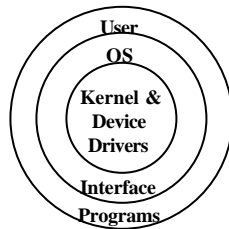
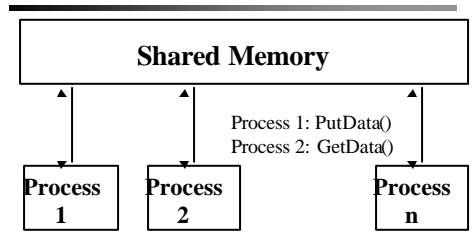


# Operating Systems



Interprocess Communication  
(IPC)

# Interprocess Communication



- Processes frequently need to communicate with other processes
- Use shared memory
- Need a well structured way to facilitate interprocess communication
  - Maintain integrity of the system
  - Ensure predicable behavior
- Many mechanisms exist to coordinate interprocess communication.

# Race Conditions

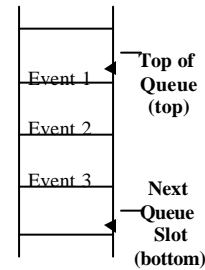
- In most modern OS, processes that are working together often share some common storage
  - Shared memory
  - Shared file system
- A **Race Condition** is when the result of an operation depends on the ordering of when individual processes are run
  - Process scheduling is controlled by the OS and is non-deterministic
- Race conditions result in intermittent errors that are very difficult to debug
  - Very difficult to test programs for race conditions
    - Must recognize where race conditions can occur
  - Programs can run for months or years before a race condition is discovered

# Race Condition Example

```

Enqueue( Data )
{
    Q[bottom] = Data
    bottom = bottom + 1
}

Dequeue( )
{
    Data = Q[top]
    top = top + 1
    return Data
}
    
```



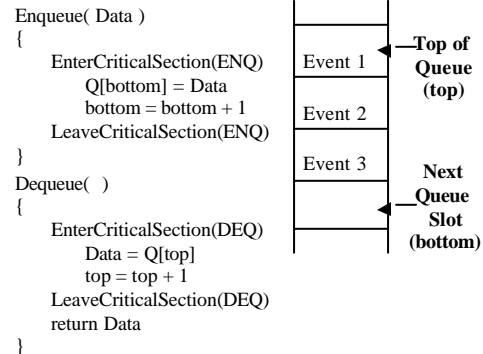
*If 2 or more processes are using the Queue where are the race conditions?*

- Queues are very useful operating systems data structures
  - Spollers, Disk request management, ...

## Critical Sections

- Need to **recognize** and **avoid** race conditions
- Desired state - Achieve mutual exclusion
- **Mutual Exclusion** ensures that if one process is using a shared variable or file then all other processes will be excluded from doing the same thing
- A **Critical Section** is a block of code where shared resources (memory, files) are accessed
- Goal: Avoid race conditions by ensuring that no two processes are in their critical section at the same time
- By managing critical sections we ensure that the code in a critical section appears atomic with respect to the other processes in the system

## Critical Section Example



*Does the above code avoid race conditions?*

- If a process tries to enter a named critical section, it will:
  - Blocks: Critical section in use
  - Enter: Critical section not in use

## Rules for Avoiding Race Conditions

*Four conditions must hold to ensure that parallel processes cooperate correctly and efficiently using shared data*

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

*If the above conditions hold, then race conditions will be avoided*

## Techniques for Avoiding Race Conditions

### METHODS

- Disabling Interrupts
- Strict Alternation
- Test and Set Lock
- Sleep and Wakeup
- Semaphores
  - Event
  - Mutex
- Monitors
- Message Passing

### IMPLEMENTATION

- Busy Waiting
- Blocking

## Disabling Interrupts

- Disable interrupts to enter critical section
- Enable interrupts to exit critical section
- By enabling/disabling interrupts we achieve mutual exclusion because no task switches can occur with interrupts disabled
  - Scheduler uses a timer interrupt to facilitate task switching
- Not good for user processes
  - What if process crashes, interrupts never reenabled
  - Valuable for OS
- Problems
  - Computers with 2 or more CPUs
  - Disabling interrupts affects all processes, not just the ones that share common resources
  - No mechanism to arbitrate fairness

## Mutual Exclusion with Interrupt Management

```

Enqueue( Data )
{
    DisableInterrupts()
    Q[bottom] = Data
    bottom = bottom + 1
    EnableInterrupts()
}

Dequeue( )
{
    DisableInterrupts()
    Data = Q[top]
    top = top + 1
    EnableInterrupts()
    return Data
}
    
```

## Lock Variables

- Use a single shared lock variable to manage each critical section
- When a user wants to enter a critical section it tests the lock
- If the lock is clear (0), set the lock and enter the critical section
- If the lock set ( $\neq 0$ ), then wait for the lock to clear

```

EnterCriticalSec( lock )    LeaveCriticalSec( lock )
{
    while(lock <> 0)        {
        lock = 1           }
    }
    
```

*Does the above code avoid all race conditions?  
What is the efficiency of the above code?*

## Strict Altercation

- With **strict altercation** the programmer develops code to ensure that race conditions do not occur
- Problems:
  - Hard to scale if additional processes are added
  - Programmer managing things that the OS should be responsible for
  - Not fair, especially when one process has more/less work to do than the other processes

```

while(1)                    while(1)
{
    while(turn!=0);          while(turn!=1);
    critical_section()      critical_section()
    turn = 1                 turn = 0
    non_critical_section()  non_critical_section()
}
    
```

**Process A**
**Process B**

## Test and Set Lock (TSL)

- Most CPUs come with a test and set lock instruction
- With TSL we can use the simple lock variable approach without any risk of a race condition
- TSL instruction
  - Requires a shared memory variable (*flag*)
  - Copies the value of *flag* to a register and sets the *flag* to 1 in a single, **non-interruptible** instruction

```
EnterCriticalSection:      LeaveCriticalSection:
    tsl    register, flag    mov    flag, #0
    cmp    register, #0      ret
    jnz   EnterCriticalSection
    ret
```

## Busy-Waiting

- **Busy-waiting**
  - When a process wants to enter a critical section it checks if the entry is allowed
  - If not, the process executes a tight loop, constantly checking if it is allowed to enter a critical section
  - Lock variable, strict alternation, TSL
- Busy-waiting problems
  - Waste of CPU
  - Priority Inversion Problem
    - Low and high priority processes
    - Scheduler favors the high priority process
    - If the low priority process is running in a critical section and the high priority process becomes ready to run
    - If the high priority process needs to enter the same critical section it busy-waits resulting in the low priority process never finishing its critical section (this is known as **starvation**)

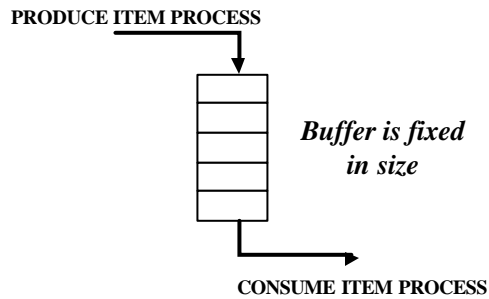
## Avoiding Busy-Waiting

- Recall that our desire is to allow the Operating System to **efficiently** coordinate access to shared resources
- Goal: Achieve mutual exclusion by implementing critical sections with **blocking** primitives
- Approach
  - Attempt to enter a critical section
  - If critical section available, enter it
  - If not, register interest in the critical section and block
  - When the critical section becomes available, the OS will unblock a process waiting for the critical section, if one exists
- Using blocking constructs greatly improves the CPU utilization

## Bakery Algorithm

- Also used for deadlock prevention, discussed later
- Critical section for  $n$  processes
  - Before entering its critical section, process receives a number.
  - Holder of the smallest number enters the critical section.
  - If processes  $P_i$  and  $P_j$  receive the same number, if  $i < j$ , then
  - $P_i$  is served first; else  $P_j$  is served first.
  - The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

## The Producer Consumer Problem



- We will use the producer consumer problem to study various IPC techniques
- Producer places elements in the buffer
- Consumer removes elements from the buffer
- Must ensure that the buffer does not over- or under-flow

## Mutual Exclusion with Sleep() and Wakeup()

- Sleep() and Wakeup() are IPC primitives that block
  - Do not waste CPU time when a process is not allowed to enter its critical section
  - System calls provided by the OS
- Sleep() causes the calling process to block
- Wakeup() awakes a process that is sleeping
- Both Sleep() and Wakeup() take a single parameter to match up Sleep() and Wakeup() calls
- Wakeup() calls are not buffered, thus they do nothing if the process being waked up is not currently sleeping

## Producer Consumer Problem with Sleep() and Wakeup()

```
#define N 100
int count = 0;
void Producer(void)
    int item;
    while(true)
        produce_item(&item);
        if (count == N) sleep(producer);
        enter_item(item); //put item in buffer
        count++;
        if (count==1) wakeup(consumer);

void Consumer(void)
    int item;
    while(true)
        if (count == 0) sleep(consumer);
        remove_item(&item); //get item from buffer
        count--;
        if(count == N-1) wakeup(producer);
        consume_item(item);
```

## Bug with Sleep() and Wakeup() Solution

```
void Producer(void)
    while(true)
        produce_item(&item);
        if (count == N) sleep(producer);
        enter_item(item); //put item in buffer
        count++;
        if (count==1) wakeup(consumer);

void Consumer(void)
    int item;
    while(true)
        if (count == 0) sleep(consumer);
        ...
```

*Task switch here when count is 0*

- Consumer suspended after it checks that count is zero. Producer runs, creates an item and sends a **wakeup()** to the consumer who is not yet sleeping. Consumer restarted and sleeps. Producer fills buffer and sleeps. Deadlock!

## Semaphores

- The problem with `sleep()` and `wakeup()` is that wakeups that are sent to non sleeping processes are lost
- **Semaphores** are a special type of variable that have the following properties:
  - A value of 0 if no wakeups were saved
  - A value  $> 0$  indicating the number of pending wakeups
- Semaphores “*save*” wakeups to compensate for future sleep calls
- Checking, changing, and possibly going to sleep is all done in a single indivisible atomic action
  - Requires support from the OS
  - Implemented as system calls
- Semaphores were proposed by Dijkstra in 1965

## Semaphore Operations

- **Down()**
  - The Down() operation checks the value of a semaphore
  - If 0 then the calling process is put to sleep
    - Blocks waiting for an Up() call
  - If  $> 0$  the semaphore is decremented
    - Uses up a *stored* wakeup and continues normally
- **Up()**
  - The Up() operation increments the value of the semaphore
  - The Up() operation is non-blocking
- Both down and up take a semaphore variable as a parameter
- Other common semaphore notation:
  - P() = Down()
  - V() = Up()

## Binary Semaphores

- Binary Semaphores
  - Accomplished by initializing the semaphore value to 1
  - Critical sections created by:
    - Performing a Down() on the semaphore to enter a critical section
    - Performing an Up() to leave the critical section
    - Ensures that only one process can be in a critical section protected by a semaphore
  - Sometimes referred to as *mutex* or mutual exclusion semaphores

```
semaphore mutex = 1;
down(&mutex);
/* Critical
Section */
up(&mutex);
```

## Event Semaphores

- Use the atomic nature of semaphore APIs to ensure proper synchronization
- Event semaphores are typically initialized to a value indicating the number of events that are allowed to happen
  - Example: the number of elements that are allowed to be in the bounded buffer
- Event semaphores are managed so that no more than N events can occur if the semaphore is initialized to N
- Event semaphores use the Up() and Down() system calls to manage the event semaphores

## Producer Consumer Problem with Semaphores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1; //mutex for critical sections
semaphore empty = N; //event for empty slots
semaphore full = 0; //event for full slots
void Producer(void)
{
    int item;
    while(true)
    {
        produce_item(&item);
        down(&empty);
        down(&mutex);
        enter_item(item); //put item in buffer
        up(&mutex);
        up(&full);
    }
}
```

## Producer Consumer Problem with Semaphores

```
void Consumer(void)
{
    int item;
    while(true)
    {
        down(&full);
        down(&mutex);
        remove_item(&item); //get item from buffer
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

*Unlike the Sleep() and Wakeup() implementation, the semaphore version of the program does not have any unwanted race conditions*

## Monitors

- Semaphores are relatively low-level operations
- Accidents with misplaced Up() and Down() calls can lead to incorrect program behavior
- Monitors were developed to make it easier to develop correct concurrent programs
- **Monitors** are a collection of procedures, variables and data structures that are grouped into a special kind of module or package
- Monitors are high level constructs and are typically supported by a compiler that maps the monitor into lower-level constructs
  - Not supported by many languages, Java's **synchronized** keyword is close to a monitor
  - Monitors can be simulated by using semaphores

## Monitors

- Monitor property
  - Only one process can be active in a monitor at any instance
  - Enables simple mutual exclusion
  - If a process attempts to enter a monitor
    - It will be allowed if the monitor is not in use
    - It will block if the monitor is in use. The process will automatically be awakened when the monitor is available

```
monitor example
{
    procedure p1()
    { ... }
    procedure p2()
    { ... }
}
end monitor
```

**Monitor Structure**

## Monitors

- Monitors automatically provide mutual exclusion with respect to executing procedures managed by the monitor
- Sometimes a process can not continue executing inside a monitor and must wait for an external event
  - Example: What should the producer do if it has produced an item and the buffer is full?
  - This problem is solved with **condition variables** and the **Wait()** and **Signal()** operations that operate on the condition variables
- The **Wait()** operation puts the calling process to sleep
- The **Signal()** operation wakes up a sleeping process
  - The Signal() operation, if needed, must be the last call that a process makes when exiting a monitor

## Producer Consumer Problem with Monitors

```
monitor ProducerConsumer
condition full, empty;
integer count;
procedure enter;
begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty);
end
procedure remove;
begin
    if count = 0 then wait(empty)
    remove_item;
    count := count - 1;
    if count = (N - 1) then signal(full);
end
count := 0;
end monitor;
```

## Producer Consumer Problem with Monitors

```
procedure producer;
begin
    while true do
        begin
            produce_item;
            ProducerConsumer.enter;
        end
    end
end
procedure consumer;
begin
    while true do
        begin
            Producer.consumer.remove;
            consume_item;
        end
    end
end
```

***Why is signal() coded as the last statement in the monitor procedures?***

## Problems with Synchronization Constructs

- Sleep() / Wakeup(), Semaphores, and Monitors work well for managing mutual exclusion problems in computer architectures that consist of a single shared memory
  - All use a shared global variable to coordinate and synchronize processes
  - This shared global variable must be visible to all processes in the system
    - Thus we must have a single shared memory
  - All can be implemented by the TSL instruction
    - System calls provided by the OS
    - TSL does not work across distributed memory
- Need another synchronization construct
  - Message Passing

## Message Passing

- Message passing is a method of interprocess communication that works well in systems that:
  - Have multiple CPUs
  - Do not have a single shared memory
  - Are connected by a standard communications network
- Primitives:
  - **send**(destination, &message)
  - **receive** (source, &message)
- Receive() should be setup to block until a message is sent to the receiving process

## Message Passing Issues

- Because message passing can pass messages across CPU or system (via a network) boundaries we must protect against lost messages
  - Communication networks are not 100% reliable
  - Need to use acknowledgement messages to confirm the receipt of a message
- We need a consistent naming convention for the sender and receiver processes
  - *process@machine.domain*
  - A domain is a collection of individual computers
- We need an authentication mechanism to ensure that the sending and receiving processes trust each other
  - Must prevent *spoofing*

## Producer Consumer Problem with Message Passing

```
#define N      100           //size of bounded buffer
#define MSIZE  4            //size of a message
typedef int message[MSIZE]; //message buffer

void producer()
{
    int item;
    message m;
    while (true) {
        produce_item(&item);
        receive(consumer, &m); //receive a null message
        build_message(&m, &item);
        send(consumer, &m); //send item to consumer
    }
}
```

## Producer Consumer Problem with Message Passing

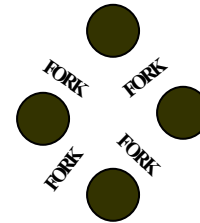
```
void consumer()
{
    int item;
    message m, e;
    create_empty_message(&e);
    for(int i = 0; i < N, i++)
        send(producer, &m); //send N empty messages
    while (true) {
        receive(producer, &m); //receive a message
        extract_item(&m, &item);
        send(producer, &e); //send empty message
        consume_item(item);
    }
}
```

***Notice the importance of using empty messages to synchronize the critical sections***

## Equivalence of Primitives

- Each synchronization primitive that we have studied has pro's and con's associated with their usage
- Messages, monitors, and semaphores are equivalent because each construct can be used to implement or simulate the other two constructs
- We will use a Java package that simulates semaphores, monitors and message passing
  - More on this later

## Classical IPC Problem: Dining Philosophers



- There are N philosophers
- There are N forks
- Can be in **thinking**, **hungry** or **eating** state
- Can only eat if the philosopher can obtain its 2 adjacent forks
- Must be careful to avoid deadlock
  - Each philosopher has one fork and needs one fork

## Dining Philosophers Solution

```
#define N 5 //num. of philosophers
#define LEFT (i-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define EATING 1
#define HUNGRY 2

typedef int semaphore;
int state[N]; //state of philosophers
semaphore mutex = 1;
semaphore s[N]; //one sem per philosopher

void philosopher(int i)
{
    while(TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}
```

## Dining Philosophers Solution

```
void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i); //try to get 2 forks
    up(&mutex);
    down(&s[i]); //block if both forks not
                //available
}

void put_forks(int i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT); //see if left neighbor
                //needs to eat
    test(RIGHT); //see if right neighbor
                //needs to eat
    up(&mutex);
}
```

## Dining Philosophers Solution

---

```
void test(int i)
{
    if ((state[i] == HUNGRY) &&
        (state[LEFT(i)] != EATING) &&
        (state[RIGHT(i)] != EATING))
    {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

- Acquire both forks in a critical section
- When done eating see if left and right neighbors can eat
- If trying to eat but both forks not available then block on a semaphore
- Release semaphore when both forks are available