

OPERATING SYSTEMS

CONCURRENCY
MUTUAL EXCLUSION AND SYNCHRONIZATION



Introduction

Concurrency

- Interleaving and overlapping
- The **relative speed of execution** of processes cannot be predicted in uniprocessor systems because it depends on
 - Activities of other processes
 - The way the OS handles interrupts
 - Scheduling policies of the OS
- Difficulties
 - **Sharing** of global resources
 - Management of the **optimal allocation** of resources
 - Locate programming errors as **results are not deterministic** and reproducible

Operating System Concerns




Be able to keep track of various processes



Allocate and de-allocate resources for each active process



Protect the data and physical resources of each process against unintended interference by other processes



The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes

Goals

- The core concept is the **Critical Section** , where two or more processes compete to acquire the same resource
- The related **hardware and software approaches** to address the issue will be presented
- The **classical** mutual exclusion and synchronization **problems** will be presented

A revised version of the Producer / Consumer problem

use of a *shared variable* count set to 0

Producer

```
while (true) {  
    /* produce an item in next_product */  
    while (count == DIM_BUFFER)  
        /* do nothing */;  
    buffer[in] = next_product;  
    in = (in + 1) % DIM_BUFFER;  
    count++;  
}
```

```
register1 := count  
register1 := register1 + 1  
count := register1
```

A revised version of the Producer / Consumer problem

Consumer

```
while (true) {  
    while (count == 0)  
        /* do nothing */;  
    next_consumed = buffer[out];  
    out = (out + 1) % DIM_BUFFER;  
    count--;  
    /* consume an item in next_consumed */  
}
```

```
register1 := count  
register1 := register1 - 1  
count := register1
```

Race condition

- When the result of a sequence of instructions from concurrent processes depends on the order in which they are executed-

Let's assume that `count == 5`

T_0	<i>prod</i>	<code>register₁ = count</code>	<code>{register₁ == 5}</code>
T_1	<i>prod</i>	<code>register₁ = register₁ + 1</code>	<code>{register₁ == 6}</code>
T_2	<i>cons</i>	<code>register₂ = count</code>	<code>{register₂ == 5}</code>
T_3	<i>cons</i>	<code>register₂ = register₂ - 1</code>	<code>{register₂ == 4}</code>
T_4	<i>prod</i>	<code>count = register₁</code>	<code>{count == 6}</code>
T_5	<i>cons</i>	<code>count = register₂</code>	<code>{count == 4}</code>

Critical Section

Process execution and Critical Section

```
do {
```

```
  entry section
```

```
  critical section
```

```
  exit section
```

```
  remainder section
```

```
} while (true);
```

waits for its turn to enter the critical section

Process **w**riting on shared variables, **u**psdating tables, etc.

When one process in critical section, no other should be in its critical section

- Any protocol to address the Critical Section issue must satisfy
 - **Mutual exclusion**
 - **Progress**
 - **Bounded Waiting**

Critical Section and kernel tasks

Two approaches depending on the kernel being pre-emptive or non pre-emptive

- **Pre-emptive**

allows pre-emption of process when running in kernel mode

- **Non pre-emptive**

runs until exits kernel mode, blocks, or voluntarily yields CPU

- Essentially free of race conditions in kernel mode



Software Solutions to the Critical Section problem



Peterson Algorithm

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);
```

two processes P_i and P_j where $i = 1 - j$

Shared variables

```
int turn;           the next process to execute the critical section  
boolean flag[2];  process requesting the critical section
```



Hardware Support for the Critical Section

Critical Section Using Locks

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

Hardware Solutions

- Disabling interrupts in uniprocessor systems has the effect of a *lock*
 - No pre-emption
 - Difficult to implement in multiprocessor systems
- Special **atomic** operations are available in current hardware architectures to implement *locks*

test_and_set atomic instruction

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to "TRUE".

Critical Section with test_and_set

- Shared Boolean variable lock, initialized to FALSE

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
        /* critical section */  
    lock = false;  
        /* remainder section */  
} while (true);
```

compare_and_swap instruction

```
int compare_and_swap(int *value,  
                    int expected, int new_value) {  
  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

1. Executed atomically
2. Returns the original value of passed parameter value
3. Set the variable value the value of the passed parameter new_value but only if value == expected.

Critical Section with compare_and _swap

Shared integer lock initialized to 0;

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
        /* critical section */  
    lock = 0;  
        /* remainder section */  
} while (true);
```

Bounded waiting with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

shared variables
boolean waiting[n];
boolean lock;
initialized to false

Mutex Locks

Mutex Lock

The simplest OS built-in tool

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

- Calls to **acquire()** and **release()** must be **atomic**
 - Usually implemented via hardware atomic instructions

acquire() and release()

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

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

- This solution requires **busy waiting** and therefore the lock is called a **spinlock**

Semaphores

Definition

- 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()**

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

wait() and signal()

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

Mutual exclusion

- Consider **P1** and **P2** that require **S1** to happen before **S2**

Semaphore synch initialized to 0

P1:

```
S1;  
signal(synch);
```

P2:

```
wait(synch);  
S2;
```

Semaphore implementation

- No two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- **The implementation becomes the critical section problem**
- Can be implemented in hardware or firmware
- Software schemes such as Peterson's algorithm
- Could now have **busy waiting** in critical section implementation
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Another alternative is to use one of the hardware-supported schemes for mutual exclusion

Semaphore implementation without busy waiting

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

- Each semaphore has an associated **waiting queue**
- 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

Semaphore
implementation
without busy
waiting
wait()

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


Semaphore implementation without busy waiting signal()

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

Strong/Weak Semaphores

Strong Semaphores

- The process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- The order in which processes are removed from the queue is not specified

Simulating semaphores and other concurrency primitives

- BACI – Ben Ari Concurrent Interpreter
A Mutual Exclusion Toolkit
https://inside.mines.edu/~tcamp/baci/baci_index.html

Problems with semaphores

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

P_0

wait(S);

wait(Q);

...

signal(S);

signal(Q);

P_1

wait(Q);

wait(S);

...

signal(Q);

signal(S);

Starvation

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

Priority Inversion

- When a process with a high priority (H) is waiting for a lock held by a lower priority process (L) that is executing
 - If a process with a medium priority level (M) pre-empts process L, it delays the execution of H
- Solution: **priority-inheritance protocol**
process L inherits the priority H until it releases the lock requested by H

Problems with semaphores

- Incorrect use of semaphore operations
 - `signal(mutex) ... wait(mutex)`
 - `wait(mutex) ... wait(mutex)`
 - Omitting of `wait(mutex)` or `signal(mutex)` (or both)
- Deadlock and starvation are possible

Classical Problems of Synchronization

Three problems

- Bounded buffer Producer/Consumer problem
- Readers/Writers problem
- Dining Philosophers problem

Bounded buffer Producer/ Consumer problem

- The producer can put its product in the **buffer** until it is **full**
 - if no **empty** cell is available, the producer must **wait**
- The consumer can take products in the **buffer** until it is **empty**
 - if **no products** are available, the consumer must **wait**
- Operations on the buffer represent the **critical section** of both the producer and the consumer

Producer/ Consumer problem with Semaphores

- buffer of size N
- Semaphore s initialized to 1
 - s plays the role of the mutex
- Semaphore n initialized to 0
 - n used to count the number of items in the buffer
- Semaphore e initialized to N
 - e used to count the number of empty cells in the buffer

Producer process

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n= 0, e= sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
```

Consumer process

```
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}

void main()
{
    parbegin (producer, consumer);
}
```

Readers / Writers Problem

- A data area is **shared** among many processes
 - Some processes only read the data area, (**readers**) and some only write to the data area (**writers**)
- **Conditions** that must be satisfied
 - Any number of readers may **simultaneously read** the file
 - Only **one writer at a time** may write to the file
 - If a writer is **writing** to the file, **no reader** may read it

The Writer process readers have priority

```
/* program readersandwriters */  
int readcount;  
semaphore x = 1, wsem = 1;
```

```
void writer()  
{  
    while (true) {  
        semWait (wsem);  
        WRITEUNIT();  
        semSignal (wsem);  
    }  
}
```


The Reader process readers have priority

```
void reader()
{
    while (true) {
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
```

The Writer process writers have priority

```
void writer ()
{
    while (true) {
        semWait (y);
        writecount++;
        if (writecount == 1) semWait (rsem);
        semSignal (y);
        semWait (wsem);
        WRITEUNIT();
        semSignal (wsem);
        semWait (y);
        writecount--;
        if (writecount == 0) semSignal (rsem);
        semSignal (y);
    }
}

void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

The Reader process writers have priority

```
/* program readersandwriters */
int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
        semWait (z);
        semWait (rsem);
        semWait (x);
        readcount++;
        if (readcount == 1) semWait (wsem);
        semSignal (x);
        semSignal (rsem);
        semSignal (z);
        READUNIT();
        semWait (x);
        readcount--;
        if (readcount == 0) semSignal (wsem);
        semSignal (x);
    }
}
```

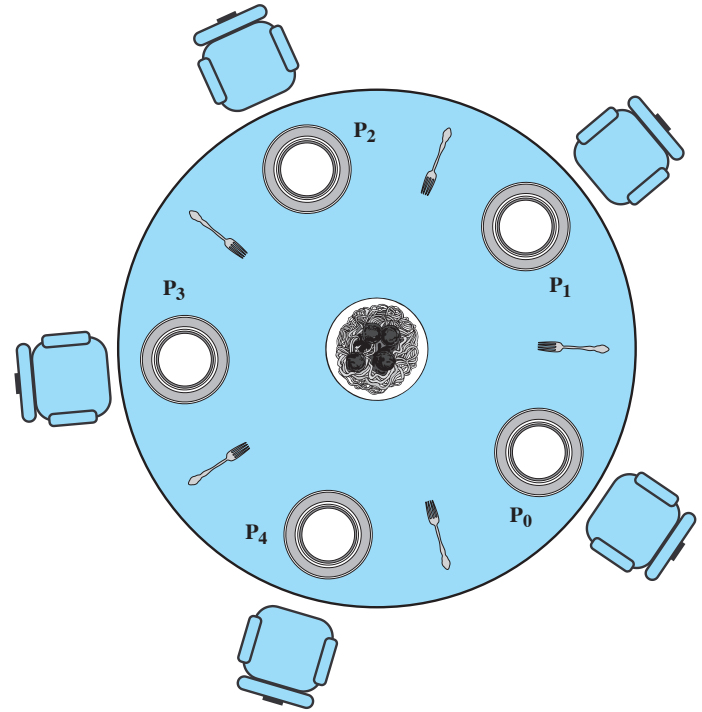
Dining Philosophers Problem

Five dining philosopher
They spend their time
alternating thinking with
eating

They share

- A bowl full of rice or spaghetti
- Chopsticks or forks

Each philosopher finds a
chopstick or fork at the
right and one at the left



First solution to the Dining Philosophers Problem

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
            philosopher (3), philosopher (4));
}
```

Deadlock and starvation are possible...

Other solutions to the Dining Philosophers Problem

- To avoid deadlock, one of the following constraints should be added
 - Only four out of five philosophers can request the forks
 - Forks must be picked in pairs and not one at a time
 - Each philosopher occupying an odd position must pick the fork in this order: first the left and then the right fork. Each philosopher occupying an even position must pick the fork in this order: first the right and then the left fork.

Only four
philosopher
are allowed to
request the
forks

```
/* program diningphilosophers */  
semaphore fork[5] = {1};  
semaphore room = {4};  
int i;  
void philosopher (int i)  
{  
    while (true) {  
        think();  
        wait (room);  
        wait (fork[i]);  
        wait (fork [(i+1) mod 5]);  
        eat();  
        signal (fork [(i+1) mod 5]);  
        signal (fork[i]);  
        signal (room);  
    }  
}  
void main()  
{  
    parbegin (philosopher (0), philosopher (1), philosopher (2),  
             philosopher (3), philosopher (4));  
}
```

Monitor

Monitor

- **Programming language construct** that provides equivalent functionality to that of semaphores and is **easier to control**
- A Monitor is a **software module** consisting of
 - one or more procedures
 - an initialization sequence
 - and local data
- Implemented in a number of programming languages
 - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java

Monitor Structure

```
monitor monitor-name
{
    // shared variable declarations

    procedure P1 (...) { ... }


    procedure Pn (...) {.....}

    Initialization code (...) { ... }

}
}
```

Monitor Characteristics

Local data variables are accessible only by the monitor's procedures and not by any external procedure

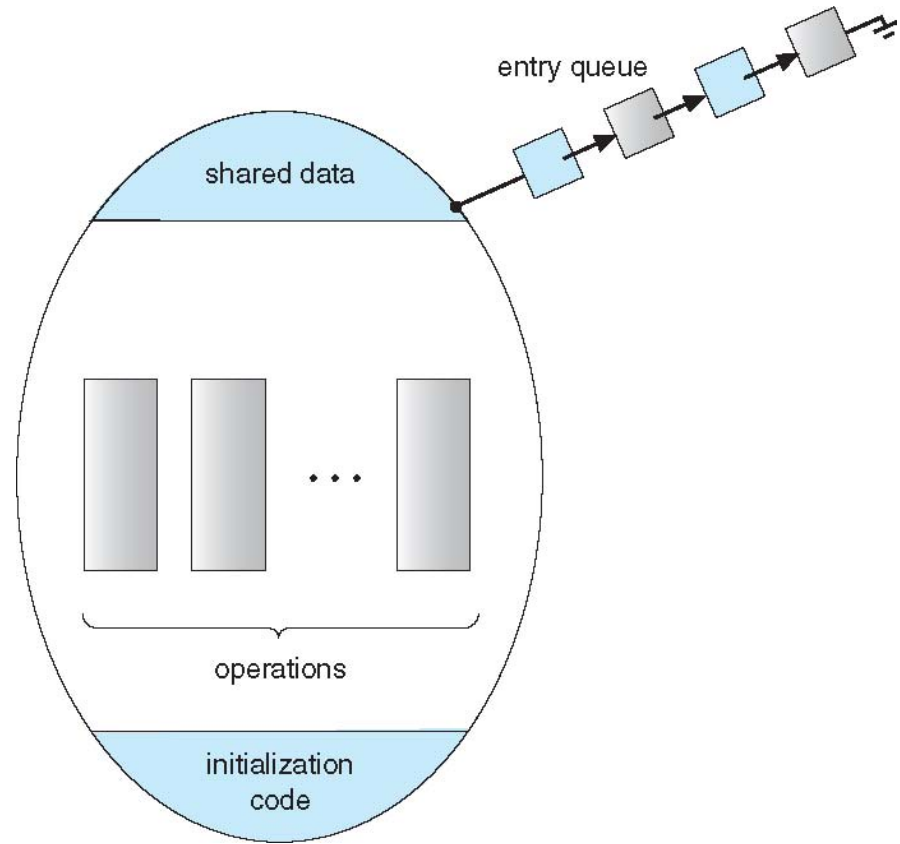


Process enters monitor by invoking one of its procedures



Only one process may be executing in the monitor at a time

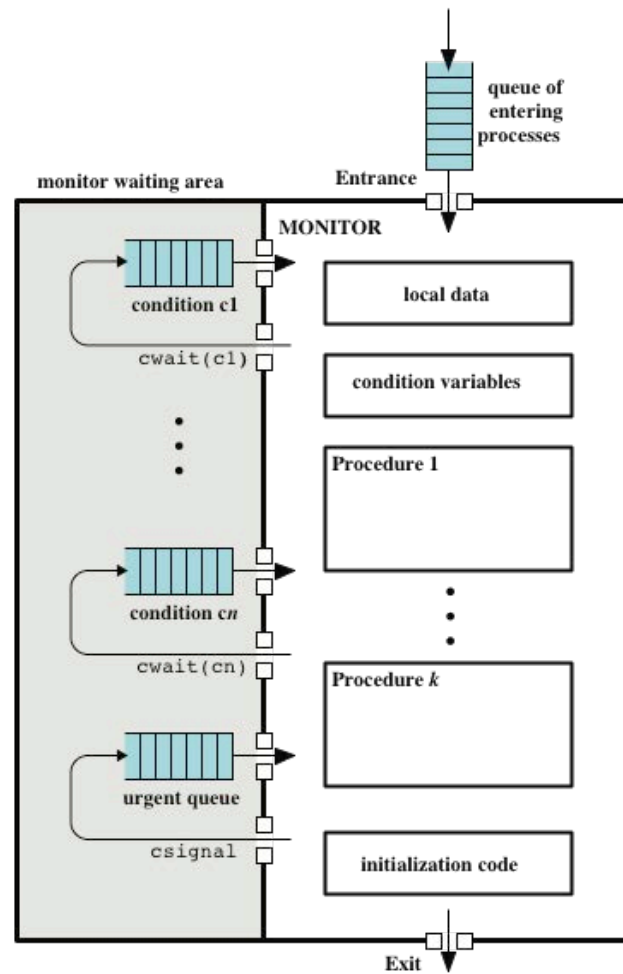
Schematic view of a Monitor



Sinchronization

- A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
- **Condition variables** are a **special data type in monitors** which are operated on by two functions
 - **cwait(c)**: suspend execution of the calling process on condition **c**
 - **csignal(c)**: resume execution of some process blocked after a **cwait** on the same condition

Structure of a Monitor with Condition Variables



Producer / Consumer problem using a Monitor

```
void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Producer / Consumer problem using a Monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];           /* space for N items */
int nextin, nextout;      /* buffer pointers */
int count;                /* number of items in buffer */
cond notfull, notempty;  /* condition variables for synchronization */

void append (char x)
{
    if (count == N) cwait(notfull);    /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                /* resume any waiting consumer */
}

void take (char x)
{
    if (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    /* one fewer item in buffer */
    csignal(notfull);                 /* resume any waiting producer */
}

{
    /* monitor body */
    nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
}
```


Solution to the Dining Philosophers Problem with a Monitor

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
    while (true) {
        <think>;
        get_forks(k);                /* client requests two forks via monitor */
        <eat spaghetti>;
        release_forks(k);           /* client releases forks via the monitor */
    }
}
```

Solution to the Dining Philosophers Problem with a Monitor

```
monitor dining_controller;
cond ForkReady[5]; /* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */

void get_forks(int pid) /* pid is the philosopher id number */
{
    int left = pid;
    int right = (++pid) % 5;
    /*grant the left fork*/
    if (!fork[left])
        cwait(ForkReady[left]); /* queue on condition variable */
    fork[left] = false;
    /*grant the right fork*/
    if (!fork[right])
        cwait(ForkReady[right]); /* queue on condition variable */
    fork[right] = false;
}
```

Solution to the Dining Philosophers Problem with a Monitor

```
void release_forks(int pid)
{
    int left = pid;
    int right = (++pid) % 5;
    /*release the left fork*/
    if (empty(ForkReady[left])      /*no one is waiting for this fork */
        fork[left] = true;
    else                               /* awaken a process waiting on this fork */
        csignal(ForkReady[left]);
    /*release the right fork*/
    if (empty(ForkReady[right])     /*no one is waiting for this fork */
        fork[right] = true;
    else                               /* awaken a process waiting on this fork */
        csignal(ForkReady[right]);
}
```

Mutual Exclusion and Synchronization in Linux and Windows

Semaphores in UNIX

- Generalization of the `semWait` and `semSignal` primitives
- No other process may access the semaphore until all operations have completed

Consists of

- Current value of the semaphore
- Process ID of the last process to operate on the semaphore
- Number of processes waiting for the semaphore value to be greater than its current value
- Number of processes waiting for the semaphore value to be zero

Signals

- A software mechanism that informs a process of the occurrence of **asynchronous events**
 - Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by
 - Performing some default action
 - Executing a signal-handler function
 - Ignoring the signal

UNIX Signals

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Spinlocks

- Most common technique for protecting a critical section in Linux
- Can only be acquired by one thread at a time
 - Any other thread will keep trying (spinning) until it can acquire the lock – busy waiting
- Built on an integer location in memory that is checked by each thread before it enters its critical section
- Effective in situations where the wait time for acquiring a lock is expected to be very short

Linux Spinlocks

<code>void spin_lock(spinlock_t *lock)</code>	Acquires the specified lock, spinning if needed until it is available
<code>void spin_lock_irq(spinlock_t *lock)</code>	Like spin lock, but also disables interrupts on the local processor
<code>void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)</code>	Like spin lock irq, but also saves the current interrupt state in flags
<code>void spin_lock_bh(spinlock_t *lock)</code>	Like spin lock, but also disables the execution of all bottom halves
<code>void spin_unlock(spinlock_t *lock)</code>	Releases given lock
<code>void spin_unlock_irq(spinlock_t *lock)</code>	Releases given lock and enables local interrupts
<code>void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)</code>	Releases given lock and restores local interrupts to given previous state
<code>void spin_unlock_bh(spinlock_t *lock)</code>	Releases given lock and enables bottom halves
<code>void spin_lock_init(spinlock_t *lock)</code>	Initializes given spinlock
<code>int spin_trylock(spinlock_t *lock)</code>	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
<code>int spin_is_locked(spinlock_t *lock)</code>	Returns nonzero if lock is currently held and zero otherwise

Semaphores

- User level
 - Linux provides a semaphore interface corresponding to that in UNIX SVR4
- Internally
 - Implemented as functions within the kernel and are more efficient than user-visible semaphores
- Three types of kernel semaphores
 - Binary semaphores
 - Counting semaphores
 - Reader-writer semaphores

Linux Semaphores

Traditional Semaphores	
<code>void sema_init(struct semaphore *sem, int count)</code>	Initializes the dynamically created semaphore to the given count
<code>void init_MUTEX(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
<code>void init_MUTEX_LOCKED(struct semaphore *sem)</code>	Initializes the dynamically created semaphore with a count of 0 (initially locked)
<code>void down(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
<code>int down_interruptible(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns <code>-EINTR</code> value if a signal other than the result of an up operation is received
<code>int down_trylock(struct semaphore *sem)</code>	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
<code>void up(struct semaphore *sem)</code>	Releases the given semaphore
Reader-Writer Semaphores	
<code>void init_rwsem(struct rw_semaphore, *rwsem)</code>	Initializes the dynamically created semaphore with a count of 1
<code>void down_read(struct rw_semaphore, *rwsem)</code>	Down operation for readers
<code>void up_read(struct rw_semaphore, *rwsem)</code>	Up operation for readers
<code>void down_write(struct rw_semaphore, *rwsem)</code>	Down operation for writers
<code>void up_write(struct rw_semaphore, *rwsem)</code>	Up operation for writers

Windows Concurrency Mechanisms

- Windows provides synchronization among threads as part of the object architecture

Most important methods are:

- Executive dispatcher objects
- User mode critical sections
- Slim reader-writer locks
- Condition variables
- Lock-free operations