Classical Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

The Classical Thread Model (3)

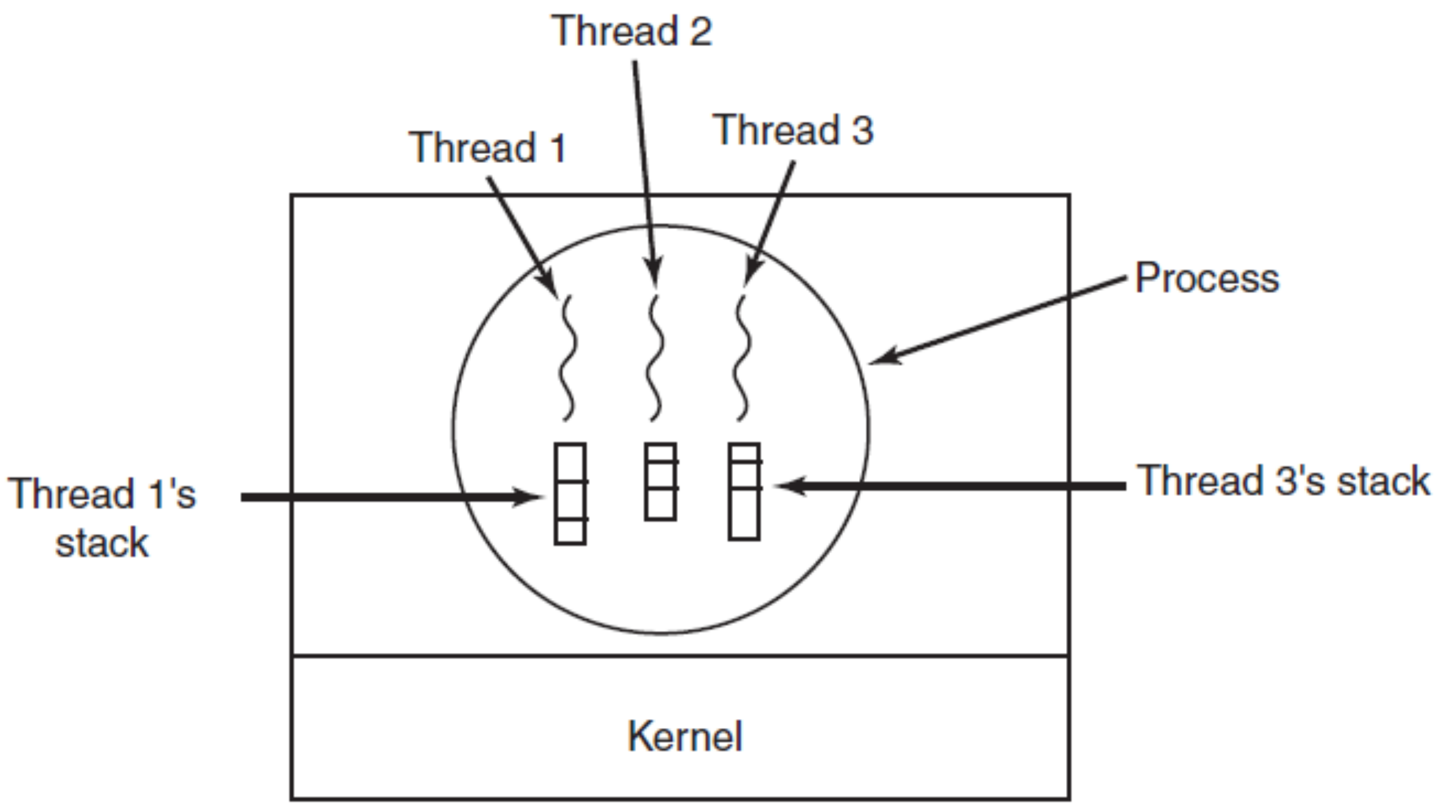


Figure 2-13. Each thread has its own stack.

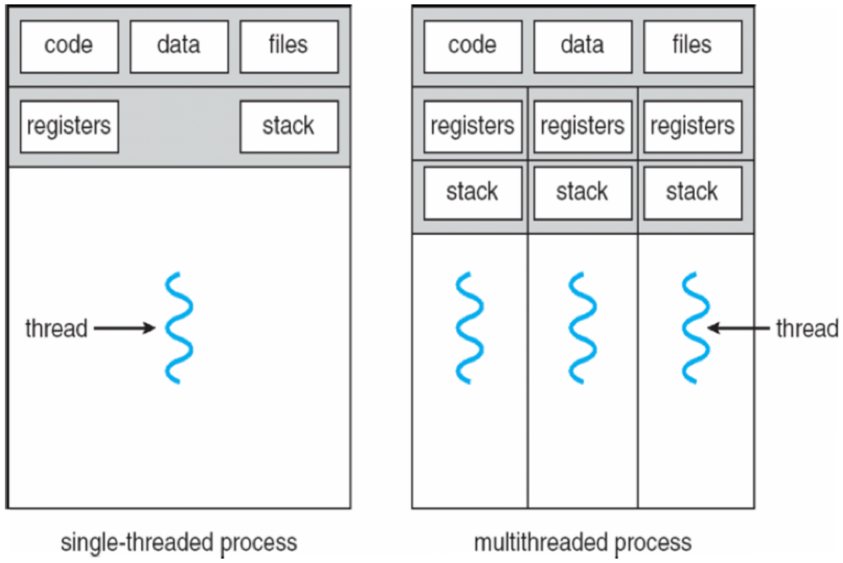
The Classical Thread Model

- Having multiple threads running in parallel is analogous to multiple processes running in parallel. With the exception:
  - In the former the threads share an address space and resources
  - In the latter the the processes share physical memory, disks, printers, etc.

Lightweight Processes

» Because threads share some of the characteristics of process, threads are sometimes referred to as lightweight processes.

The Classical Thread Model



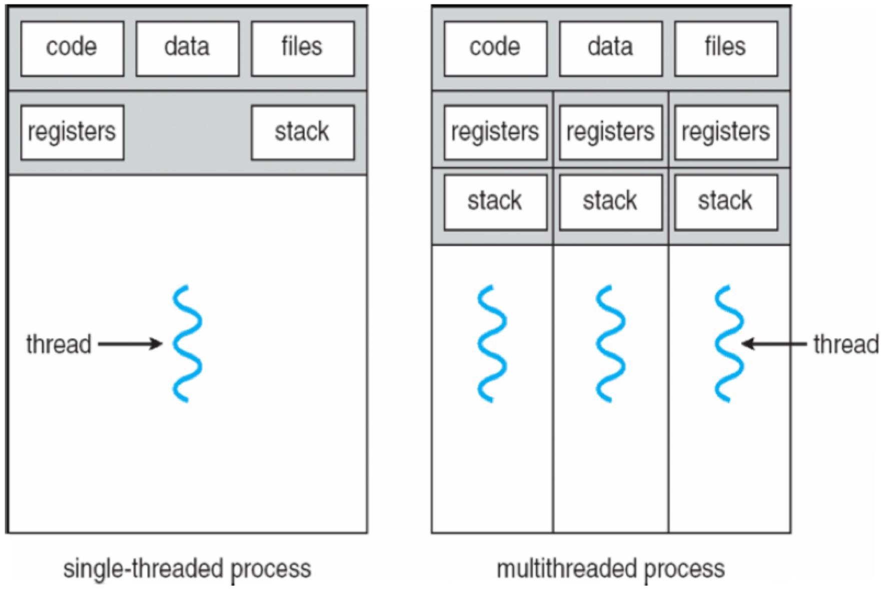
- Different threads are not as independent as processes.
  - Shared address space means no protection between threads.
  - All threads share the same global variables.

The Classical Thread Model (2)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Figure 2-12. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

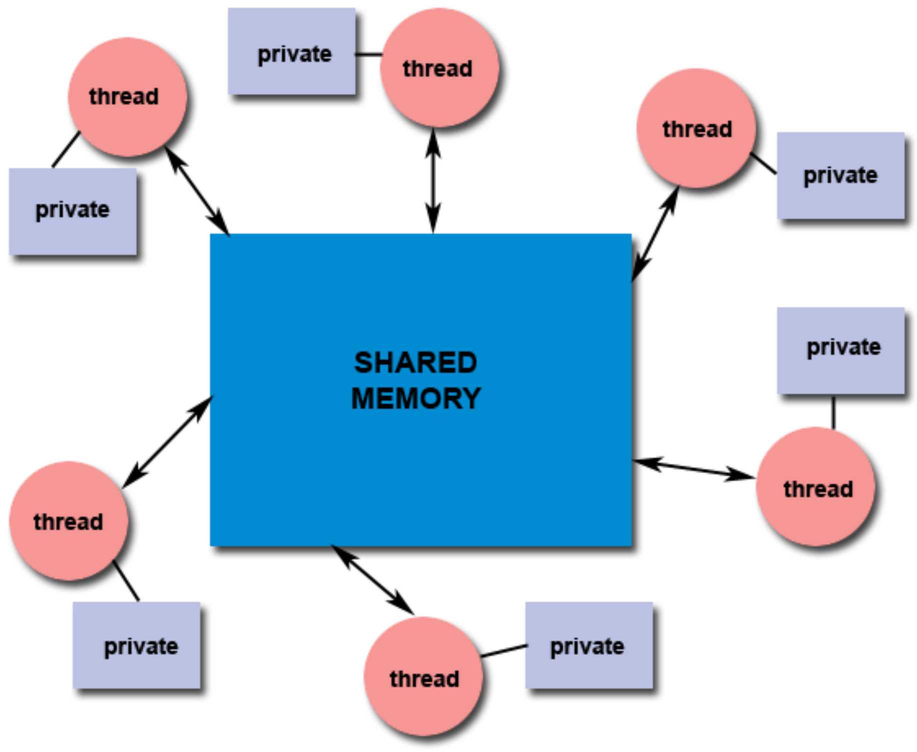
The Classical Thread Model



» Since each thread calls different procedures and has a different execution history, it needs its own stack.

» As with processes, each thread will follow the process lifecycle.

Shared Memory Model



» All threads have access to the same global, shared memory

» Threads also have their own private data

» Programmers are responsible for synchronizing access (protecting) globally shared data.

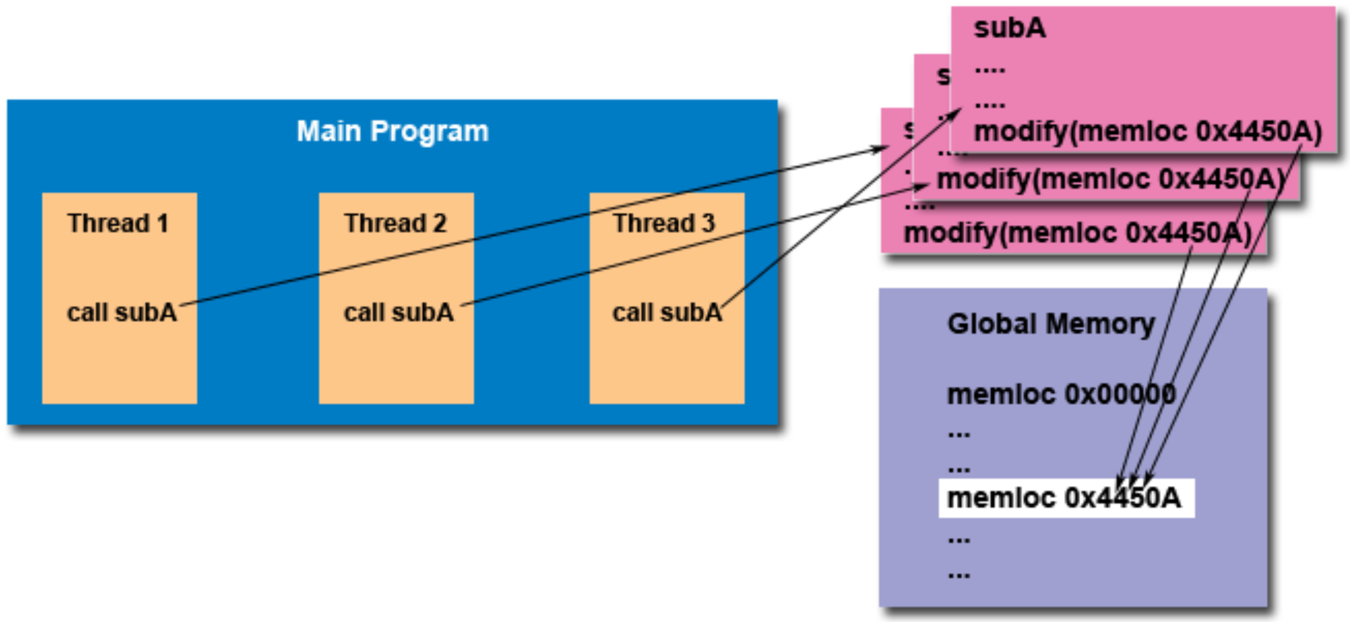
Thread Safeness

» Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.

» For example, suppose that your application creates several threads, each of which makes a call to the same library routine:

» This library routine accesses/modifies a global structure or location in memory.

Thread Safeness



- As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
- If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.

Thread Safeness

- The implication to users of external library routines is that if you aren't 100% certain the routine is thread-safe, then you take your chances with problems that could arise.
- Recommendation: Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine, etc.

Potentially Thread Unsafe

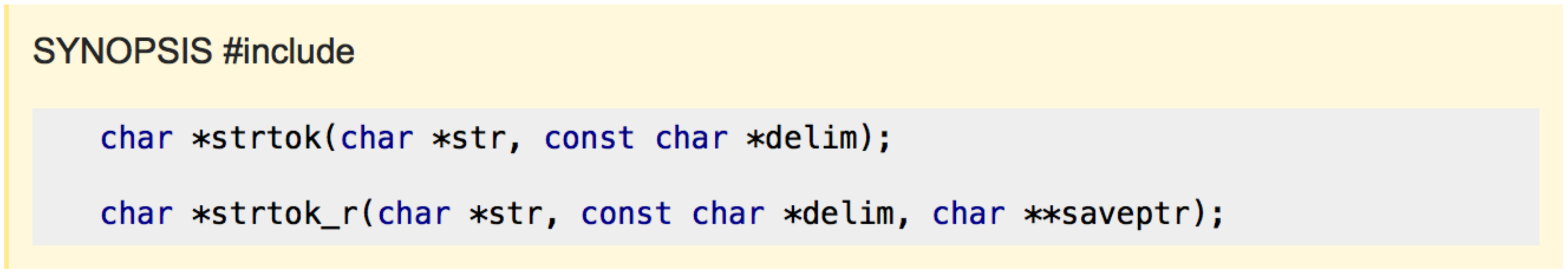
All functions defined by this volume of POSIX.1-2008 shall be thread-safe, except that the following functions need not be thread-safe

- asctime()
- basename()
- catgets()
- crypt()
- ctime()
- dbm_clearerr()
- dbm_close()
- dbm_delete()
- dbm_error()
- dbm_fetch()
- dbm_firstkey()
- dbm_nextkey()
- dbm_open()
- dbm_store()
- dirname()
- dlerror()
- drand48()
- encrypt()
- endgrent()
- endpwent()
- endutxent()
- ftw()
- getc_unlocked()
- getchar_unlocked()
- getdate()
- getenv()
- getgrent()
- getgrgid()
- getgrnam()
- gethostent()
- getlogin()
- getnetbyaddr()
- getnetbyname()
- getnetent()
- getopt()
- getprotobyname()
- getprotobynumber()
- getprotoent()
- getpwent()
- getpwnam()
- getpwuid()
- getservbyname()
- getservbyport()
- getservent()
- getutxent()
- getutxid()
- getutxline()
- gmtime()
- hcreate()
- hdestroy()
- hsearch()
- inet_ntoa()
- l64a()
- lgamma()
- lgammaf()
- lgammal()
- localeconv()
- localtime()
- lrand48()
- mblen()
- mbtowc()
- mrand48()
- nftw()
- nl_langinfo()
- ptsname()
- putc_unlocked()
- putchar_unlocked()
- putenv()
- pututxline()
- rand()
- readdir()
- setenv()
- setgrent()
- setkey()
- setpwent()
- setutxent()
- strerror()
- strsignal()
- strtok()
- system()
- ttyname()
- unsetenv()
- wctomb()

Potentially Thread Unsafe

If a glibc function is not thread-safe then the man page will say so, and there will (most likely) be a thread safe variant also documented.

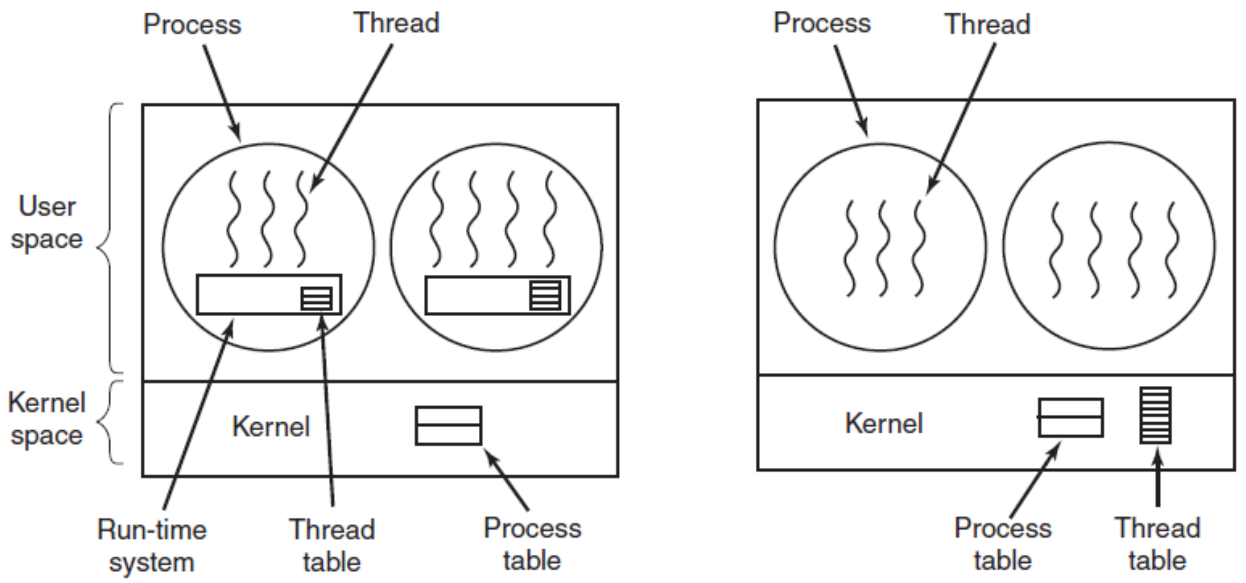
See, for example, man strtok:



SYNOPSIS #include

```
char *strtok(char *str, const char *delim);

char *strtok_r(char *str, const char *delim, char **saveptr);
```

Implementing Threads



- Two main places to implement threads in an OS:
  - user space
  - kernel space

User Space Threads

- Advantage:
  - OS doesn't need to be thread aware and can be implemented on any OS
- Each process needs its own private thread table to track threads
- Runtime tracks when threads are ready to run or block.

User Space Threads

- Since context switching doesn't involving trapping to the kernel it is at least an order of magnitude faster.
- Each process can have a custom thread scheduling algorithm
- Scale better since kernel threads require table space and stack space in the kernel. Can be a problem with a large number of threads.

User Space Threads

- Some major disadvantages.
  - Blocking calls made by a thread will cause all threads to block.
    - Could wrap all system calls and preface them with `select()`
  - A page fault (discussed in chapter 3) caused by one thread will block all threads.

User Space Threads

- » Some major disadvantages continued
  - » No preemptive scheduling of threads. Each thread must yield on its own.
  - » Threads are most useful in application that block often, for example multithreaded web server.

Kernel Space

- Kernel maintains a thread table.
  - Creating / destroying a thread requires a kernel call.
  - Kernel table holds threads registers, state and other information
- All calls that might block are implemented as system call, at a greater cost.

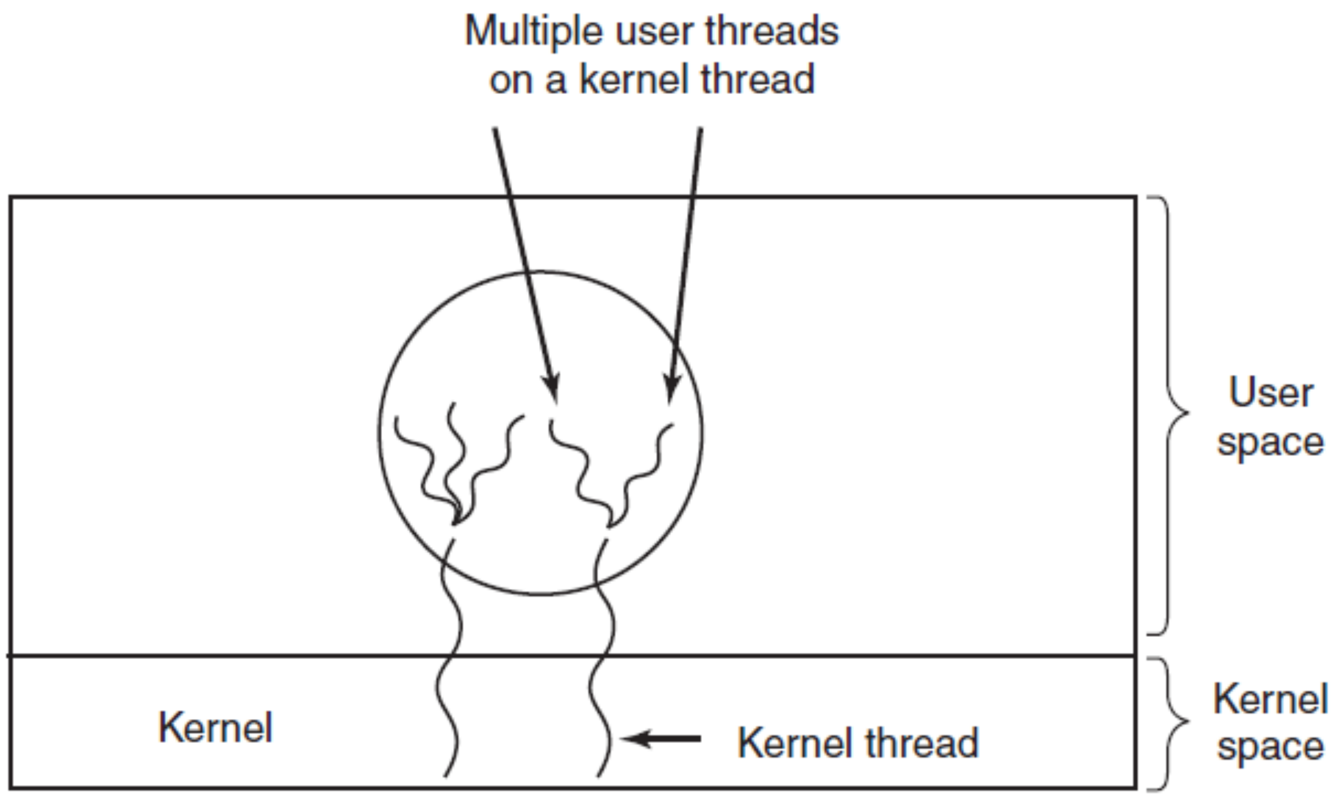
Kernel Space

- When a thread blocks, the OS can choose to run:
  - A ready thread from the current process
  - A ready thread from another process
- Due to large cost of creating a kernel thread, OS can recycle threads.

Kernel Space

- Some problems:
    - What happens when a process forks?
        - Kernel copies each parent threads?
        - Kernel implements one thread?
    - Signals are sent to processes. Which thread handles it?

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads.

Pop-Up Threads

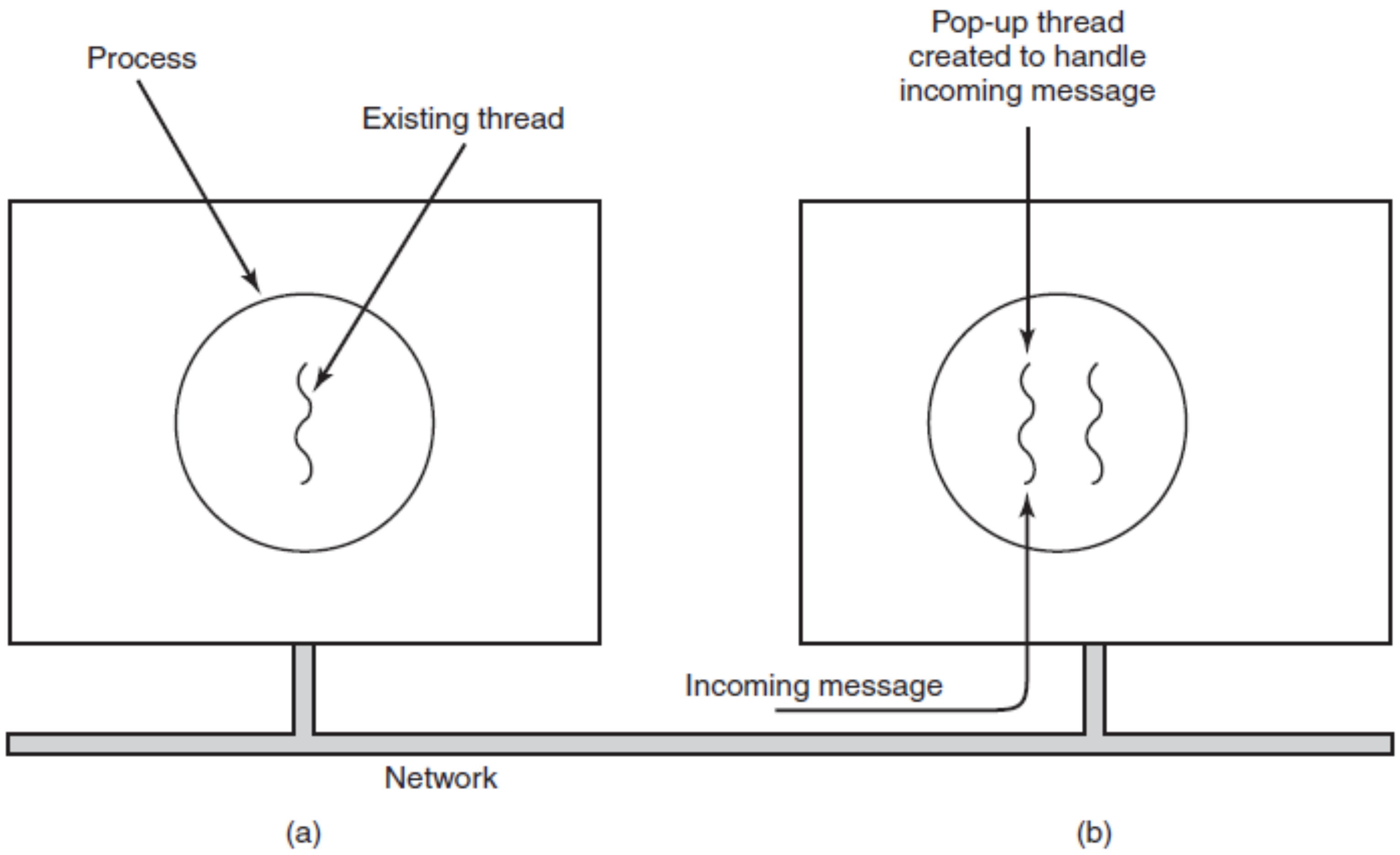


Figure 2-18. Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

Making Single-Threaded Code Multithreaded (1)

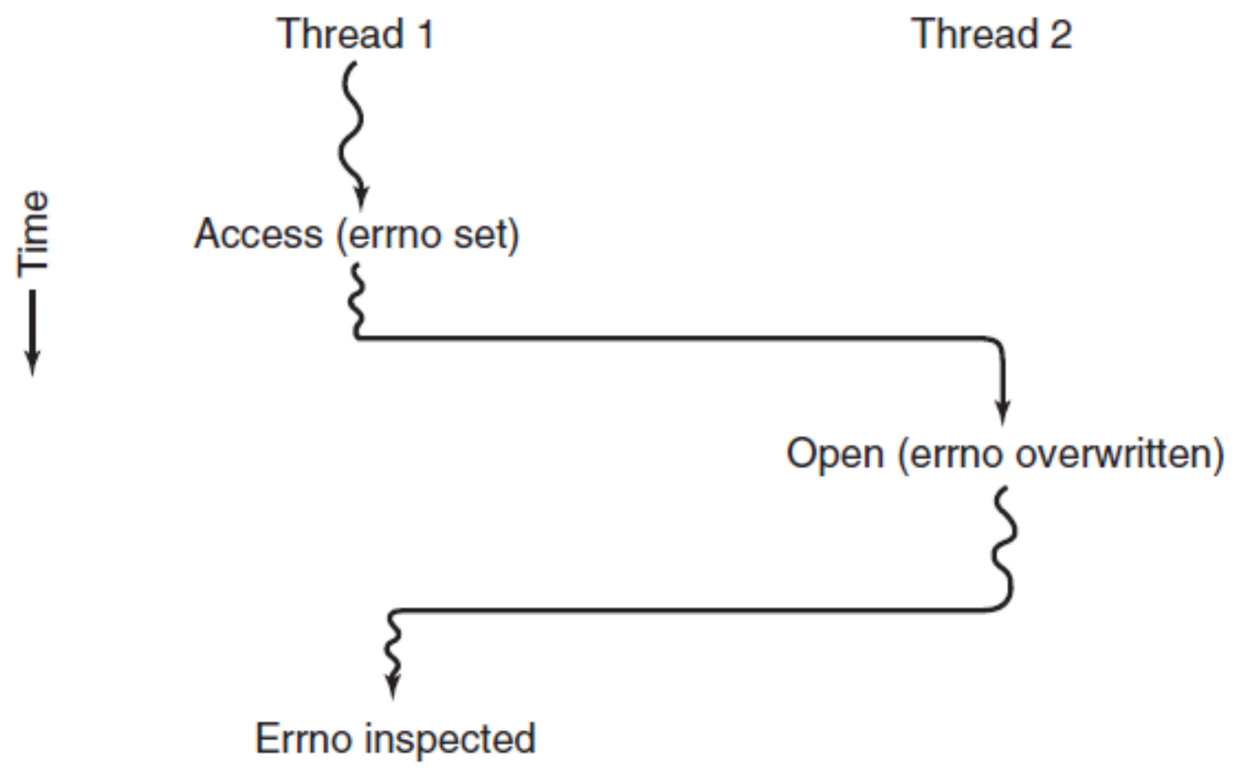


Figure 2-19. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded (2)

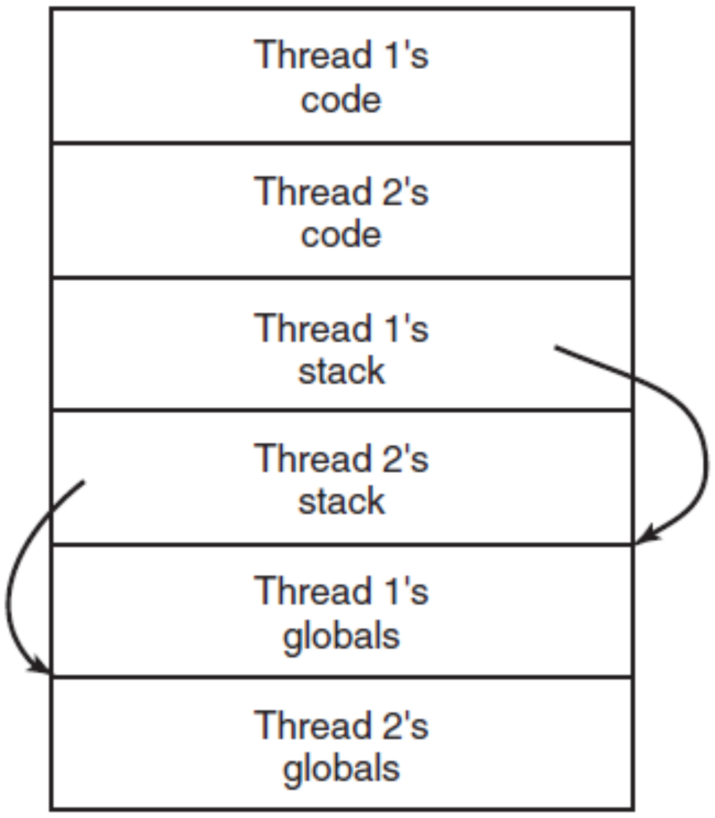


Figure 2-20. Threads can have private global variables.

IPC Issues

1. How can one process pass information to another

   Easy for Threads

2. Keep two processes form getting in each other's way
3. Proper sequencing when dependencies are present.

Race Conditions

- » Processes working together may share common storage.
- » A race condition is where two or more processes are reading or writing shared data and the final result depends on who runs precisely when.
- » Very common as multiprogramming with more and more cores

Race Condition Example

```c
#include <assert.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

static void * simple_thread(void *)  ;

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER  ;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER  ;
int x = 5 ;
int y = 0 ;


int main()
{
  pthread_t tid = 0  ;
  int count = 0  ;

  srand (time(NULL)) ;

  int retval = pthread_create(&tid, 0, &simple_thread, 0)  ;

  while ( 1 )
  {
    x = 5 ;
    if ( x == 5 )
    {
      y = x * 2 ;
      printf("y = %d\n", y ) ;
      fflush( NULL ) ;

      assert( y == 10 ) ;
    }
    usleep( rand() % 100 ) ;
  }

  return 0  ;
}


static void * simple_thread(void * unused)
{
  while ( 1 )
  {
    x = 100 ;
    usleep( rand() % 100 ) ;
  }

  return NULL  ;
}
```

```c
#include <assert.h>
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

static void * simple_thread(void *)  ;

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER  ;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER  ;
int x = 5 ;
int y = 0 ;


int main()
{
  pthread_t tid = 0  ;
  int count = 0  ;

  srand (time(NULL)) ;

  int retval = pthread_create(&tid, 0, &simple_thread, 0)  ;

  while ( 1 )
  {
    x = 5 ;
    if ( x == 5 )
    {
      y = x * 2 ;
      printf("y = %d\n", y ) ;
      fflush( NULL ) ;

      assert( y == 10 ) ;
    }
    usleep( rand() % 100 ) ;
  }

  return 0  ;
}


static void * simple_thread(void * unused)
{
  while ( 1 )
  {
    x = 100 ;
    usleep( rand() % 100 ) ;
  }

  return NULL  ;
}
```

Shared data

Both threads modify/compare x with no protection against concurrent access

Critical Regions

- Key is to prevent more than one process form reading and writing from a shared area at the same time
  - Mutual exclusion
  - pthread mutexes

Mutex Variables

```
pthread_mutex_lock (mutex)
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)
```

- The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.
- `pthread_mutex_trylock()` will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.
- `pthread_mutex_unlock()` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error will be returned if:
  - If the mutex was already unlocked
  - If the mutex is owned by another thread

Critical Regions

- » Critical region is the part of a program where shared memory is accesses.
- » Four conditions to avoid race conditions
  1. No two processes may be simultaneously inside the critical region
  2. No assumptions may be made about speeds or number of CPUs
  3. No process running outside its critical region may block any process
  4. No process should have to wait forever to enter its critical

Critical Regions (2)

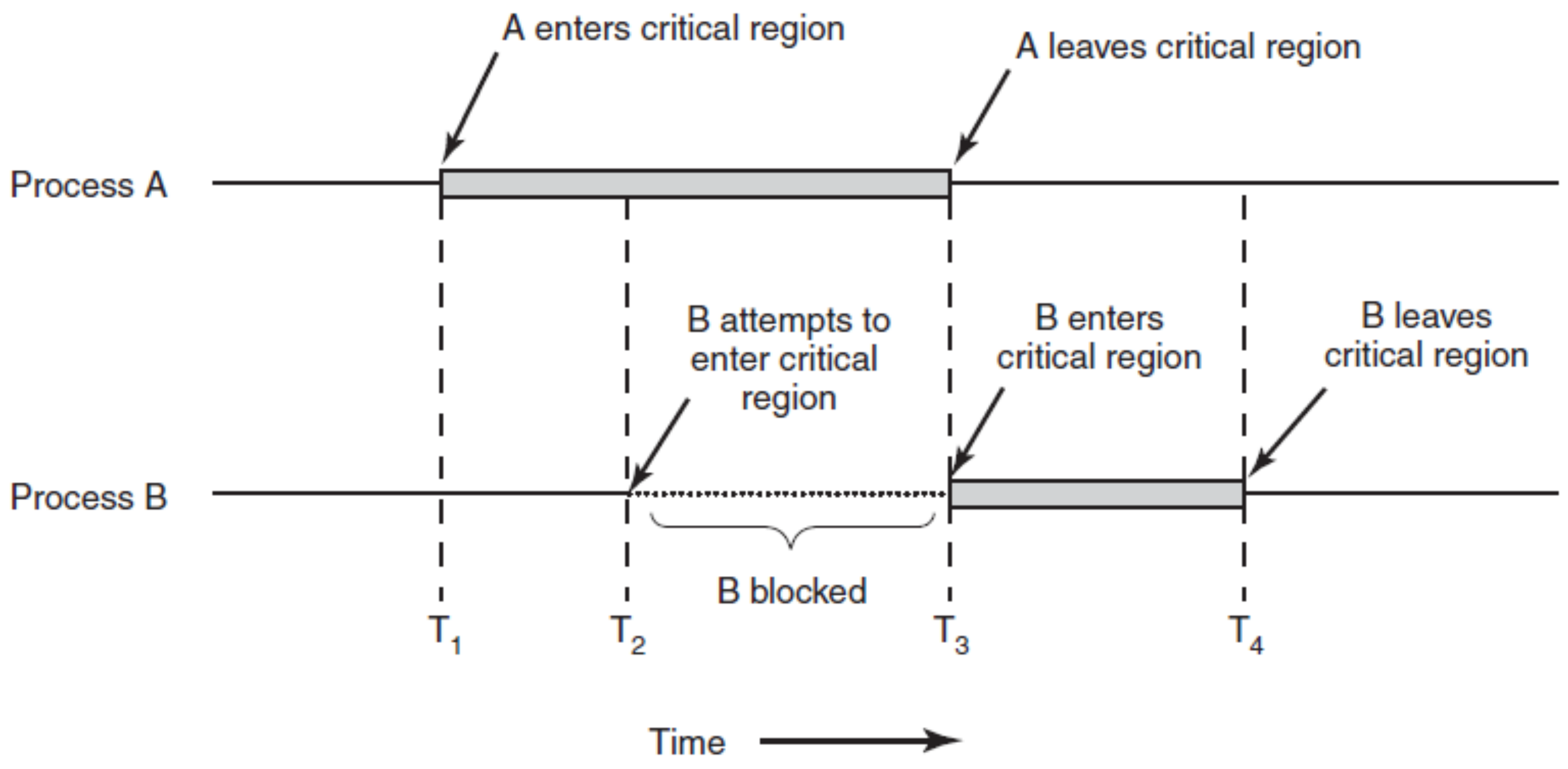


Figure 2-22. Mutual exclusion using critical regions.

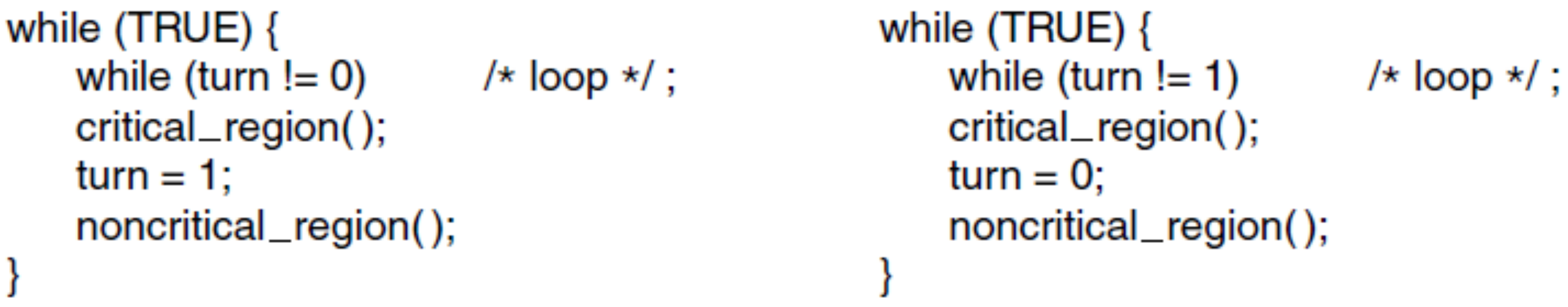
Mutual exclusion: Busy waiting

- » Disabling Interrupts
  - » Only works on single processor system
  - » Disable interrupts before entering critical region.
  - » Re-enable before leaving the critical region.
  - » Generally a bad idea
    - » Kernel does it, though

Mutual exclusion: Busy waiting

- Lock variables
  - Shared lock variable, initially 0
  - When a process wants to enter the critical region, test then set
  - Bad. Value can change after check but before set.
    - See my real world example

Mutual exclusion: Busy waiting



```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

» Strict alternation

  » Ok for C. Bad for garbage collected languages.

  » Violates criteria #3

Mutual exclusion: Busy waiting

- Busy waiting wastes CPU time
  - Only should be used when there is a reasonable expectation that that wait will be short.
  - spin-lock is a lock that uses busy waiting

Peterson's Solution

```
bool flag[2] = {false, false};
int turn;
```

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] && turn == 1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] && turn == 0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

» 1981, G. L. Peterson came up with a mutual exclusion algorithm

TSL Instructions

- » Test and Set Lock
  - » Atomic instruction
  - » An instruction used to write 1 (set) to a memory location and return its old value as a single atomic (i.e., non-interruptible) operation.
  - » If multiple processes may access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished.

Fix Attempt #2

Sleep and Wakeup

- » Peterson's and TSL are correct but. have the defect of busy waiting.
  - » Wastes CPU time
  - » Also can have cause priority inversion and starvation

Priority Inversion and Starvation

- Indefinite blocking or starvation
  - process is not deadlocked
  - but is never removed from the semaphore queue

- Priority inversion
  - lower-priority process holds a lock needed by higher-priority process !

- Assume three processes L, M, and H
  - Priorities in the order L < M < H
  - L holds shared resource R, needed by H
  - M preempts L, H needs to wait for both L and M !!

Priority Inversion and Starvation

- Solutions
  - Only support at most two priorities
  - Priority inheritance protocol – lower priority process accessing shared resource inherits higher priority

Mars Pathfinder

- July 4th 1997 - Mars Pathfinder lands on Mars

Mars Pathfinder

- VxWorks OS on the spacecraft provided priority with preemption scheduler
- Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft.
- A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

Mars Pathfinder

- The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex.
- The spacecraft also contained a communications task that ran with medium priority.

Mars Pathfinder

- Very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread.
- After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

Mars Pathfinder

- JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch".

Mutex

- Simplified version of a semaphore
- Shared variable that can be in one of two states: locked or unlocked.
- Can't be in a disjoint address space.

Mutex Variables

```
pthread_mutex_lock (mutex)
pthread_mutex_trylock (mutex)
pthread_mutex_unlock (mutex)
```

- The `pthread_mutex_lock()` routine is used by a thread to acquire a lock on the specified mutex variable. If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked.

- `pthread_mutex_trylock()` will attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

- `pthread_mutex_unlock()` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An error will be returned if:

  - If the mutex was already unlocked

  - If the mutex is owned by another thread

Mutex Example

POSIX Semaphores

- There are two actions defined on semaphores (we'll go with the classic terminology): P(Sem s) and V(Sem s).
- P and V are the first letters of two Dutch words proberen (to test) and verhogen (to increment).
  - The inventor of semaphores was Edsger Dijkstra who was very Dutch.

POSIX Semaphores

- » P(Sem s) decrements s->value, and if this is less than zero, the thread is blocked, and will remain so until another thread unblocks it. This is all done <u>atomically.</u>
- » V(Sem s) increments s->value, and if this is less than or equal to zero, then there is at least one other thread that is blocked because of s.

POSIX Semaphores

» All POSIX semaphore functions and types are prototyped or defined in `semaphore.h`. To define a semaphore object, use

```
sem_t sem_name;
```

Initializing a semaphore

» To initialize a semaphore, use `sem_init`:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `sem` points to a semaphore object to initialize
- `pshared` is a flag indicating whether or not the semaphore should be shared with `fork()`ed processes.
- `value` is an initial value to set the semaphore to

Example of use:

```
        sem_init(&sem_name, 0, 10);
```

Waiting on a Semaphore

- To wait on a semaphore, use `sem_wait`:

```
int sem_wait(sem_t *sem);
```

- Example of use:

```
sem_wait(&sem_name);
```

- sem_wait is an implementation of the DOWN operation. If the value of the semaphore is 0, the calling process blocks; one of the blocked processes wakes up when another process calls `sem_post`.

Incrementing a Semaphore

» To increment the value of a semaphore, use `sem_post`:

```
    int sem_post(sem_t *sem);
```

Example of use:

```
  sem_post(&sem_name);
```

» `sem_post` is an implementation of the UP operation. It increments the value of the semaphore and wakes up a blocked process waiting on the semaphore, if any.

Query a Semaphore Value

- » To find out the value of a semaphore, use

```
int sem_getvalue(sem_t *sem, int *valp);
```

- » gets the current value of sem and places it in the location pointed to by valp

Example of use:

```
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphors is %d\n", value);
```

Destroy a Semaphore

» To destroy a semaphore, use

```
    int sem_destroy(sem_t *sem);
```

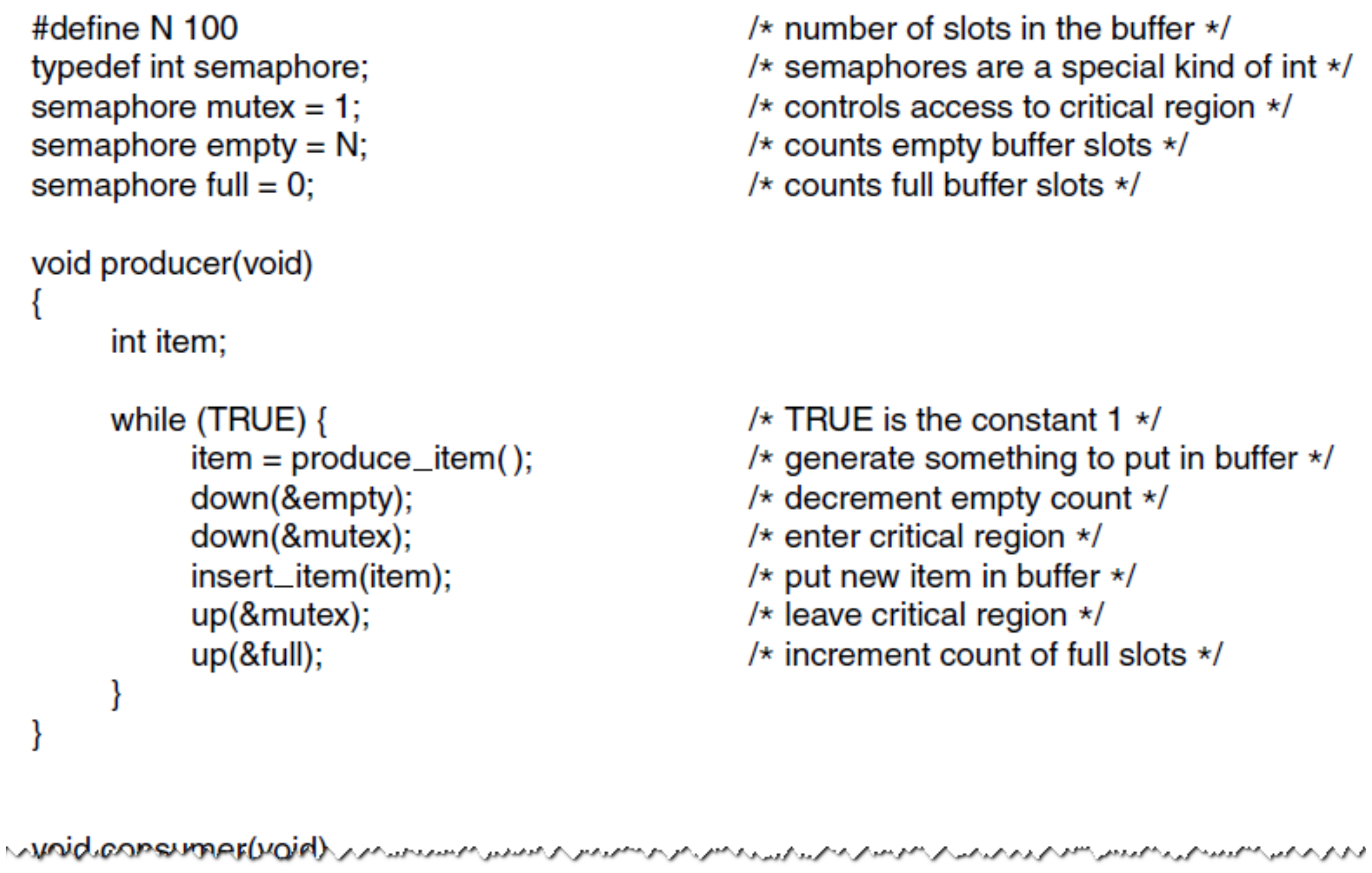
» destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:

```
    sem_destroy(&sem_name);
```

Semaphore Example

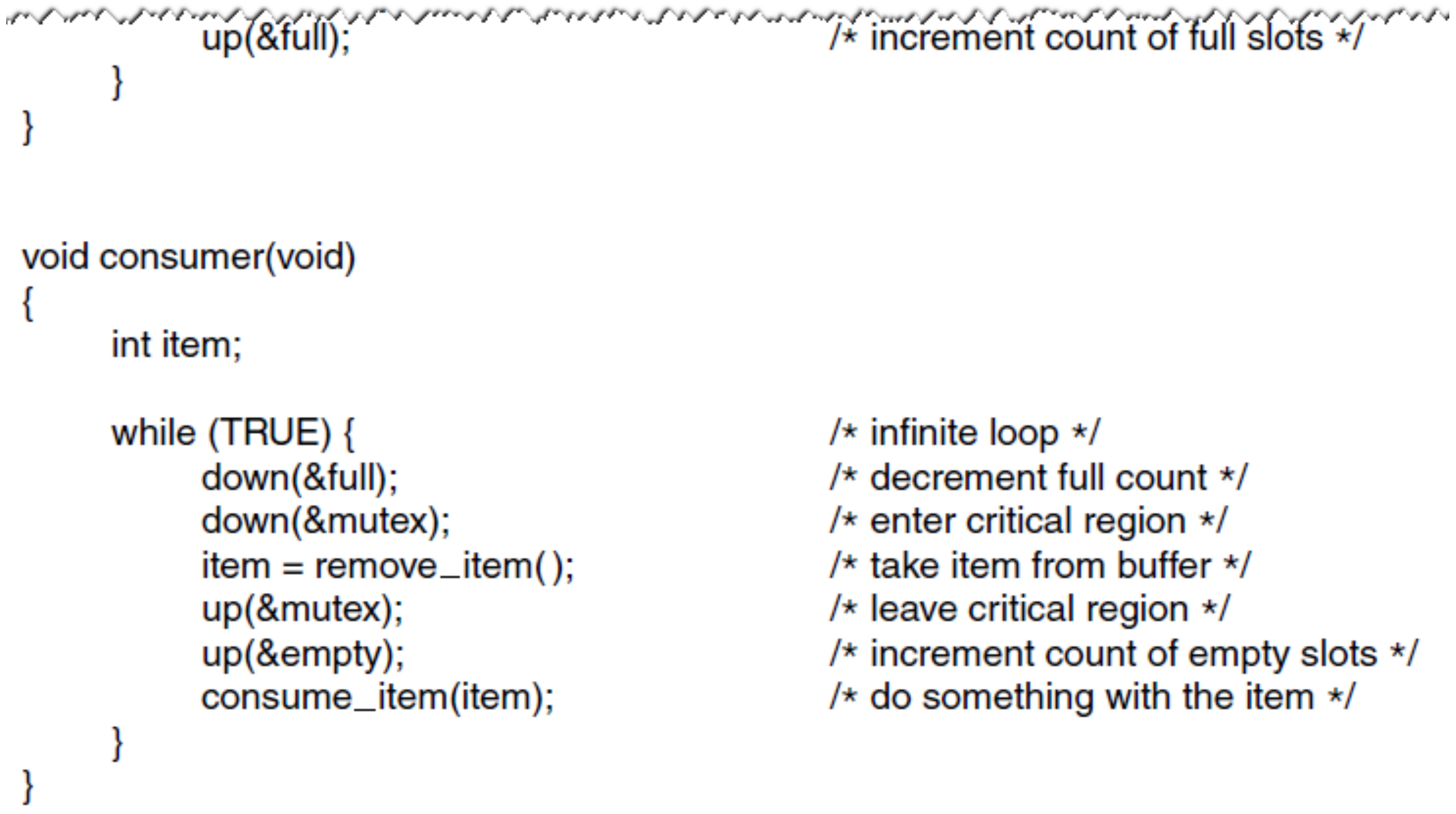
Semaphores (1)



```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}

void consumer(void)
```

Figure 2-28. The producer-consumer problem using semaphores.

Semaphores (2)



```
                up(&full);                              /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                                  /* infinite loop */
                down(&full);                            /* decrement full count */
                down(&mutex);                           /* enter critical region */
                item = remove_item( );                  /* take item from buffer */
                up(&mutex);                             /* leave critical region */
                up(&empty);                             /* increment count of empty slots */
                consume_item(item);                     /* do something with the item */
        }
}
```

Figure 2-28. The producer-consumer problem using semaphores.