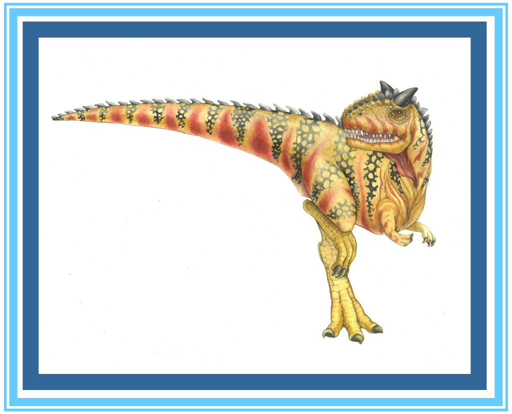
Chapter 5: Process Synchronization



Chapter 5: 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
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

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

Concurrent Access to Shared Data

- **Suppose that two processes A and B have access to a shared variable “Balance”.**

PROCESS A:	**PROCESS B:**
Balance = Balance - 100	**Balance = Balance - 200**

- **Further, assume that Process A and Process B are executing *concurrently* in a time-shared, multiprogrammed system.**

Concurrent Access to Shared Data

- The statement “ Balance = Balance – 100” is implemented by several machine level instructions such as:

```
A1.  LOAD R1, BALANCE   // load Balance from memory
                                     into Register 1 (R1)
A2. SUB R1, 100          // Subtract 100 from  R1
A3. STORE BALANCE, R1  // Store R1’s  contents back to
                           the memory location of Balance.
```

- Similarly, “Balance = Balance – 200” can be implemented by the following:

```
B1. LOAD R1, BALANCE
B2. SUB R1, 200
B3. STORE BALANCE, R1
```

Race Conditions

- **Observe:** In a *time-shared* system the *exact instruction execution order* cannot be predicted!

- Scenario 1:

  A1. LOAD R1, BALANCE

  A2. SUB R1, 100

  A3. STORE BALANCE, R1

  Context Switch!

  B1. LOAD R1, BALANCE

  B2. SUB R1, 200

  B3. STORE BALANCE, R1

- Balance is decreased by 300!

- Scenario 2:

  A1. LOAD R1, BALANCE

  A2. SUB R1, 100

  Context Switch!

  B1. LOAD R1, BALANCE

  B2. SUB R1, 200

  B3. STORE BALANCE, R1

  Context Switch!

  A3. STORE BALANCE, R1

- Balance is decreased by 100!

Race Condition

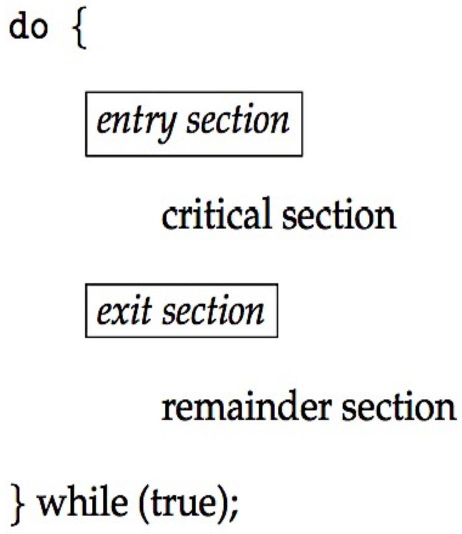
- We would arrive at this incorrect state because we allowed **both** processes to manipulate the variable **BALANCE** **concurrently.**
- This situation is called a **race condition.**
- To guard against the race condition above, we need to ensure that **only** one process at a time can be manipulating the variable BALANCE .
- Situations such as the one just described occur frequently in Applications and **operating systems** as different parts of the system manipulate **shared** resources.
- In **multicore** systems, an increased emphasis on developing multithreaded applications. In such applications, several threads—which are quite possibly sharing data—are running in parallel on different processing cores.
- we require that the processes be **synchronized** in some way.

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 **Shared** (items) like: **common variables**, updating **table**, writing **file**, etc
  - When one process in **critical** section, **no** other may be in its critical section
- ***Critical section problem*** is to design **protocol** to solve this.
- 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 ***P_i***



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

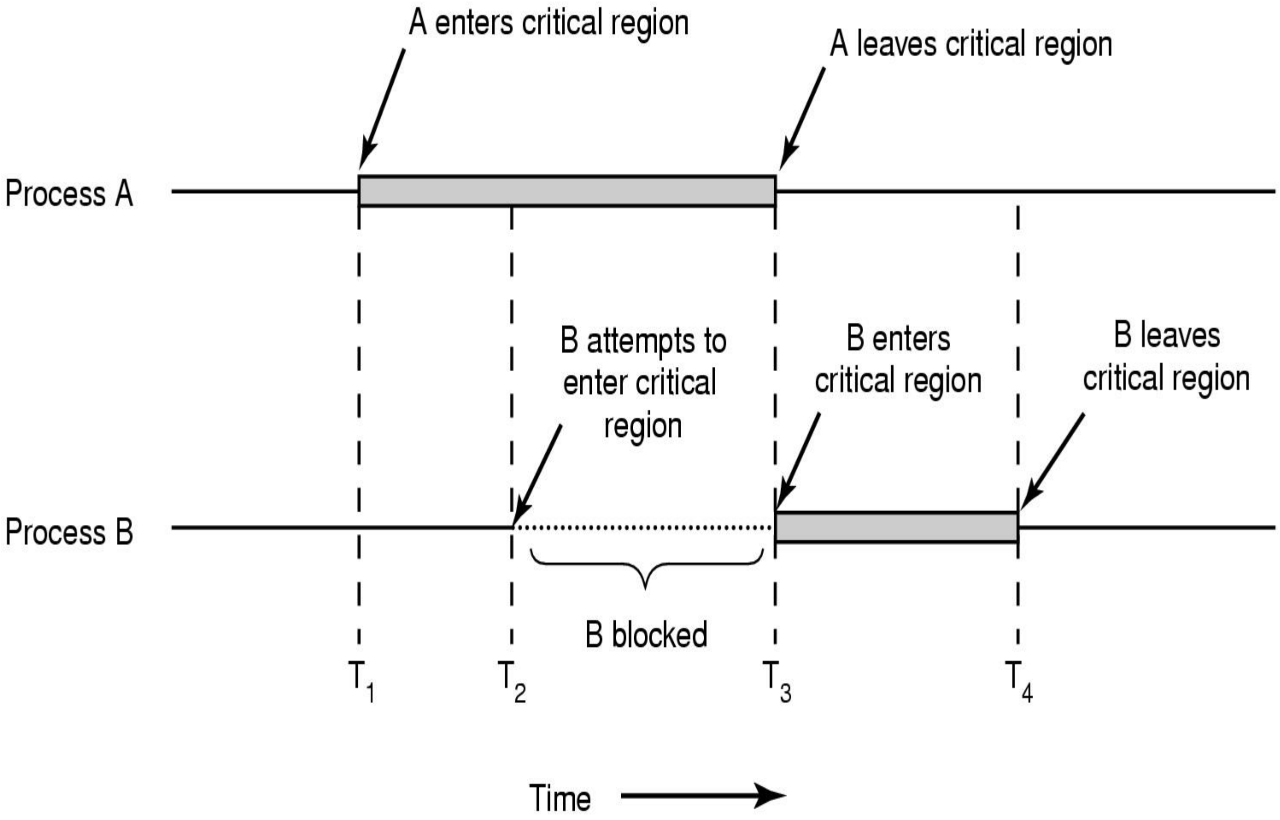
Algorithm for Process P_i

```
do {

    while (turn == j);

             critical section

    turn = j;

             remainder section

 } while (true);
```

Solution to Critical-Section Problem

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:**
   - **No process running outside its critical section may block any process from entering its Critical section.**
   - the **selection** of the processes that will enter the critical section **next** cannot be postponed **indefinitely.**
3. **Bounded Waiting** - A bound (**limit**) 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.

Mutual Exclusion



Critical-Section Handling in OS

Two approaches depending on if **kernel** is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in **kernel** mode
- **Non-preemptive** – runs until **exits** kernel mode, **blocks**, or voluntarily **yields** (gives up) CPU
  - Essentially free of race conditions in kernel mode

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 P_i is ready!

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);
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement 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: **Pi** will enter the critical section (progress) after **at most one** entry by **Pj**.

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**
  - Currently running code would execute without **preemption**
  - Generally too inefficient on **multiprocessor** systems
    - Operating systems using this are 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

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”.

Entering and leaving a critical region using the TSL instruction

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was not zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Mutual Exclusion 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);
```

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.

Entering and leaving a critical region using the XCHG instruction

```
enter_region:
        MOVE REGISTER,#1                    | put a 1 in the register
        XCHG REGISTER,LOCK                  | swap the contents of the register and lock variable
        CMP REGISTER,#0                     | was lock zero?
        JNE enter_region                    | if it was non zero, lock was set, so loop
        RET                                 | return to caller; critical region entered


leave_region:
        MOVE LOCK,#0                        | store a 0 in lock
        RET                                 | return to caller
```

Mutual Exclusion 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);
```

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);
```

Mutex Locks

- Previous solutions are **complicated** and generally inaccessible to application **programmers.**
- OS designers build software **tools** to solve critical section problem.
- Simplest is **mutex** lock. (mutex is from Mutual exclusion)
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be **atomic**
  - Usually implemented via hardware **atomic** instructions
- But has a main disadvantage (**busy waiting).**
- This lock therefore called a **spinlock,** because the process "spins" while waiting for the lock to become available.

Solution to Critical-section Problem Using Locks

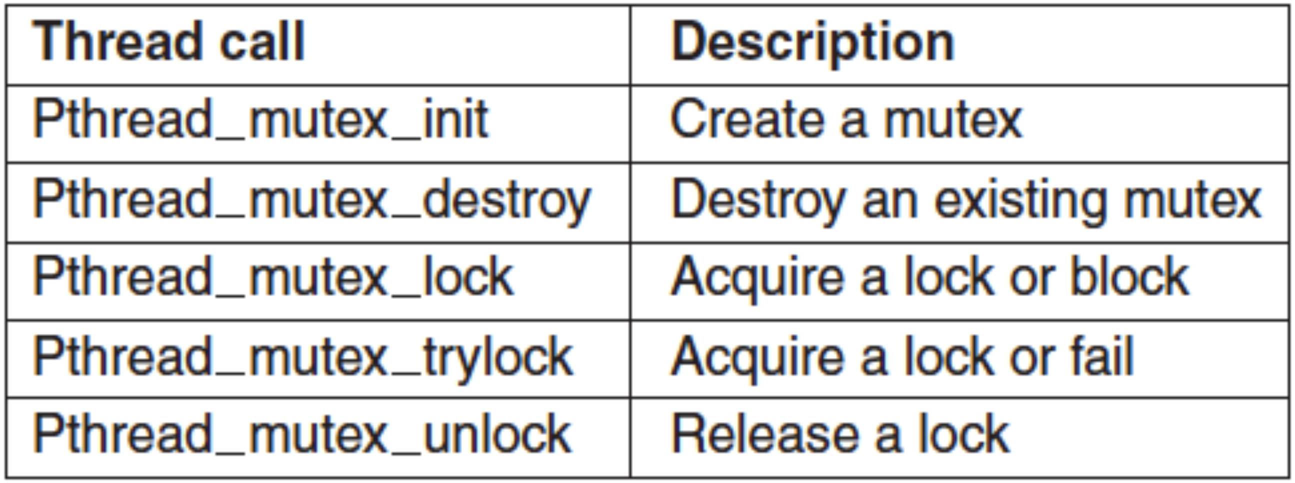
```
do {
     acquire lock
                critical section
     release lock
                remainder section
} while (TRUE);
```

acquire() and release()

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

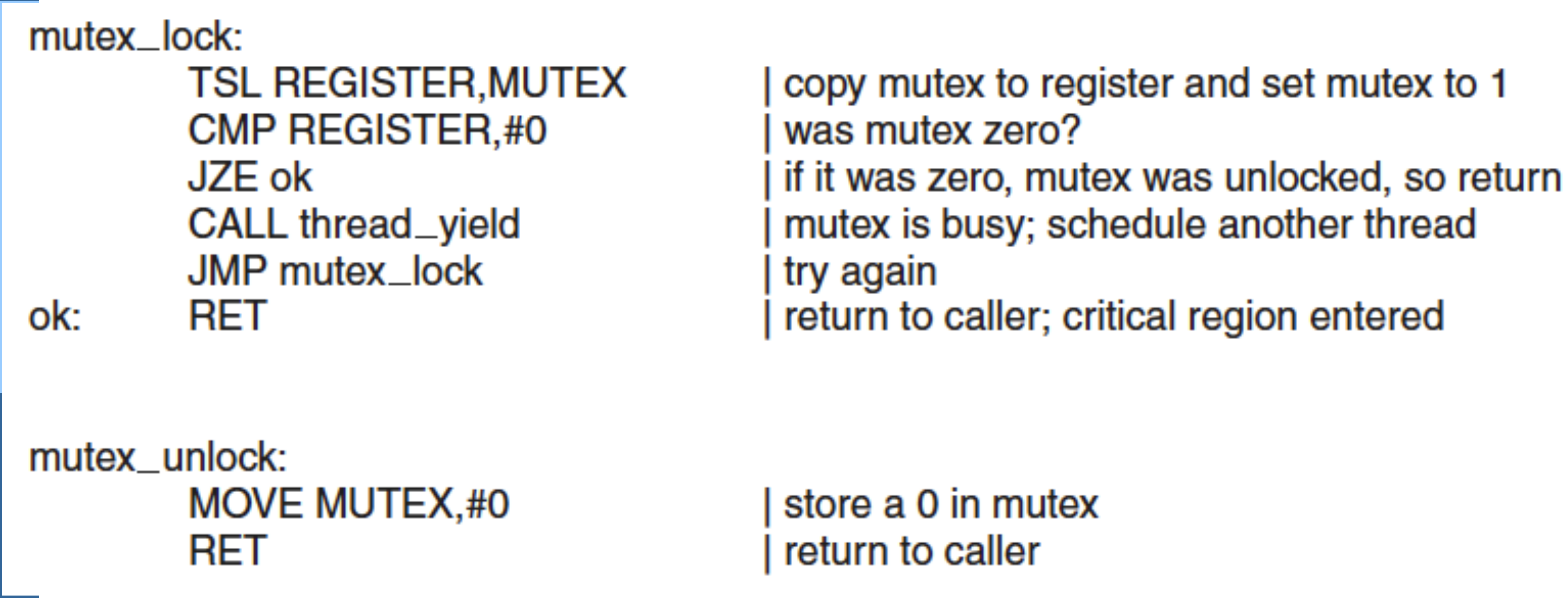
-
  ```
  release() {
      available = true;
  }
  ```

Some of the Pthreads calls relating to mutexes



Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Implementation of mutex lock and mutex unlock



```
mutex_lock:
          TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
          CMP REGISTER,#0             | was mutex zero?
          JZE ok                      | if it was zero, mutex was unlocked, so return
          CALL thread_yield           | mutex is busy; schedule another thread
          JMP mutex_lock              | try again
ok:       RET                         | return to caller; critical region entered


mutex_unlock:
          MOVE MUTEX,#0               | store a 0 in mutex
          RET                         | return to caller
```

Semaphore

- **Synchronization** tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (**atomic**) operations
  - **`wait()`** and **`signal()`**
- Each process that wishes to **use** a resource performs a **wait**() operation on the semaphore, thus decrementing the count.
- When a process **releases** a resource, it performs a **signal**() operation on the semaphore, thus incrementing the count.

Semaphore

- 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:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- 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
  - These two operations are provided by the operating system as basic system **calls**.

Semaphore Definition with no Busy waiting

```
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** – 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);
```

- **Starvation** – **indefinite blocking**
  - A process may **never** be removed from the semaphore queue in which it is **suspended**
- **Priority Inversion** – Scheduling problem when **lower-priority** process holds a **lock** needed by **higher-priority** process
  - Solved via **priority-inheritance protocol**

Priority Inversion

- **Priority Inversion** – Scheduling problem when **lower-priority** process holds a **lock** needed by **higher-priority** process
- Solved via **priority-inheritance protocol:**
  - all processes that are accessing resources needed by a higher-priority process **inherit** the higher priority until they are finished with the resources.
  - When they are finished, their priorities **revert** to their original values.

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

- Producer produces and stores in buffer, Consumer consumes from buffer.
- Trouble when – Producer produces, but buffer is **full** – Consumer consumes, but buffer is **empty**.
- ***Solution with Semaphore:*** both Consumer and Producer share the following Data Structure:
- ***int n, indicates n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - provides **mutual exclusion** for accesses to the **buffer** pool.
- Semaphore **`full`** initialized to the value 0
  - count the number of full buffers
- Semaphore **`empty`** initialized to the value n
  - count the number of empty buffers

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

- Suppose that a database is to be **shared** among several concurrent processes.
- Some of these processes may want only to **read** the database.
- Others may want to **update** (that is, to **read** and **write**) the database.
- We distinguish between these two types of processes by referring to the former as **readers** and to the latter as **writers.**
- **No problem with only Reading.**
- We require that the **writers** have exclusive access to the shared database while writing to the database.

Readers-Writers Problem

- **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**
- The simplest one, referred to as the **first** readers–writers problem:
  - no reader be kept waiting unless a writer has already obtained permission to use the **shared** object.

the *First* Readers-Writers Problem

- **Shared Data:**
  - Semaphore `rw_mutex` initialized to 1
    - mutual exclusion semaphore for the **writers**.
  - Semaphore `mutex` initialized to 1
    - used to ensure mutual exclusion when the variable `read_count` is updated.
  - Integer `read_count` initialized to 0
    - keeps track of how many processes are currently **reading** the object.

The *First* Readers-Writers Problem (Cont.)

- The structure of a **writer** process

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

The *First* Readers-Writers Problem (Cont.)

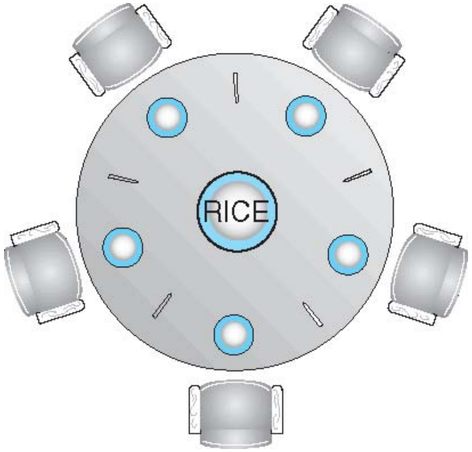
- The structure of a **reader** process

```
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);
```

Readers-Writers Problem Variations

- ***First*** variation – **no** reader kept waiting **unless** writer has permission to use **shared** object.
- ***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
- Problem is solved on some systems by kernel providing **reader-writer 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
- 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 **forks** 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

- Although semaphores provide a convenient and effective mechanism for process synchronization:
  - using them **incorrectly** can result in timing errors that are difficult to detect.
- **Incorrect** use of semaphore operations, a Process **change** order:
  - signal (mutex) /Critical section/ wait (mutex)
    - In this situation, several processes maybe executing in their **critical** sections **simultaneously**, violating the **mutual-exclusion** requirement.
  - wait (mutex) /Critical section/ wait (mutex)
    - In this case, a **deadlock** will occur.
  - **Omitting** of wait (mutex) or signal (mutex) (or both)
    - In this case, **either mutual** exclusion is violated or a **deadlock** will occur.
- **starvation** is possible.

Monitors

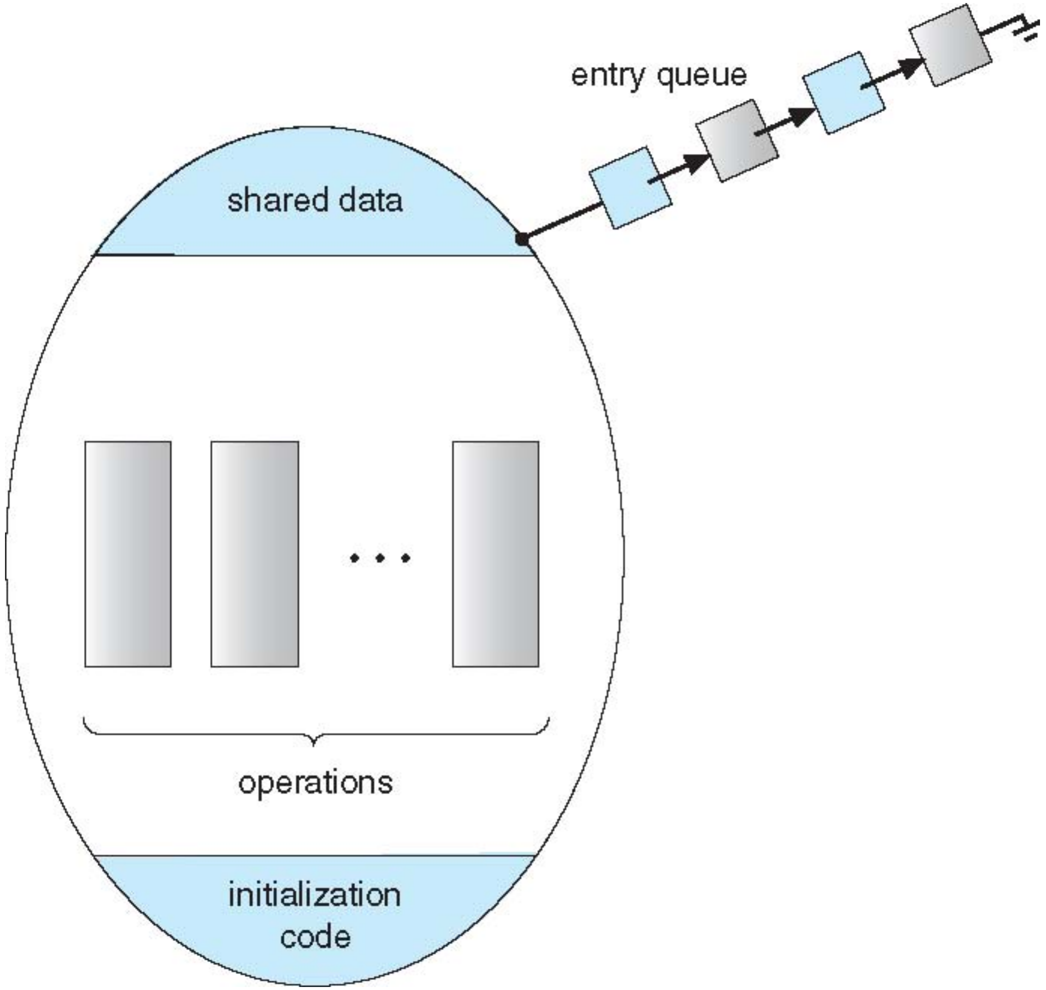
- To deal with such semaphore errors, researchers have developed **high-level language Constructs**.
- recall: An **abstract data type—or ADT**—encapsulates data with a set of functions to operate on that data that are **independent** of any specific implementation of the ADT.
- A ***monitor*** type is an ADT that includes a set of programmer defined **operations** that are provided with **mutual** exclusion within the monitor.
- The monitor type also declares the **variables** whose values define the **state** of an instance of that type, along with the bodies of functions that operate on those variables.
- a function defined within a monitor can access **only** those **variables** declared locally within the monitor and its formal parameters.
- The local variables of a monitor can be accessed by **only** the local functions.

Monitors

- Only **one** process may be **active** within the monitor at a time.
- But **not** powerful enough to model some synchronization schemes

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

  procedure Pn (…) {……}

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

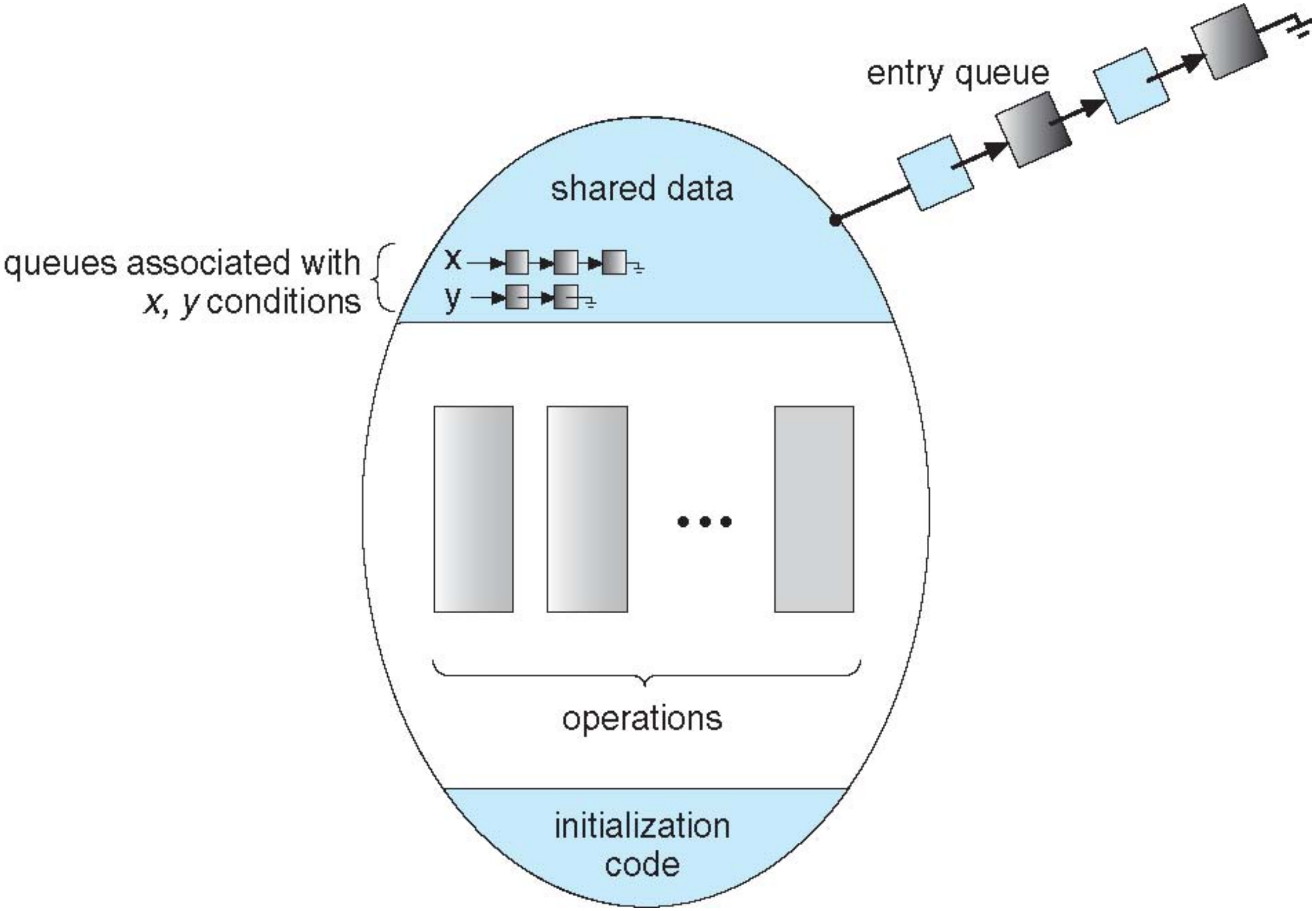
Schematic view of a Monitor



adding Condition Variables to Monitor

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until another process invokes **`x.signal()`**
  - **`x.signal()`** – resumes exactly one suspended process.
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

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. since P was already executing in the monitor
  - 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
- Monitor is Implemented in other languages including 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);
}
```

Solution to Dining Philosophers (Cont.)

```
      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;
```

- 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

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

 R.release;
```

- Where R is an instance of type `ResourceAllocator`

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 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

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)

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

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

Transactional Memory

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

```
void update()
 {
        /* 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 5