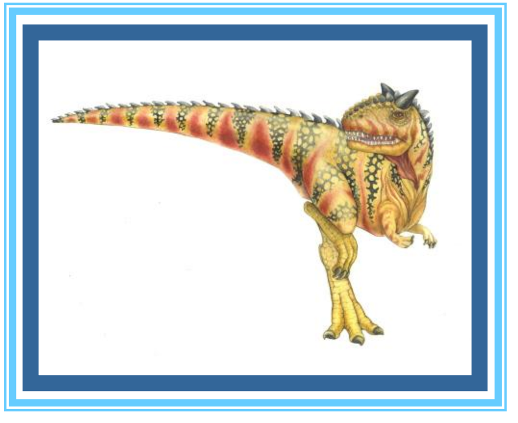
Chapter 6: Process Synchronization



Chapter 6: 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

- Processes can execute concurrently
  - 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.

Producer & Consumer

- Producer

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

- Consumer

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

Race Condition

- counter++ could be implemented as

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

In assembly,

```
MOV EAX, DWORD PTR DS:[ESI]
INC EAX
MOV DWORD PTR DS:[ESI], EAX
```

- counter-- could be implemented as

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

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

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

Critical Section Problem

- Consider a system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process is in critical section, no others may be in its critical section
- ***Critical section problem*** is to design protocol to solve it
- Each process must
  - Ask permission to enter critical section in **entry section**,
  - May follow critical section with **exit section**
  - Then **remainder section**

Critical Section

- General structure of process **Pi**

do {

entry section

critical section

exit section

remainder section

} while (true);

Algorithm for Process P_i & P_j

```
do {

     while (turn == j)
        ; /* do nothing */

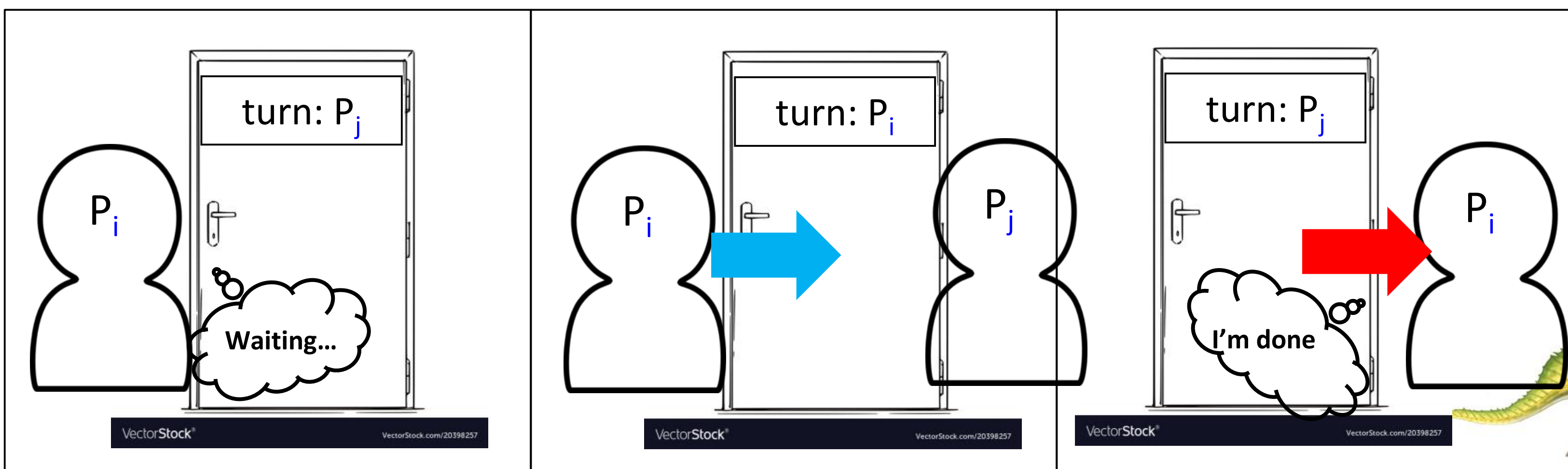
            critical section
     turn = j;

            remainder section

 } while (true);
```

P_j is working in the critical section (turn == j) so P_i should wait

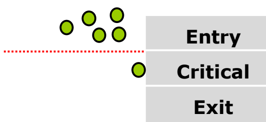
P_i finishes its task and returns a turn to access the critical section to P_j



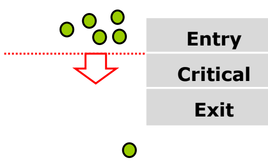
Solution to Critical-Section Problem

- Three requirements:

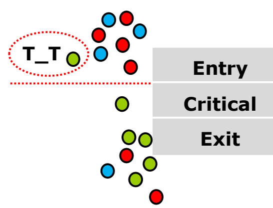
1. **Mutual Exclusion** - If process **Pi** is executing in its critical section, then no other processes can be executing in their critical sections



2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next 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 relative speed of the n processes

Critical-Section Handling in OS

- Two approaches exist depending on the kernel policy
  - **Preemptive kernel** – allows preemption of another process when a process is running in kernel mode
  - **Nonpreemptive kernel** – runs a process until it exits kernel mode, blocks, or voluntarily yields CPU

    - Essentially free of race conditions on kernel data structures, as only one process is active in the kernel at a time
  - Preemptive kernel is especially difficult to design for Symmetric Multiprocessing (SMP) architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors

    - A preemptive kernel may be more responsive, and more suitable for real-time programming

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
  - Assume that the *load* and *store* machine-language instructions are **atomic**; that is, cannot be interrupted
  - The two processes share two variables:

    - int turn;

    - boolean flag[2]

  - The variable turn indicates whose turn it is to enter the critical section
  - The flag array is used to indicate if a process is ready to enter the critical section. *flag[i] = true* implies that process **Pi** is ready!

Peterson's Solution (Cont.)

- Algorithm for Process Pi & Pj

```
do {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn == j)
            ; /* do nothing */
                critical section
        flag[i] = false;
                remainder section
 } while (true);
```

- Provable that the three CS requirements are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either **flag[j] == false** or **turn == i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts (sei() / cli())
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this is not broadly scalable
- Modern machines provide special atomic hardware instructions

  Atomic = non-interruptible
  - Either test memory word and set value or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

```
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".

Solution using test_and_set()

- Shared boolean variable *lock*, initialized to FALSE
- Solution:

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

- Are the three CS requirement met?

  Mutual Exclusion
  Progress
  Bounded Waiting

compare_and_swap Instruction

Definition:

```
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.

Solution using compare_and_swap

- Shared integer "*lock*" initialized to 0;
- Solution:

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

- Are the three CS requirement met?

Mutual Exclusion
Progress
Bounded Waiting

Bounded-waiting Mutual Exclusion 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);
```

P_i waits for opening the lock using its key (TRUE). When the lock is available, then the key becomes FALSE and the lock is disabled as TRUE.

P_i seeks for the next process which is waiting for the available lock.

P_i hands the key over to P_j by setting the waiting[j] as FALSE. Through these steps, the lock is still disabled as TRUE.

- Are the three CS requirement met?

Mutual Exclusion

Progress

Bounded Waiting

Mutex Locks

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

acquire() and release()

- ```
  acquire() {
        while (!available)
         ; /* busy wait */
     available = false;;
   }
  ```
- ```
  release() {
     available = true;
   }
  ```
- ```
  do {
    acquire lock
     critical section
    release lock
    remainder section
  } while (true);
  ```
  ```

Semaphore

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

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

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

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

Semaphore Usage

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

P1:

    S_1;

    signal(synch);

$\longrightarrow$ **Synch -> 1**

P2:

    wait(synch);

$\longrightarrow$ **Synch -> 0**

    S_2;

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

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

Semaphore Implementation with no Busy waiting

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

Implementation with no Busy waiting (Cont.)

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

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

Deadlock and Starvation

- **Deadlock** – the state where 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	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

S0: P0 wait(S) {S = 1 -> S = 0}
S1: C.S.
S2: P1 wait(Q) {Q = 1 -> Q = 0}
S3: P1 wait(S) {S = 0 -> S = -1, block}
S4: C.S.
S5: P0 wait(Q) {Q = 0 -> Q = -1, block}
S and Q become deadlock

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when a lower-priority process holds a lock needed by higher-priority process
  - E.g., priority *L* < *M* < *H*. Process *L* holds a lock, process *H* waits for *L*, process *M* preempts *L*, i.e., *M* affects *H*
  - Solved via **priority-inheritance protocol**: Allow *L* holding a lock to temporarily inherit the priority of *H*, thereby preventing *M* from preempting *L*'s execution

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n

- A producer holds **empty** for acquiring an empty buffer and a consumer holds **full** for acquiring a filled buffer
  - One of them then tries to hold a **mutex** for mutual exclusive access to the buffer structure

Bounded Buffer Problem (Cont.)

- 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);
```

Bounded Buffer Problem (Cont.)

- 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);
```


Readers-Writers Problem

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

- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read_count** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

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.

- The structure of a reader process

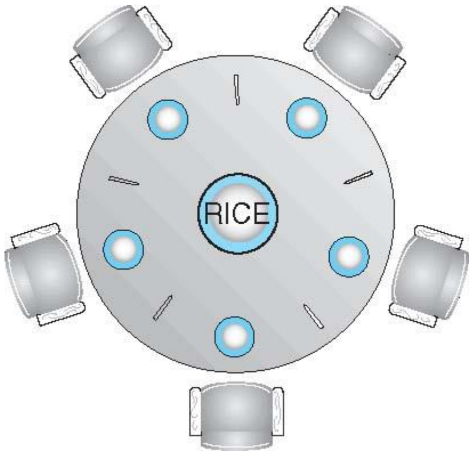
```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1)
        wait(rw_mutex);
     signal(mutex);
        ...
     /* reading is performed */
        ...
     wait(mutex);
     read_count--;
     if (read_count == 0)
        signal(rw_mutex);
     signal(mutex);
} while (true);
```

- Any problem?

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object.
  - In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- ***Second*** variation – once writer is ready, it performs the write ASAP
  - In other words, if a writer is waiting to access the object, no new readers may start reading
- Both may have starvation leading to even more variations
- On some systems, kernel provides **reader-writer** locks for the problem
  - Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.
  - Require more overhead to establish than semaphores or mutual-exclusion locks

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
  - An approximated model to describe processes requiring two resources to enter their critical sections
- In the case of 5 philosophers
  - Shared data

    - Bowl of rice (data set)

    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

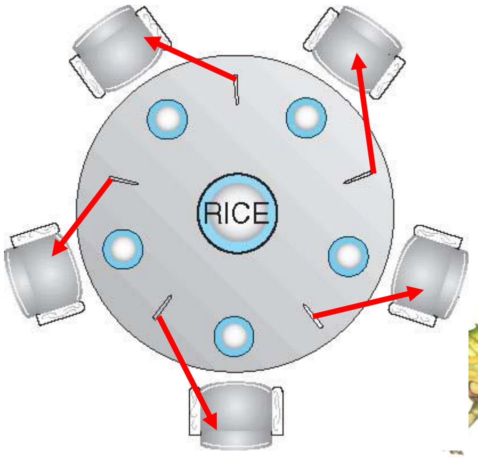
        //  eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

        //  think

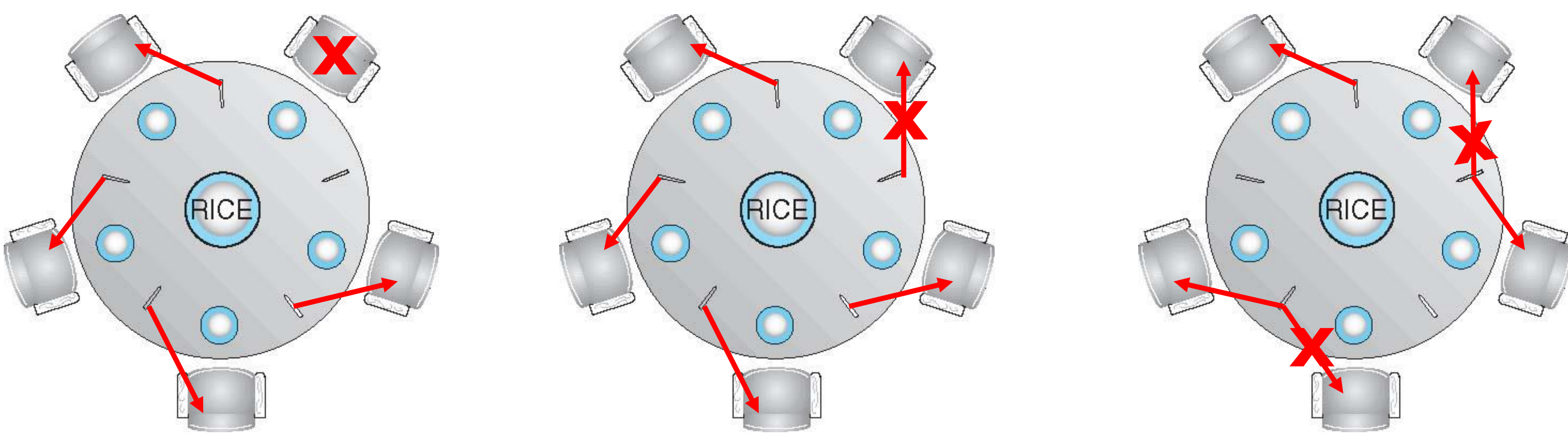
} while (TRUE);
```

- What is the problem with this algorithm?



Dining-Philosophers Problem Algorithm (Cont.)

- 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.



Problems with Semaphores

- Incorrect use of semaphore operations:

  Case1: signal (mutex) …. wait (mutex)

  Case2: wait (mutex) … wait (mutex)

  Case3: Omitting of wait (mutex) or signal (mutex) (or both)

- Deadlock and starvation are possible.
- To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct—the **monitor** type.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type (ADT)* encapsulates internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
  - That is, ADT includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor
- But not powerful enough to model some synchronization schemes

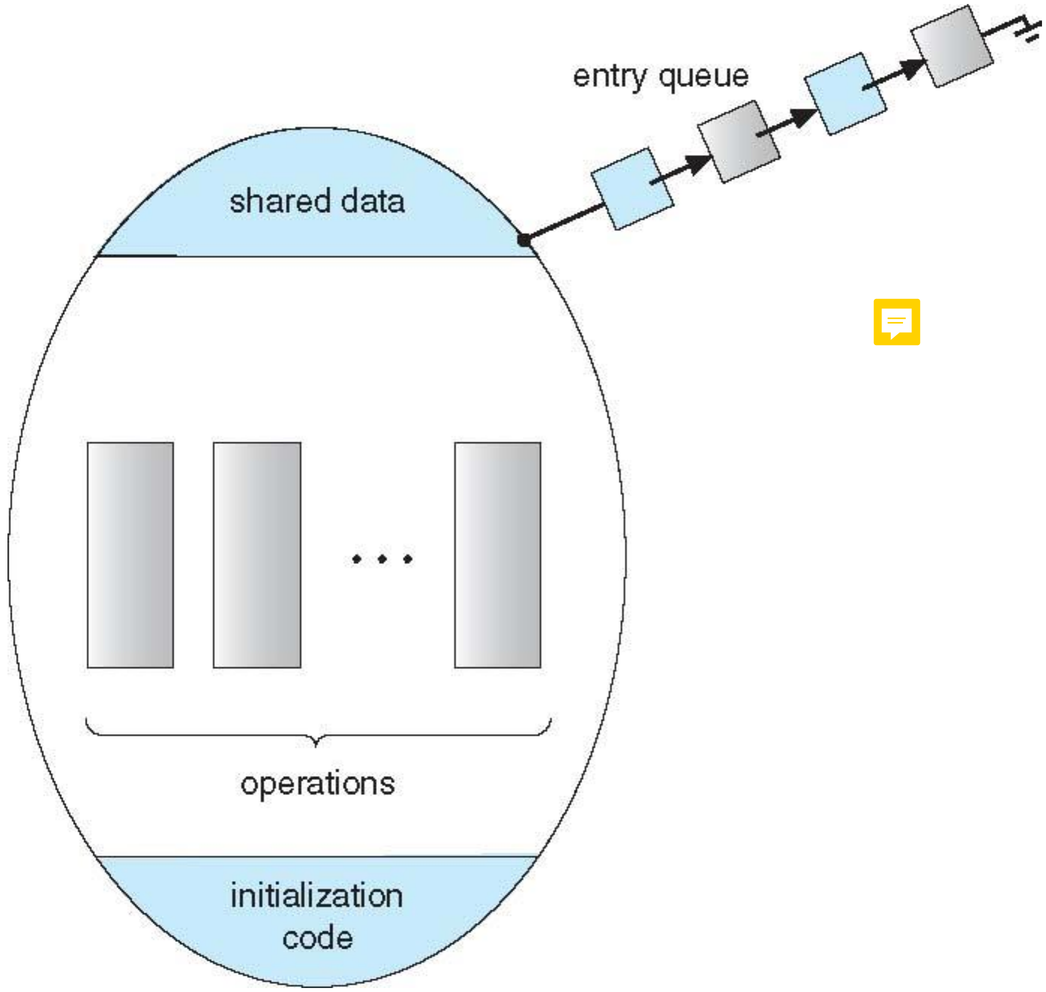
```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }

   procedure Pn (…) {……}

   Initialization code (…) { … }

}
```

Schematic view of a Monitor

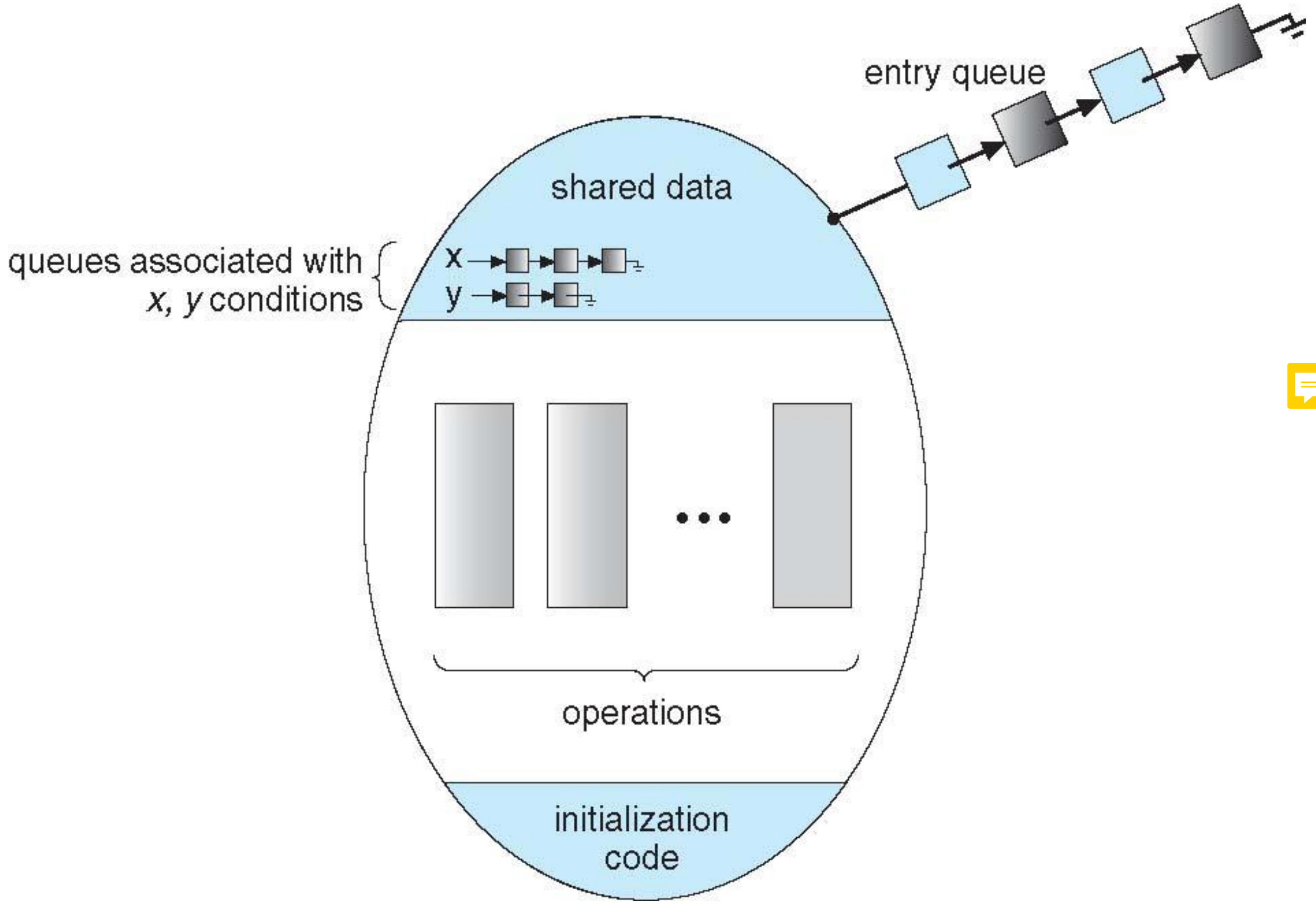


Condition Variables

- **condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**

    - If no **x.wait()** on the variable, then it has no effect on the variable (cf. Contrast this operation with the **signal()** operation associated with semaphores, which always affects the state of the semaphore)

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **x.signal(),** and process Q is suspended in **x.wait()**, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait

- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Solution to Dining Philosophers

```
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;
    }
}
```

Solution to Dining Philosophers (Cont.)

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

```
DiningPhilosophers.pickup(i);

        EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially  = 1)
semaphore next;  // (initially  = 0)
int next_count = 0;
```

for each monitor

Since a signaling process must wait until the resumed process either leaves or waits

# of processes suspended on **next**

- Each procedure ***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

Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
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--;
```

Monitor Implementation (Cont.)

- The operation **x.signal** can be implemented as:

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

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

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 (shortest time-allocation has the highest priority).

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

R.release;
```

- Where R is an instance of monitor type **ResourceAllocator** (next slide)

A Monitor to Allocate Single Resource

```
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

Solaris Synchronization

- Implements a variety of locks (adaptive mutex locks, condition variables, semaphores, reader–writer locks, and turnstiles) 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 the lock is held by a thread running on another CPU, it spins because the lock ‘might’ be released soon
  - If the lock is held by non-run-state thread (waiting for signal), then it is blocked and waits for the lock being released
- Uses **reader-writer** locks when longer sections of code need access to data

Solaris Synchronization (cont.)

- Uses **turnstiles** (queues) 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
  - The turnstile for the first thread to block on a synchronized object becomes the turnstile for the object. Threads subsequently blocking on the lock will be added to this turnstile
  - Priority-**inheritance** per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- **Kernel** uses **spinlocks** only to protect short code segments on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Outside the kernel, provides **dispatcher objects** which may act mutexes, semaphores, events, and timers
  - Gain ownership of a mutex to access the data and release the ownership when it is finish
  - Semaphores behave as described
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects are either of **signaled-state** (object available) or **non-signaled state** (thread will block)

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
  - Mutex locks
  - spinlocks & semaphores (reader-writer versions of both)
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- Pthreads standard provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - semaphores (read-write locks)
  - spinlocks

Alternative Approaches

- Transactional Memory
  - commit, abort & roll back
- OpenMP
  - critical-section compiler directive
- Functional Programming Languages
  - Most of the problems addressed in this chapter are nonexistent in functional languages

Transactional Memory

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

```
void update()
{
    atomic {
        /* read/write memory */
    }
}
```

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.

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.

End of Chapter 6