

Lecture 4: Inter-process Communication and Synchronization



Contents

- Cooperating processes
- Where is the problem?
- Race Condition and Critical Section
- Possible Solutions
- Semaphores
- Deadlocks
- Classical Synchronization Tasks
- Monitors
- Examples

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
- 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

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- IPC Implementation
 - Message system – processes communicate with each other without resorting to shared variables
 - Shared memory – not available for distributed systems
- Message system facility provides two operations:
 - **send**(*message*) – message size fixed or variable
 - **receive**(*message*)
- If P and Q wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., hardware bus, network)
 - logical (e.g., logical properties)

Direct & Indirect Communication

■ Direct Communication

- Processes must name each other explicitly:
 - ▶ **send** ($P, message$) – send a message to process P
 - ▶ **receive**($Q, message$) – receive a message from process Q
- Properties of communication link
 - ▶ Links are established automatically
 - ▶ A link is associated with exactly one pair of communicating processes
 - ▶ Between each pair there exists exactly one link
 - ▶ The link may be unidirectional, but is usually bi-directional

■ Indirect Communication

- Messages are directed and received from *mailboxes* (also referred to as *ports*)
 - ▶ Each mailbox has a unique *id* and is created by the kernel on request
 - ▶ Processes can communicate only if they share a mailbox
- Properties of communication link
 - ▶ Link established only if processes share a common mailbox
 - ▶ A link may be associated with many processes
 - ▶ Each pair of processes may share several communication links
 - ▶ Link may be unidirectional or bi-directional

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send:** the sender blocks until the message is received by the other party
 - **Blocking receive:** the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send:** the sender sends the message and continues executing
 - **Non-blocking receive:** the receiver gets either a valid message or a null message (when nothing has been sent to the receiver)
- Often a combination:
 - Non-blocking send and blocking receive

Producer & Consumer Problem

Message passing:

```
#define BUF_SZ = 20 /* depends on the mailbox size */
typedef struct { ... } item_t;
```

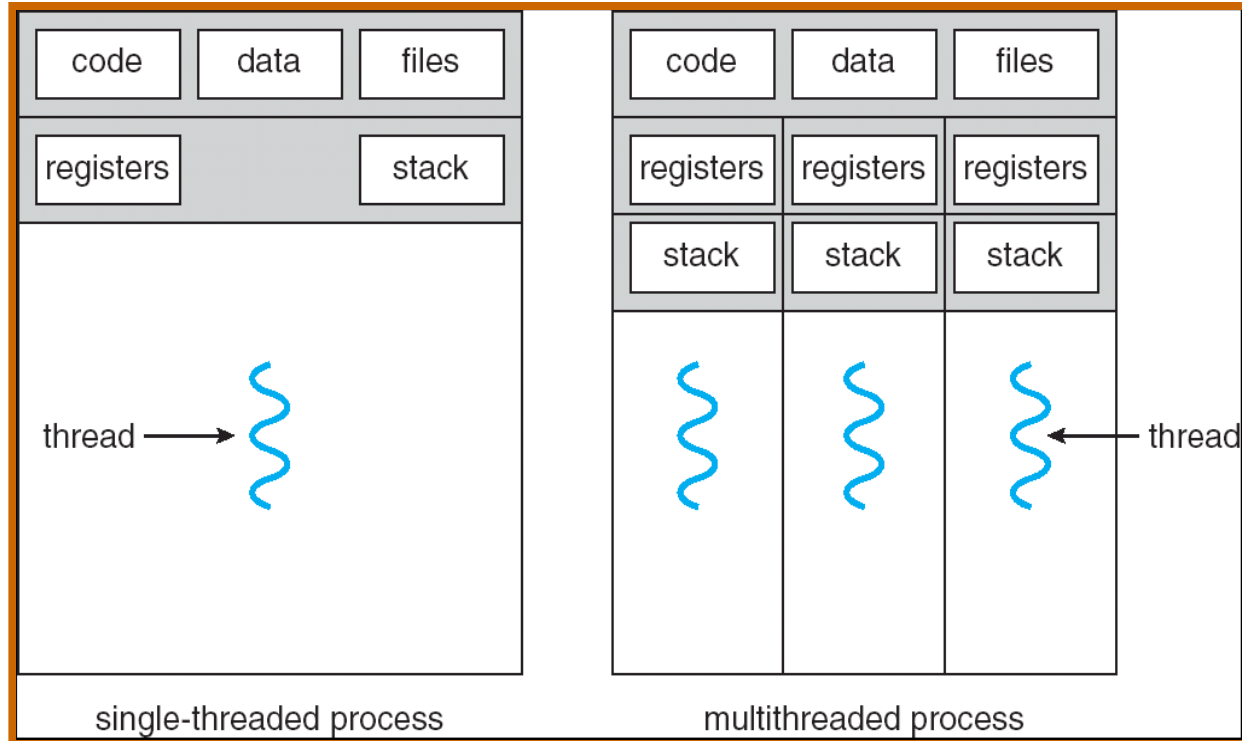
Producer:

```
void producer() {
    item_t item;
    message m;
    while (1) {
        /* Generate new item */
        receive(consumer, &m);
        /* free slot */
        build_msg(&m, item);
        send(consumer, &m);
    }
}
```

Consumer:

```
void consumer() {
    item_t item;
    message m;
    for (i=0; i<BUF_SZ; i++)
        send(producer, &m);
    while (1) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m);
        /* Process nextConsumed */
    }
}
```

Single and Multithreaded Processes



■ Benefits of Multi-threading

- Responsiveness
- Easy Resource Sharing
- Economy
- Utilization of Multi-processor Architectures

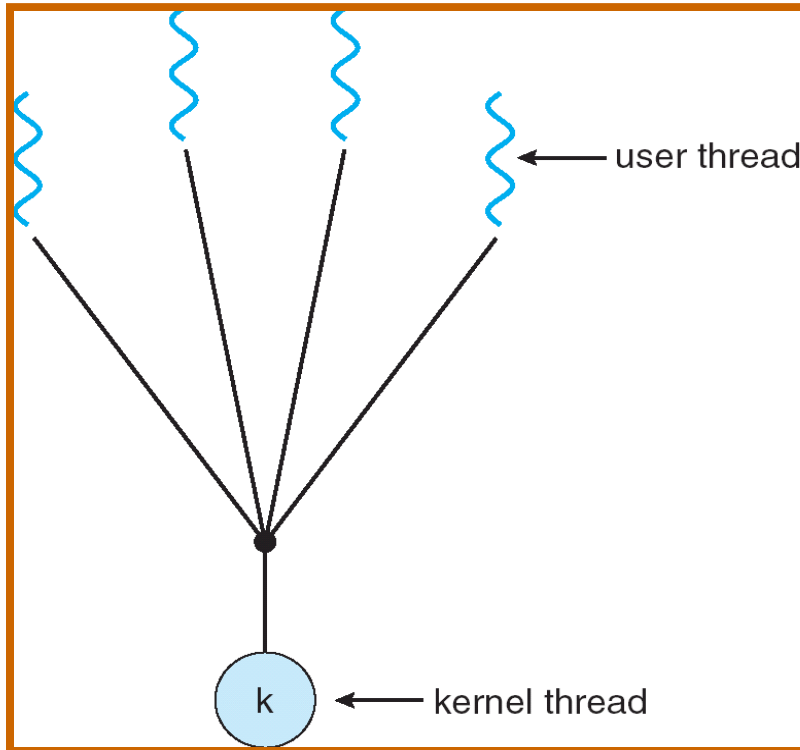
Data sharing with threads

■ Processes and threads

- process: unit that contains resources (memory, open files, user rights)
- thread: unit for scheduling
- One process can have more threads

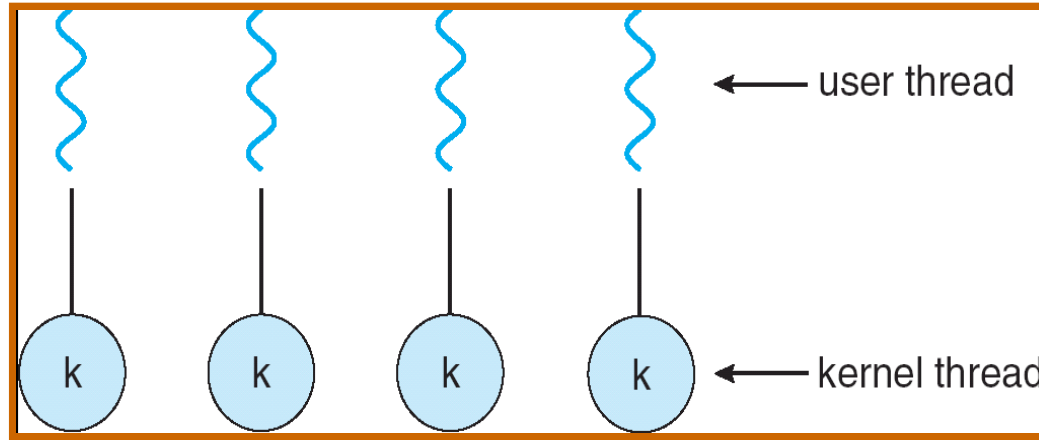
Program code:	process
Local and working variables:	thread
Global data:	process
System resources:	process
Stack:	thread
Memory management:	process
PC – program counter:	thread
CPU register:	thread
Scheduling state:	thread
User identification and rights:	process

User threads - Many-to-One Model



- Thread management done by user-level threads library
- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads
- Only old operating systems without thread support

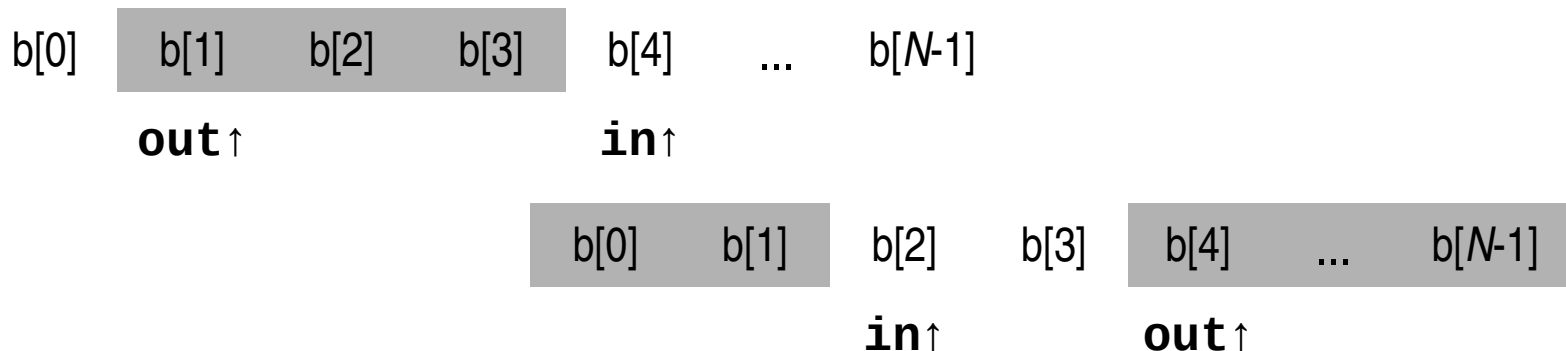
One-to-one Model



- Supported by the Kernel
- Better scheduling – one waiting thread cannot block other threads from the same process
- Examples: Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X

Example

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the producer-consumer problem:
 - We have a limited size buffer (N items). The *producer* puts data into the buffer and the *consumer* takes data from the buffer
 - We can have an integer **count** that keeps track of the number of occupied buffer entries. Initially, count is set to 0.
 - It is incremented by the producer after it inserts a new item in the buffer and is decremented by the consumer after it consumes a buffer item



Producer & Consumer Problem

Shared data:

```
#define BUF_SZ = 20
typedef struct { ... } item;
item buffer[BUF_SZ];
int count = 0;
```

Producer:

```
void producer() {
    int in = 0;
    item nextProduced;
    while (1) {
        /* Generate new item */
        while (count == BUF_SZ) ;
        /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUF_SZ;
        count++ ;
    }
}
```

Consumer:

```
void consumer() {
    int out = 0;
    item nextConsumed;
    while (1) {
        while (count == 0) ;
        /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUF_SZ;
        count-- ;
        /* Process nextConsumed */
    }
}
```

- This is a naive solution that does **not work**

Race Condition

- `count++` could be implemented as

```
reg1 = count
reg1 = reg1 + 1
count = reg1
```

- `count--` could be implemented as

```
reg2 = count
reg2 = reg2 - 1
count = reg2
```

- Consider this execution interleaving with “`count = 5`” initially:

S0: producer executes	<code>reg1 = count</code>	{reg1 = 5}
S1: producer executes	<code>reg1 = reg1 + 1</code>	{reg1 = 6}
S2: consumer executes	<code>reg2 = count</code>	{reg2 = 5}
S3: consumer executes	<code>reg2 = reg2 - 1</code>	{reg2 = 4}
S4: consumer executes	<code>count = reg2</code>	{count = 4}
S5: producer executes	<code>count = reg1</code>	{count = 6}

- Variable `count` represents a **shared resource**

Critical-Section Problem

What is a **CRITICAL SECTION**?

Part of the code when one process tries to access a **particular** resource shared with another process. We speak about a **critical section related to that resource**.

1. **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections related to that resource
2. **Progress** – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then one of the processes that wants to enter the critical section should be allowed as soon as possible
3. **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the N processes

Critical Section Solution

Critical section has two basic operation: `enter_CS` and `leave_CS`

Possible implementation of this operation:

- Only SW at application layer
- Hardware support for operations
- SW solution with support of OS

SW solution for 2 processes

- Have a variable *turn* whose value indicates which process may enter the critical section. If *turn* == 0 then P_0 can enter, if *turn* == 1 then P_1 can.

```
 $P_0$   
while(TRUE) {  
    while(turn!=0); /* wait */  
    critical_section();  
    turn = 1;  
    noncritical_section();  
}
```

```
 $P_1$   
while(TRUE) {  
    while(turn!=1); /* wait */  
    critical_section();  
    turn = 0;  
    noncritical_section();  
}
```

- However:

- Suppose that P_0 finishes its critical section quickly and sets *turn* = 1; both processes are in their non-critical parts. P_0 is quick also in its non-critical part and wants to enter the critical section. As *turn* == 1, it will have to wait even though the critical section is free.
 - ▶ The requirement #2 (Progression) is violated
 - ▶ Moreover, the behaviour inadmissibly depends on the relative speed of the processes

Peterson's Solution

- Two processes solution from 1981
- Assume that the LOAD and STORE 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 ($i = 0, 1$)

```
    j = 1-i;
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
        // CRITICAL SECTION
    flag[i] = FALSE;
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Dangerous to disable interrupts at application level
 - ▶ Disabling interrupts is usually unavailable in CPU user mode
 - Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this are not broadly scalable
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic = non-interruptible**
 - Test memory word and set value
 - Swap contents of two memory words
 - For computers with 2 or more cores – real problem of synchronization
 - ▶ Locking bus
 - ▶ Cache snooping – synchronization of L1 and L2 caches

TestAndSet Instruction

■ Semantics:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

■ Shared boolean variable lock, initialized to false.

■ Solution:

```
while (TestAndSet (&lock )) ; // active waiting
// critical section
lock = FALSE;
// remainder section
```

Swap Instruction

■ Semantics:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

■ Shared Boolean variable lock initialized to FALSE; each process has a local Boolean variable key.

■ Solution:

```
    key = TRUE;
while (key == TRUE) { // waiting
    Swap (&lock, &key );
}
    // critical section
lock = FALSE;
    // remainder section
```

Synchronization without active waiting

■ Active waiting waste CPU

- Can lead to failure if process with high priority is actively waiting for process with low priority

■ Solution: blocking by system functions

- `sleep()` the process is inactive
- `wakeup(process)` wake up process after leaving critical section

```
void producer() {
    while (1) {
        if (count == BUFFER_SIZE) sleep();           // if there is no space wait - sleep
        buffer[in] = nextProduced; in = (in + 1) % BUFFER_SIZE;
        count++;
        if (count == 1) wakeup(consumer);           // if there is something to consume
    }
}

void consumer() {
    while (1) {
        if (count == 0) sleep();                     // cannot do anything – wait - sleep
        nextConsumed = buffer[out]; out = (out + 1) % BUFFER_SIZE;
        count--;
        if (count == BUFFER_SIZE-1) wakeup(producer); // now there is space for new product
    }
}
```

Synchronization without active waiting (2)

■ Presented code is not good solution:

- Critical section for shared variable `count` and function `sleep()` is not solved
 - ▶ Consumer read `count == 0` and then Producer is switch before it call `sleep()` function
 - ▶ Producer insert new product into buffer and try to wake up Consumer because `count == 1`. But Consumer is not sleeping!
 - ▶ Producer is switched to Consumer that continues in program by calling `sleep()` function
 - ▶ When producer fill the buffer it call function `sleep()` – both processes are sleeping!

■ Better solution: Semaphores

Semaphore

■ Synchronization tool that does not require busy waiting

- Busy waiting wastes CPU time

■ Semaphore S – system object

- With each semaphore there is an associated waiting queue. Each entry in waiting queue has two data items:
 - ▶ value (of type integer)
 - ▶ pointer to next record in the list
- Two standard operations modify S : wait() and signal()

```
wait(S)    {
            value--;
            if (value < 0) {
                add caller to waiting queue
                block(P);    }
            }

signal(S)  {
            value++;
            if (value <= 0) {
                remove caller from the waiting queue
                wakeup(P);   }
            }
```


Semaphore as General Synchronization Tool

- **Counting** semaphore – the integer value can range over an unrestricted domain
- **Binary** semaphore – the integer value can be only 0 or 1
 - Also known as **mutex lock**
- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion (mutex)

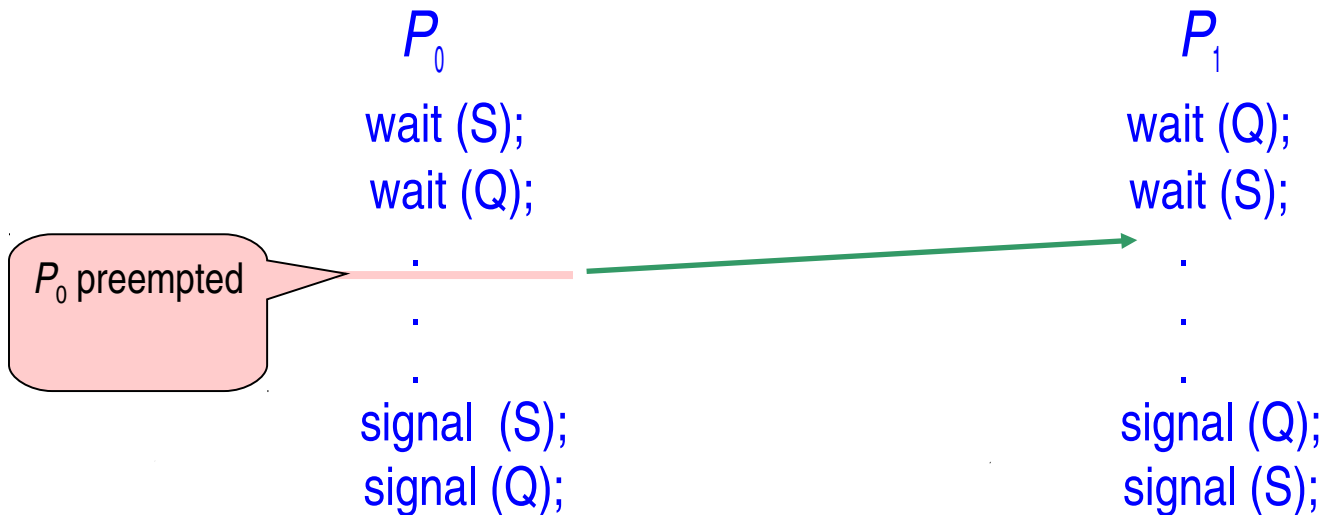
```
Semaphore S; // initialized to 1
wait (S);
    Critical Section
signal (S);
```

Spin-lock

- **Spin-lock** is a general (counting) semaphore using busy waiting instead of blocking
 - Blocking and switching between threads and/or processes may be much more time demanding than the time waste caused by short-time busy waiting
 - One CPU does busy waiting and another CPU executes to clear away the reason for waiting
- Used in multiprocessors to implement short critical sections
 - Typically inside the OS kernel
- Used in many multiprocessor operating systems
 - Windows 2k/XP, Linuxes, ...

Deadlock and Starvation

- **Overlapping critical sections related to different resources**
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to 1



- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

■ Bounded-Buffer Problem

- Passing data between 2 processes

■ Readers and Writers Problem

- Concurrent reading and writing data (in databases, ...)

■ Dining-Philosophers Problem from 1965

- An interesting illustrative problem to solve deadlocks
 - ▶ Five philosophers sit around a table; they either think or eat
 - ▶ They eat slippery spaghetti and each needs two sticks (forks)
 - ▶ What happens if all five philosophers pick-up their right-hand side stick?

“They will die of hunger”

Bounded-Buffer Problem using Semaphores

■ Three semaphores

- **mutex** – for mutually exclusive access to the buffer – initialized to 1
- **used** – counting semaphore indicating item count in buffer – initialized to 0
- **free** – number of free items – initialized to BUF_SZ

```
void producer() {
    while (1) { /* Generate new item into nextProduced */
        wait(free);
        wait(mutex);
        buffer[in] = nextProduced; in = (in + 1) % BUF_SZ;
        signal(mutex);
        signal(used);
    }
}

void consumer() {
    while (1) { wait(used);
        wait(mutex);
        nextConsumed = buffer[out]; out = (out + 1) % BUF_SZ;
        signal(mutex);
        signal(free);
        /* Process the item from nextConsumed */
    }
}
```

Readers and Writers

■ The task: Several processes access shared data

- ▶ Some processes read the data – **readers**
- ▶ Other processes need to write (modify) the data – **writers**
- Concurrent reads are allowed
 - ▶ An arbitrary number of readers can access the data with no limitation
- Writing must be mutually exclusive to any other action (reading and writing)
 - ▶ At a moment, only one writer may access the data
 - ▶ Whenever a writer modifies the data, no reader may read it

■ Two possible approaches

- Priority for readers
 - ▶ No reader will wait unless the shared data are locked by a writer. In other words: Any reader waits only for leaving the critical section by a writer
 - ▶ **Consequence: Writers may starve**
- Priority for writers
 - ▶ Any ready writer waits for freeing the critical section (by reader or writer). In other words: Any ready writer overtakes all ready readers.
 - ▶ **Consequence: Readers may starve**

Readers and Writers with Readers' Priority

Shared data

- semaphore wrt, readcountmutex;
- int readcount

Initialization

- wrt = 1; readcountmutex = 1; readcount = 0;

Implementation

Writer:

```
wait(wrt);
```

```
....
```

```
    writer modifies data
```

```
....
```

```
signal(wrt);
```

Reader:

```
wait(readcountmutex);
```

```
readcount++;
```

```
if (readcount==1) wait(wrt);
```

```
signal(readcountmutex);
```

```
... read shared data ...
```

```
wait(readcountmutex);
```

```
readcount--;
```

```
if (readcount==0) signal(wrt);
```

```
signal(readcountmutex);
```

Readers and Writers with Writers' Priority

Shared data

- semaphore wrt, rdr, readcountmutex, writecountmutex;
int readcount, writecount;

Initialization

- wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
readcount = 0; writecount = 0;

Implementation

Reader:

```
wait(rdr);  
wait(readcountmutex);  
readcount++;  
if (readcount == 1) wait(wrt);  
signal(readcountmutex);  
signal(rdr);
```

... read shared data ...

```
wait(readcountmutex);  
readcount--;  
if (readcount == 0) signal(wrt);  
signal(readcountmutex);
```

Writer:

```
wait(writecountmutex);  
writecount++;  
if (writecount==1) wait(rdr);  
signal(writecountmutex);  
wait(wrt);
```

... modify shared data ...

```
signal(wrt);  
wait(writecountmutex);  
writecount--;  
if (writecount==0) release(rdr);  
signal(writecountmutex);
```

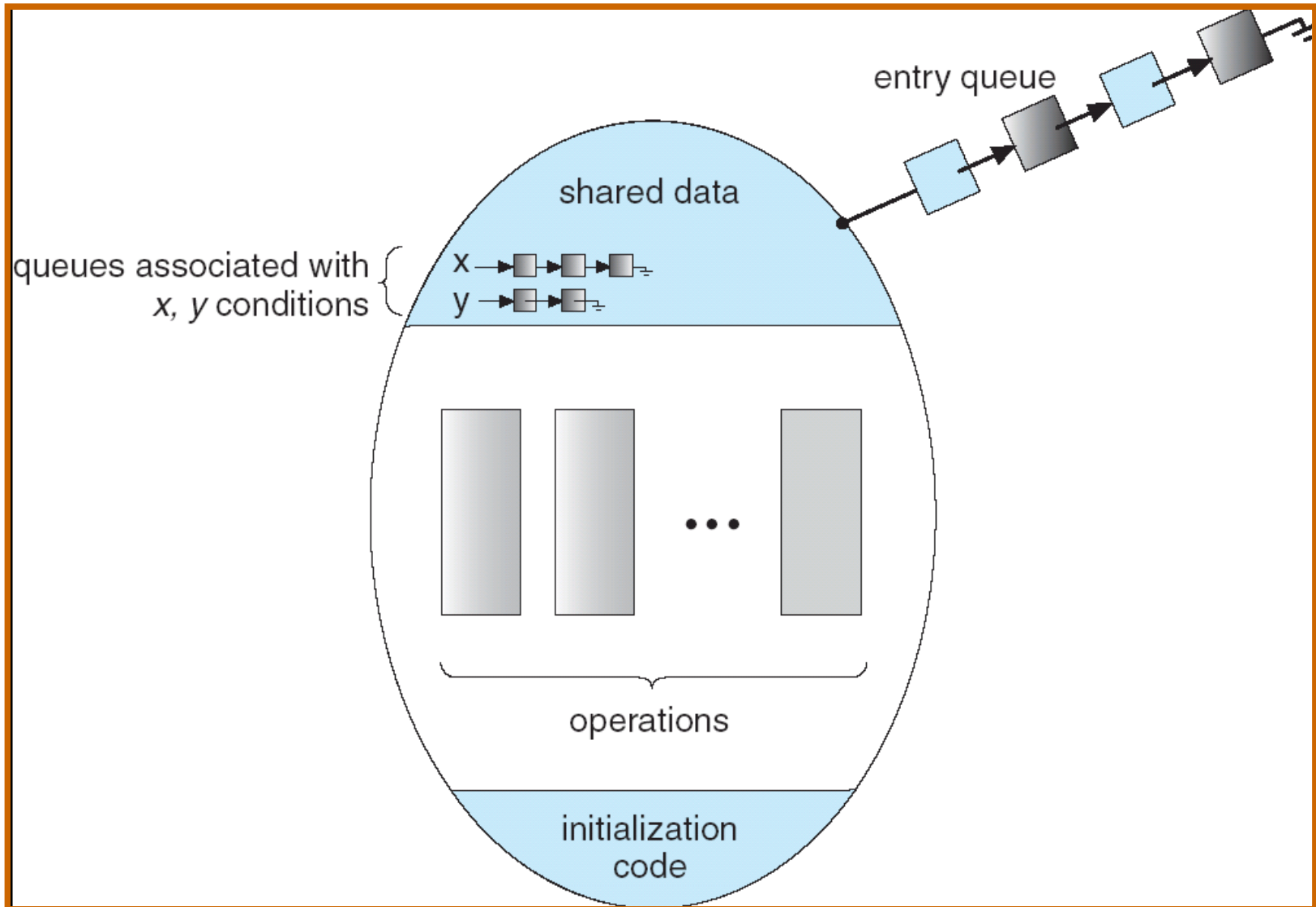

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor_name
{
    // shared variable declarations
    condition x, y; // condition variables declarations
    procedure P1 (...) { .... }
    ...
    procedure Pn (...) {.....}
    Initialization code ( ....) { ... }
    ...
}
}
```

- Two operations on a condition variable:
 - `x.wait ()` – a process that invokes the operation is suspended.
 - `x.signal ()` – resumes one of processes (if any) that invoked `x.wait ()`

Monitor with Condition Variables



Semaphores in Java

- Java is using Monitor for synchronization
- User can define counting semaphore as follows:

```
public class CountingSemaphore {  
    private int signals = 1;  
  
    public synchronized void wait() throws InterruptedException{  
        while(this.signals == 0) wait();  
        this.signals--;  
    }  
  
    public synchronized void signal() {  
        this.signals++;  
        this.notify();  
    }  
}
```

Synchronization Examples

■ Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
- Also provides **dispatcher objects** which may act as either mutexes and semaphores
- Dispatcher objects may also provide **events**
 - ▶ An event acts much like a condition variable

■ Linux Synchronization

- Disables interrupts to implement short critical sections
- Provides semaphores and spin locks

■ Pthreads Synchronization

- Pthreads API is OS-independent and the detailed implementation depends on the particular OS
- By POSIX, it provides
 - ▶ mutex locks
 - ▶ condition variables (monitors)
 - ▶ read-write locks (for long critical sections)
 - ▶ spin locks

End of Lecture 5

Questions?

