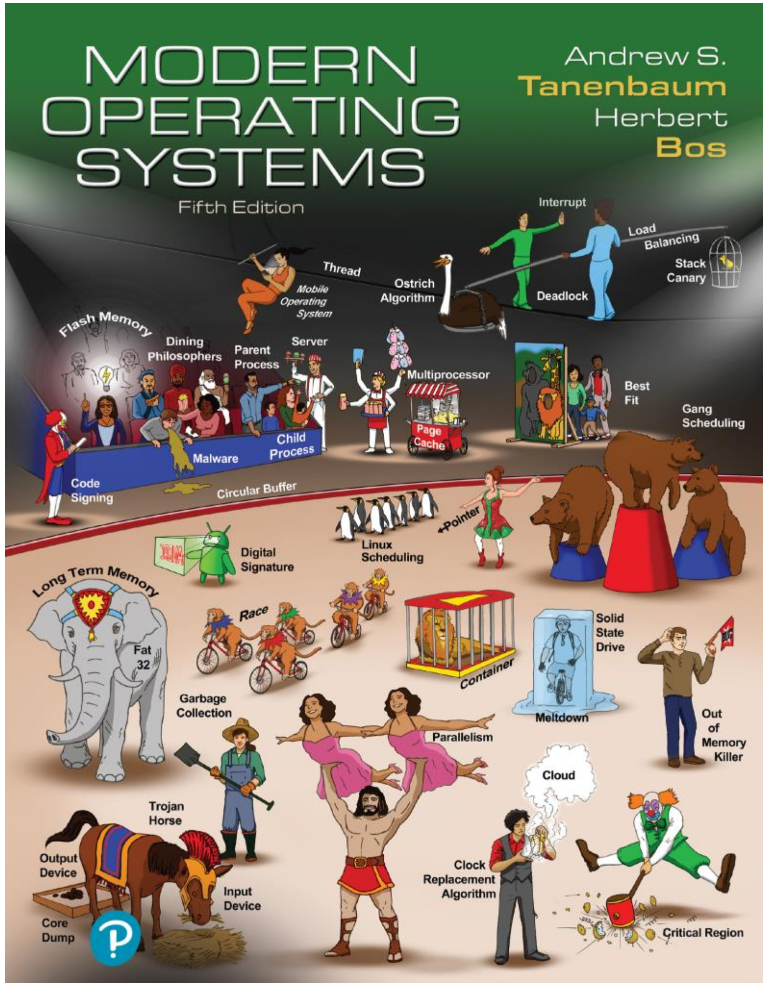
Modern Operating Systems

Fifth Edition



Chapter 2

Synchronization and Inter-Process Communication

Pearson

Synchronization and Inter-Process Communication (IPC)

- Why?
- Processes need some way to **communicate:**
  - To share data throughout the execution
- No explicit cross-process sharing:
  - $\rightarrow$ 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

Producer

```
while (true) {
      /* produce an item in next produced */


      while (counter == BUFFER_SIZE) ;

            /* do nothing */

      buffer[in] = next_produced;

      in = (in + 1) % BUFFER_SIZE;

      counter++;

}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
             counter--;
        /* consume the item in next consumed
*/
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Critical Regions

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
  - No two processes may be simultaneously in their critical regions
  - No assumptions may be made about speeds or nr. of CPUs
  - No process running outside its critical region may block others
  - 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

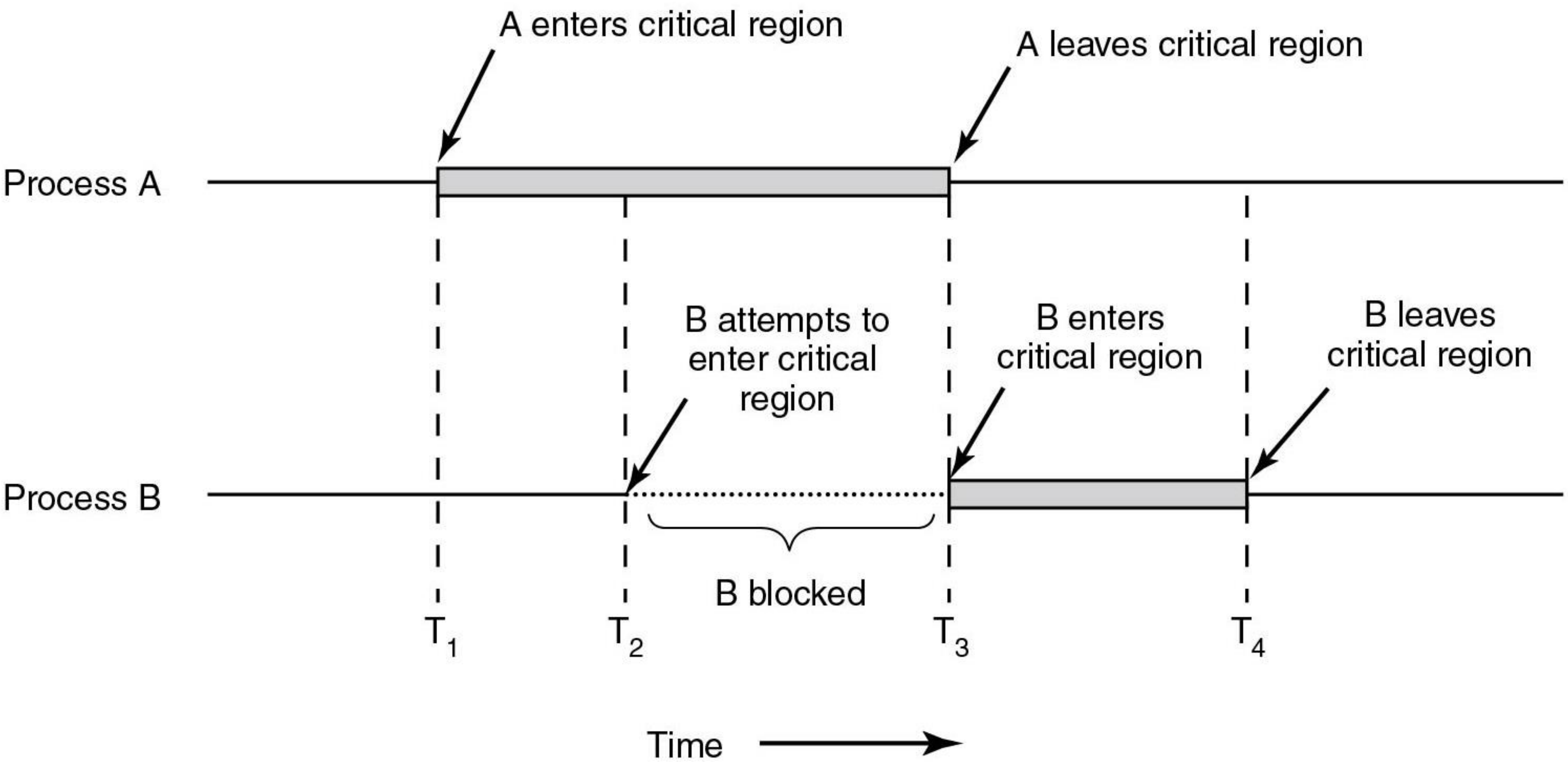


Figure 2.22 Mutual exclusion using critical regions.

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm for Process P_i

```
do {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn = = j);
               critical section
     flag[i] = false;
               remainder section
 } while (true);
```

Mutual Exclusion with Busy Waiting: The TSL Instruction

- 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
```

Pearson

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;

while (true) {
       while (in == out)

              ; /* do nothing */
       next_consumed = buffer[out];

       out = (out + 1) % BUFFER_SIZE;


       /* consume the item in next consumed */

}
```

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(){
 set own state to BLOCKED;
 give CPU to scheduler;
}
```

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

Producer-Consumer (1 of 4)

```
#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 (2 of 4)

Producer `sleeps` **when buffer is** ***full***

```
#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 (3 of 4)

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 (4 of 4)

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

```
#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);
 }
}
```

Pearson

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

```
do {
  acquire lock
    critical section
  release lock
    remainder section
} while (true);
```

Mutexes in Pthreads

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

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.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal()` operation**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-

```
typedef struct{
int value;
struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Full Example on Linux

```c
#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;
}
```

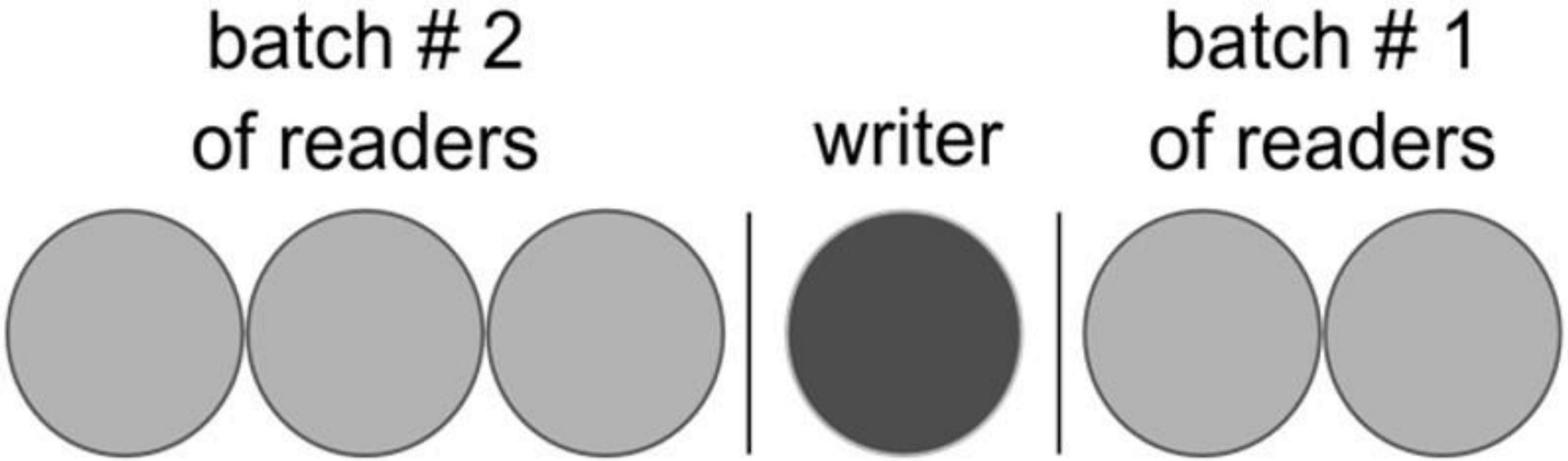
```c
/* Main entry point. */
int main(int argc, char **argv)
{
 pthread_t producer_tid, consumer_tid;

 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;
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

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?



Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
     wait(rw_mutex);
       ...
     /* writing is performed */
       ...
   signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

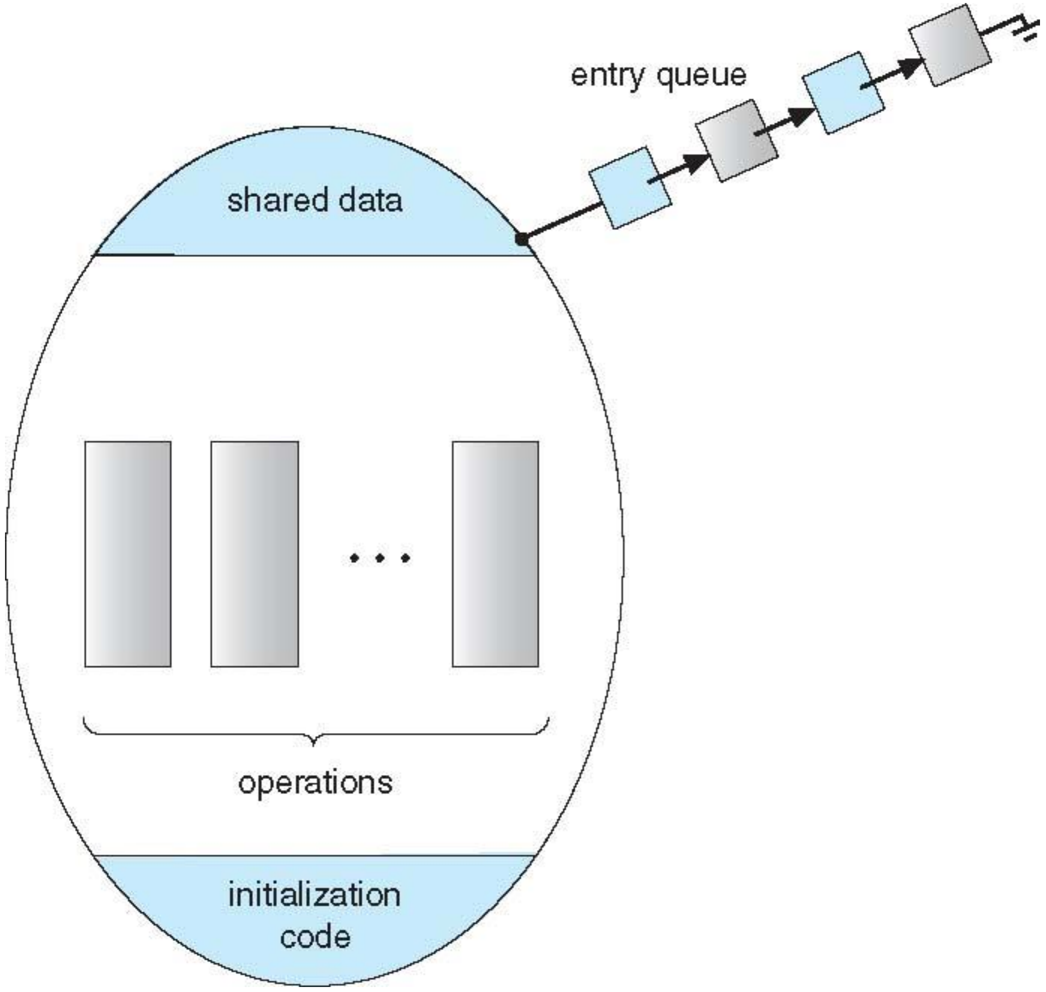
- The structure of a reader process

```
do {
       wait(mutex);
       read_count++;
       if (read_count == 1)
       wait(rw_mutex);
    signal(mutex);
        ...
       /* reading is performed */
        ...
    wait(mutex);
       read count--;
       if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Monitors

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

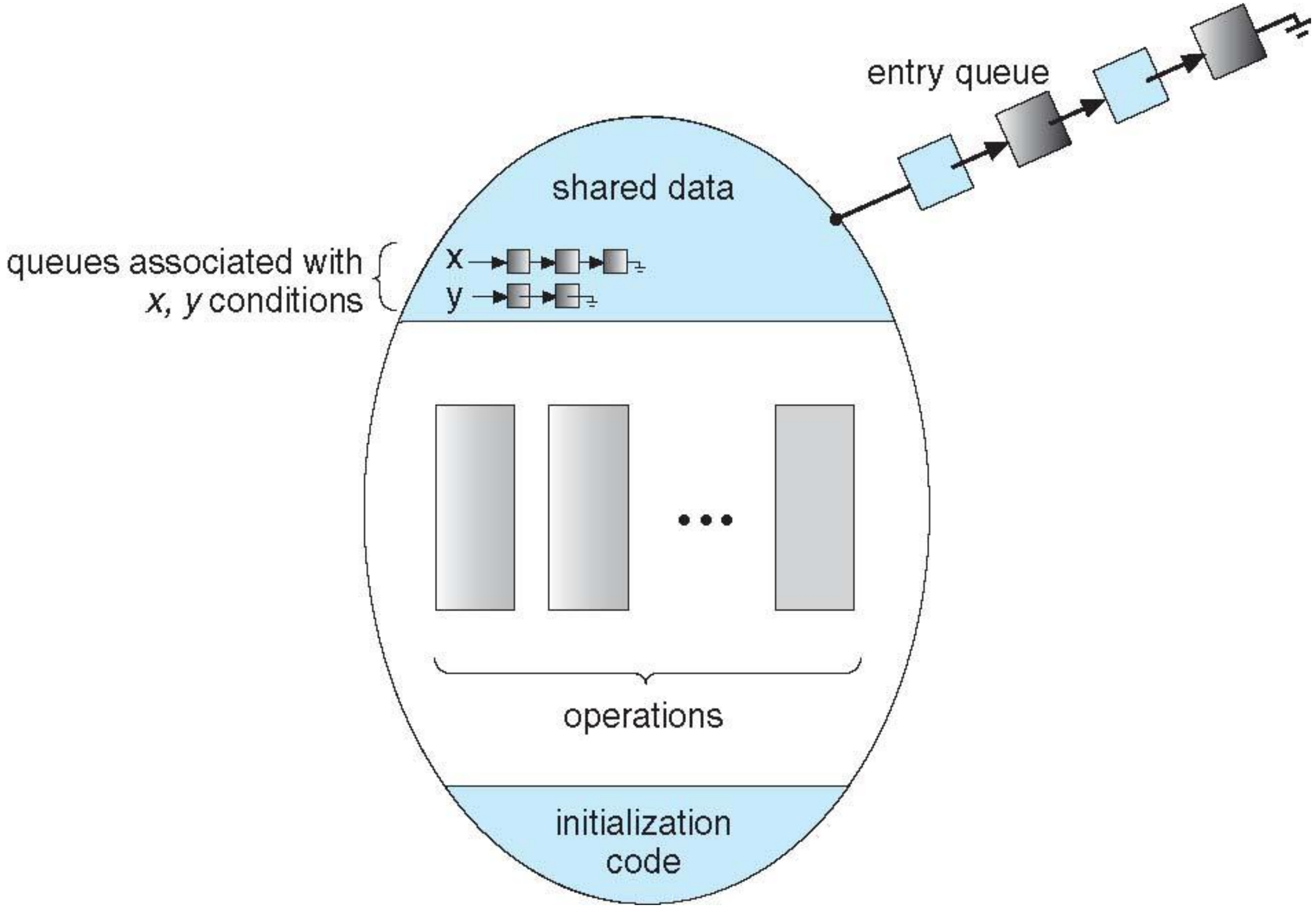
Schematic view of a Monitor



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Monitors: Producer-Consumer (1 of 5)

<pre>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);
 }
}</pre>
	<pre>void producer(){
	 int item;
	 while(TRUE){
	  item = produce_item();
	  ProdCons.enter(item);
	 }
	}
	
	void consumer(){
	 int item;
	 while(TRUE){
	  ProdCons.remove(&item);
	  consume_item(item);
	 }
	}</pre>

Monitors: Producer-Consumer (2 of 5)

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 (3 of 5)

`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 (4 of 5)

`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 (5 of 5)

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);
 }
}
```

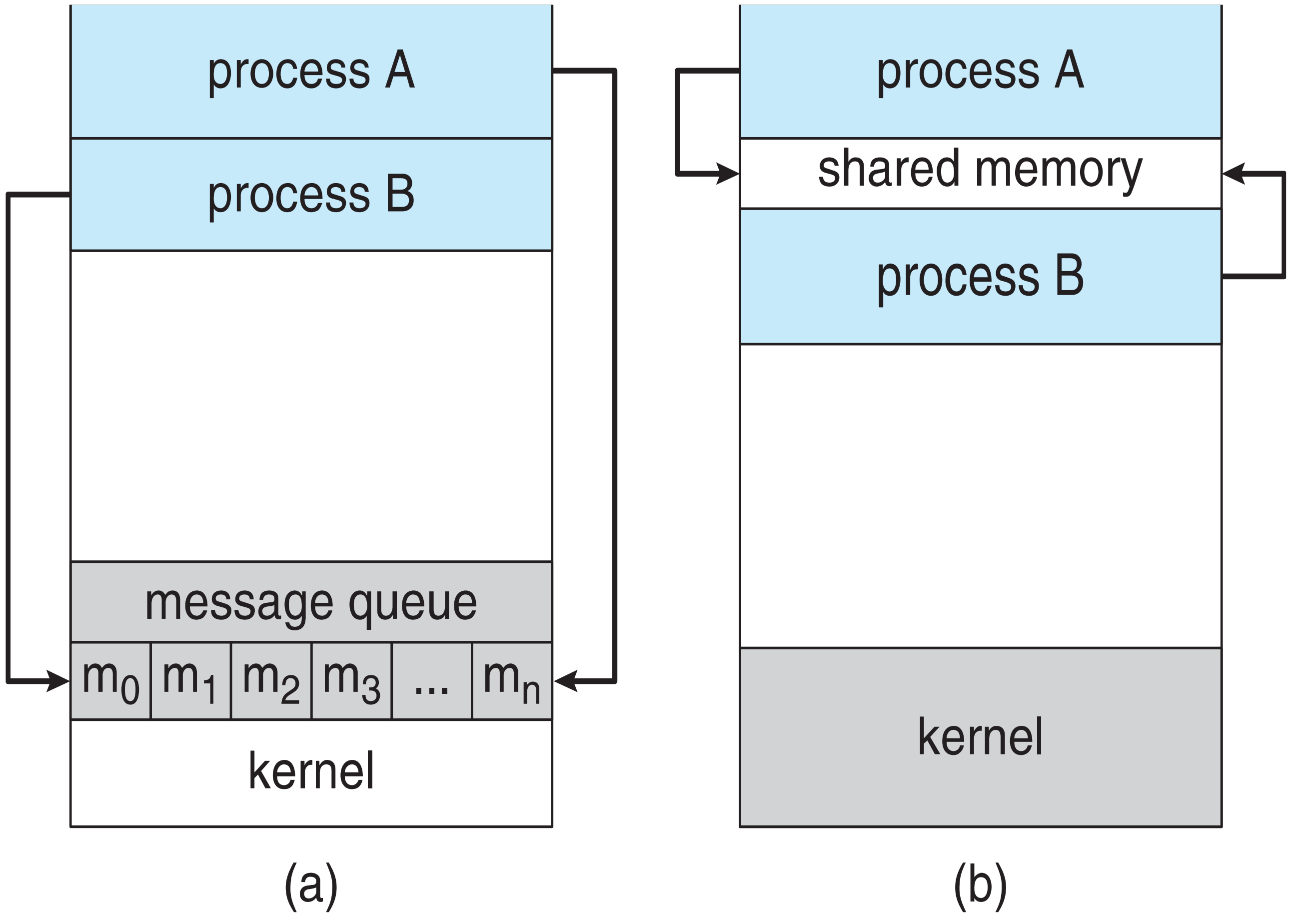
```
void producer(){
 int item;
 while(TRUE){
  item = produce_item();
  ProdCons.enter(item);
 }
}

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

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```
  - Also used to open an existing segment to share it
  - Set the size of the object
    ```
    ftruncate(shm fd, 4096);
    ```
  - Now the process could write to the shared memory
    ```
    sprintf(shared memory, "Writing to shared
    memory");
    ```

IPC POSIX Producer

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

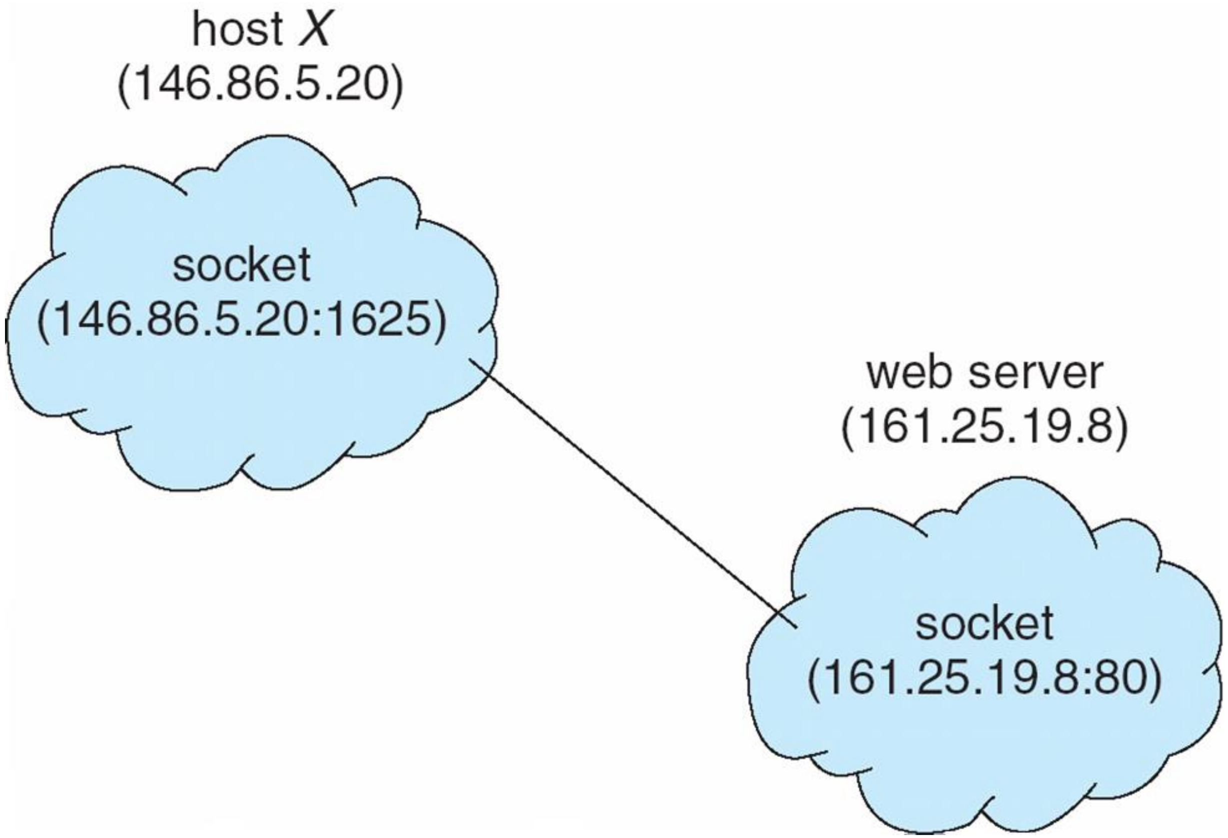
IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Sockets

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication



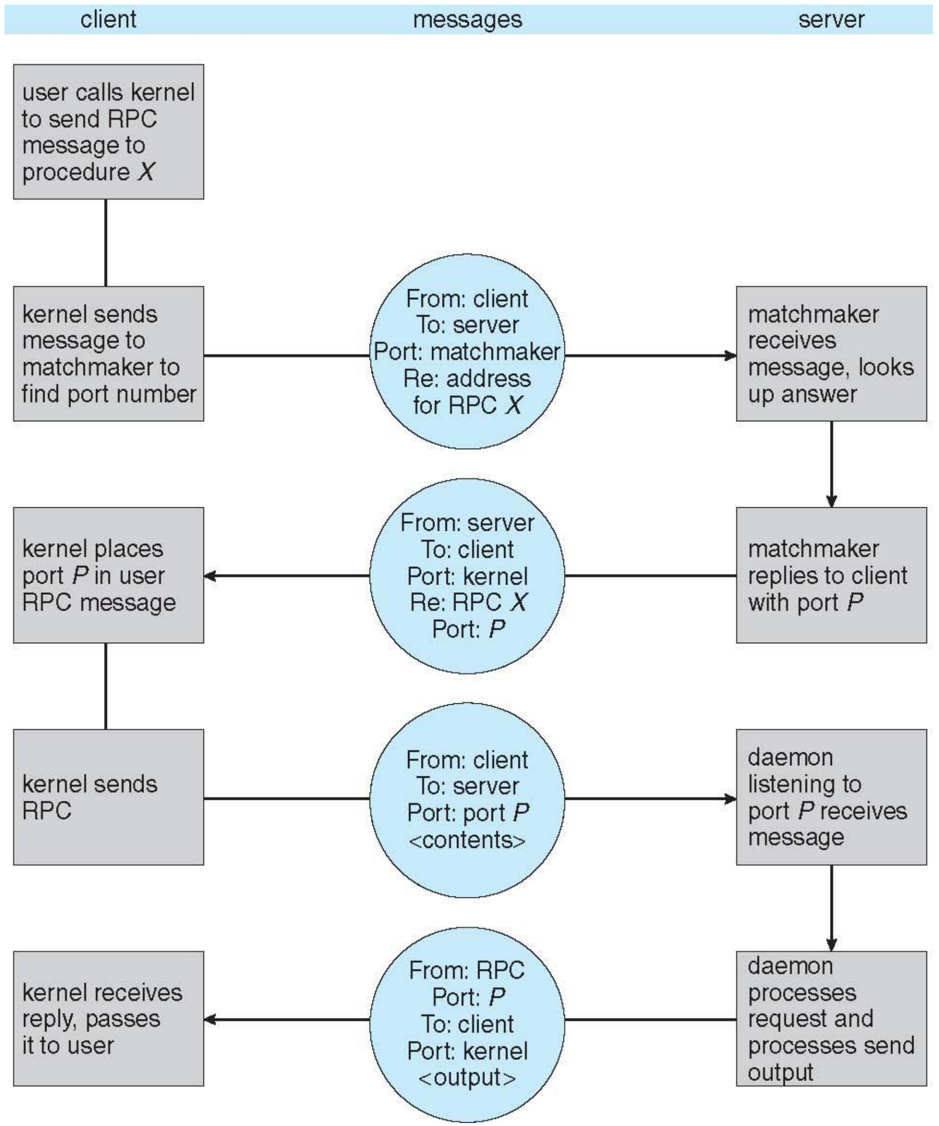
Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Messages can be delivered ***exactly once*** rather than ***at most once***
- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC

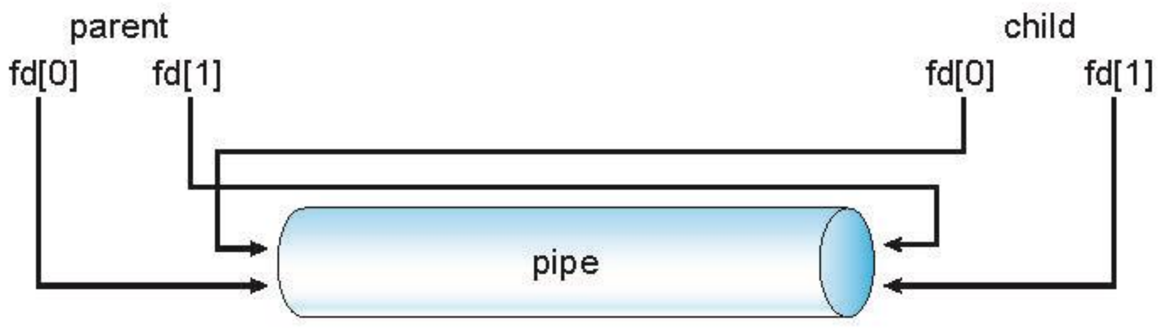


Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**
- See Unix and Windows code samples in textbook

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Priority Inversion

Consider the Mars Pathfinder

- It had three subsystems
  - A data-distribution system (**High** Prio)
  - A Communication system (**Med** Prio)
  - 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 semaphore

Question: What can happen?

NASA/JPL-Caltech

Priority Inversion

Consider the Mars Pathfinder

- It had three subsystems
  - A data-distribution system (**High** Prio)
  - A Communication system (**Med** Prio)
  - A meteorological data gathering (**Low** Prio)

NASA/JPL-Caltech

There was also a common hardware subsystem:

The data bus used by Task 1 and 3.

It is protected by a semaphore

Question: What can happen?

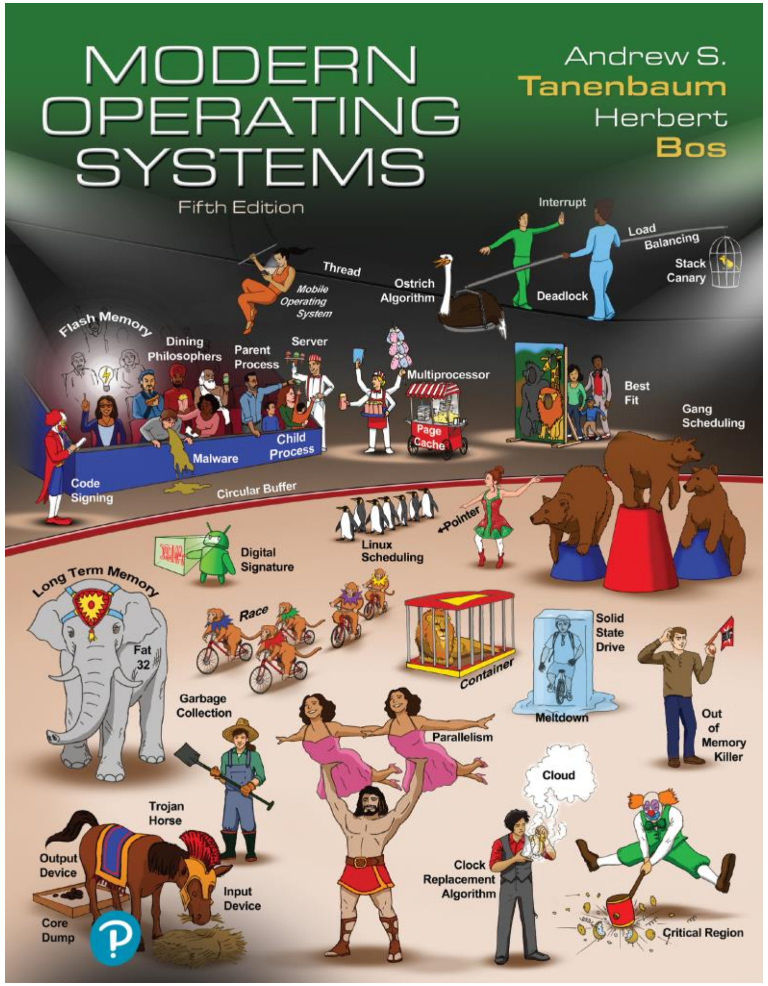
Imagine the following scenario: (1) Low Prio task takes mutex, (2) gets interrupted by Med Prio task, (3) which is interrupted by High Prio task (which blocks because Low Prio task holds lock) Medium Prio task runs even though there is a High Prio task to run (priority is inverted)

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

Modern Operating Systems

Fifth Edition



Chapter 2

Synchronization and Inter-Process Communication - End