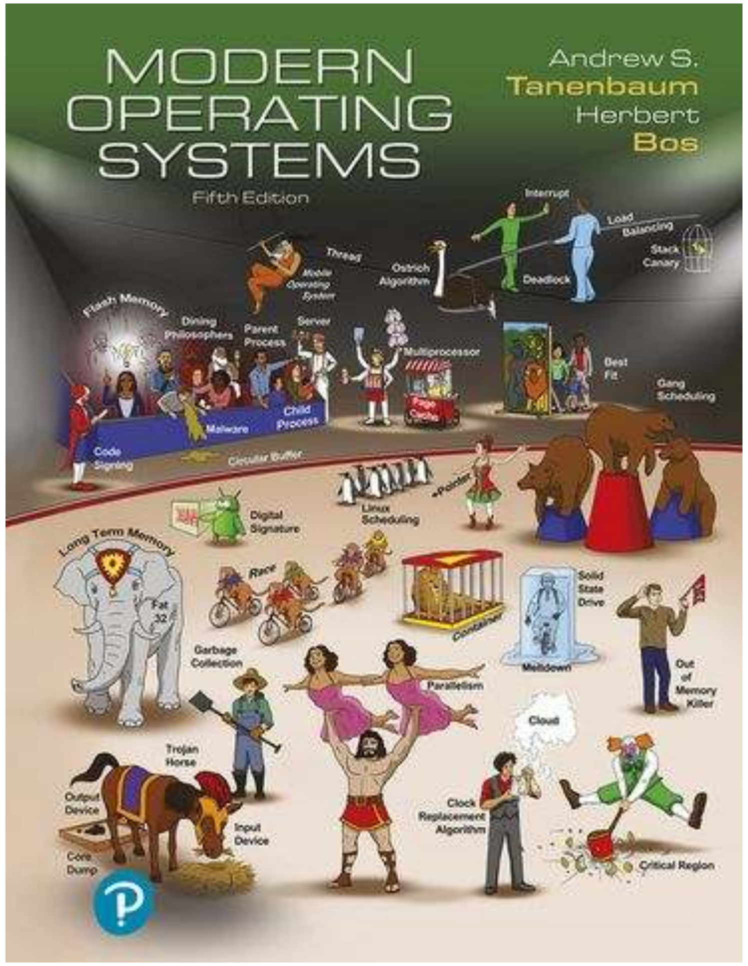
Modern Operating Systems

Fifth Edition



Chapter 2

Processes and Threads

Processes and threads

- Introduction to Processes
  - The Process Model
  - Process Management
  - Process States
  - Threads
  - Signal Handling
- Inter-Process Communication
  - IPC Mechanisms
  - Classical IPC Problems
- Scheduling

The Process Model

Process = Program in execution

- How many processes for each program?
- A fundamental operating system **abstraction**
- Allows the OS to simplify:
  - Resource **allocation**
  - Resource **accounting**
  - Resource **limiting**
- OS maintains information on the resources and the internal state of every single process in the system

The Process Model (1 of 3)

- Single program counter.
- Each process in unique location.
- CPU switches back and forth from process to process

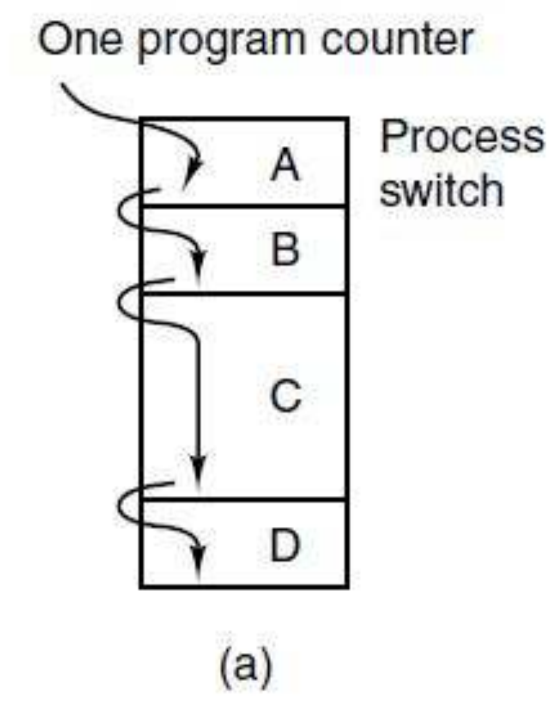


Figure 2-1. (a) Multiprogramming of four programs.

The Process Model (2 of 3)

- Each process has own flow of control (own logical program counter)
- Each time we switch processes, we save the program counter of first process and restore the program counter of the second

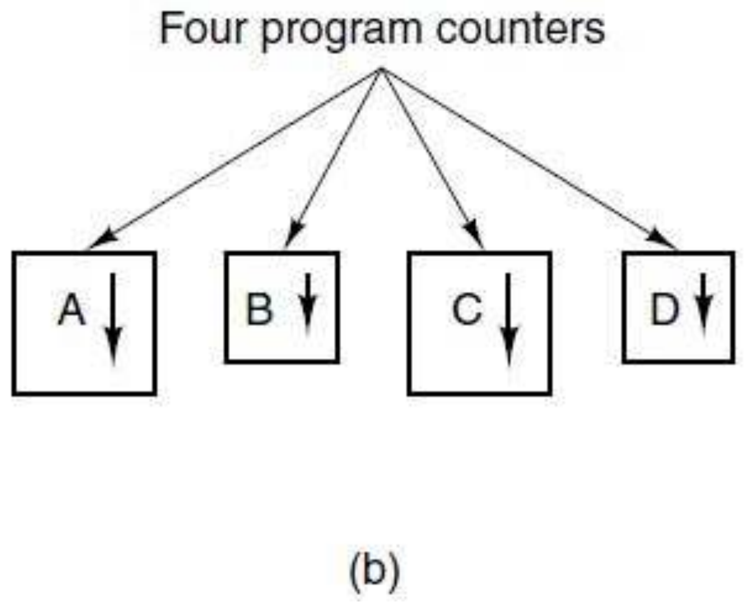


Figure 2-1. (b) Conceptual model of four independent, sequential processes.

The Process Model (3 of 3)

- All processes make progress, but only one is active at any given time

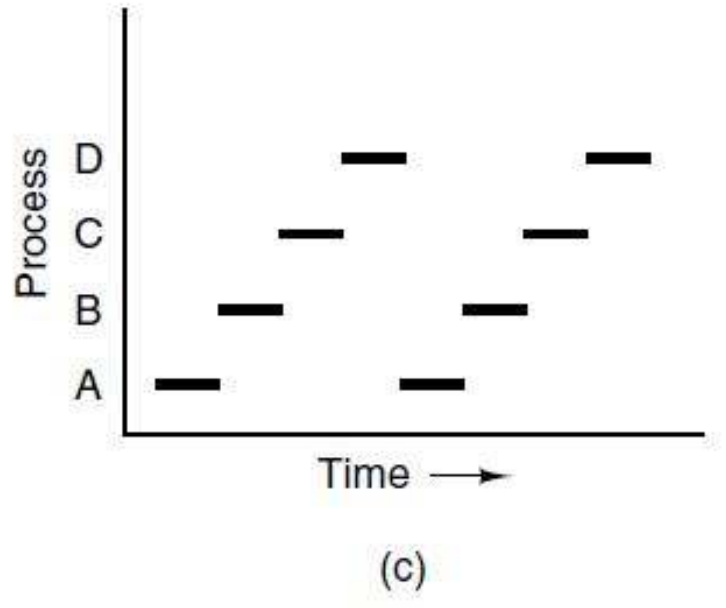
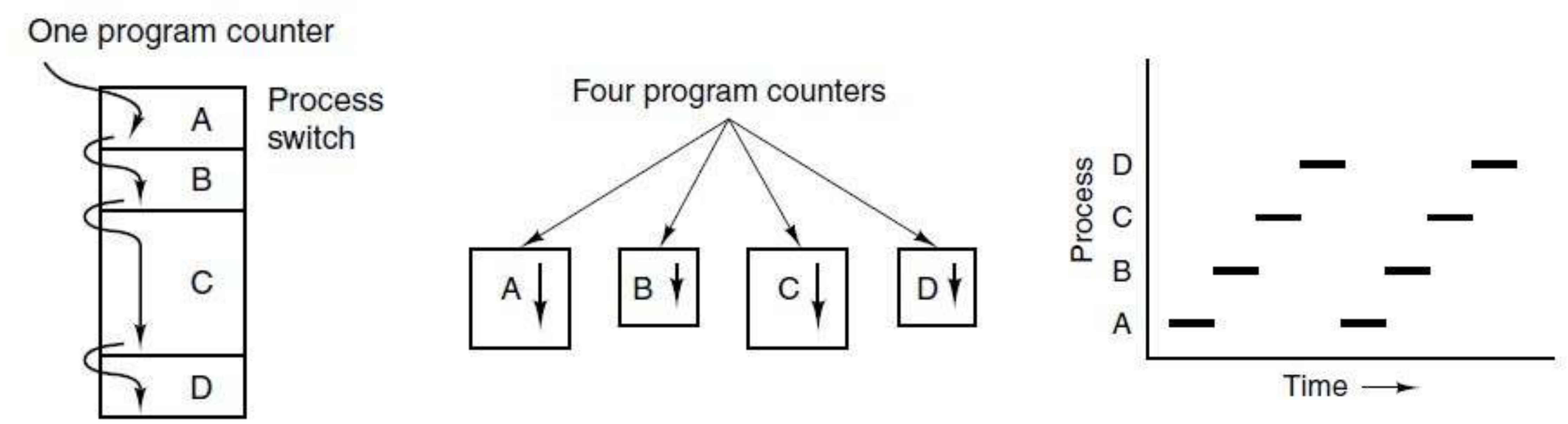


Figure 2-1. (c) Only one program is active at once.

Concurrent processes



- In principle, multiple processes are mutually independent
- They need explicit means to interact with each other
- The CPU can be allocated in turns to different processes
- OS normally offers **no timing or ordering** guarantees

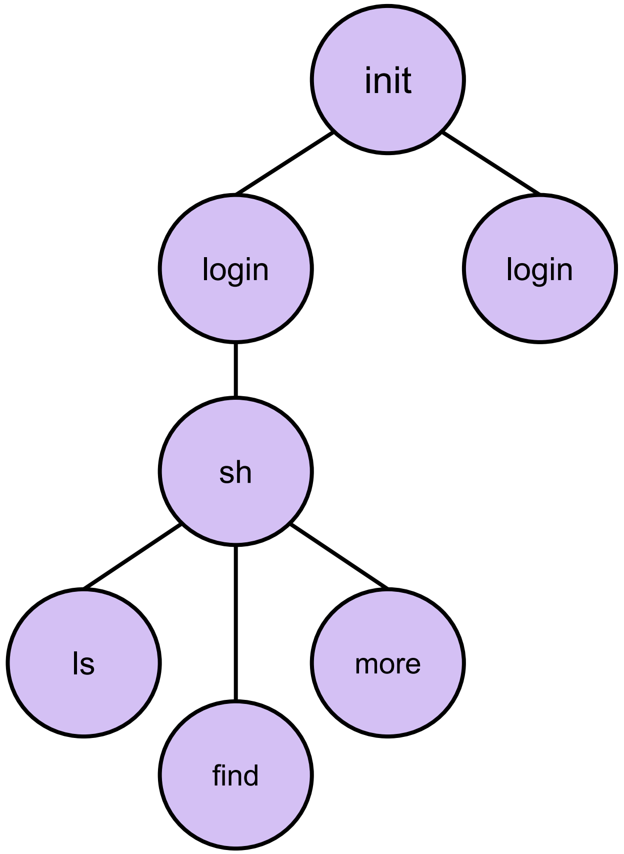
Process hierarchies

OS typically creates only 1 `init` process

Subprocesses created independently:

- A **parent** process can create a **child** process
- This results in a tree-like structure and **process groups**
- E.g., shell executes commands:

```
$ find /tmp &> t.log &
$ ls | more
```



Process Creation

Four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job.

Process Termination

Typical conditions which terminate a process:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Process management

- `fork`: create a new process
  - Child is a “private” **clone** of the parent
  - Shares **some** resources with the parent
- `exec`: execute a new process image
  - Used in combination with `fork`
  - `exec` on Windows?
- `exit`: cause voluntary process termination
  - Exit status returned to the parent
  - Involuntary process termination?
- `kill`: send a signal to a process (or group)
  - Can cause involuntary process termination

Process States (1 of 3)

Three states a process may be in:

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).

Process States (2 of 3)

- The OS allocates resources (e.g., CPU) to processes
- To allocate the CPU, the OS needs to track **process states**:
    - **Running**: the process is currently executed by the CPU
    - **Blocked**: the process is waiting for available resources
    - **Ready**: the process is ready to be selected
- The scheduler (de)allocates the CPU (see transitions 2&3)

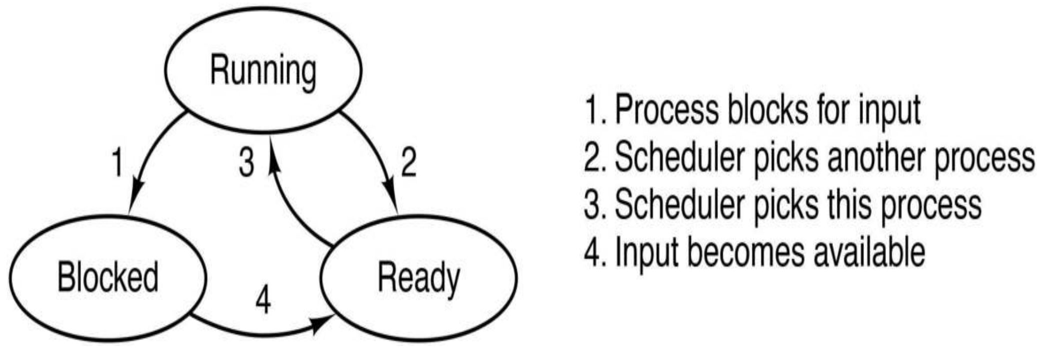

Running
Blocked
Ready
1
2
3
4
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Process States (3 of 3)

- Scheduler periodically **switches** processes
- **Sequential processes** lay on the layer above
- This leads to a simple process organization
- What are the missing fundamental OS abstractions?

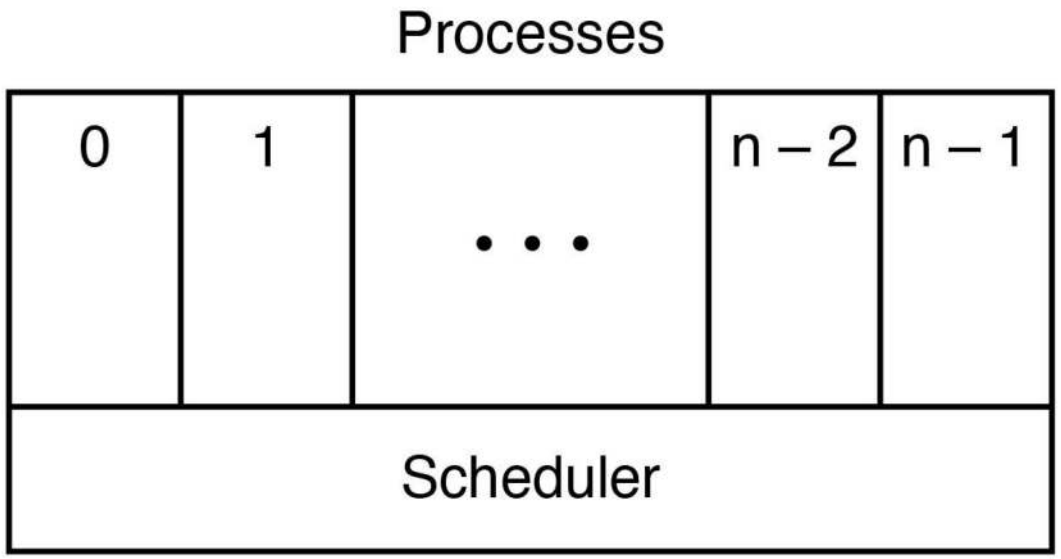


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

Information associated with a process

- ID (PID), User (UID), Group (GID)
- Memory address space
- Hardware registers (e.g., program counter)
- Open files
- Signals
- ...

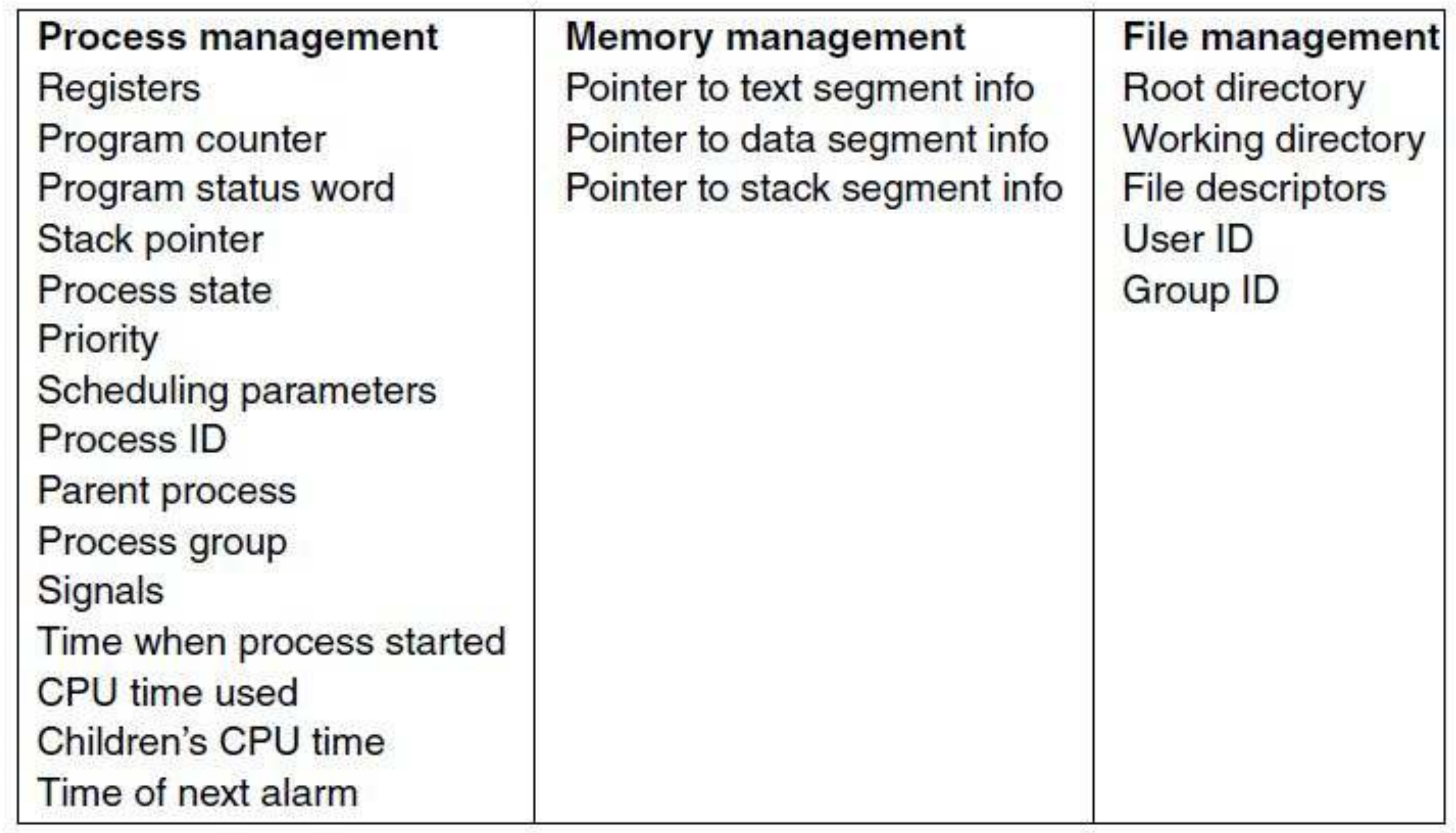
This information is stored in the operating system's **Process Table**

Information associated with a process

- ID (PID), User (UID), Group (GID)
- Memory address space
- Hardware registers (e.g., program counter)
- Open files
- Signals
- ...

This information is stored in the operating system’s **Process Table**

Process Control Blocks



Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process-table entry

MINIX vs Linux

```
struct proc {
  struct stackframe_s p_reg;
  struct segframe p_seg;
  proc_nr_t p_nr;
  struct priv *p_priv;
  volatile u32_t p_rts_flags;
  volatile u32_t p_misc_flags;

  char p_priority;
  u64_t p_cpu_time_left;
  unsigned p_quantum_size_ms;

  struct proc *p_scheduler;
  unsigned p_cpu;

  /* ... */
};

EXTERN struct proc proc[NR_TASKS + NR_PROCS];
```

```
struct task_struct {
  volatile long state;
  void *stack;
  atomic_t usage;
  unsigned int flags;
  unsigned int ptrace;

  int prio, static_prio, normal_prio;
  unsigned int rt_priority;
  const struct sched_class *sched_class;
  struct sched_entity se;
  struct sched_rt_entity rt;

  struct list_head tasks;

  /* ... */
};

struct task_struct init_task = INIT_TASK(init_task);
```

Interrupts

- **Idea**: to deallocate the CPU in favor of the scheduler, we rely on hardware-provided interrupt handling support
- Allows the scheduler to periodically get control, i.e., whenever the hardware generates an **interrupt**
- **Interrupt vector:**
    - Associated with each I/O device and interrupt line
    - Part of the **interrupt descriptor table** (IDT)
    - Contains the start address of an OS-provided internal procedure (**interrupt handler**)
- The **interrupt handler** continues the execution
- **Interrupt types**: sw, hw device (async), exceptions

Implementation of Processes (2 of 2)

Skeleton of what the lowest level of the operating system does when an interrupt occurs.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

- Every time an interrupt occurs, the scheduler gets control □acts as a mediator
- A process cannot give the CPU to another process (context switch) without going through the scheduler

Modeling Multiprogramming

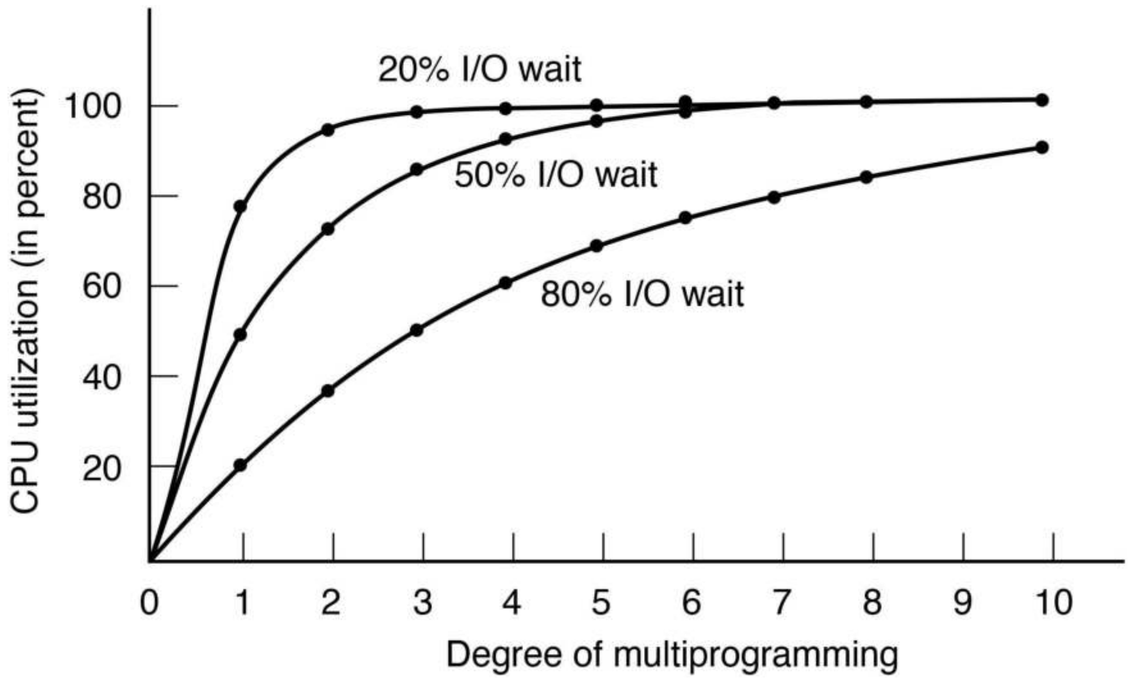


Figure 2-6. CPU utilization as a function of the number of processes in memory.

Signal handling

- Signal types:
  - Hardware-induced (e.g., `SIGILL`)
  - Software-induced (e.g., `SIGQUIT` or `SIGPIPE`)
- Actions:
  - **Term**, **Ign**, **Core**, **Stop**, **Cont**
  - Default action on per-signal basis, typically overridable
  - Signals can be typically blocked and actions delayed
- Catching signals:
  - Process registers signal handler
  - OS delivers signal and allows process to run handler
  - Current execution context needs to be saved/restored

Catching Ctrl-C

```
void signalHandler( int signum ) {
   printf ("Interrupt signal &d received\n”, signum );
   // cleanup and terminate program
   exit(signum);
}
int main () {
   // register signal SIGINT and signal handler
   signal(SIGINT, signalHandler);
   while(1) {
      printf ("Going to sleep....\n");;
      sleep(1);
   }
   return 0;
}
```

Signal handling

Kernel delivers signal

– Stops code currently executing

– Saves context

– Executes signal handling code

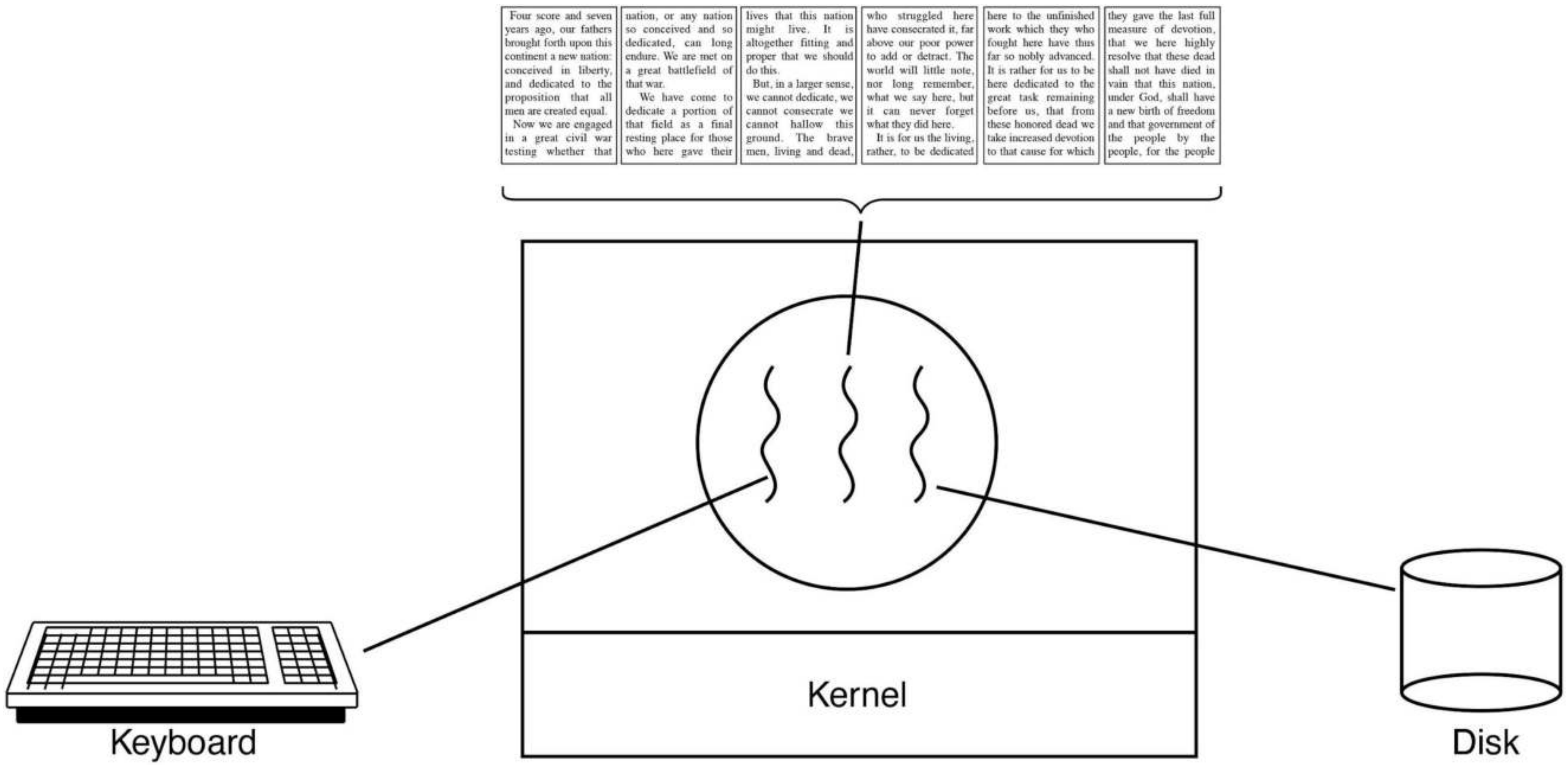
– Restores original context

Threads

Threads

- Implicit assumption so far:
    - 1 process → 1 thread of execution
- Multithreaded execution:
    - 1 process → N threads of execution
- Why allow multiple **threads** per process?
    - Lightweight processes
        - Allow space- and time- efficient parallelism
    - Organized in thread groups
        - Allow simple communication and synchronization

Thread Usage (1 of 3)



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

Figure 2-7. A word processor with three threads.

Thread Usage (2 of 3)

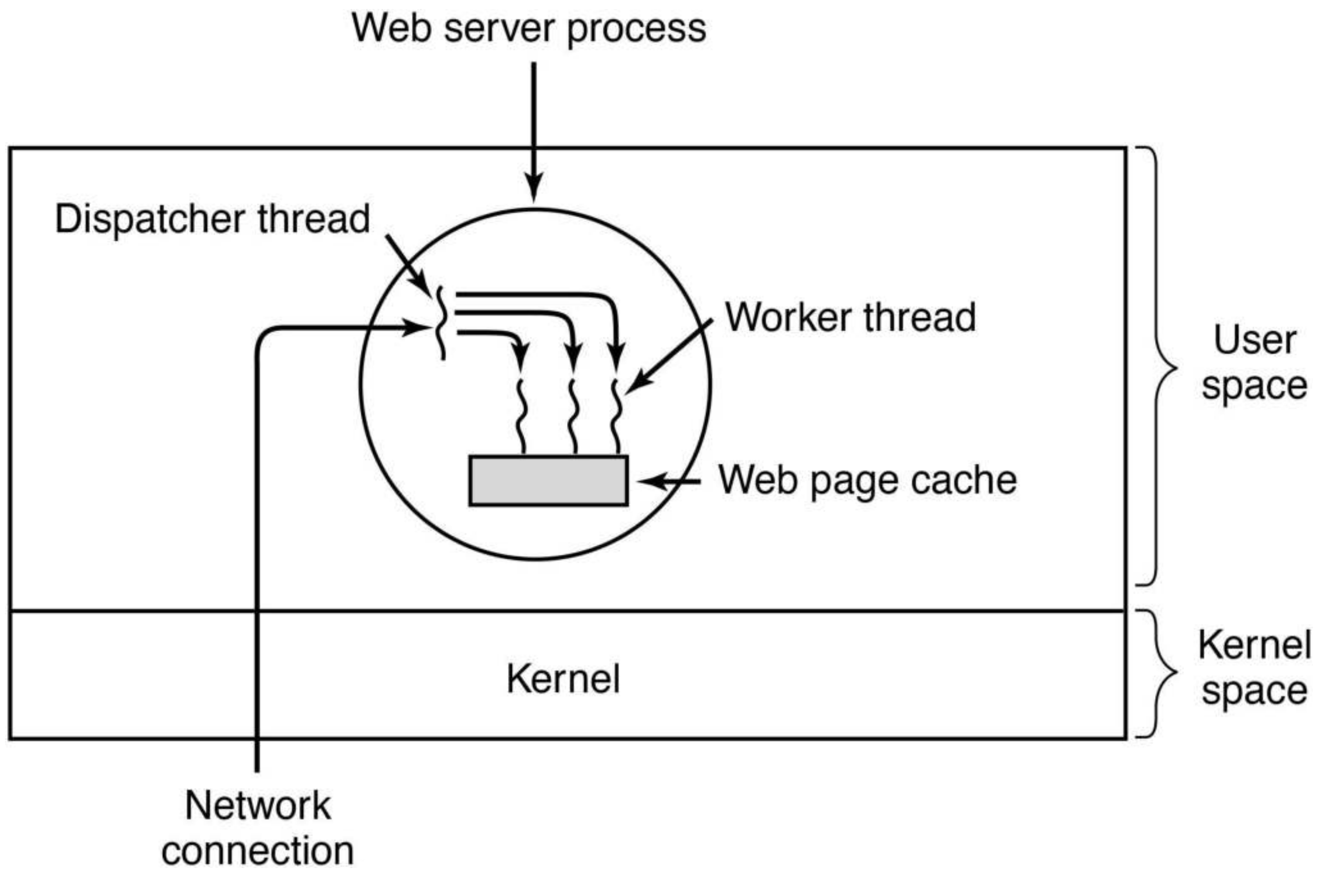


Figure 2-8. A multithreaded Web server.

Thread Usage (3 of 3)

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

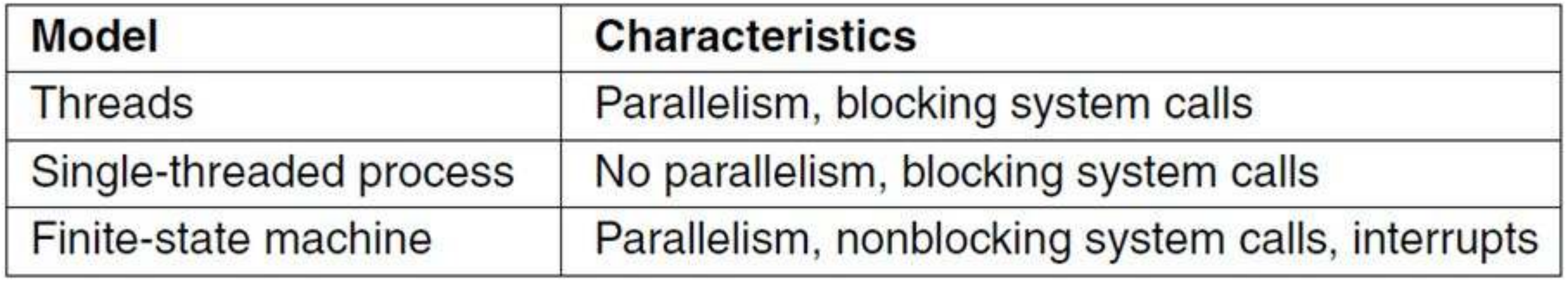
(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Figure 2-9. A rough outline of the code for Fig. 2-8. (a) Dispatcher thread. (b) Worker thread.`

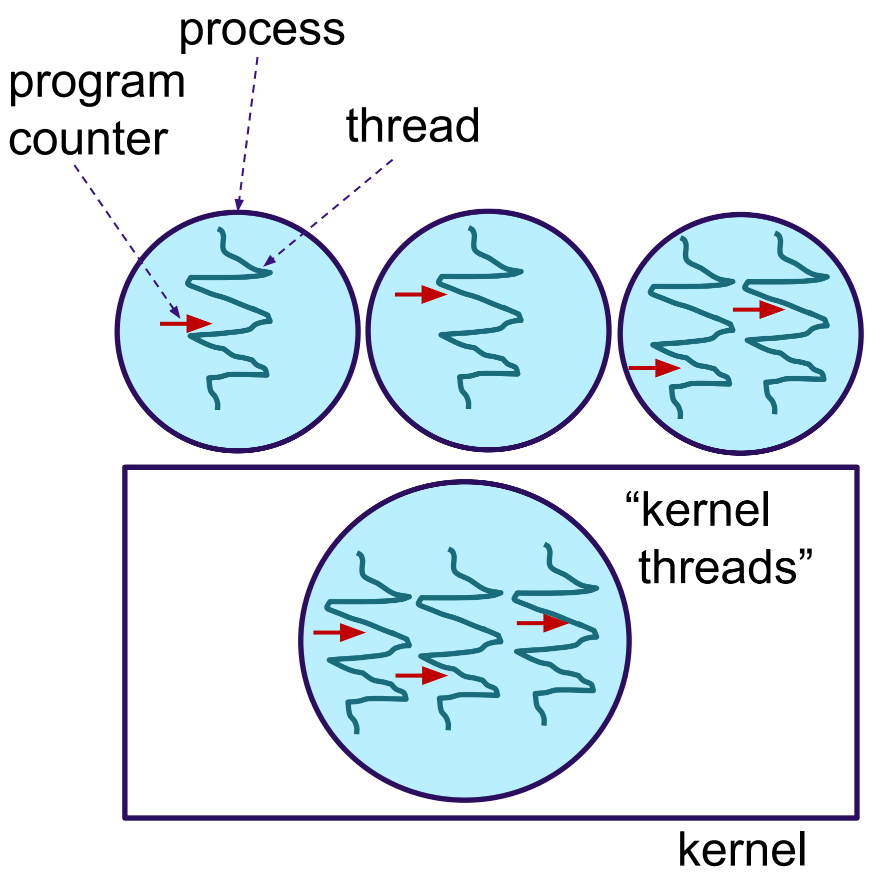
Threads, process, finite state machine



Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Threads and processes

- **Threads** reside in the **same address space** of a single process
- All information exchange is via **data shared between the threads**
- Threads **synchronize** via simple primitives
- Each thread has its own stack, hardware registers, and state
- Thread table/switch: a lighter process table/switch
- Each thread may **call** any OS-supported system call **on behalf of the process to which it belongs**



The Classical Thread Model (1 of 3)

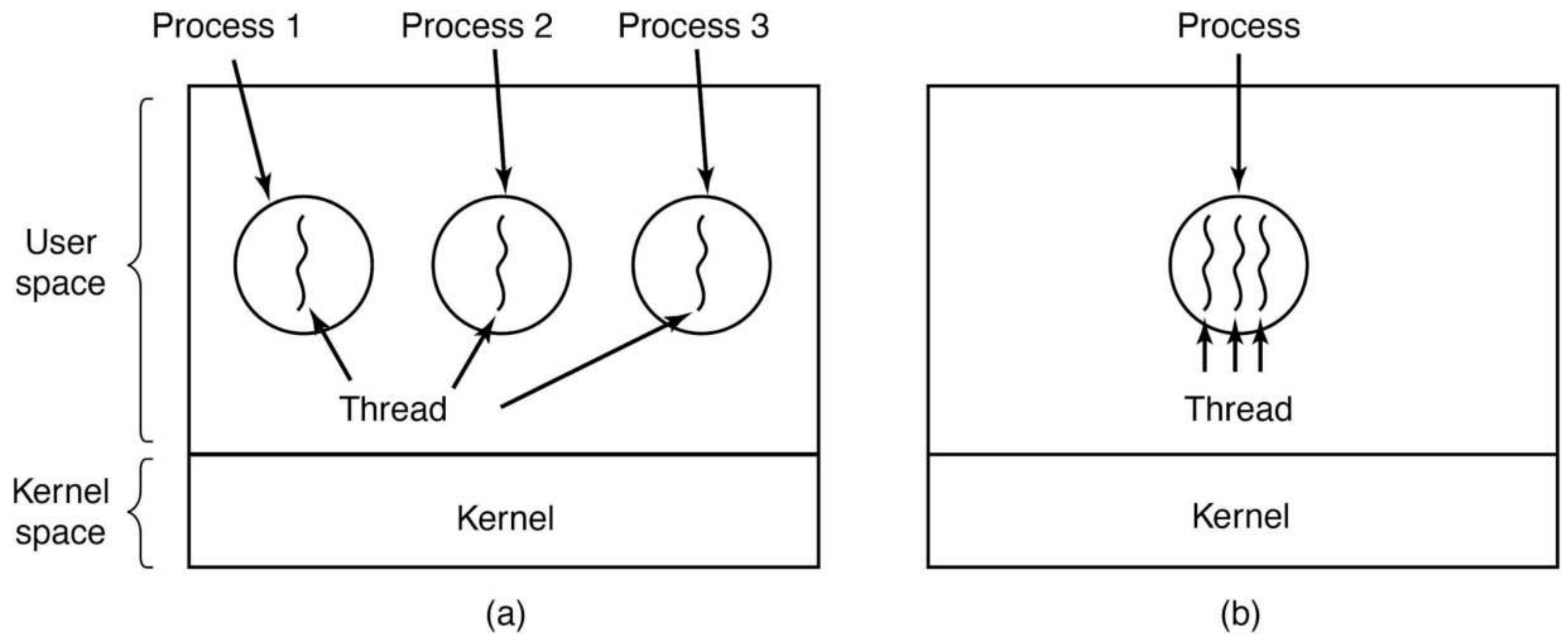


Figure 2-10. (a) Three processes each with one thread. (b) One process with three threads.

The Classical Thread Model (2 of 3)

Figure 2-11. The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

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	

The Classical Thread Model (3 of 3)

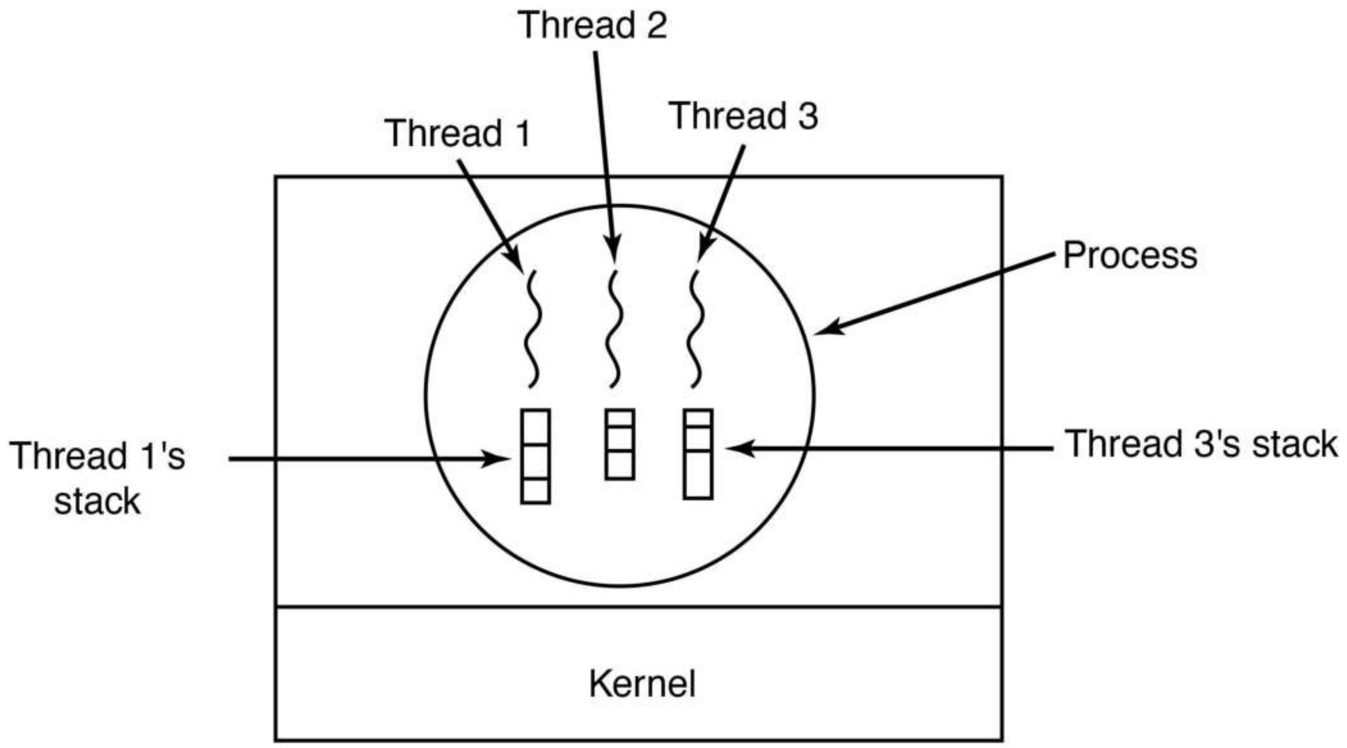


Figure 2-12. Each thread has its own stack.

POSIX Threads (1 of 2)

Figure 2-13. Some of the Pthreads function calls.

Thread call	Description
pthread_create	Create a new thread
pthread_exit	Terminate the calling thread
pthread_join	Wait for a specific thread to exit
pthread_yield	Release the CPU to let another thread run
pthread_attr_init	Create and initialize a thread's attribute structure
pthread_attr_destroy	Remove a thread's attribute structure

Pthreads

What will the output be?

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

#define NUMBER_OF_THREADS  10
void * print_hello_world(void * tid)
{
  printf("Hello World. Greetings from thread %d\n", tid);
  pthread_exit(NULL);
}
int main(int argc, char * argv[])
{
  pthread_t threads[NUMBER_OF_THREADS];
  int status, i;
  for(i=0; i < NUMBER_OF_THREADS; i++) {
    status = pthread_create(&threads[i], NULL, print_hello_world, (void * )i);
    if (status != 0) {
      exit(-1);
    }
  }
  return 0;
}
```

Implementing Threads in User Space

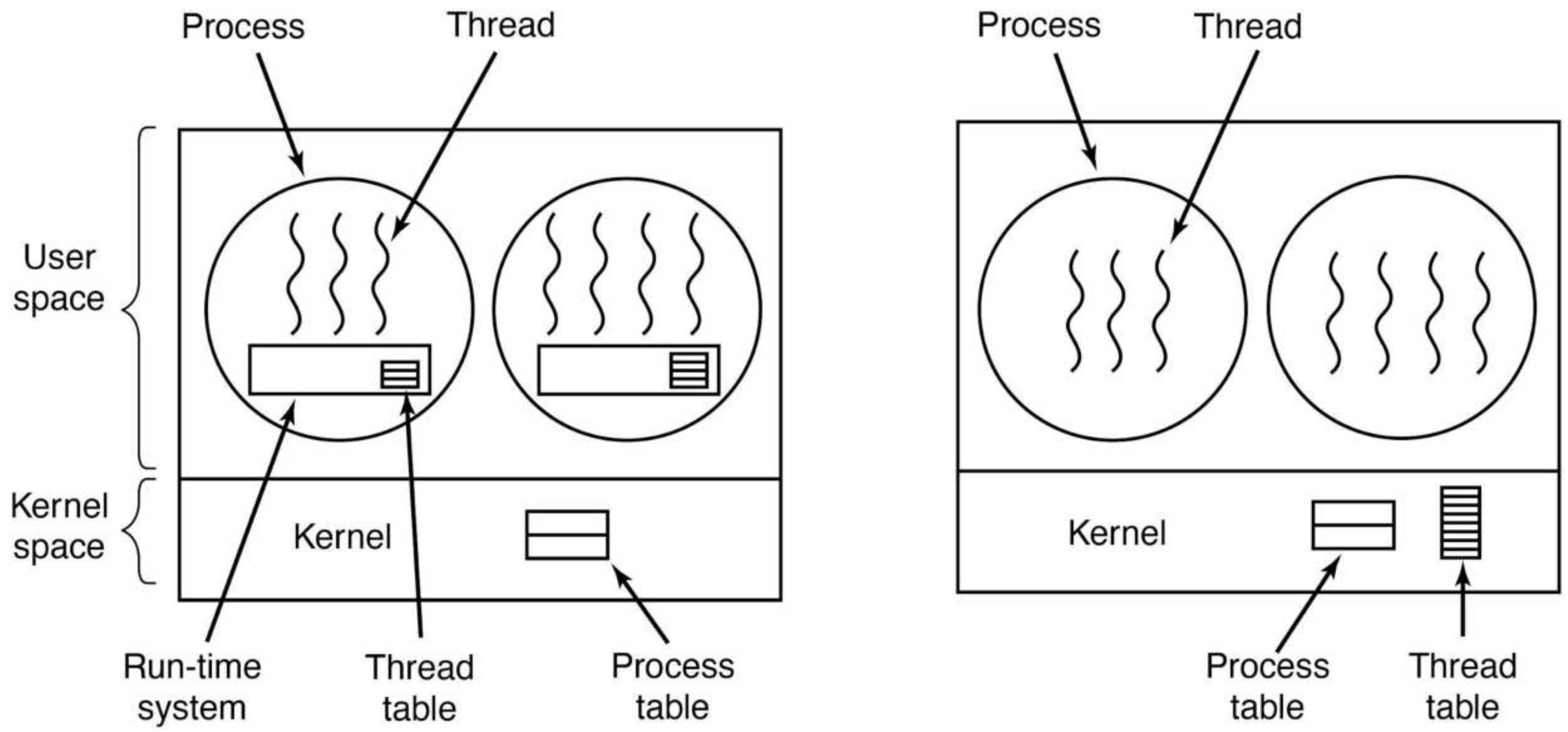
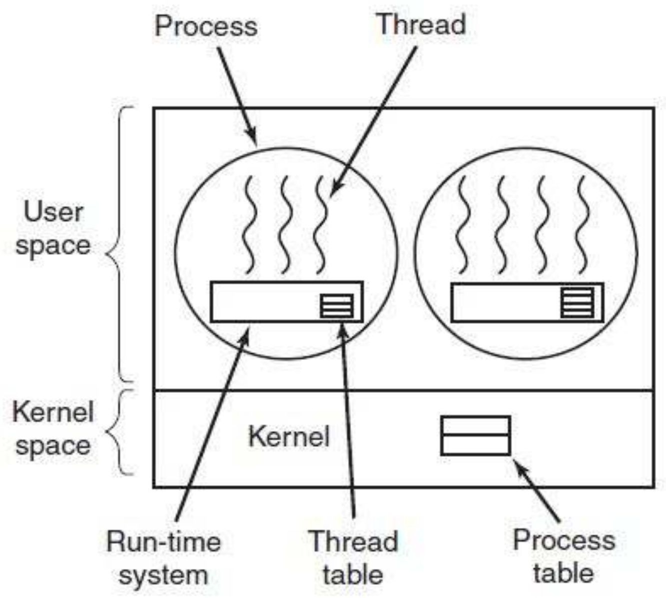
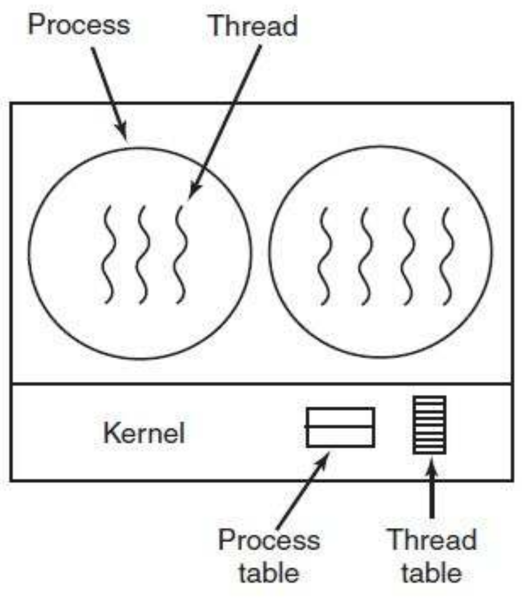


Figure 2-15. (a) A user-level threads package. (b) A threads package managed by the kernel.

User threads: pros and cons





+ Thread switching time (no mode switch)
+ Scalability, customizability (no in-kernel management)
- Transparency (typically requires app cooperation)
- Parallelism (blocking syscalls are problematic)

Hybrid Implementations

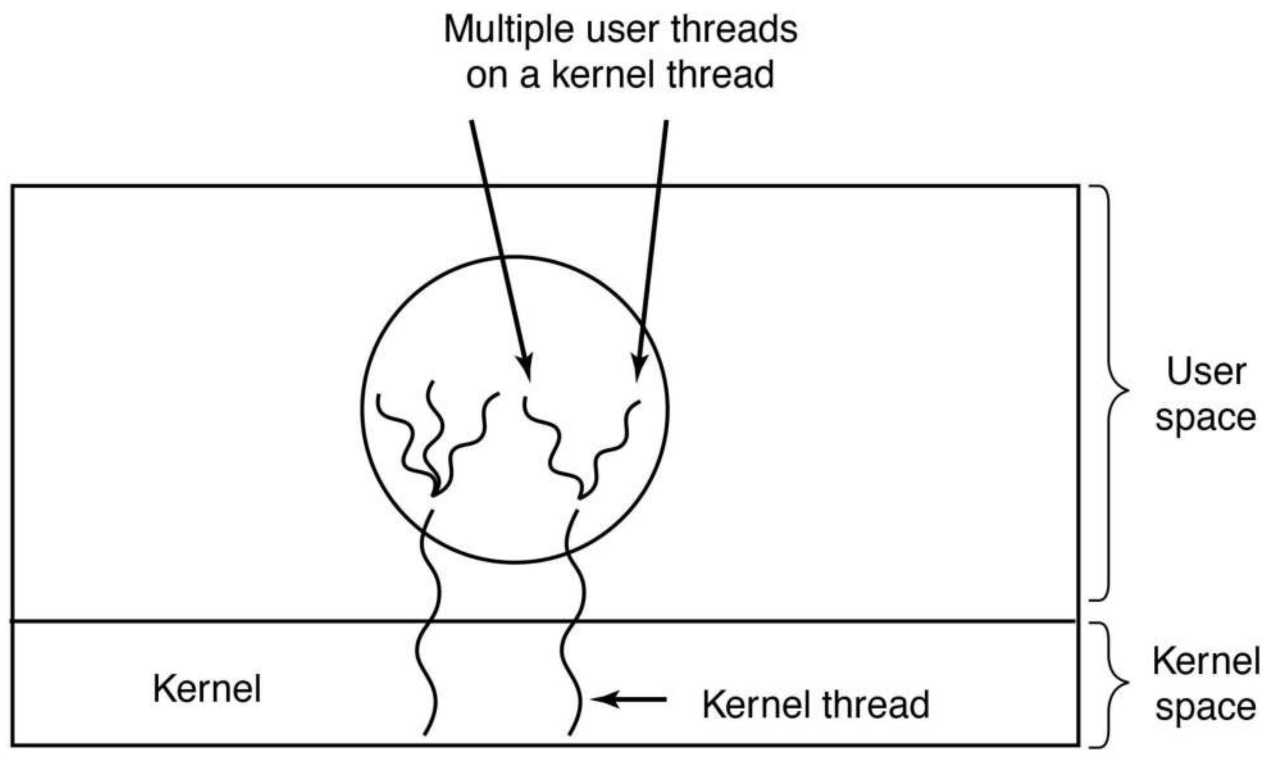


Figure 2-16. Multiplexing user-level threads onto kernel-level threads.

Making Single-Threaded Code Multithreaded (1 of 2)

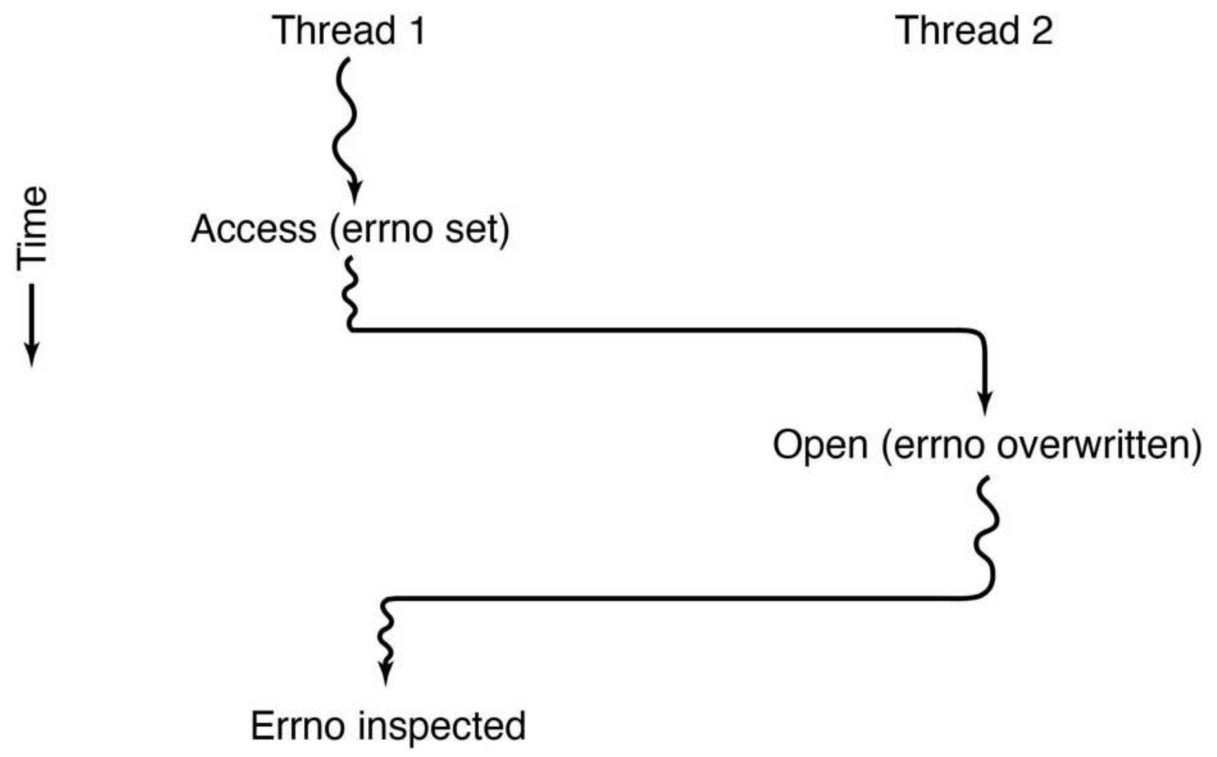


Figure 2-17. Conflicts between threads over the use of a global variable.

Making Single-Threaded Code Multithreaded (2 of 2)

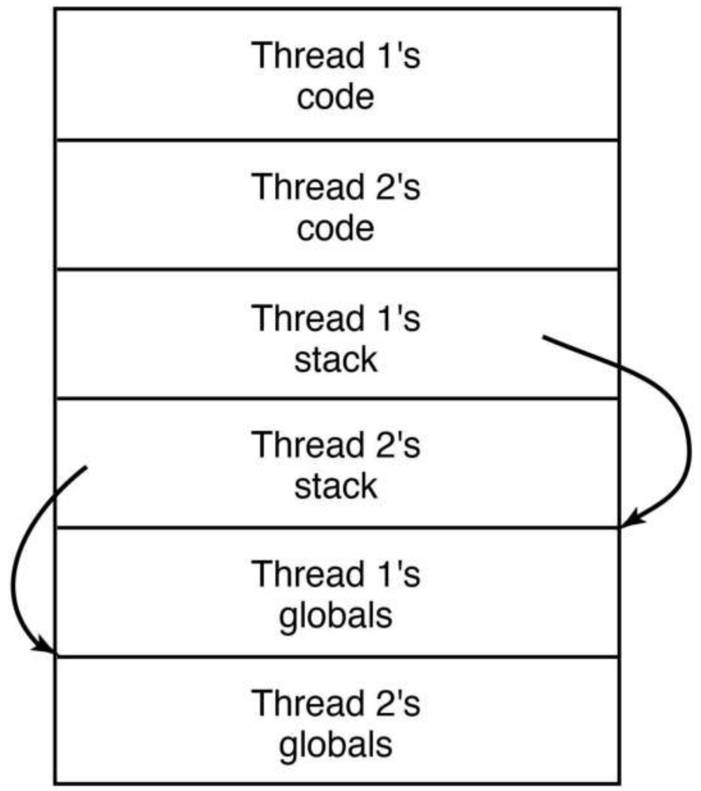


Figure 2-18. Threads can have private global variables.

Threads: issues

- Does the OS keep track of threads?
    - **Kernel threads** vs. **user threads**
- What to do on `fork`?
    - Clone **all threads** vs. **calling thread**
    - What if a thread is currently blocking on a systems call?
- What to do with signals?
    - Send signal to **all threads** vs. **single thread**
    - **Per-process** or **per-thread** signal handlers
- Where to store per-thread variables?
- Does sharing come at a cost?
- Are threads required inside an operating system?

Event-Driven Servers

- Implement server as finite-state machine that responds to events using asynchronous system calls
  - E.g., the availability of data on a socket
- Implementation can be very efficient
- Every event leads to a burst of activity without blocking
- Most OS offer event notification interfaces for asynchronous I/O
  - Linus: epoll
  - FreeBSD: kqueue

```
/* Preliminaries:
  svrSock : the main server socket, bound to TCP port 12345
  toSend : database to track what data we still have to send to the client
    - toSend.put (fd, msg) will register that we need to send msg on fd
    - toSend.get (fd) returns the string we need to send msg on fd
   -  toSend.destroy (fd) removes all infor mation about fd from toSend */

inFds = { svrSock } /* file descriptors to watch for incoming data */
outFds = { } /* file descriptors to watch to see if sending is possible */
exceptFds = { } /* file descriptors to watch for exception conditions (not used) */

char msgBuf [MAX MSG SIZE] /* buffer in which to receive messages */
char *thankYouMsg = "Thank you!" /* reply to send back */

while (TRUE)
{
  /* block until some file descriptors are ready to be used */
  rdyIns, rdyOuts, rdyExcepts = select (inFds, outFds, exceptFds, NO TIMEOUT)

  for (fd in rdyIns) /* iterate over all the connections that have something for us */
  {
    if (fd == svrSock) /* a new connection from a client */
    {
       newSock = accept (svrSock) /* create new socket for client */
       inFds = inFds ∪ { newSock } /* must monitor it also */
    }
   else
   { /* receive the message from the client */
      n = receive (fd, msgBuf, MAX MSG SIZE)
      printf ("Received: %s.0, msgBuf)

      toSend.put (fd, thankYouMsg) /* must still send thankYouMsg on fd */
      outFds = outFds ∪ { fd } /* so must monitor this fd */
    }
  }
  for (fd in rdyOuts) /* iterate over all the connections that we can now thank */
  {
     msg = toSend.get (fd) /* see what we need to send on this connection */
      n = send (fd, msg, strlen(msg))
      if (n < strlen (thankYouMsg)
      {
         toSend.put (fd, msg+n) /* remaining characters to send next time*/
       } else
      {
         toSend.destroy (fd)
         outFds = outFds \ { fd } /* we have thanked this one already */
      }
   }
}
```

Single-Threaded Versus Multi-threaded Versus Event-Driven Servers

Figure 2-20. Three ways to construct a server.

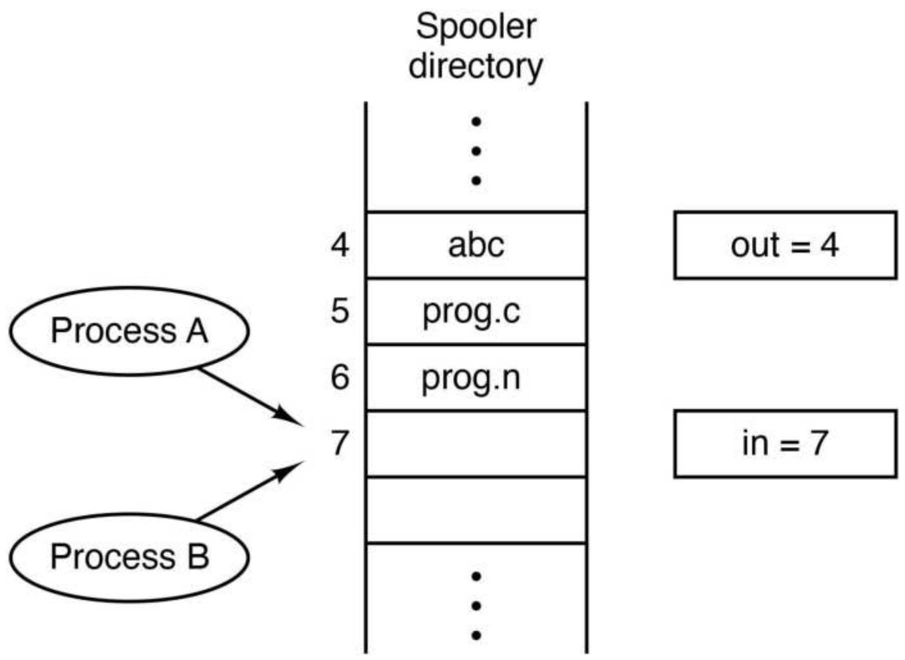
Model	**Characteristics**
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Synchronization and Inter-Process Communication (IPC)

- Why?
- Processes need some way to **communicate:**
  - To share data throughout the execution
- No explicit cross-process sharing:
  - → Data must be normally exchanged between processes
- Processes need some way to **synchronize**:
  - To account for dependencies
  - To avoid they get in each other’s way
  - Also applies to multithreaded execution

Race Conditions

- Process **A** reads `in=7` and decides to append its file at that position
- **A** is suspended by OS (because its slott expired)
- Process **B** also reads `in=7` and puts its file at that position
- **B** sets `in=8` and eventually gets suspended.
- **A** writes its file to position `7`



- **Problem**: reading/updating in should be an atomic action. If it is not, processes can race each other and come to wrong conclusions

Critical Regions (1 of 2)

Requirements to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Critical regions

- **Critical region**: a code region with access to shared resources
  1. No two processes may be simultaneously in their critical regions
  2. No assumptions may be made about speeds or nr. of CPUs
  3. No process running outside its critical region may block others
  4. No process should have to wait forever to enter its critical region
- **(Non)solutions**:
  - **Disable interrupts**: simply prevent that the CPU can be reallocated. Works for single-CPU systems only
  - **Lock variables**: guard critical regions with 0/1 variables. Races now occur on the lock variables themselves

Critical Regions (2 of 2)

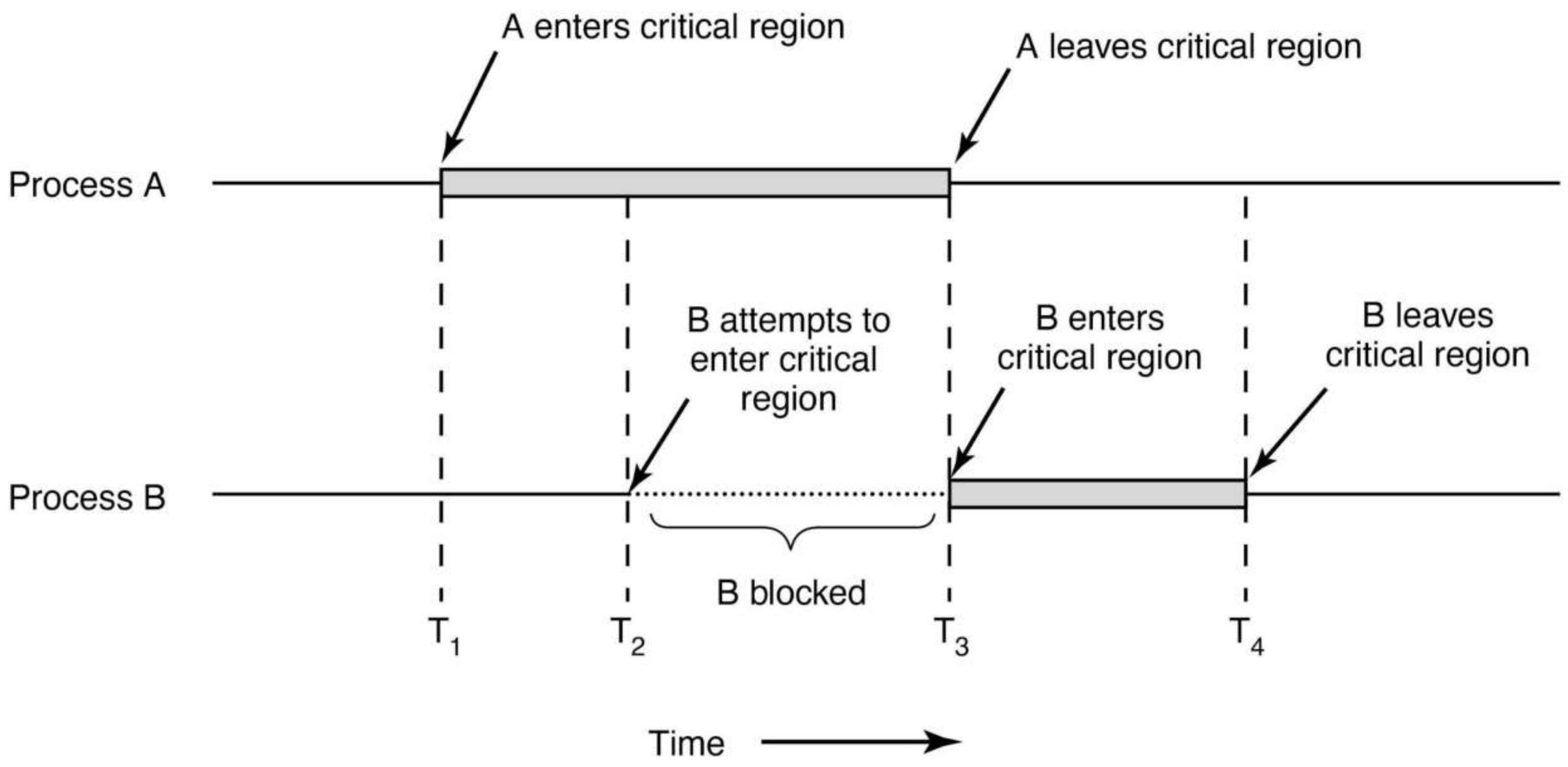


Figure 2-22. Mutual exclusion using critical regions.

Mutual Exclusion with Busy Waiting: Strict Alternation

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

Unfortunately, this is yet another (non)solution:

- Does not permit processes to enter their critical regions two times in a row
- A process outside the critical region can effectively block another one

Mutual Exclusion with Busy Waiting: Peterson’s Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
      int other;                          /* number of the other process */

      other = 1 – process;                /* the opposite of process */
      interested[process] = TRUE;         /* show that you are interested */
      turn = process;                     /* set flag */
      while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
      interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Figure 2-24. Peterson’s solution for achieving mutual exclusion.

Mutual Exclusion with Busy Waiting: Peterson’s algorithm

```
#define N 2

int turn;
int interested[N];

void enter_region(int process){
 int other = 1 - process;
 interested[process] = TRUE;
 turn = process;
 while(turn==process && interested[other]==TRUE);
}

void leave_region(int process){
 interested[process] = FALSE;
}
```

Mutual Exclusion with Busy Waiting: The TSL Instruction (1 of 2)

```
enter_region:
        TSL REGISTER,LOCK               | copy lock to register and set lock to 1
        CMP REGISTER,#0                 | was lock zero?
        JNE enter_region                | if it was not zero, lock was set, so loop
        RET                             | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                    | store a 0 in lock
        RET                             | return to caller
```

Figure 2-25. Entering and leaving a critical region using the TSL instruction.

Mutual Exclusion with Busy Waiting: The TSL Instruction (1 of 2)

- Hardware-assisted solution to the mutual exclusion problem
- **Atomic** test and set of a memory value
- Spin until **LOCK** is acquired

```
enter_region:
TSL REGISTER,LOCK  |copy LOCK to register and set LOCK to 1
CMP REGISTER,#0    |was LOCK zero?
JNE ENTER_REGION   |if it was non zero, LOCK was set, so
                   |loop
RET                |return to caller; critical regn entered

leave_region:
MOVE LOCK,#0       |store a 0 in LOCK
RET                |return to caller
```

Mutual Exclusion with Busy Waiting: The XCHG Instruction (2 of 2)

enter region:

```
   MOVE REGISTER,#1        | put a 1 in the register
   XCHG REGISTER,LOCK      | swap the contents of the register and lock variable
   CMP REGISTER,#0         | was lock zero?
   JNE enter region        | if it was non zero, lock was set, so loop
   RET                     | retur n to caller; critical region entered
```

leave region:

```
   MOVE LOCK,#0            | store a 0 in lock
   RET                     | retur n to caller
```

Figure 2-26. Entering and leaving a critical region using the XCHG instruction

Spinlocks and spinlock problems

```
void spin_lock(spinlock_t *lock);
void spin_unlock(spinlock_t *lock);

void my_irq_handler(void *shared_data, spinlock_t *lock)
{
  spin_lock(lock);
  update(shared_data);
  spin_unlock(lock);
}
void my_syscall_handler(void *shared_data, spinlock_t *lock)
{
  spin_lock(lock);
  read(shared_data);
  spin_unlock(lock);
}
```

What would happen when we get an interrupt after the syscall handler has taken the lock?

Avoiding Busy Waiting

- The solutions so far let a process keep the CPU busy waiting until it can enter its critical region (**spin lock**)
- **Solution**: let a process waiting to enter its critical region return the CPU to the scheduler voluntarily

```
void sleep(){                         void wakeup(process){
 set own state to BLOCKED;             set state of process to READY;
 give CPU to scheduler;                give CPU to scheduler;
}                                     }
```

Producer-Consumer

```
#define N 100
int count=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  if(count==N) sleep();
  insert_item(item);
  count++;
  if(count==1) wakeup(cons);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
  if(count==0) sleep();
  item = remove_item();
  count--;
  if(count==N-1) wakeup(prod);
  consume_item(item);
 }
}
```

Producer-Consumer

```
#define N 100
int count=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  if(count==N) sleep();
  insert_item(item);
  count++;
  if(count==1) wakeup(cons);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
  if(count==0) sleep();
  item = remove_item();
  count--;
  if(count==N-1) wakeup(prod);
  consume_item(item);
 }
}
```

Producer `sleeps` when buffer is *full*

Producer-Consumer

Consumer `sleeps` when buffer is *empty*

```
#define N 100
int count=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  if(count==N) sleep();
  insert_item(item);
  count++;
  if(count==1) wakeup(cons);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
  if(count==0) sleep();
  item = remove_item();
  count--;
  if(count==N-1) wakeup(prod);
  consume_item(item);
 }
}
```

Producer-Consumer

```
#define N 100
int count=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  if(count==N) sleep();
  insert_item(item);
  count++;
  if(count==1) wakeup(cons);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
  if(count==0) sleep();
  item = remove_item();
  count--;
  if(count==N-1) wakeup(prod);
  consume_item(item);
 }
}
```

***Problem*: wake up events may get lost!**
Sample run:
1.Con, 2.Prd, 3.Con
→ Cause? Effect?

Semaphores

- **Idea**: introduce a special `sema` integer type with 2 operations:
  - **`down`:**
    - if `sema` ≤ 0 then block the calling process
    - `sema=sema-1` otherwise
  - **`up`:**
    - if there is a process blocking on `sema`, wake it up
    - `sema=sema+1` otherwise
- OS guarantees all the operations are **atomic** by design
  - **Disable interrupts** on single processors
  - **Spin locking** on multiprocessors

- Back to busy waiting problems?
- Uses for binary semaphores (aka **mutexes**)?

Semaphores: Producer-Consumer

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  down(&empty);
  down(&mutex);
  insert_item(item);
  up(&mutex);
  up(&full);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
   down(&full);
   down(&mutex);
   item = remove_item();
   up(&mutex);
   up(&empty);
   consume_item(item);
 }
}
```

Semaphores: Producer-Consumer

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  down(&empty);
  down(&mutex);
  insert_item(item);
  up(&mutex);
  up(&full);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
   down(&full);
   down(&mutex);
   item = remove_item();
   up(&mutex);
   up(&empty);
   consume_item(item);
 }
}
```

mutex serializes access to the shared buffer

Semaphores: Producer-Consumer

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  down(&empty);
  down(&mutex);
  insert_item(item);
  up(&mutex);
  up(&full);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
   down(&full);
   down(&mutex);
   item = remove_item();
   up(&mutex);
   up(&empty);
   consume_item(item);
 }
}
```

`empty` semaphore blocks the producer when the shared buffer is *full*

Semaphores: Producer-Consumer

full semaphore blocks the consumer when the buffer is *empty*

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  down(&empty);
  down(&mutex);
  insert_item(item);
  up(&mutex);
  up(&full);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
   down(&full);
   down(&mutex);
   item = remove_item();
   up(&mutex);
   up(&empty);
   consume_item(item);
 }
}
```

Semaphores: Producer-Consumer

```
#define N 100

typedef int sema;
sema mutex=1;
sema empty=N, full=0;

void producer(void){
 int item;
 while(TRUE){
  item = produce_item();
  down(&empty);
  down(&mutex);
  insert_item(item);
  up(&mutex);
  up(&full);
 }
}
```

```
void consumer(void){
 int item;
 while(TRUE){
   down(&full);
   down(&mutex);
   item = remove_item();
   up(&mutex);
   up(&empty);
   consume_item(item);
 }
}
```

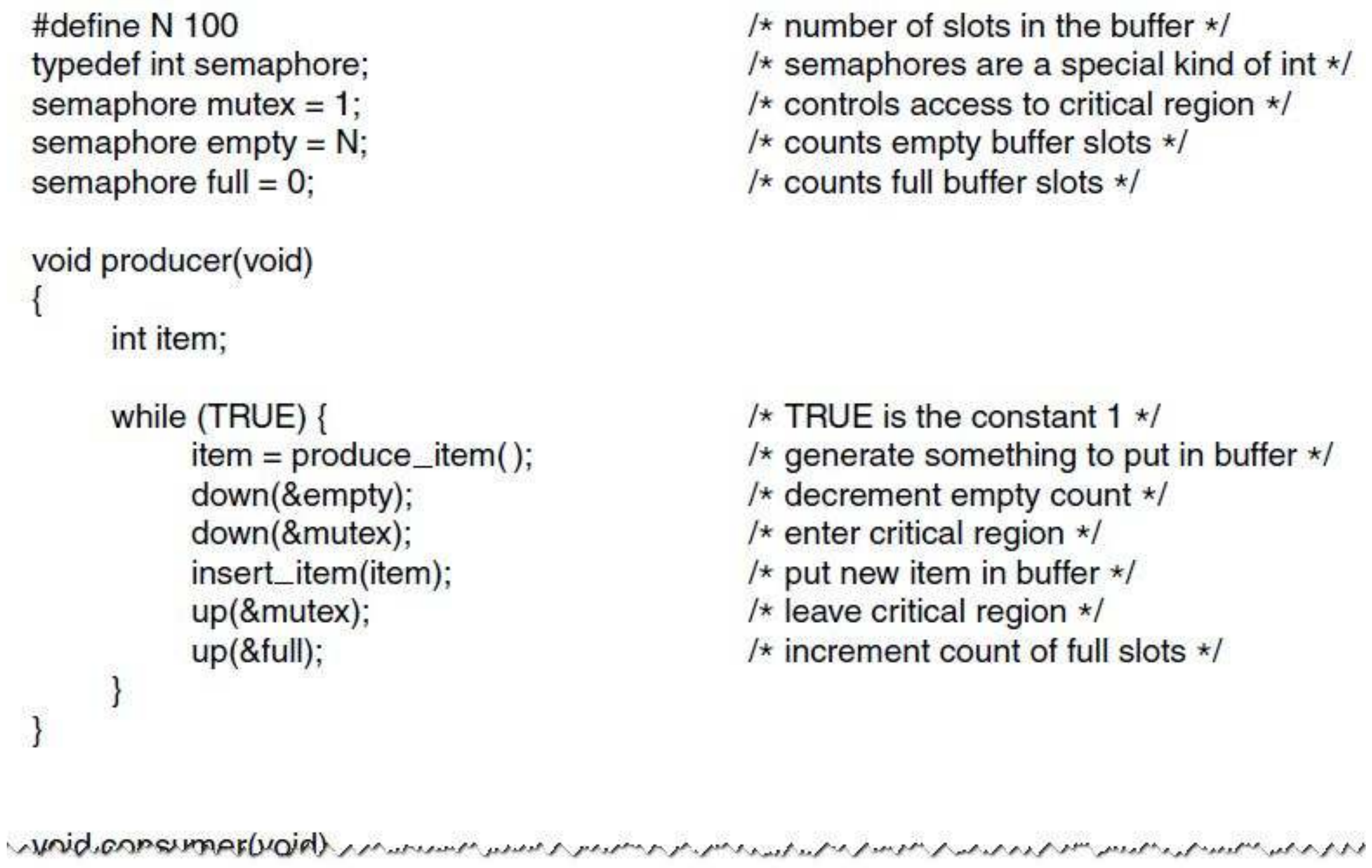
3 semaphores used in our solution.
→ Can we use 2?
→ Lost wakeups?

Full example on Linux

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define N 100
#define down sem_wait
#define up sem_post

/* Semaphores. */
sem_t mutex;
sem_t empty, full;
int mutex_init=1, empty_init=N, full_init=0;

/* Pthread wrappers. */
void producer(void);
static void *pthread_producer(void* args)
{
 producer();
 return NULL;
}

void consumer(void);
static void *pthread_consumer(void* args)
{
 consumer();
 return NULL;
}
```

```
/* Main entry point. */
int main(int argc, char **argv)
{
 pthread_t producer_tid, consumer_tid;

 srand(time(0));
 sem_init(&mutex, 0, mutex_init);
 sem_init(&empty, 0, empty_init);
 sem_init(&full, 0, full_init);

 fprintf(stderr, "Running threads...\n");
 pthread_create(&producer_tid, pthread_producer,...);
 pthread_create(&consumer_tid, pthread_consumer,...);
 sleep(3);

 fprintf(stderr, "Canceling threads...\n");
 pthread_cancel(producer_tid);
 pthread_cancel(consumer_tid);
 pthread_join(producer_tid, NULL);
 pthread_join(consumer_tid, NULL);

 fprintf(stderr, "Cleaning up...\n");
 sem_destroy(&mutex);
 sem_destroy(&empty);
 sem_destroy(&full);

 return 0;
}
```

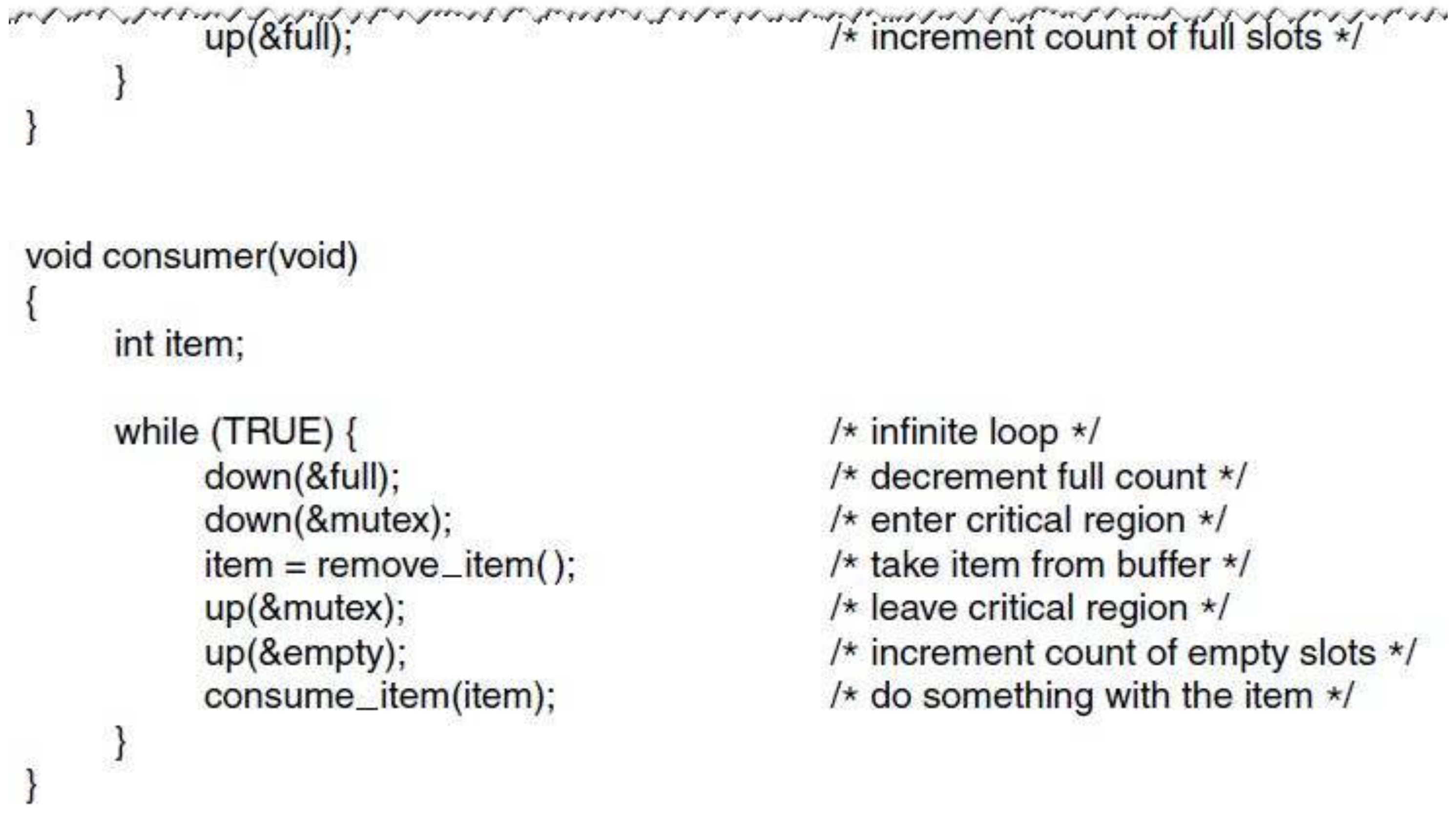
Semaphores (1 of 2)



```
#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 of 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.

Readers/Writers

- *N* processes access (i.e., read or write) some shared data
- At any given time: *R* readers **or** *1* writer allowed. Basic solution:

```
typedef int sema;
sema mutex = 1;
sema db = 1;
int rc = 0;
```

```
void reader(){
 while(TRUE){
  down(&mutex);
  rc++;
  if(rc==1) down(&db);
  up(&mutex);
  read_db();
  down(&mutex);
  rc--;
  if(rc==0) up(&db);
  up(&mutex);
  use_data_read();
 }
}
```

```
void writer(){
 while(TRUE){
  think_up_data();
  down(&db);
  write_db();
  up(&db);
 }
}
```

Readers/Writers

- *N* processes access (i.e., read or write) some shared data
- At any given time: *R* readers **or** *1* writer allowed. Basic solution:

```
typedef int sema;
sema mutex = 1;
sema db = 1;
int rc = 0;
```

```
void reader(){
 while(TRUE){
  down(&mutex);
  rc++;
  if(rc==1) down(&db);
  up(&mutex);
  read_db();
  down(&mutex);
  rc--;
  if(rc==0) up(&db);
  up(&mutex);
  use_data_read();
 }
}
```

```
void writer(){
 while(TRUE){
  think_up_data();
  down(&db);
  write_db();
  up(&db);
 }
}
```

`mutex` serializes access to the shared `rc` counter

Readers/Writers

- *N* processes access (i.e., read or write) some shared data
- At any given time: *R* readers **or** *1* writer allowed. Basic solution:

```
typedef int sema;
sema mutex = 1;
sema db = 1;
int rc = 0;
```

```
void reader(){
 while(TRUE){
  down(&mutex);
  rc++;
  if(rc==1) down(&db);
  up(&mutex);
  read_db();
  down(&mutex);
  rc--;
  if(rc==0) up(&db);
  up(&mutex);
  use_data_read();
 }
}
```

```
void writer(){
 while(TRUE){
  think_up_data();
  down(&db);
  write_db();
  up(&db);
 }
}
```

db semaphore controls RW access to the shared db

Readers/Writers

- *N* processes access (i.e., read or write) some shared data
- At any given time: *R* readers **or** *1* writer allowed. Basic solution:

```
typedef int sema;
sema mutex = 1;
sema db = 1;
int rc = 0;
```

```
void reader(){
 while(TRUE){
  down(&mutex);
  rc++;
  if(rc==1) down(&db);
  up(&mutex);
  read_db();
  down(&mutex);
  rc--;
  if(rc==0) up(&db);
  up(&mutex);
  use_data_read();
 }
}
```

```
void writer(){
 while(TRUE){
  think_up_data();
  down(&db);
  write_db();
  up(&db);
 }
}
```

db is a regular mutex from the writers' perspective

Readers/Writers

- *N* processes access (i.e., read or write) some shared data
- At any given time: *R* readers **or** *1* writer allowed. Basic solution:

```
typedef int sema;
sema mutex = 1;
sema db = 1;
int rc = 0;
```

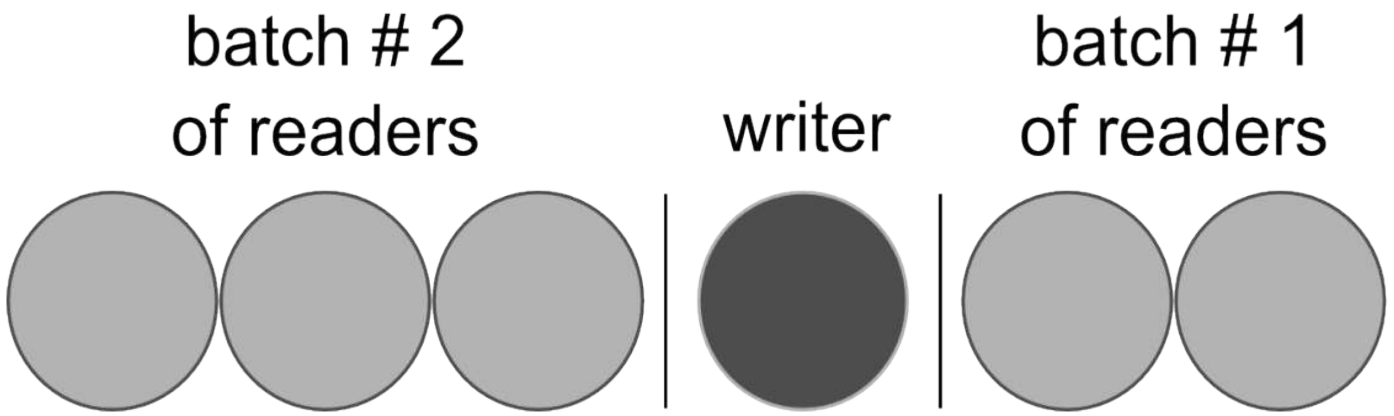
```
void reader(){
 while(TRUE){
  down(&mutex);
  rc++;
  if(rc==1) down(&db);
  up(&mutex);
  read_db();
  down(&mutex);
  rc--;
  if(rc==0) up(&db);
  up(&mutex);
  use_data_read();
 }
}
```

```
void writer(){
 while(TRUE){
  think_up_data();
  down(&db);
  write_db();
  up(&db);
 }
}
```

First / last reader issues `down` / `up` operations on `db`

Readers/Writers

- **Idea**: Build a queue of readers and writers
- Let several readers in at the same time
- Allow 1 writer when no readers are active
- How long may the writer have to wait?



The Readers and Writers Problem

```
typedef int semaphore;                /* use your imagination */
semaphore mutex = 1;                  /* controls access to rc */
semaphore db = 1;                     /* controls access to the database */
int rc = 0;                           /* # of processes reading or wanting to */

void reader(void)
{
      while (TRUE) {                  /* repeat forever */
            down(&mutex);             /* get exclusive access to rc */
            rc = rc + 1;              /* one reader more now */
            if (rc == 1) down(&db);   /* if this is the first reader ... */
            up(&mutex);               /* release exclusive access to rc */
            read_data_base();         /* access the data */
            down(&mutex);             /* get exclusive access to rc */
            rc = rc − 1;              /* one reader fewer now */
            if (rc == 0) up(&db);     /* if this is the last reader ... */
            up(&mutex);               /* release exclusive access to rc */
            use_data_read();          /* noncritical region */
      }
}


void writer(void)
{
      while (TRUE) {                  /* repeat forever */
            think_up_data();          /* noncritical region */
            down(&db);                /* get exclusive access */
            write_data_base();        /* update the data */
            up(&db);                  /* release exclusive access */
      }
}
```

Figure 2-29. A solution to the readers and writers problem.

Mutexes: simple implementation

```
pthread_mutex_lock:
1:    TSL REGISTER,MUTEX  | copy mutex to register and set mutex to 1
      CMP REGISTER,#0     | was mutex zero?
      JZE ok      | if it was zero, mutex was unlocked, so return
      CALL pthread_yield  | mutex is busy; schedule another thread
      JMP 1           | try again
ok:   RET         | return to caller; critical region entered


pthread_mutex_unlock:
     MOVE MUTEX,#0    | store a 0 in mutex
     RET          | return to caller
```

Figure 2-30. Implementation of **mutex_lock** and **mutex_unlock.**

(Extra material)

Pthread Mutexes: Linux-style implementation

```
pthread_mutex_lock:
1:   TSL REGISTER,MUTEX  | copy mutex to register and set mutex to 1
     CMP REGISTER,#0| was mutex zero?
     JZE ok       | if it was zero, mutex was unlocked, so return
     CALL futex_wait     | mutex is busy; request kernel to block thread
     JMP 1        | try again
ok:  RET          | return to caller; critical region entered


pthread_mutex_unlock:
    MOVE MUTEX,#0    | store a 0 in mutex
    CALL futex_wake | request kernel to unblock a waiter (if any)
    RET          | return to caller
```

Mutexes in Pthreads (1 of 5)

Thread Call	Description
pthread_mutex_init	Create a mutex
pthread_mutex_destroy	Destroy an existing mutex
pthread _mutex_lock	Acquire a lock or block
pthread_mutex_trylock	Acquire a lock or fail
pthread_mutex_unlock	Release a lock

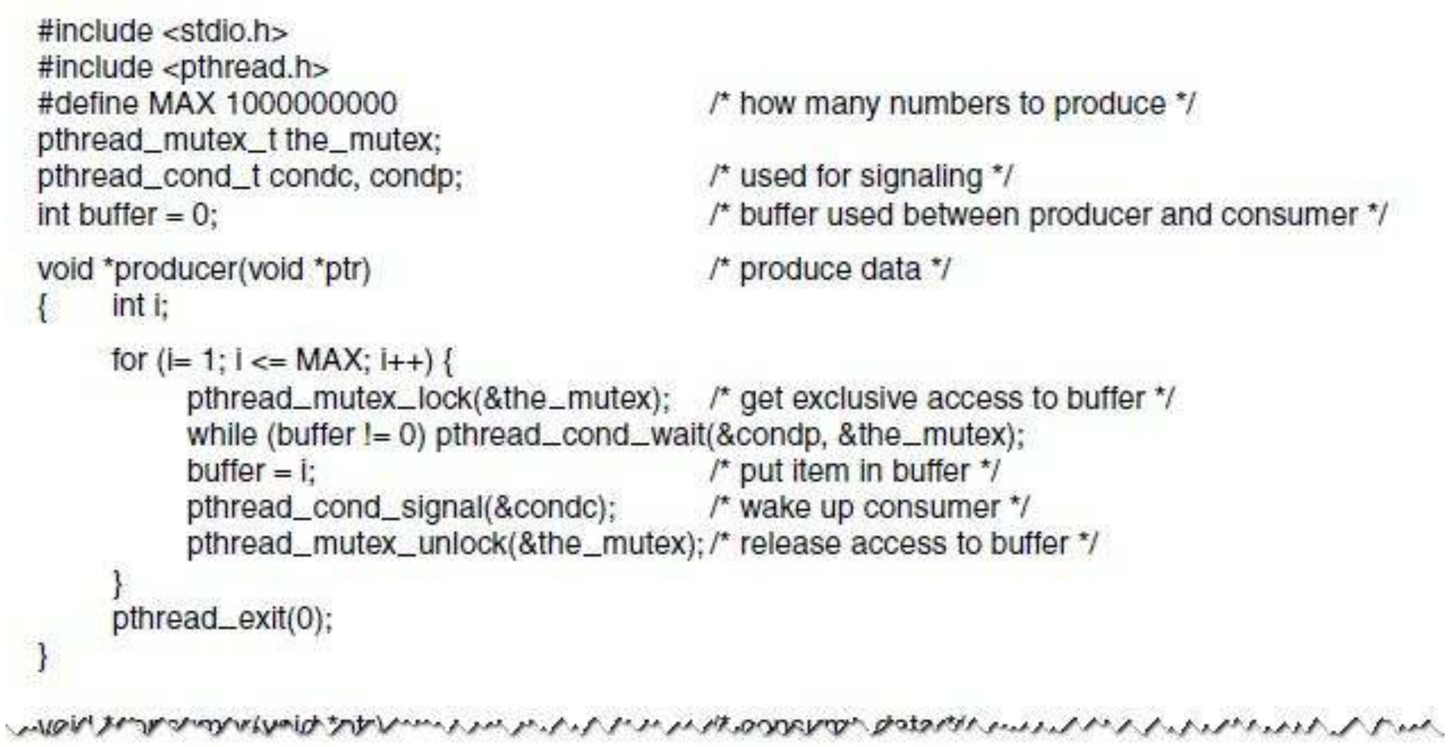
Figure 2-31. Some of the Pthreads calls relating to mutexes.

Mutexes in Pthreads (2 of 5)

Thread Call	Description
pthread_cond_init	Create a condition variable
pthread_cond_destroy	Destroy a condition variable
pthread _cond_wait	Block waiting for a signal
pthread_cond_signal	Signal another thread and wake it up
pthread_cond_broadcast	Signal multiple threads and wake all of them

Figure 2-32. Some of the Pthreads calls relating to condition variables.

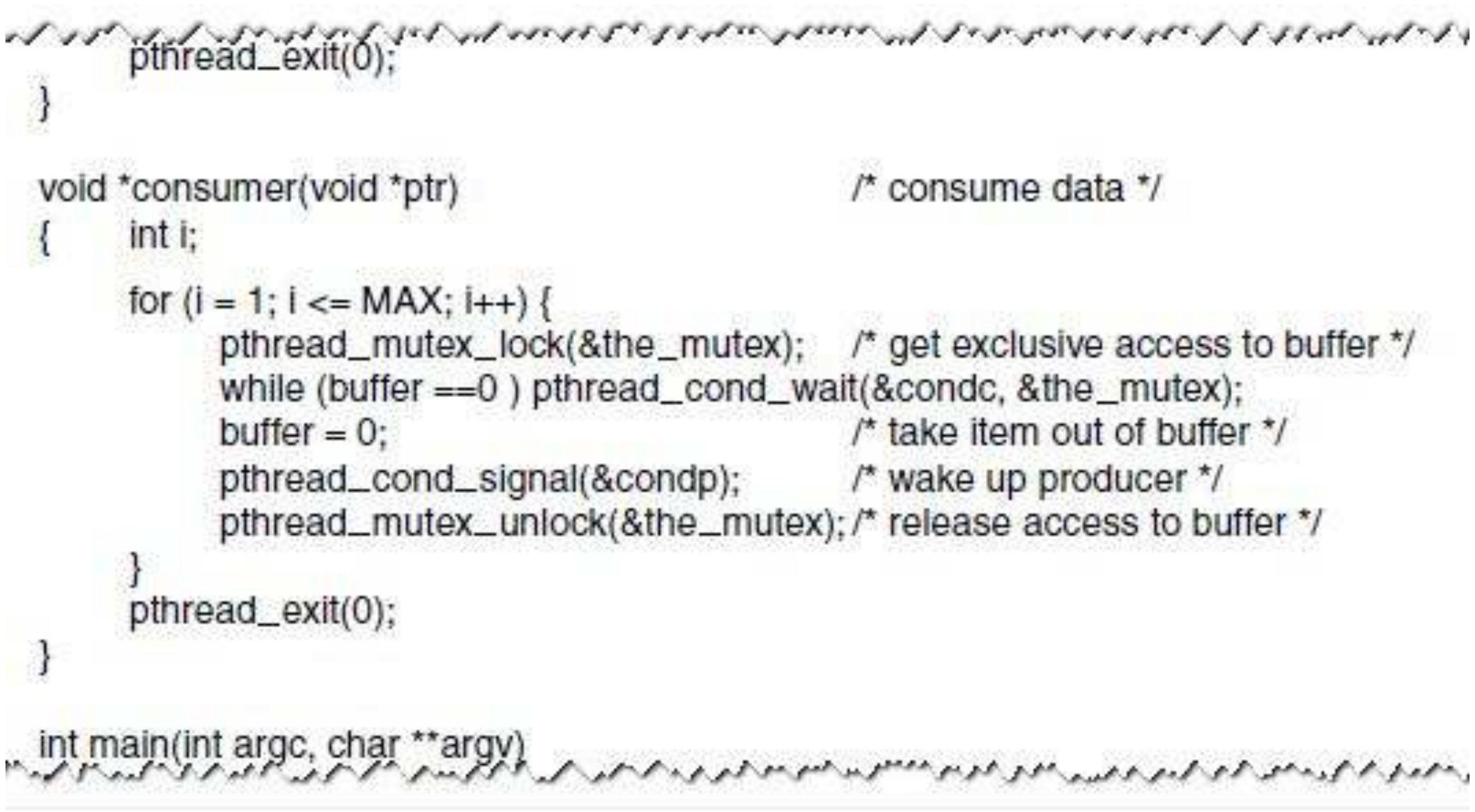
Mutexes in Pthreads (3 of 5)



```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* used for signaling */
int buffer = 0;                                 /* buffer used between producer and consumer */

void *producer(void *ptr)                       /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                       /* put item in buffer */
                pthread_cond_signal(&condc);      /* wake up consumer */
                pthread_mutex_unlock(&the_mutex); /* release access to buffer */
        }
        pthread_exit(0);
}
```

Figure 2-33. Using threads to solve the producer-consumer problem.

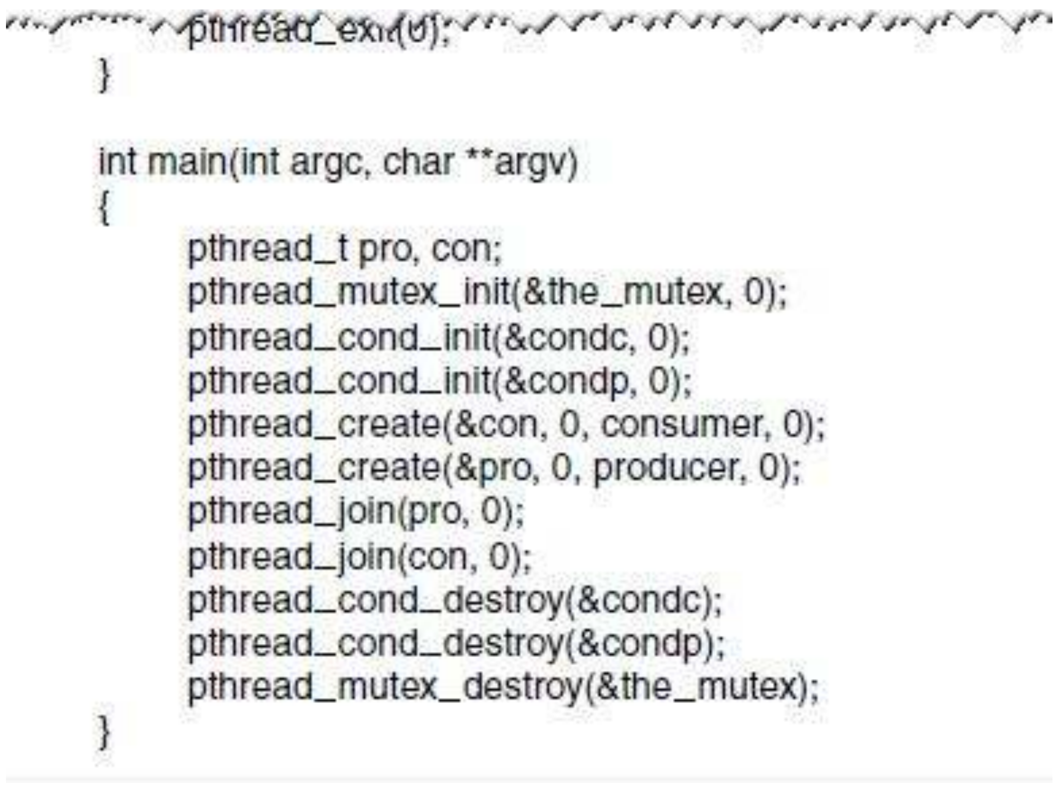
Mutexes in Pthreads (4 of 5)



```
          pthread_exit(0);
}

void *consumer(void *ptr)                        /* consume data */
{         int i;

          for (i = 1; i <= MAX; i++) {
                    pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
                    while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                    buffer = 0;                       /* take item out of buffer */
                    pthread_cond_signal(&condp);      /* wake up producer */
                    pthread_mutex_unlock(&the_mutex); /* release access to buffer */
          }
          pthread_exit(0);
}

int main(int argc, char **argv)
```

Figure 2-33. Using threads to solve the producer-consumer problem.

Mutexes in Pthreads (5 of 5)



```
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Figure 2-33. Using threads to solve the producer-consumer problem.

Monitors

- Semaphores have been heavily criticized for the chaos they can introduce in programs
- **Monitors**: more structured approach towards process synchronization:
  - Serialize the procedure calls on a given module
  - Use condition variables to `wait` / `signal` processes
- **Requires** dedicated language support
- Popular in managed languages, e.g., Java:
  - `synchronized` methods / blocks
  - `wait`, `notify`, `notifyall` primitives

Monitors (1 of 6)

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;


        procedure consumer( );
        .          .          .
        end;
end monitor;
```

Figure 2-34. A monitor.

Monitors: Producer-Consumer

```
monitor ProdCons{
 condition full, empty;
 int count=0;
 void enter(int item) {
  if(count==N) wait(full);
  insert_item(item);
  count++;
  if(count==1) signal(empty);
 }
 void remove(int *item) {
  if(count==0) wait(empty);
  *item = remove_item();
  count--;
  if(count==N-1) signal(full);
 }
}
```

```
void producer(){
 int item;
 while(TRUE){
  item = produce_item();
  ProdCons.enter(item);
 }
}


void consumer(){
 int item;
 while(TRUE){
  ProdCons.remove(&item);
  consume_item(item);
 }
}
```

Monitors: Producer-Consumer

Access to enter and remove is serialized by the monitor

```
monitor ProdCons{
 condition full, empty;
 int count=0;
 void enter(int item) {
  if(count==N) wait(full);
  insert_item(item);
  count++;
  if(count==1) signal(empty);
 }
 void remove(int *item) {
  if(count==0) wait(empty);
  *item = remove_item();
  count--;
  if(count==N-1) signal(full);
 }
}
```

```
void producer(){
 int item;
 while(TRUE){
  item = produce_item();
  ProdCons.enter(item);
 }
}


void consumer(){
 int item;
 while(TRUE){
  ProdCons.remove(&item);
  consume_item(item);
 }
}
```

Monitors: Producer-Consumer

`wait` **suspends caller on a condition variable.**
→ *Monitor state?*

```
monitor ProdCons{
 condition full, empty;
 int count=0;
 void enter(int item) {
  if(count==N) wait(full);
  insert_item(item);
  count++;
  if(count==1) signal(empty);
 }
 void remove(int *item) {
  if(count==0) wait(empty);
  *item = remove_item();
  count--;
  if(count==N-1) signal(full);
 }
}
```

```
void producer(){
 int item;
 while(TRUE){
  item = produce_item();
  ProdCons.enter(item);
 }
}


void consumer(){
 int item;
 while(TRUE){
  ProdCons.remove(&item);
  consume_item(item);
 }
}
```

Monitors: Producer-Consumer

`signal` **wakes up one waiter on a condition variable.**
→ *Monitor state?*
→ *Lost wakeups?*

```
monitor ProdCons{
 condition full, empty;
 int count=0;
 void enter(int item) {
  if(count==N) wait(full);
  insert_item(item);
  count++;
  if(count==1) signal(empty);
 }
 void remove(int *item) {
  if(count==0) wait(empty);
  *item = remove_item();
  count--;
  if(count==N-1) signal(full);
 }
}
```

```
void producer(){
 int item;
 while(TRUE){
  item = produce_item();
  ProdCons.enter(item);
 }
}


void consumer(){
 int item;
 while(TRUE){
  ProdCons.remove(&item);
  consume_item(item);
 }
}
```

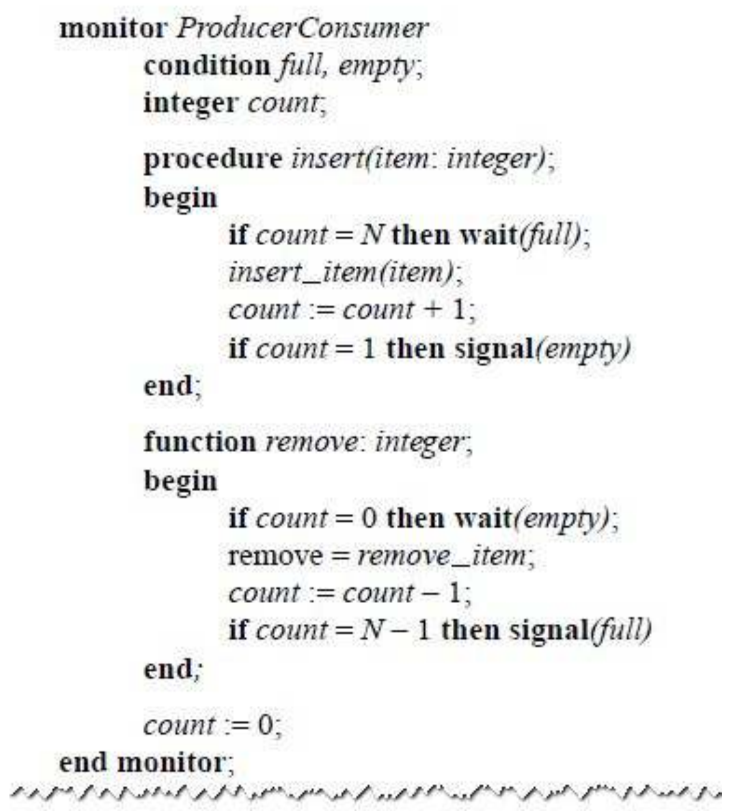
Monitors: Producer-Consumer

Monitors make parallel programming much *easier.*
→ ***Do we need more?***

```
monitor ProdCons{
 condition full, empty;
 int count=0;
 void enter(int item) {
  if(count==N) wait(full);
  insert_item(item);
  count++;
  if(count==1) signal(empty);
 }
 void remove(int *item) {
  if(count==0) wait(empty);
  *item = remove_item();
  count--;
  if(count==N-1) signal(full);
 }
}
```

```
void producer(){
 int item;
 while(TRUE){
  item = produce_item();
  ProdCons.enter(item);
 }
}


void consumer(){
 int item;
 while(TRUE){
  ProdCons.remove(&item);
  consume_item(item);
 }
}
```

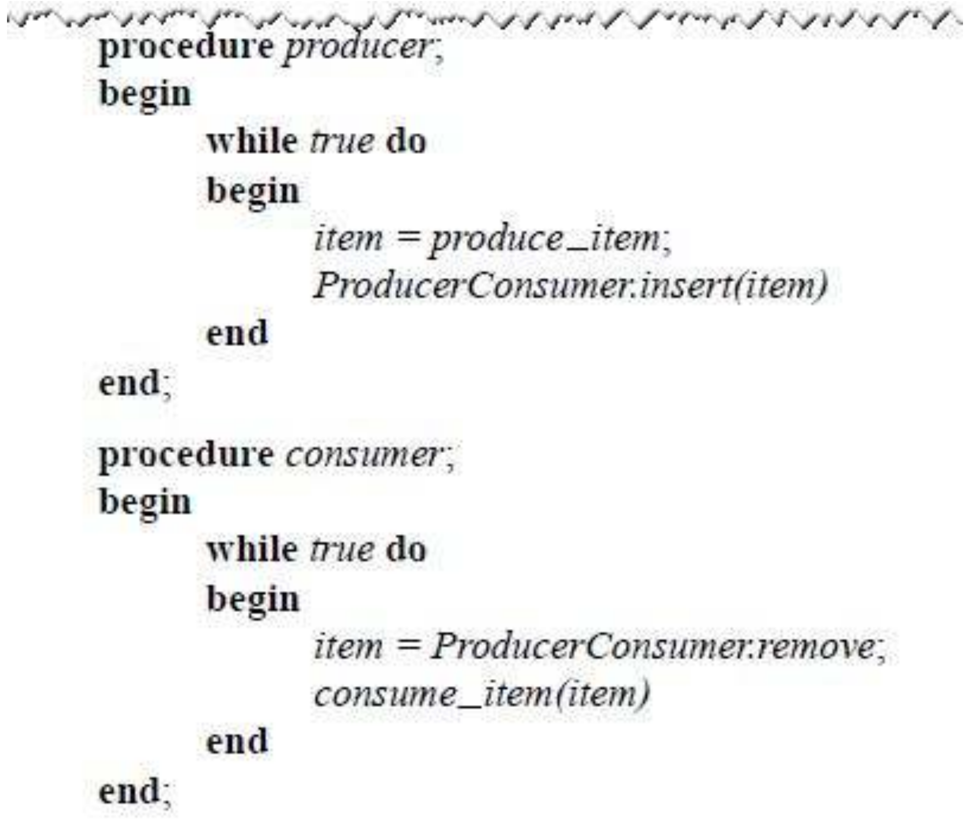
Monitors (2 of 6)



```
monitor ProducerConsumer
    condition full, empty;
    integer count;

    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;

    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count − 1;
        if count = N − 1 then signal(full)
    end;

    count := 0;
end monitor;
```

Figure 2-35. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

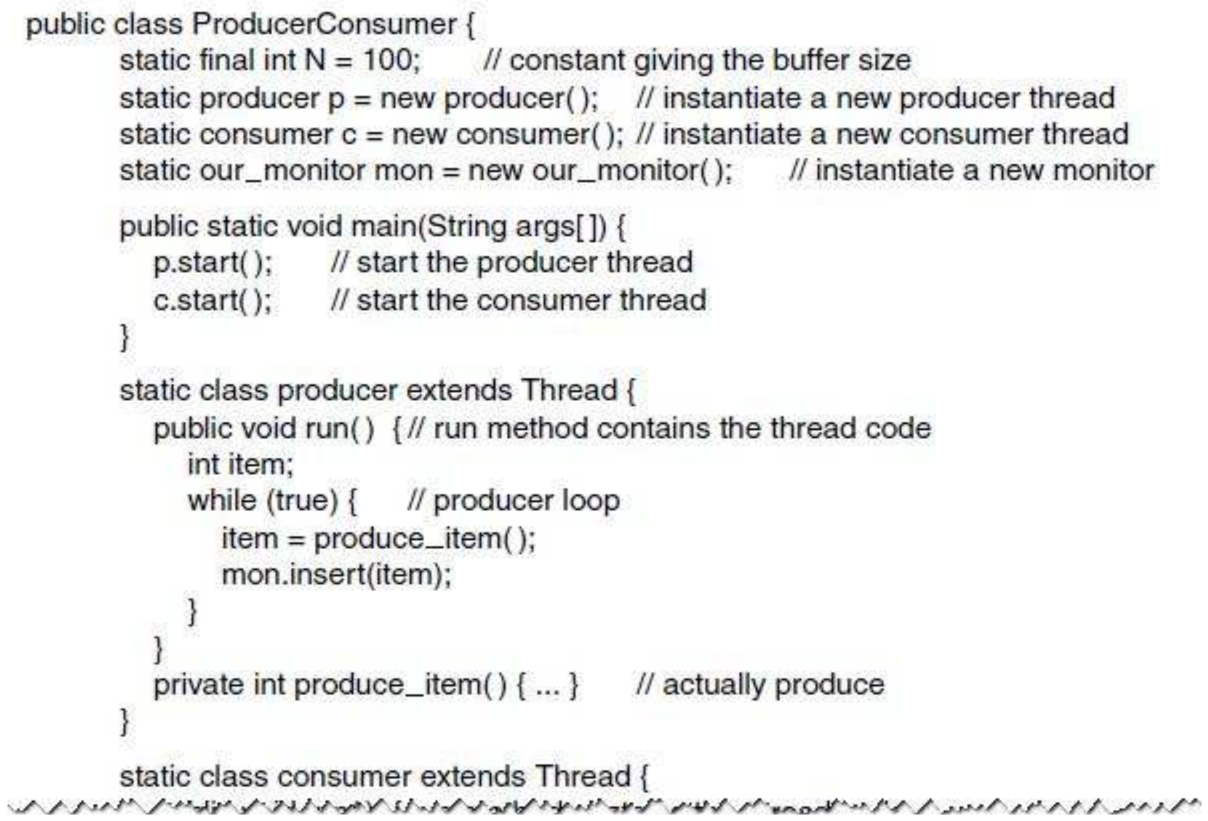
Monitors (3 of 6)



```
procedure producer;
begin
      while true do
      begin
            item = produce_item;
            ProducerConsumer.insert(item)
      end
end;

procedure consumer;
begin
      while true do
      begin
            item = ProducerConsumer.remove;
            consume_item(item)
      end
end;
```

Figure 2-35. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

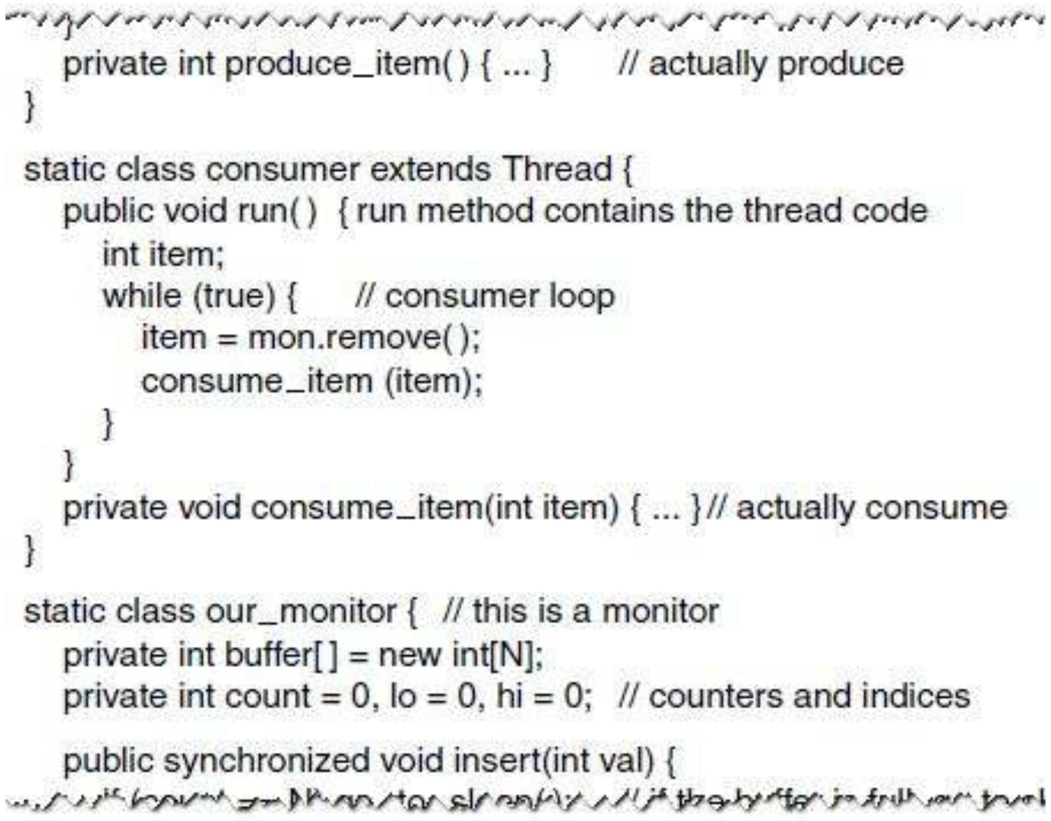
Monitors (4 of 6)



```
public class ProducerConsumer {
        static final int N = 100;      // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );      // instantiate a new monitor

        public static void main(String args[ ]) {
          p.start( );       // start the producer thread
          c.start( );       // start the consumer thread
        }

        static class producer extends Thread {
          public void run( )  {// run method contains the thread code
            int item;
            while (true) {      // producer loop
              item = produce_item( );
              mon.insert(item);
            }
          }
          private int produce_item( ) { ... }      // actually produce
        }

        static class consumer extends Thread {
```

Figure 2-36. A solution to the producer-consumer problem in Java.

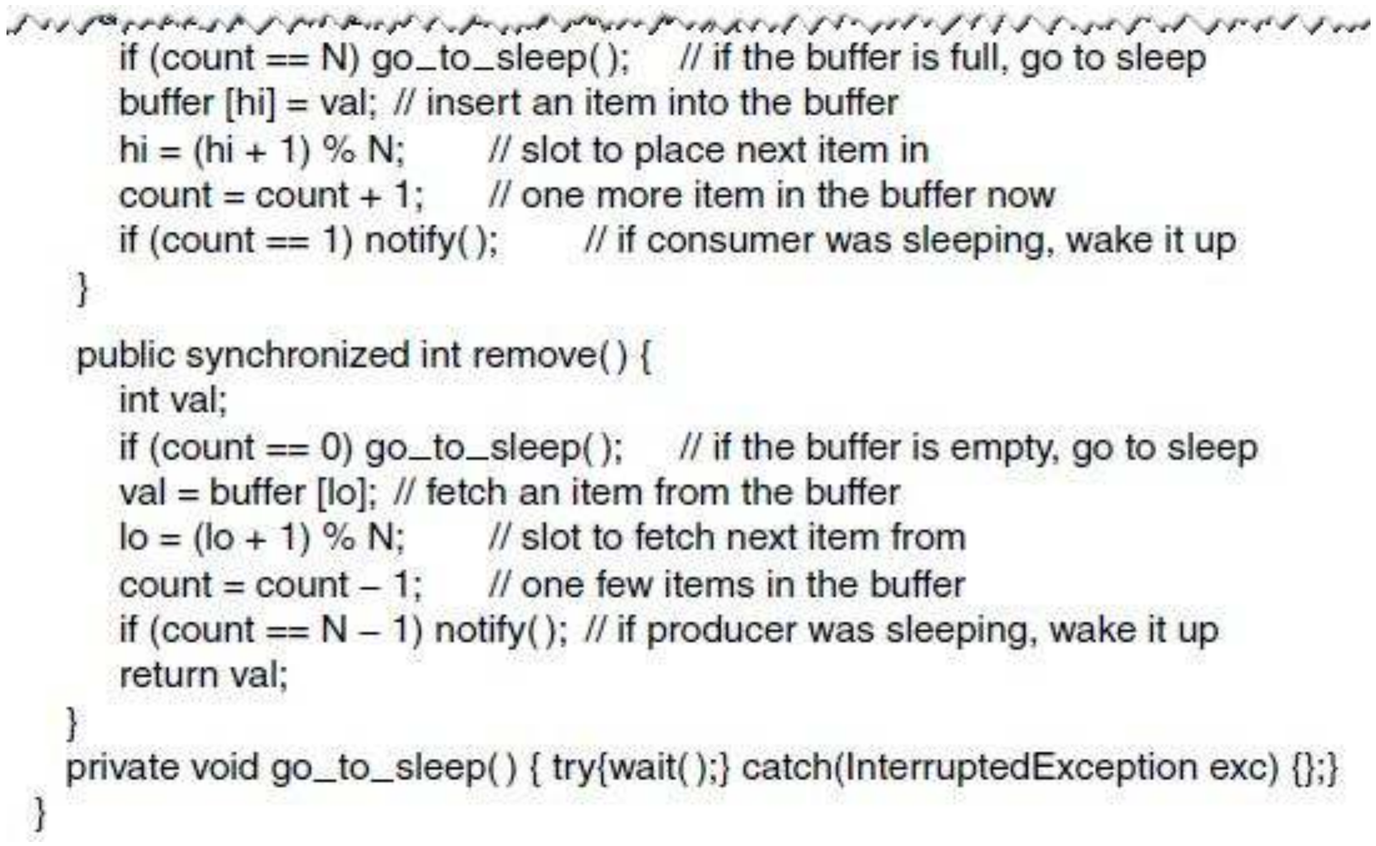
Monitors (5 of 6)



```
    private int produce_item( ) { ... }       // actually produce
}

static class consumer extends Thread {
    public void run( )  { run method contains the thread code
        int item;
        while (true) {      // consumer loop
            item = mon.remove( );
            consume_item (item);
        }
    }
    private void consume_item(int item) { ... }// actually consume
}

static class our_monitor {   // this is a monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0;   // counters and indices

    public synchronized void insert(int val) {
```

Figure 2-36. A solution to the producer-consumer problem in Java.

Monitors (6 of 6)



```
        if (count == N) go_to_sleep();    // if the buffer is full, go to sleep
        buffer [hi] = val; // insert an item into the buffer
        hi = (hi + 1) % N;       // slot to place next item in
        count = count + 1;       // one more item in the buffer now
        if (count == 1) notify();       // if consumer was sleeping, wake it up
    }
    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep();    // if the buffer is empty, go to sleep
        val = buffer [lo]; // fetch an item from the buffer
        lo = (lo + 1) % N;       // slot to fetch next item from
        count = count – 1;       // one few items in the buffer
        if (count == N – 1) notify(); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
}
```

Figure 2-36. A solution to the producer-consumer problem in Java.

Message passing

- Solution to both the process **synchronization** and the process **communication** problems
- Most common choice in **multiserver** OS designs
- Processes interact by sending and receiving **messages**:
  - `send(destination, &message);`
  - `receive(source, &message);`
  - `receive(ANY, &message);`

Message Passing: Producer-Consumer

```
#define N 100

void producer(){
 int item;
 message msg;


 while(TRUE){
  item = produce_item();
  receive(consumer, &msg);
  build_message(&msg, item);
  send(consumer, &msg);
 }
}
```

```
void consumer(){
 int item, i;
 message msg;
 for(i=0; i<N; i++)
  send(producer, &msg);
 while(TRUE){
  receive(producer, &msg);
  item = extract_item();
  send(producer, &msg);
  cosume_item(item);
 }
}
```

Message Passing: Producer-Consumer

N messages buffered by the system, initially all *empty* (producer's queue)

```
#define N 100

void producer(){
 int item;
 message msg;


 while(TRUE){
  item = produce_item();
  receive(consumer, &msg);
  build_message(&msg, item);
  send(consumer, &msg);
 }
}
```

```
void consumer(){
 int item, i;
 message msg;
 for(i=0; i<N; i++)
  send(producer, &msg);
 while(TRUE){
  receive(producer, &msg);
  item = extract_item();
  send(producer, &msg);
  cosume_item(item);
 }
}
```

Message Passing: Producer-Consumer

Producer `receives` an *empty* message and "*replaces*" it with a *full* message

```
#define N 100

void producer(){
 int item;
 message msg;


 while(TRUE){
  item = produce_item();
  receive(consumer, &msg);
  build_message(&msg, item);
  send(consumer, &msg);
 }
}
```

```
void consumer(){
 int item, i;
 message msg;
 for(i=0; i<N; i++)
  send(producer, &msg);
 while(TRUE){
  receive(producer, &msg);
  item = extract_item();
  send(producer, &msg);
  cosume_item(item);
 }
}
```

Message Passing: Producer-Consumer

Consumer `receives` a *full* message and "*replaces*" it with an *empty* message

```
#define N 100

void producer(){
 int item;
 message msg;


 while(TRUE){
  item = produce_item();
  receive(consumer, &msg);
  build_message(&msg, item);
  send(consumer, &msg);
 }
}
```

```
void consumer(){
 int item, i;
 message msg;
 for(i=0; i<N; i++)
  send(producer, &msg);
 while(TRUE){
  receive(producer, &msg);
  item = extract_item();
  send(producer, &msg);
  cosume_item(item);
 }
}
```

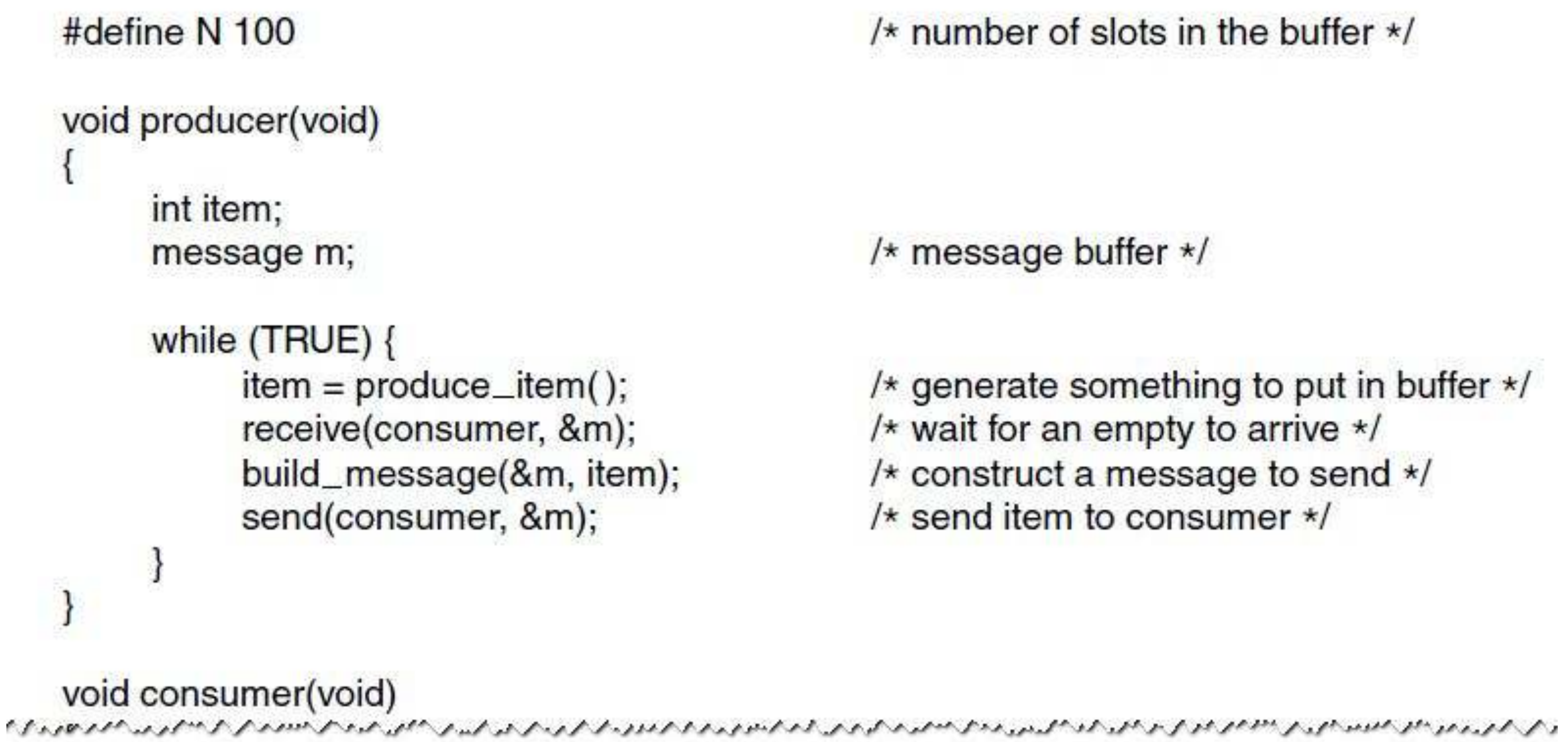
Message Passing: Producer-Consumer

receive blocks producer / consumer when N messages are *full / empty*

```
#define N 100

void producer(){
 int item;
 message msg;


 while(TRUE){
  item = produce_item();
  receive(consumer, &msg);
  build_message(&msg, item);
  send(consumer, &msg);
 }
}
```

```
void consumer(){
 int item, i;
 message msg;
 for(i=0; i<N; i++)
  send(producer, &msg);
 while(TRUE){
  receive(producer, &msg);
  item = extract_item();
  send(producer, &msg);
  cosume_item(item);
 }
}
```

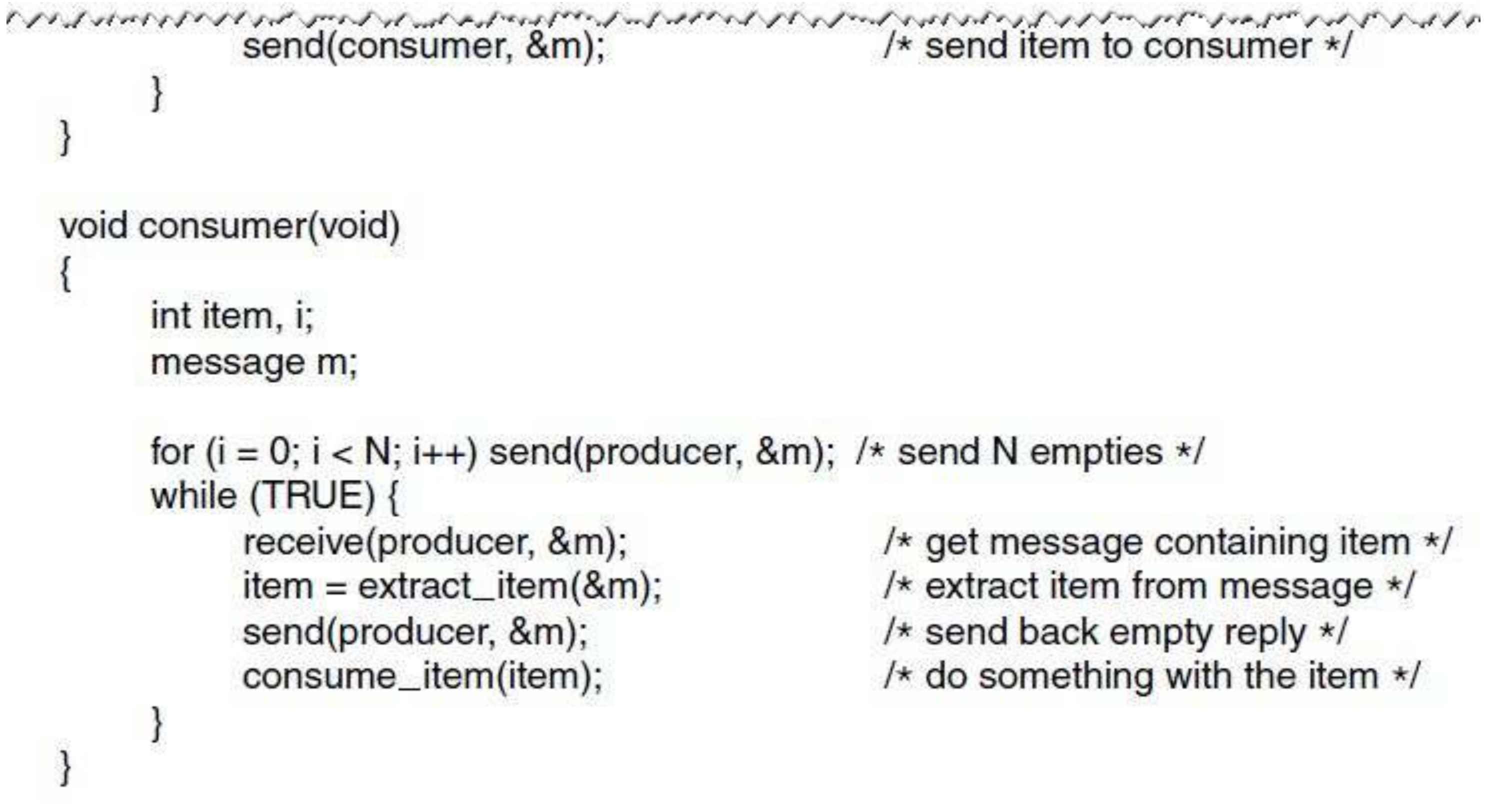
The Producer-Consumer Problem with Message Passing (1 of 2)



```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
        int item;
        message m;                              /* message buffer */

        while (TRUE) {
                item = produce_item( );         /* generate something to put in buffer */
                receive(consumer, &m);          /* wait for an empty to arrive */
                build_message(&m, item);        /* construct a message to send */
                send(consumer, &m);             /* send item to consumer */
        }
}

void consumer(void)
```

Figure 2-37. The producer-consumer problem with **N** messages.

The Producer-Consumer Problem with Message Passing (2 of 2)



```
            send(consumer, &m);                      /* send item to consumer */
        }
}

void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
            receive(producer, &m);                   /* get message containing item */
            item = extract_item(&m);                 /* extract item from message */
            send(producer, &m);                      /* send back empty reply */
            consume_item(item);                      /* do something with the item */
        }
}
```

Figure 2-37. The producer-consumer problem with **N** messages.

Barriers

no process may proceed into the next phase until all processes are ready to proceed to the next phase

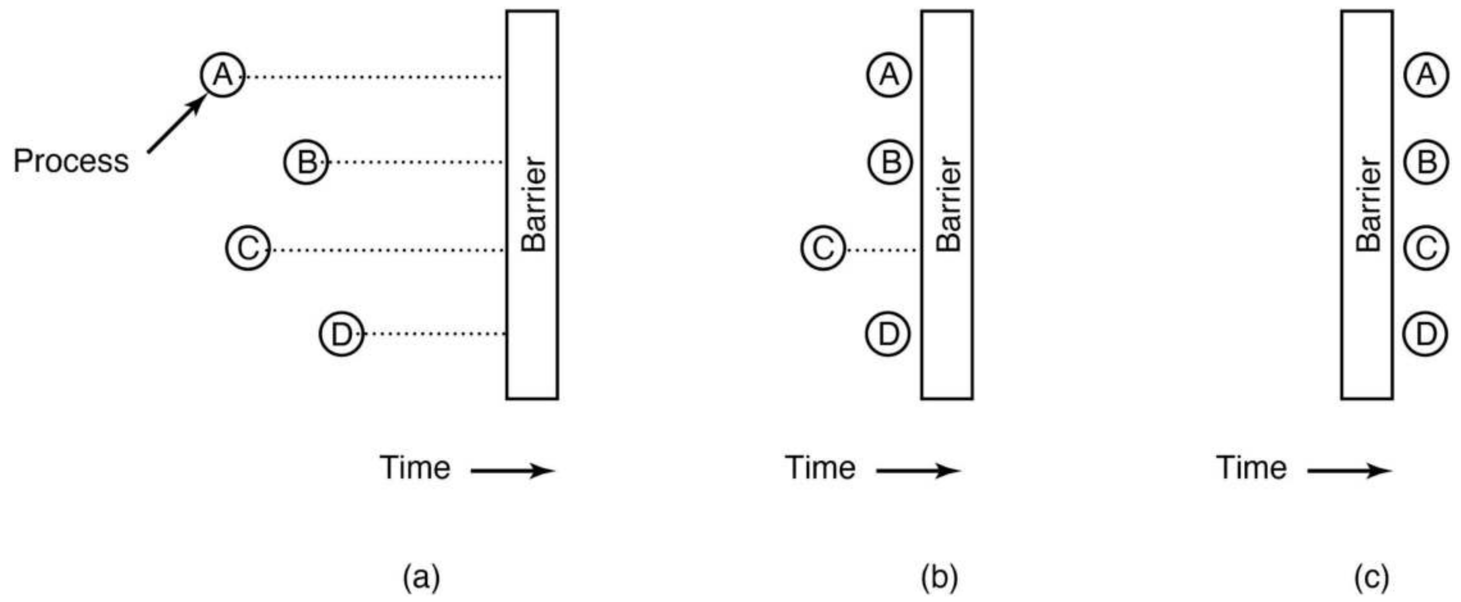


Figure 2-38. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

Priority Inversion

Consider the Mars Pathfinder

It had three subsystems

1. A data-distribution system (**High** Prio)
2. A Communication system (**Med** Prio)
3. A meteorological data gathering (**Low** Prio)

There was also a common hardware subsystem:

- The data bus used by Task 1 and 3.
- It is protected by a mutex

Question: What can happen?

Priority Inversion

Consider the Mars Pathfinder

It had three subsystems

1. A data-distribution system (**High** Prio)
2. A Communication system (**Med** Prio)
3. A meteorological data gathering (**Low** Prio)

There was also a common hardware subsystem:

- The data bus used by Task 1 and 3.
- It is protected by a mutex

Question: What can happen?

Imagine the following scenario: (1) Low Prio task takes mutex, (2) gets interrupted by Med Prio task, (3) which is interrupted by High Prio task (which blocks because Low Prio task holds lock) □ Medium Prio task runs even though there is a High Prio task to run (priority is inverted)

Priority Inversion

- Several methods to solve priority inversion
  - Disable all interrupts while in the critical region
  - Priority ceiling: associate a priority with the mutex and assign that to the process holding it
  - Priority inheritance: A low-priority task holding the mutex temporarily inherits the priority of the high-priority task trying to obtain it
  - Random boosting: randomly assigning mutex-holding threads a high priority until they exit the critical region

Avoiding Locks: Read-Copy-Update

- An instance of **relativistic programming**
- Do not try to avoid conflicts between readers and writers □ tolerate them and ensure a correct result regardless of the order of events
- Allow writer to update data structure even if other processes are still using it.
    - □ Parallel copies of a data structure.
    - Ensure that each reader either reads the old or the new version, but not some weird combination of the 2.
    - Single-pointer readers/writers scheme
- Readers execute in read-side sections
- Writer operates 3 steps:
    - Atomically update pointer to new copy
    - Wait for existing readers (**grace period**)
    - Reclaim old copy

Avoiding Locks: Read-Copy-Update (1 of 2)

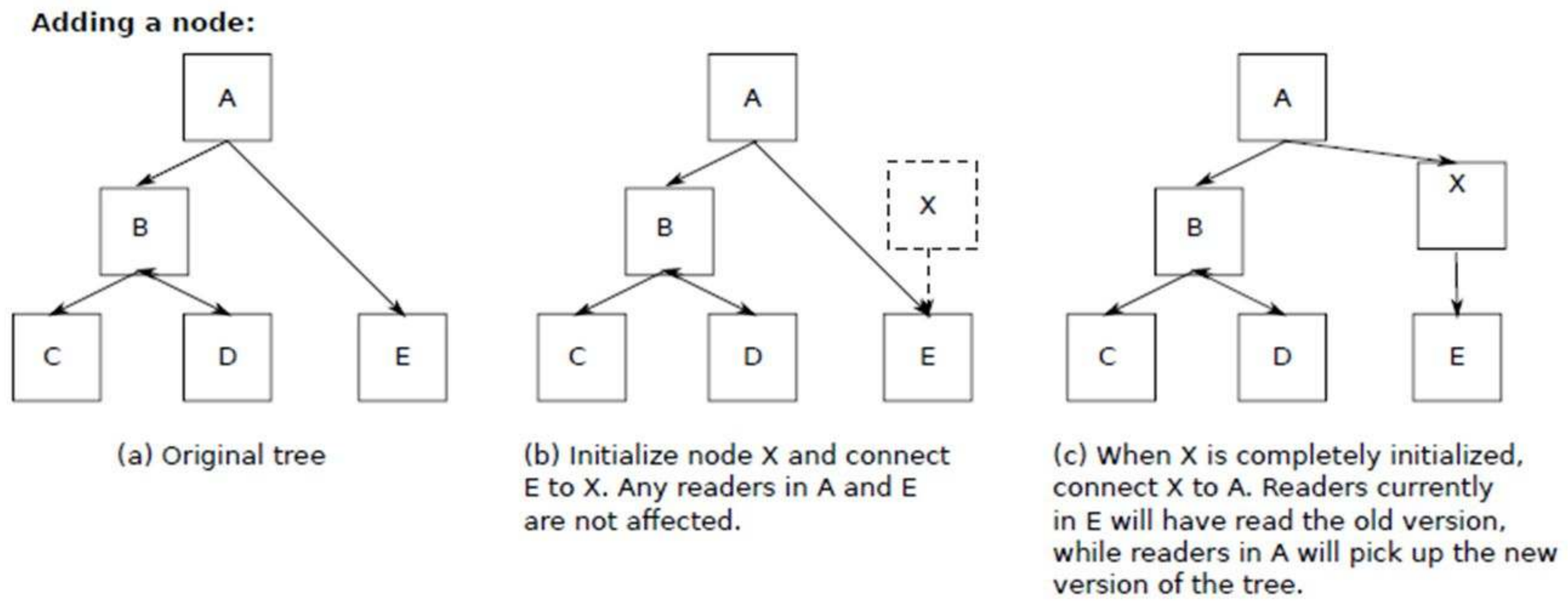
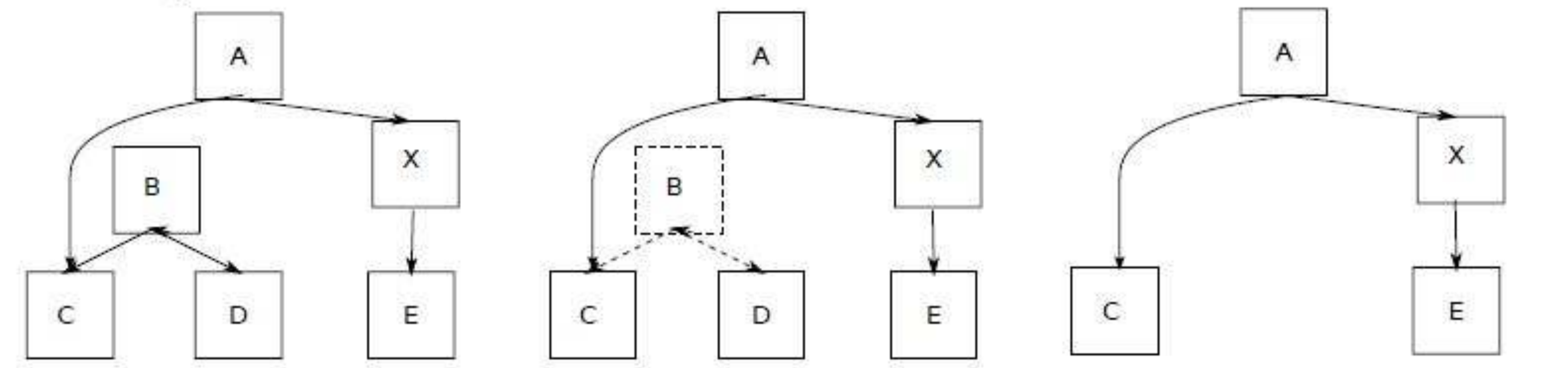

Adding a node:
A
B
C
D
E
X
(a) Original tree
(b) Initialize node X and connect E to X. Any readers in A and E are not affected.
(c) When X is completely initialized, connect X to A. Readers currently in E will have read the old version, while readers in A will pick up the new version of the tree.

Figure 2-39. Read-Copy-Update: inserting a node in the tree and then removing a branch-all without locks

Avoiding Locks: Read-Copy-Update (2 of 2)

Removing nodes:



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.

(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.

(f) Now we can safely remove B and D

Figure 2-39. Read-Copy-Update: inserting a node in the tree and then removing a branch-all without locks

Scheduling

Introduction to Scheduling Process Behavior

CPU bound vs. I/O bound processes

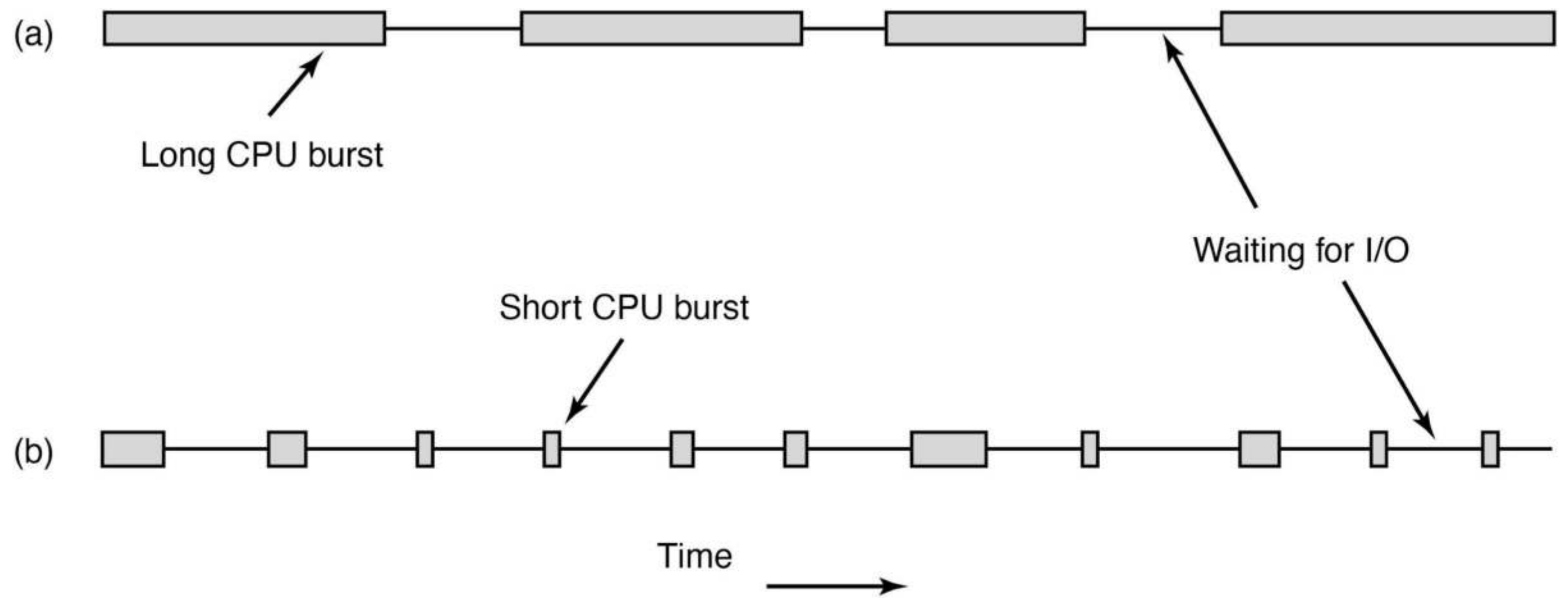
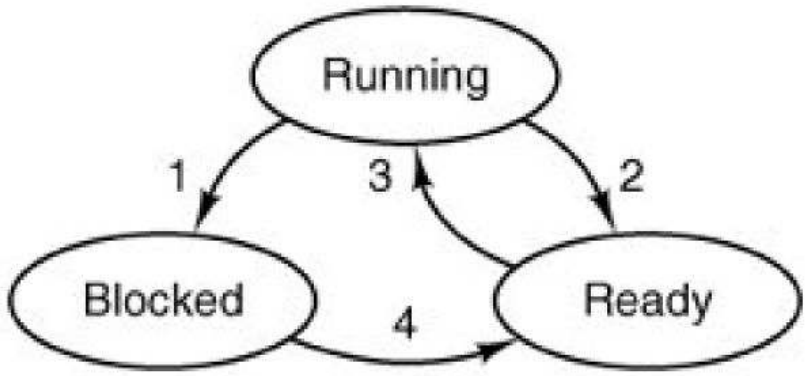


Figure 2-40. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CP U-bound process. (b) An I/O-bound process.

Process State Revisited



If more processes ready than CPUs available:

- **Scheduler** decides which process to run next
- Algorithm used by scheduler is called **scheduling algorithm**

When to schedule?

- ***Process exits***
- ***Process blocks on I/O, Semaphore, etc.***
- *When a new process is created*
- *When an interrupt occurs:*
  - *I/O, clock, syscall, etc.*

Preemptive vs **non-preemptive** scheduling?

Categories of Scheduling Algorithms

1. Batch.
2. Interactive.
3. Real time.

Scheduling Algorithm Goals (1 of 2)

Different goals for different systems

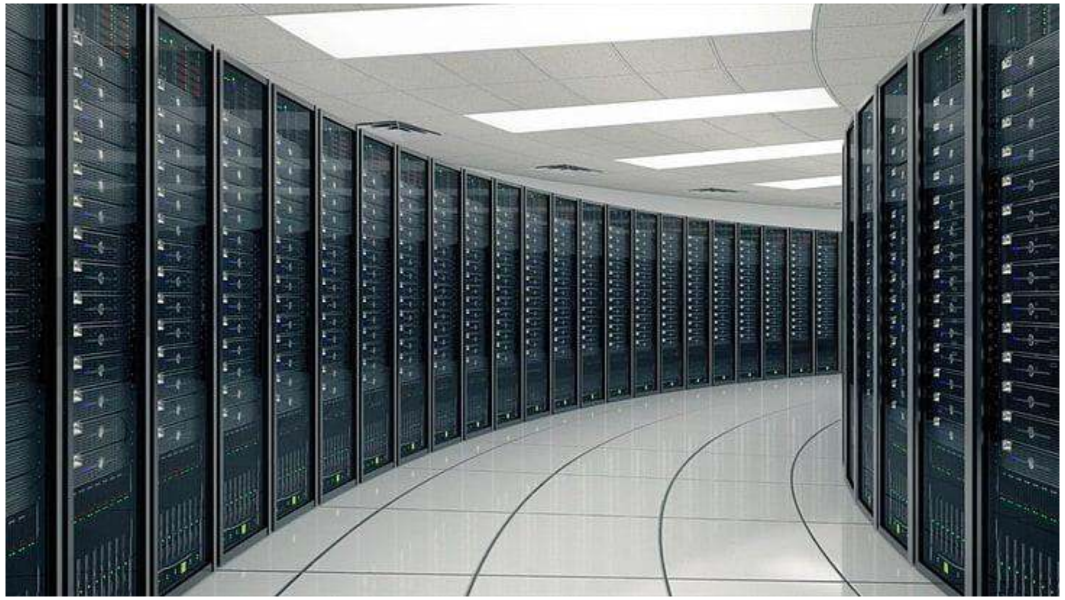
Batch.

Interactive.

Real time.

- All systems:
  - Fairness - giving each process a fair share of the CPU
  - Policy enforcement - seeing that stated policy is carried out
  - Balance - keeping all parts of the system busy

Batch Systems

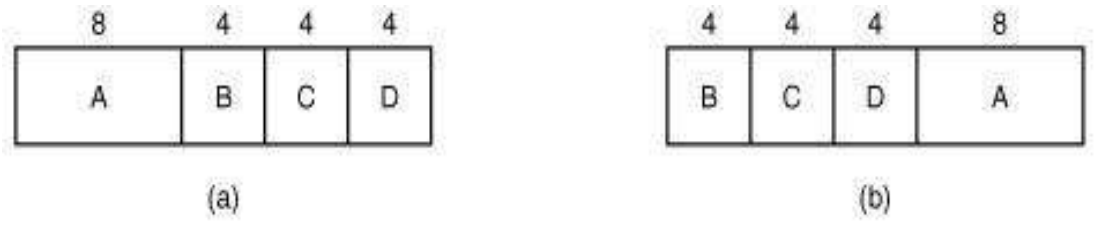


● **Throughput**	: maximize jobs per hour
● **Turnaround time**	: minimize time between submission and termination
● **CPU utilization**	: keeping the CPU busy all the time

First-Come First-Served

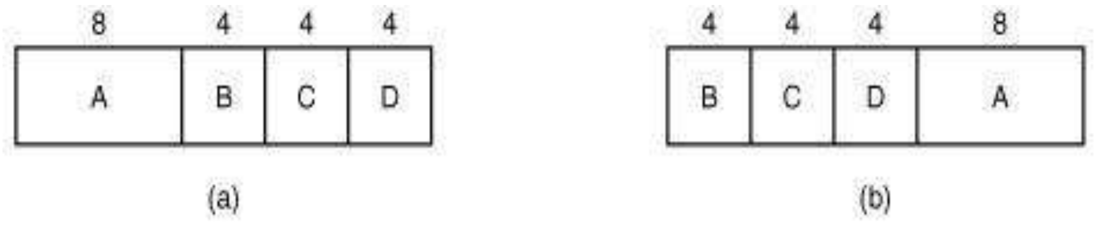
- Process jobs in order of their arrival
- Non-preemptive
- Single Process Queue
  - New jobs or blocking processes are added to the end of the queue
- “Convoy Effect” if only few CPU bound and many I/O bound processes

Shortest Job First



- Pick the job with the shortest run time
- Provably optimal:
  - Lowest turnaround time

Shortest Job First



- Pick the job with the shortest run time
- Provably optimal:
  - Lowest turnaround time
  - Only if all jobs are available simultaneously
  - If new jobs arrive, it may lead to starvation
- Runtimes have to be known in advance
- **Highest-response-ratio-next**
  - Improved version of Shortest Job First

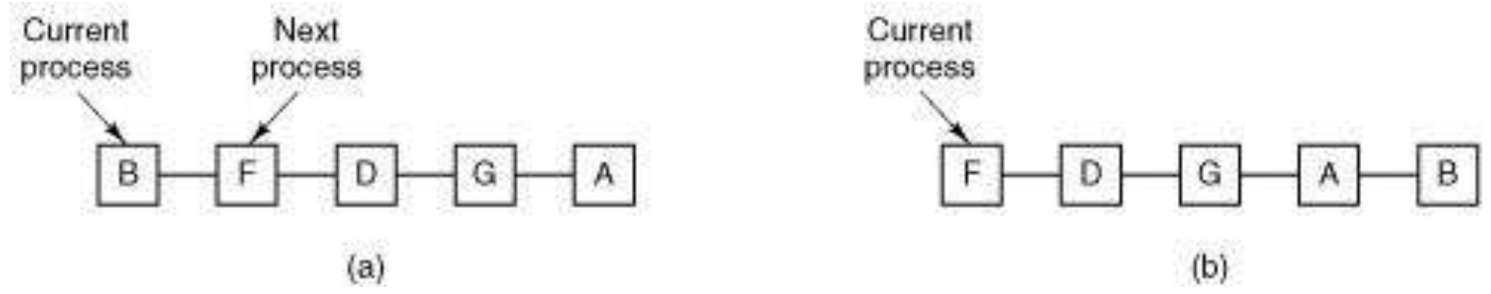
Interactive Systems

- **Response time:** respond to requests quickly
- **Proportionality:** meet users’ expectations

Apple MacIntosh (1984)

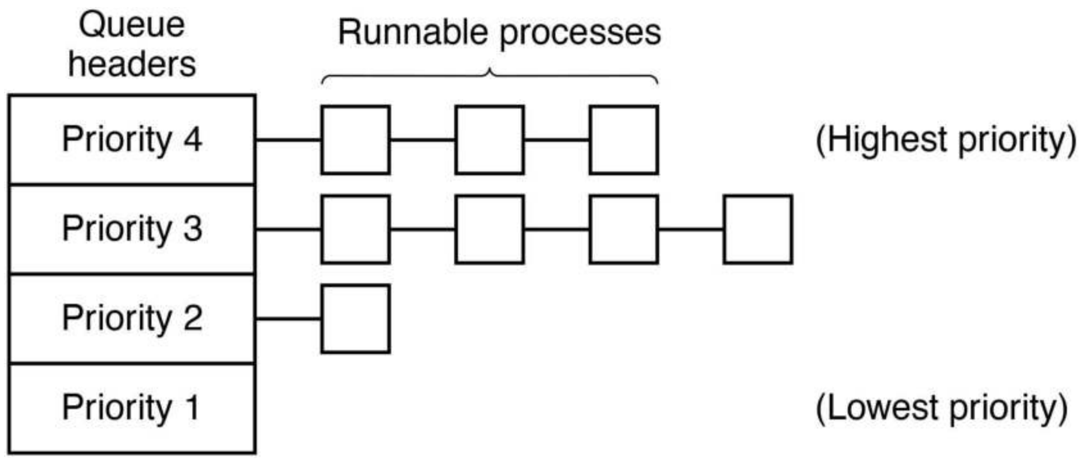
Round Robin Scheduling

- Preemptive scheduling algorithm
- Each process gets a **time slice** or **quantum**
- If process is still running at end of quantum it gets preempted end goes to end of ready queue
- Question: How big should the quantum be?   CPU utilization vs. response time



Priority Scheduling

Simplest, multiple queues



- Similar to round robin but several ready queues
- Next process is picked from queue with highest priority
- Static vs. dynamic priorities

What processes should have high priorities?

Priorities are complicated

and may not work out the way you think...

(remember priority inversion?)

Shortest Process Next

Problem: How to minimize response time for each priority queue?

Idea: Use shortest "job" first and try to best predict next running time

$\searrow$ whatever runs between the waits

Solution: Form weighted average of previous running times of process → **Aging**

$$T_{k+1} = a * T_{k-1} + (1 - a) * T_k$$

Easy to implement when a = 1/2

Guaranteed Scheduling

Idea: N processes running → each process gets 1/Nth of CPU time (also known as fair-share)

Solution:

- Calculate how much CPU time it might have gotten: Time since process creation divided by N
- Measure actual consumed CPU time and form ratio
- 0.5 → process running half the time it was entitled to
- 2.0 → process running twice as much as it was entitled to
- Pick process with the smallest ratio to run next
- How to incorporate priorities (See Linux' CFS)?
- Note: fair-share scheduling can be per-user/-process

Lottery Scheduling

- Processes get lottery tickets
- Whenever a scheduling decision has to be made the OS chooses a winning ticket randomly
- Processes can possess multiple tickets → Priorities
- Tickets can be traded between processes.
- Tickets are immediately available to newly created processes.

Policy vs mechanism

Important principle

Here: we may have a scheduling algorithm, but parameters to be filled in by user (process)

For instance, to give some child processes higher priority than others

Real Time Systems

- **Meeting deadlines:** avoid losing data
- **Predictability:** avoid quality degradation in multimedia systems

Real Time Systems

- Systems where timing plays essential role
- **Soft real time vs. Hard real time**
- Can consist of **periodic** and **aperiodic** tasks
- Schedules can be **static** or **dynamic**
- System with periodic tasks is **schedulable** when:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Real Time Systems

Consider three jobs:

	Period	Req. CPU time
P1	100 ms	50 ms
P2	200 ms	30 ms
P3	500 ms	100 ms

Is this system schedulable?

Scheduling Algorithm Goals (2 of 2)

- Interactive systems
  - Response time - respond to requests quickly
  - Proportionality - meet users' expectations
- Real-time systems
  - Meeting deadlines - avoid losing data
  - Predictability - avoid quality degradation in multimedia systems

Scheduling in Batch Systems

- First-Come First-Served
- Shortest Job First
- Shortest Remaining Time Next

Shortest Job First

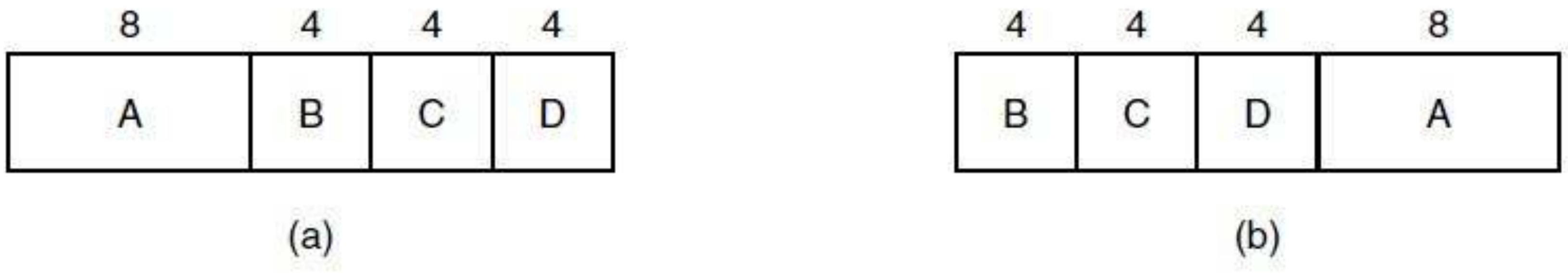


Figure 2-42. An example of shortest job first scheduling. (a) Running four jobs in the original order. (b) Running them in shortest job first order.

Scheduling in Interactive Systems

- Round-Robin Scheduling
- Priority Scheduling
- Multiple Queues
- Shortest Process Next
- Guaranteed Scheduling
- Lottery Scheduling
- Fair-Share Scheduling

Round-Robin Scheduling

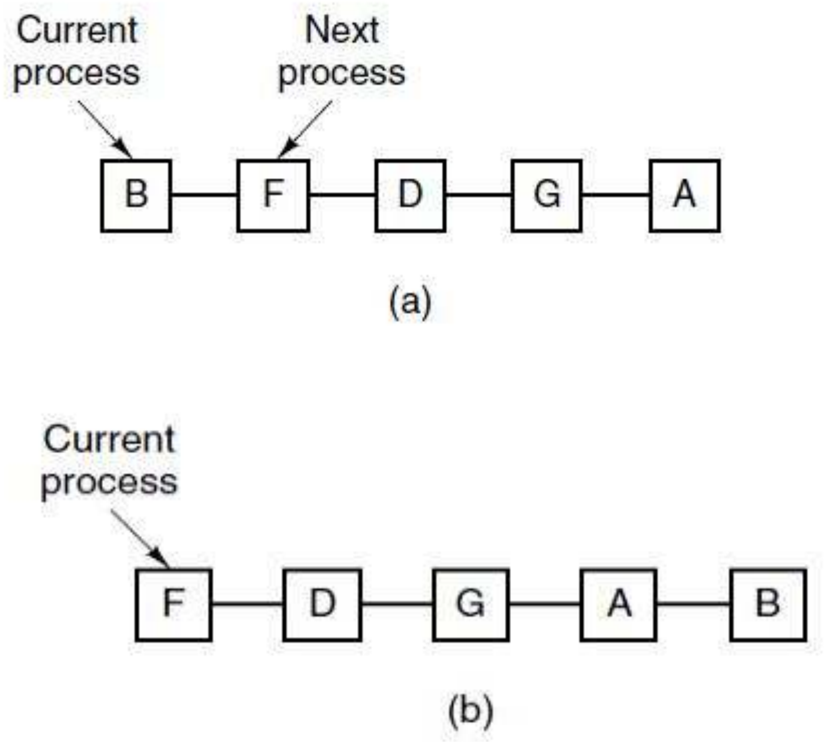


Figure 2-43. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after **B** uses up its quantum.

Priority Scheduling

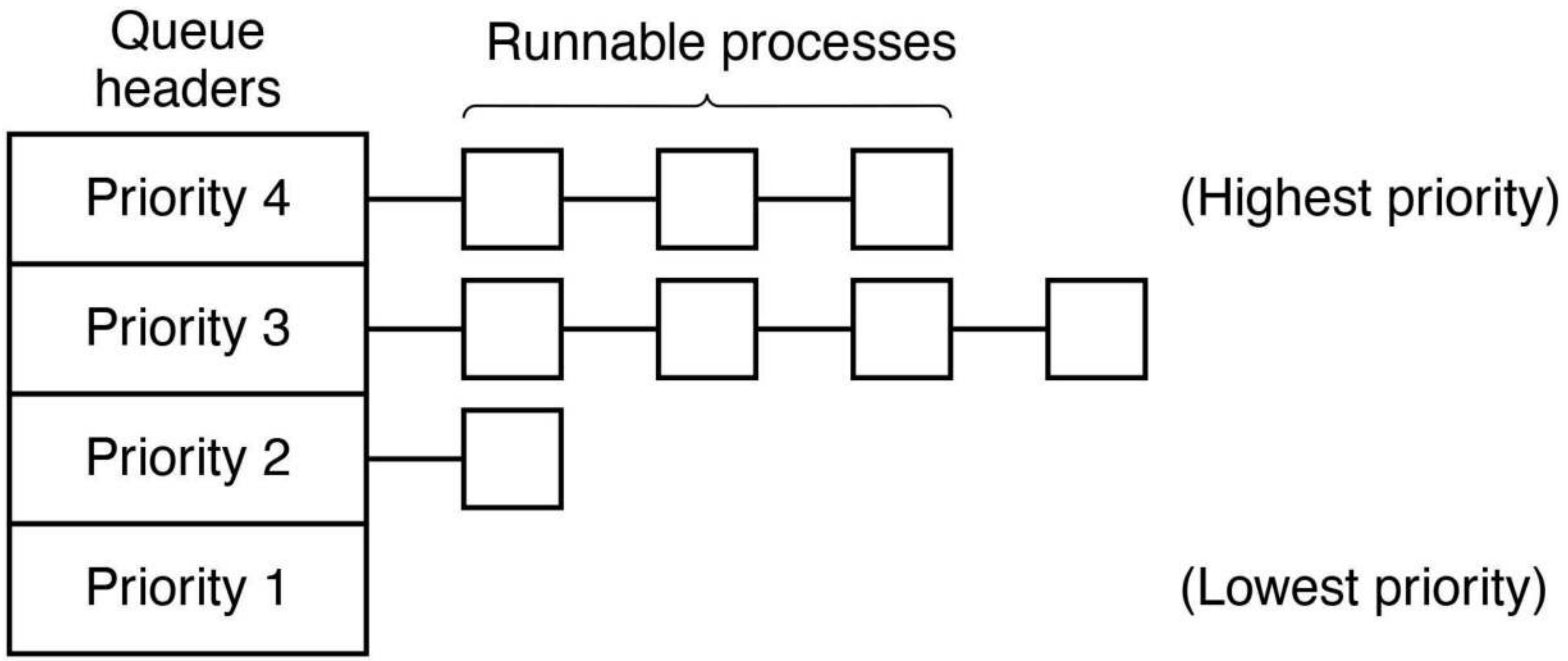


Figure 2-44. A scheduling algorithm with four priority classes.

Scheduling in Real-Time Systems

- Time plays an essential role
- Categories
  - Hard real time
  - Soft real time
  - Periodic or aperiodic
- Schedulable satisfies

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Thread Scheduling (1 of 2)

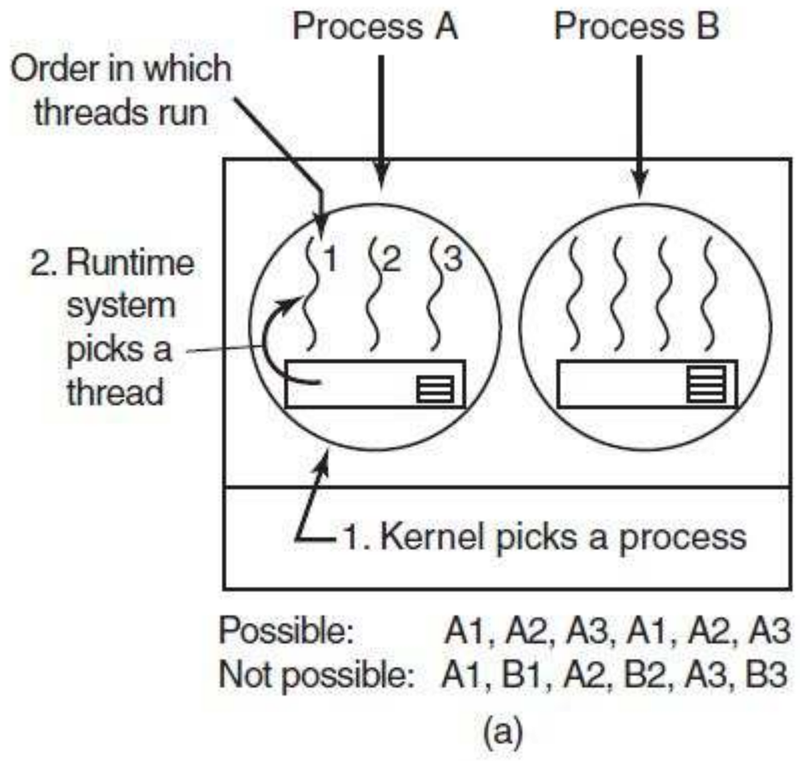

Process A
Process B
Order in which threads run
1
2
3
2. Runtime system picks a thread
1. Kernel picks a process
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3
(a)

Figure 2-45. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Thread Scheduling (2 of 2)

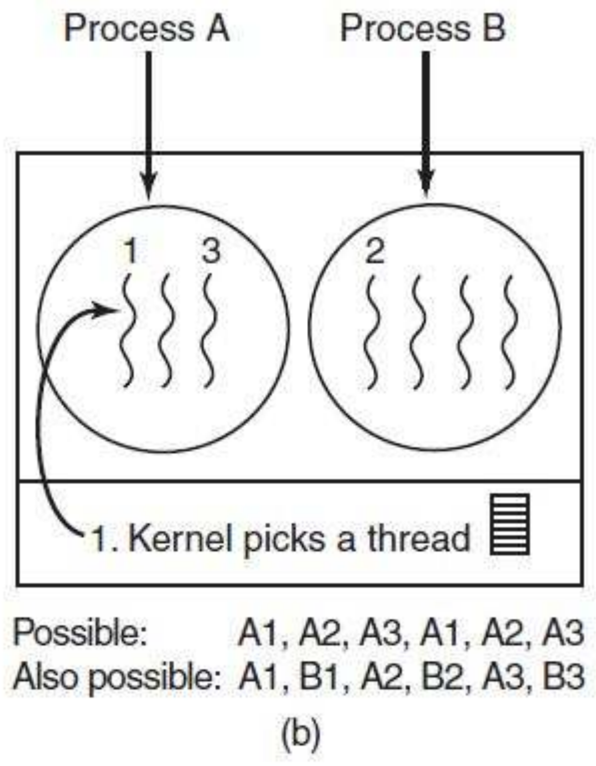

Process A
Process B
1
3
2
1. Kernel picks a thread
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3
(b)

Figure 2-45. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Chapter 2 Processes and Threads -- Summary

PROCESSES

The Process Model

Process Creation & Termination

Process Hierarchies

Process States

Implementation of Processes

Modeling Multiprogramming

THREADS

The Classical Thread Model

POSIX Threads

User / Kernel Threads

EVENT-DRIVEN SERVERS

SYNCHRONIZATION AND IPC

Race Conditions, Critical Regions, Mutual Exclusion with Busy Waiting, Sleep and Wakeup

Semaphores, Mutexes, Monitors, Message Passing

9 Barriers

Priority Inversion

Read-Copy-Update

SCHEDULING

Batch Systems

Interactive Systems

Real-Time Systems

Policy Versus Mechanism

Thread Scheduling

Copyright