Operating Systems

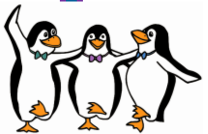
Process Synchronization

WILEY

UOIT
CHALLENGE INNOVATE CONNECT

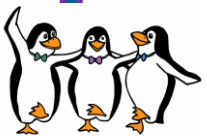
Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches



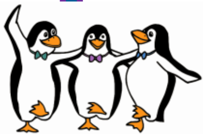
Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems



Background

- A **cooperating process** is one that can affect or be affected by other processes executing in the system.
  - They can directly share a logical address space (both code and data) → threads
  - They can share data only through files or messages.
- **Processes can execute concurrently or in parallel**
  - May be interrupted at any time, partially completing execution
  - Concurrent access to shared data may result in data inconsistency
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.



Background

- **Problem (Race Condition):**
  - If counter = 5 and that the producer and consumer processes concurrently execute the statements “counter++” and “counter--”.
  - Then what is the value of the variable counter, is it **4**, **5**, or **6**!

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 */
}
```

Background

- **Problem (Race Condition):**
  - We can show that the value of counter may be incorrect as follows:
    - `counter++` could be implemented in machine language 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:

```
T0: producer execute register1 = counter           {register1 = 5}
T1: producer execute register1 = register1 + 1     {register1 = 6}
T2: consumer execute register2 = counter           {register2 = 5}
T3: consumer execute register2 = register2 - 1     {register2 = 4}
T4: producer execute counter = register1           {counter = 6 }
T5: consumer execute counter = register2           {counter = 4}
```

  - If several processes access the same data concurrently and the outcome depends on the order in which the access takes place, this is called a **race condition.**

The Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file,...
  - When one process is in its critical section, no other process is allowed to be in its critical section
- ***Critical section problem*** is to design protocol to solve this issue
  - Each process must request permission to enter its critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**
  - General structure of process P_i is as follows:

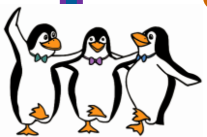
```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```

The Critical Section Problem

- Solution to Critical-Section problem must satisfy these three requirements:

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
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 the **relative speed** of the n processes



The Critical Section Problem

- Critical-Section Handling in OS
  - The kernel code of the OS is subject to race conditions.
    - Example: The data structure for maintaining a list of all open files in the system, memory allocation, process lists, interrupt handling.
  - **Two approaches are used to handle critical sections in OS:**
    - **Preemptive kernel:** allows preemption of process when running in kernel mode
    - **Non-preemptive kernel:** the process runs until it exits kernel mode, blocks, or voluntarily yields the CPU
      - It is free of race conditions on kernel data structures, as only one process is active in the kernel at a time
  - **Why then we use Preemptive kernel:**
    - It is more responsive, it has less risk that a kernel-mode process will run for long period before leaving the CPU to other processes
    - It is more suitable for real-time programming

Peterson's Solution

- It is a software-based solution to the critical-section Problem
- **Peterson's solution** is restricted to **two processes (P_i ,P_j)** that alternate execution between their critical sections and remainder sections
  - Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
  - The solution requires that the two processes share two variables:

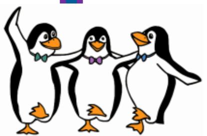
```
int turn;
Boolean flag[2];
```

  - The variable `turn` indicates whose turn it is to enter the critical section

    turn = i → process ***Pi*** is allowed to execute its 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 $\boldsymbol{P_i}$ is ready!



Peterson's Solution

- **The solution:**
  - To enter the critical section, process P_i first sets ***flag[i]=true*** and ***turn=j***
    - Gives the other process (P_j) precedence if it was ready
    - If both processes tried to set turn at the same time, then only one of these assignments (***turn=j*** or ***i***) will last;

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);
```

Algorithm for Process P_j

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

Peterson's Solution

- **The solution:**
  - Prove that the three critical section (CS) requirement are met:
    1. Mutual exclusion is preserved

       $\mathtt{P_i}$ enters CS only if: either `flag[j] = false` or `turn = i`
    2. Progress requirement is satisfied
    3. Bounded-waiting requirement is met

       – P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions are based on idea of **locking**
  - Protecting critical regions via locks
- The critical-section problem could be solved simply in a single-processor environment
  - Because uniprocessors can disable interrupts while a shared variable was being modified.
    - Currently running code would execute without preemption
  - Generally, it is too inefficient on multiprocessor systems
    - Disabling interrupts in a multiprocessor is time consuming: message has to be passed to all processors before entering the critical section
    - Operating systems using this not broadly scalable
- Modern machines provide special **atomic** hardware instructions
    - **Atomic** = non-interruptible
  - Instruction to: test and modify the content of a memory word

  or

  - Instruction to: swap the contents of two memory words atomically

Synchronization Hardware

- Using **test_and_set** Instruction
  - Definition:

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

    - This instruction is executed atomically
      - If two test_and_set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
    - Returns the original value of passed parameter
    - Set the new value of passed parameter to “TRUE”.
- Then provide mutual exclusion by shared Boolean variable **lock**, initialized to FALSE.

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

Synchronization Hardware

- Using **compare_and_swap** Instruction
  - Definition: the instruction operates on three operands:

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

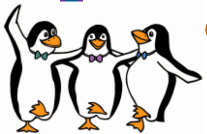
1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Synchronization Hardware

- Using **compare_and_swap** Instruction
  - Introduce a shared integer called “lock” initialized to 0;
  - Solution:

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

  - Mutual exclusion can be provided as follows: a global variable (lock) is declared and is initialized to 0. The first process that invokes compare_and_swap() will set **lock=1**. It will then enter its critical section, because the original value of lock was equal to the expected value of 0.
  - Subsequent calls to compare_and_swap() will not succeed, because lock now is not equal to the expected value of 0.
  - When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section.



Synchronization Hardware

- Bounded-waiting Mutual Exclusion with test_and_set
  - Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement
  - The following algorithm uses the test_and_set() instruction that satisfies all the critical-section requirements. The common data structures (initialized to false) are:

    boolean waiting[n];

    boolean lock;

```
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);
```

- To prove that the bounded-waiting requirement is met:
- when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n − 1, 0, ..., i − 1) to find the next process (waiting[j] == true) to enter the critical section.
- Any process waiting to enter its critical section will thus do so within n − 1 turns.

Mutex Locks

- The previous hardware-based solutions are complicated and mostly inaccessible to application programmers.
- Therefore, operating-systems designers build software tools to solve the critical-section problem such as **mutex lock** (mutual exclusion).
  - The luck has a Boolean variable indicating if the lock is available or not
  - They are used to protect critical regions and prevent race conditions
  - A process must acquire() the lock before entering a critical section and release() it when it exits the critical section.
    - The acquire() function acquires the lock, and the release() function releases the lock

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

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

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

- The acquire() and release() must be **atomic**. Thus, mutex locks are often implemented as a **hardware** mechanisms

Mutex Locks

- But this solution requires **busy waiting** (While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to *acquire()*).
  - This lock therefore called a **spinlock**

Mutex Locks

- **Question:**
  - Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
      int available;
} lock;
```

where (available == 0) indicates the lock is available; a value of 1 indicates the lock is unavailable.

Using this struct, illustrate how the following functions may be implemented using the test_and_set() and compare_and_swap() instructions.

- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.

Mutex Locks

- **Solution:**

```
// initialization
mutex->available = 0;

// acquire using compare and swap()
void acquire(lock *mutex) {
    while (compare and swap(&mutex->available, 0, 1) != 0)
    ;
    return;
}

// acquire using test and set()
void acquire(lock *mutex) {
    while (test and set(&mutex->available) != 0)
    ;
    return;
}

void release(lock *mutex) {
    mutex->available = 0;
    return;
}
```

Semaphores

Semaphore

- It is a synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** is an integer variable that can only be accessed via two indivisible (atomic) operations:
    - **wait()** and **signal()**
        - Originally called P() and V()
- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

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

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

- *These operations must be executed without interruption*

Semaphores

- Semaphore Usage
  - **Counting semaphore:** its integer value can range over an unrestricted domain
    - Counting semaphores can be used to control access to a given resource consisting of a finite number of instances (bounded buffer).
      - The semaphore is initialized to the number of resources available.
      - Each process that wishes to use a resource performs a wait() (decrementing the count).
      - When a process releases a resource, it performs a signal() (incrementing the count).
      - If the count=0, (all resources are being used), then a processes that wish to use a resource will block until count > 0.
  - **Binary semaphore:** its integer value can range only between 0 and 1
    - Used the same as a **mutex lock** to provide mutual exclusion

Semaphores

- Semaphore Usage
  - Semaphore also can be used to solve various synchronization problems:
    - Consider $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ that require $\boldsymbol{S_1}$ statement to be executed before $\boldsymbol{S_2}$ statement
      - Create a semaphore "**synch**" that initialized to 0

In process P1, use the statments:

```
S₁;
signal(synch);
```

In process P2, use the ststments:

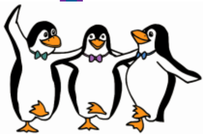
```
wait(synch);
S₂;
```

- Because **synch** is initialized to **0**, P2 will execute S_2 only after P1 has invoked signal(synch), which is after statement S_1 has been executed.

Semaphores

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- The previous definitions of the wait() and signal() semaphore operations suffer also from the **busy waiting** problem:
  - When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait by engaging in busy waiting,
  - Instead the process can block itself by placing itself in a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.



Semaphores

- Semaphore Implementation with no Busy waiting
  - With every semaphore there are an integer value and an associated waiting queue
    - Each entry in the waiting queue is a PCB that has pointer to next record in the list
  - **We need two operations that are provided by the OS as system calls:**
    - **block()**: places the process invoking the operation on the appropriate waiting queue
    - **wakeup()**: removes one of processes from the waiting queue and place it in the ready queue
  - **We can implement the semaphore as:**

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

For the list, we can use a FIFO or any other queue

Semaphores

- Semaphore Implementation with no Busy waiting

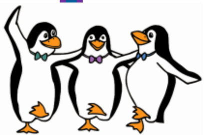
```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

-> :used when you have a pointer to a structure and you want to dereference one of the struct's fields.

http://williamstallings.com/OS-Animation/Queensland/SEMA.SWF

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

- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore
- It is critical that semaphore operations be executed atomically.
  - We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time.



Semaphores

- Deadlock and Starvation
  - **Deadlock:** two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
    - consider a system consisting of two processes, ***P*$_0$** and ***P*$_1$**, each accessing two semaphores, **S** and **Q**, set to the value 1

```
 P0                         P1
wait(S);                   wait(Q);
wait(Q);                   wait(S);
 ...                         ...
signal(S);                 signal(Q);
 signal(Q);                signal(S);
```

    - If ***P*$_0$** executes wait(S) and then ***P*$_1$** executes wait(Q).
    - When ***P*$_0$** executes wait(Q), it must wait until ***P*$_1$** executes signal(Q).
    - When ***P*$_1$** executes wait(S), it must wait until ***P*$_0$** executes signal(S).
      - Since these signal() operations cannot be executed, ***P*$_0$** and ***P*$_1$** are **deadlocked**.
  - **Starvation** **(indefinite blocking)** $\rightarrow$ another problem with deadlocks
    - A process may never be removed from the semaphore queue in which it is suspended (when the queue is implemented in LIFO)

Semaphores

- Priority Inversion
  - Scheduling problem arises when lower-priority process holds a lock needed by a higher-priority process
    - Solved via **priority-inheritance protocol**
      - A process that is accessing a resource needed by a higher-priority process inherit the higher priority until it is finished with the resource in question

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes:
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Classical Problems of Synchronization

- Bounded-Buffer Problem
  - The producer and consumer processes share the following data structures:
    - There are ***n*** buffers, each can hold one item
    - The **`mutex`** semaphore provides mutual exclusion for accesses to the buffer pool
    - The **`full`** and **`empty`** semaphores count the number of empty and full buffers
    - Initialization:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
  - We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The structure of the **producer** process

```
do {
       ...
/* produce an item in next_produced */
          ...
          wait(empty);
          wait(mutex);
          ...
/* add next produced to the buffer */
          ...
          signal(mutex);
          signal(full);
} while (true);
```

The structure of the **consumer** process

```
Do {
          wait(full);
          wait(mutex);
          ...
/* remove an item from buffer to next_consumed */
          ...
          signal(mutex);
          signal(empty);
          ...
/* consume the item in next consumed */
          ...
} while (true);
```

Classical Problems of Synchronization

- The Readers-Writers Problem
  - Assume that a database is shared among several concurrent processes:
    - Readers: processes that only read the data; they do ***not*** perform any updates
    - Writers: processes that can both read and write to the database
  - **Problem**: if a writer and some other process (either a reader or a writer) access the database simultaneously
    - Writers should have exclusive access to database while writing to it
    - All possible solutions are involving some form of **priorities**
    - Readers-Writers Problem Variations
      - *First* reader-writer problem: no reader kept waiting unless a writer has permission to use shared object (no reader should wait for other readers to finish because a writer is waiting) → writers may starve
      - *Second* reader-writer problem: once a writer is ready, it should perform the write ASAP (if a writer is waiting to access the object, no new readers may start reading) → readers may starve
  - A solution to either problem may result in starvation
  - Problem is solved on some systems by kernel providing reader-writer locks

Classical Problems of Synchronization

- The Readers-Writers Problem
  - The solution to the *first* readers–writers problem, the reader processes share the following data structures:

semaphore **rw_mutex** = 1 →(common to both reader and writer)

semaphore **mutex** = 1 →(to ensure mutual exclusion when read_count is updated)

int **read_count** = 0 →(tracks of how many processes are currently reading the object)

The structure of a **writer** process

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

The structure of a **reader** process

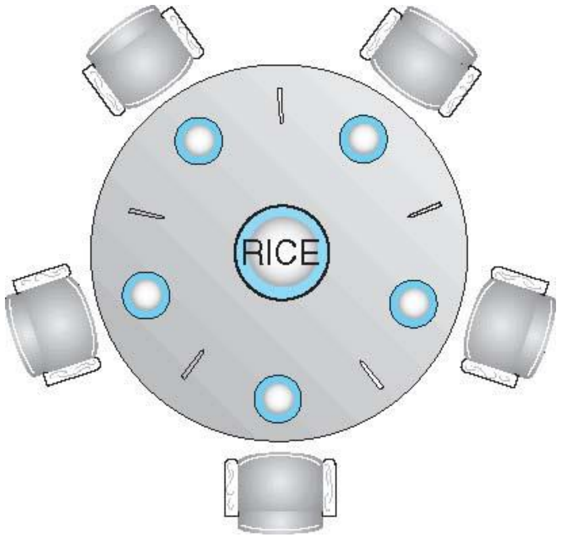
```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
        ...
/* reading is performed */
       ...
        wait(mutex);
        read count--;
        if (read_count == 0)
           signal(rw_mutex);
        signal(mutex);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

Classical Problems of Synchronization

- The Readers-Writers Problem
  - The semaphore **rw_mutex** functions as:
    - A mutual exclusion for the writers processes.
    - It is also used by the first or last reader that enters or exits the critical section (not for other readers).
  - **Notes**:
    - If a writer is in the critical section and n readers are waiting, then one reader is queued on rw_mutex, and n − 1 readers are queued on mutex.
    - When a writer executes signal(rw_mutex), we may resume the execution of either the waiting readers or a single waiting writer.
      - The selection is made by the scheduler.

Classical Problems of Synchronization

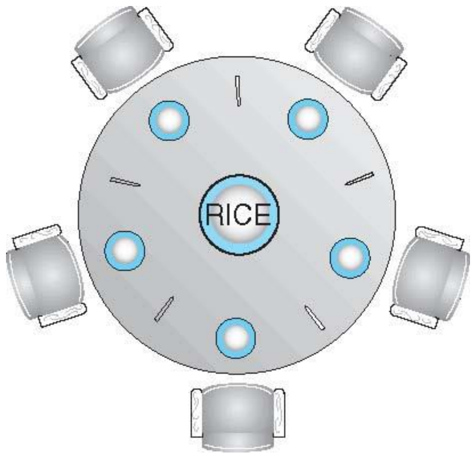
- Dining-Philosophers Problem
  - Philosophers spend their lives alternating between thinking and eating
  - They do not interact with their neighbors, and occasionally they try to pick up 2 chopsticks (one at a time) to eat from bowl of rice
    - They need both (left and right chopsticks) to eat, then they release both when done
  - In the case of 5 philosophers, the shared data:
    - Bowl of rice (data set)
    - Semaphore chopstick [5] → all initialized to 1



Classical Problems of Synchronization

- Dining-Philosophers Problem Algorithm
  - **Solution**: the structure of Philosopher ***i***:

```
do {
     wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );
        /* eat for awhile */
      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );
        /* think for awhile */
} while (TRUE);
```



  - What is the problem with this algorithm? (it creates a deadlock)
  - Deadlock handling:
    - Allow at most 4 philosophers to be sitting simultaneously at the table.
    - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
    - Use an asymmetric solution: an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Classical Problems of Synchronization

- Problems with Semaphores
  - Incorrect use of semaphore operations can result in timing errors that are difficult to detect:
  - **The correct execution of semaphore is as:**

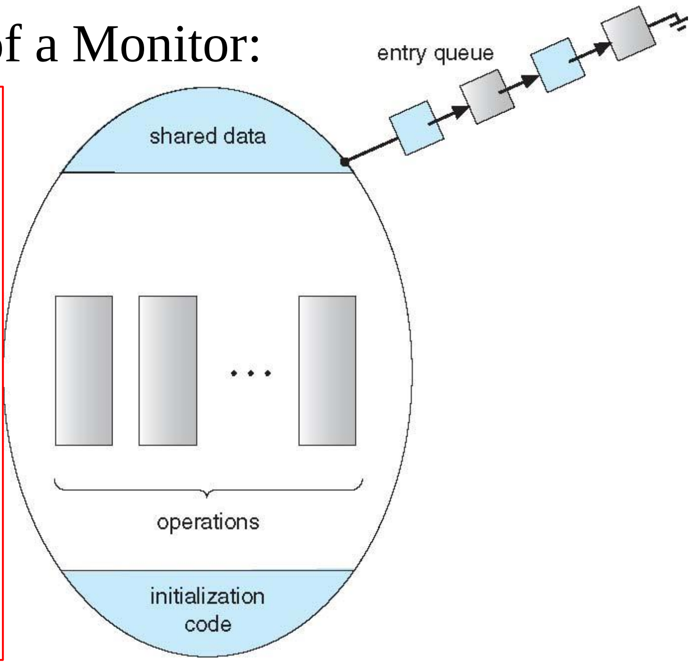
    wait(mutex) critical section signal(mutex)
  - **But if by mistake the order executed as:**
    - signal(mutex) critical section wait(mutex)
      - More than one process are simultaneously active in their critical sections
    - wait(mutex) critical section wait(mutex)
      - Deadlock will occur
    - Omitting of wait(mutex) or signal(mutex) (or both)
      - May be mutual exclusion is violated or a deadlock will occur.
  - Deadlock and starvation are possible.
  - To deal with such errors, researchers have developed high-level language constructs (the monitor type)

Monitors

- Monitor Usage
  - It is a high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - A **monitor type** is an **abstract data type (ADT)**, that includes a set of programmer defined operations and variables that are provided with mutual exclusion within the monitor.
  - Only one process may be active within the monitor at a time
  - But not powerful enough to model some synchronization schemes
    - Therefore, we need to define additional synchronization mechanisms that are provided by the condition construct.
  - Syntax of a monitor and Schematic view of a Monitor:

```
monitor monitor-name
{
  /* shared variable declarations
  function P1 (…)
     { …. }
  function P2 (…)
     { …. }
  :
  function Pn (…)
     { …… }

  Initialization code (…)
     { … }
}
```

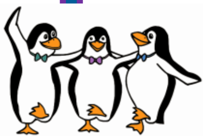


Monitors

- Monitor Usage
  - **Condition Construct** variables
    - A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

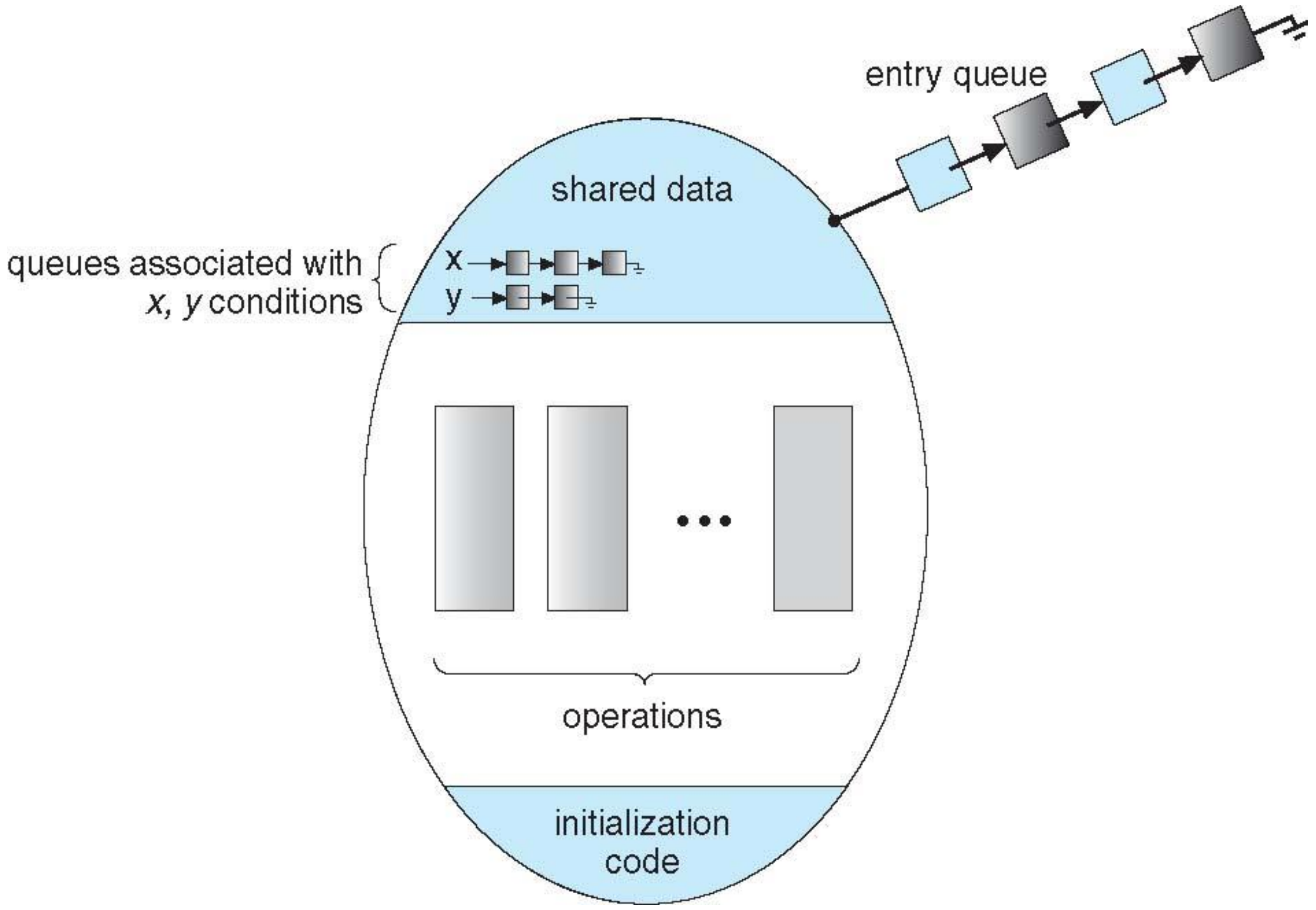
      condition x, y;

    - Two operations (wait and signal) are allowed on a condition variable:
      - **x.wait():** a process that invokes the operation is suspended until **x.signal()**is invoked
      - **x.signal():** resumes exactly one of processes (if any) that invoked **x.wait()**
        - If no **x.wait()** on the variable, then it has no effect on the variable **x**
          - The state of **x** is the same as if the operation had never been executed



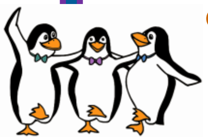
Monitors

- Monitor Usage
  - **Monitor with Condition Variables**



Monitors

- Condition Variables Choices
  - Suppose that a process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
    - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait, otherwise, both P and Q would be active simultaneously within the monitor
  - Two possibilities exist:
    - **Signal and wait:** P waits until Q either leaves the monitor or waits for another condition
    - **Signal and continue:** Q waits until P either leaves the monitor or waits for another condition
  - Both options have pros and cons – language implementer can decide
  - A compromise between these two choices was adopted in the language Concurrent Pascal
    - When P executing the signal(), it immediately leaves the monitor, and Q is resumed
  - Implemented in other languages including Mesa, C# (C-sharp), Java



Monitors

Dining-Philosophers Solution Using Monitors

- The solution is that a philosopher may pick up her chopsticks only if both of them are available.
- Therefore, we need to distinguish among three states in which we may find a philosopher in:

  enum {THINKING, HUNGRY, EATING} state[5];

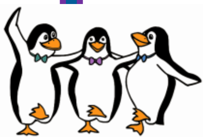
- Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating:

  (state[(i+4) % 5] != EATING) **and** (state[(i+1)% 5] != EATING)

- Each philosopher i must invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup(i);
       ….
         EAT
       ….
DiningPhilosophers.putdown(i);
```

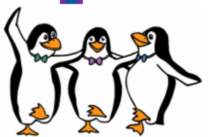
- No deadlock, but starvation is possible



Monitors

- Dining-Philosophers Solution Using Monitors

```
monitor DiningPhilosophers{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];
   void pickup (int i){
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
             self[i].wait;
   }
  void putdown (int i){
          state[i] = THINKING;
         // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);}
   void test (int i){
      if ((state[(i + 4) % 5] != EATING) &&(state[i] == HUNGRY)
                     &&(state[(i + 1) % 5] != EATING) ){
             state[i] = EATING ;
             self[i].signal () ;
      }}
  initialization_code() {
     for (int i = 0; i < 5; i++)
          state[i] = THINKING;
   }}
```



Monitors

- Implementing a Monitor Using Semaphores
  - For each monitor, a semaphore mutex (initialized to 1) is provided.
    - A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
  - A semaphore, next (initialized to 0) is introduced because a signaling process must wait until the resumed process either leaves or waits.
  - An integer variable next_count is also provided to count the number of processes suspended on next.

```
semaphore mutex = 1;
semaphore next = 0;
int next_count = 0;
```

  - For each external function ***F*** will be replaced by

```
wait(mutex);
        …
         body of F;
        …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

  - Mutual exclusion within a monitor is ensured

Monitors

- Monitor Implementation – Condition Variables
  - For each condition variable ***x***, we introduce:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

  - The operation x.wait() can be implemented as:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

  - The operation `x.signal()` can be implemented as:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Monitors

- Resuming Processes within a Monitor
  - If several processes queued on condition x, and x.signal() executed, which process should be resumed?
  - We can use First Come First Served (FCFS), but it is not adequate in most cases
  - Therefore, we can use **conditional-wait** construct of the form **x.wait(c)**
    - Where c is a **priority number**
    - Process with lowest number (highest priority) is scheduled next

Monitors

- Single Resource allocation
  - Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
...
access the resurce;
...
R.release;
```

  - Where R is an instance of type **`ResourceAllocator`**
    - Problem: the monitor concept cannot guarantee that the preceding access sequence will be observed

```
monitor ResourceAllocator
{
 boolean busy;
 condition x;
 void acquire(int time) {
 if (busy)
  x.wait(time);
  busy = TRUE;
 }
 void release() {
  busy = FALSE;
  x.signal();
 }
 initialization code() {
  busy = FALSE;
 }
}
```

Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Synchronization Examples

- Solaris Synchronization
  - Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
  - Uses **adaptive mutexes** for efficiency when protecting data from short code segments
    - Starts as a standard semaphore spin-lock
    - If lock held, and by a thread running on another CPU, spins
    - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
  - Uses **condition variables**
  - Uses **readers-writers** locks when longer sections of code need access to data
  - Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
    - Turnstiles are per-lock-holding-thread, not per-object
  - Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Synchronization Examples

- Windows Synchronization
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses **spinlocks** on multiprocessor systems
    - Spinlocking-thread will never be preempted
  - Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
    - **Events**
      - An event acts much like a condition variable
    - Timers notify one or more thread when time expired
    - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Synchronization Examples

- Linux Synchronization
  - Linux:
    - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
    - Version 2.6 and later, fully preemptive
  - Linux provides:
    - Semaphores
    - atomic integers
    - spinlocks
    - reader-writer versions of both
  - On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Synchronization Examples

- Pthreads Synchronization
  - Pthreads API is OS-independent
  - It provides:
    - mutex locks
    - condition variable
  - Non-portable extensions include:
    - read-write locks
    - spinlocks

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Alternative Approaches

- Transactional Memory
  - A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()
{
        acquire();
        /* modify shared data */
        release();
}
```

Alternative Approaches

- OpenMP
  - OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
 {
       #pragma omp critical
       {
             count += value
       }
  }
```

The code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.

Alternative Approaches

- **Functional Programming Languages**
  - Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.
  - Variables are treated as immutable and cannot change state once they have been assigned a value.
  - There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.