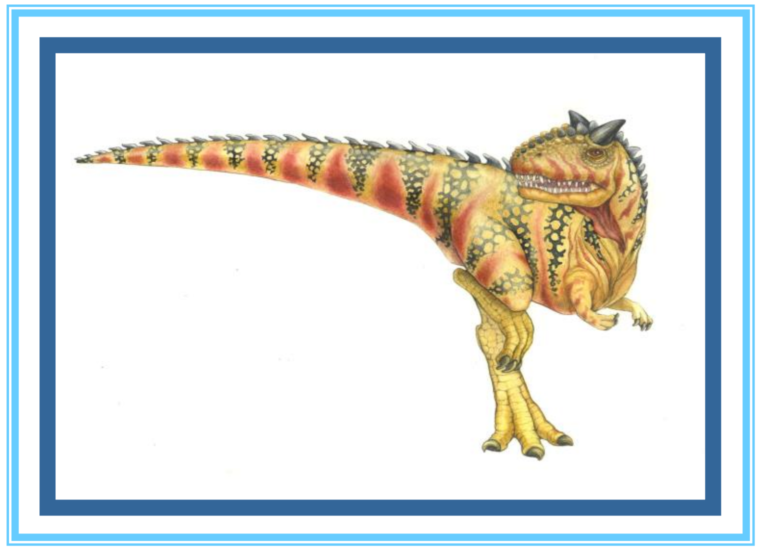
Chapter 6: Process Synchronization



Chapter 6: Process Synchronization

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

Objectives

- Introduce critical-section problem, whose solutions can be used to ensure the consistency of shared data
- Present both software and hardware solutions for the critical-section problem
- Examine several classical process-synchronization problems
- Explore tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - Processes may be interrupted at any time, partially completing execution, due to a variety of reasons.
- Concurrent access to any shared data may result in data inconsistency
- Maintaining data consistency requires mechanism(s) to ensure the orderly execution of cooperating processes

Illustration of the Problem

- Think about the Producer-Consumer problem
- An integer **`counter`** that keeps track of the number of buffers occupied.
  - Initially, **`counter`** is set to 0
  - It is incremented each time by the producer after it produces an item and places in the buffer
  - It is decremented each time by the consumer after it consumes an item in the buffer.

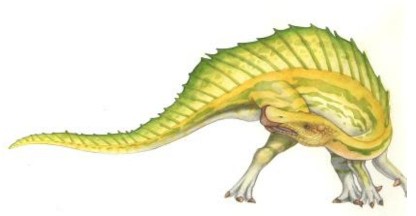
Producer-Consumer Problem

```
while (true) {
      /* produce an item in next produced */

      while (counter == BUFFER SIZE) ;
            /* do nothing */
      buffer[in] = next produced;
      in = (in + 1) % BUFFER SIZE;
      counter++;
}
```

Producer

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

Consumer



Race Condition

- `counter++` could be implemented 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:

```
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

- A Race Condition is a undesirable situation where several processes access and manipulate a shared data concurrently and the outcome of the executions depends on the particular order in which the accesses or executions take place

- Consider a system with *n* processes {P_0, P_1, … P_{n-1}}

- A process has a Critical Section segment of code (short or long), during which
  - A process may be changing shared variables, updating a table, writing a file, and etc.
  - We need to ensure when one process is in Critical Section, no other may be in its critical section
  - In a way mutual exclusion and critical section imply the same thing

- Critical section problem is to design protocol to solve this
  - Specifically, each process must ask permission before entering the critical section in entry section, may follow critical section with exit section, then remainder section

Critical Section

- General structure of process p_i is

```
do {

     entry section

          critical section

     exit section

          remainder section

} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, 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 sections, the selection of a process that will enter the critical section next cannot be postponed indefinitely – selection of one process entering

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 – any waiting process

- Assume that each process executes at a nonzero speed
- There is NO assumption concerning relative speed of the ***n*** processes

Critical-Section Problem in Kernel

- Kernel code (the code implementing an operating system) is subject to several possible race conditions
  - A kernel data structure that maintains a list of all open files can be updated by multiple kernel processes, i.e., two processes were to open files simultaneously
  - Other kernel data structures such as structures maintaining memory allocation, process lists, for interrupt handling and etc.
- Two general approaches are used to handle critical sections in operating system depending on whether the kernel is preemptive or non-preemptive
  - Preemptive – allows preemption of process when running in the kernel mode, not free from the race condition, and increasingly more difficult in SMP architectures.
  - Non-preemptive – runs until exiting the kernel mode, blocks, or voluntarily yields CPU. This is essentially free of race conditions in the kernel mode

Peterson's Solution

- A classical software-based solution.
- This provides a good algorithmic description of solving the critical-section (CS) problem
- Two-process solution
- Assume that the **`load`** and **`store`** 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 (which process) 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 and requests to enter the CS

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);
```

P_0

```
do {
              flag[0] = true;
              turn = 1;
              while (flag[1] && turn == 1);
                     critical section
              flag[0] = false;
                     remainder section
       } while (true);
```

P_1

```
do {
              flag[1] = true;
              turn = 0;
              while (flag[0] && turn == 0);
                     critical section
              flag[1] = false;
                     remainder section
       } while (true);
```

Peterson's Solution – Proof

- **Mutual exclusion**: P_i enters its critical section only if either `flag[j]==false` or `turn ==i`. If both processes are trying to enter the critical section `flag[0]==flag[1] == true`, the value of `turn` can be either `0` or `1` but not both

- P_i can be prevented from entering its critical section only if it is stuck in the `while loop` with the condition `flag[j]==true` and `turn==j;`

- If P_j is not ready to enter the critical section, then `flag[j]==false` and P_i can enter its critical section.

- If P_j is inside the critical section, once P_j exits its critical section, it will reset `flag[j]` to `false`, allowing P_i can to enter its critical section. If P_j resets `flag[j]` to `true`, it must also set `turn` to `i`. Thus since P_i does not change the value of the variable `turn` while executing the `while` statement, P_i can will enter its critical section (**progress**) after at most one entry (**bounded waiting**)

Peterson' s Solution – Discussion

- The solution works to protect "Critical Section" part of the codes, but
  - It is really complex even for a simple example
  - Codes are different for different threads, what if there are many
  - It involves "busy waiting", when one thread is inside "Critical Section", the other has to wait, wasting CPU cycles

- OS provides better solutions (high level primitives) beyond **`load`** and **`store`**

Synchronization Tools

- Operating systems provide hardware and high level API support for critical section code

Programs	Share Programs
High level APIs	`Locks, Semaphores, Monitors`
Hardware	`Load/Store, Disable Interrupts, Test&Set, Compare&Swap`

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS builds software tools to solve the critical section problem
- The simplest tool that most OSes use is mutex lock
- To access the critical regions with it by first `acquire()` a lock then `release()` it
  - Boolean variable indicating if 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
  - Spinlock wastes CPU cycles due to busy waiting, but it does have one advantage in that no context switch is required when a process must wait on a lock, and a contest switch may take considerable time. Thus, when locks are expected to be held for short times, spinlock is useful
  - Spinlocks are often used in multiprocessor systems where one thread can “spin” on one processor while another thread performs its critical section on another processor

acquire() and release()

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

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

Solutions based on the idea of **lock** to protect critical section

- Operations are **atomic** (non-interruptible) – at most one thread acquires a lock
- Lock before entering critical section for accessing share data
- Unlock upon departure after accessing shared data
- Wait if locked - all synchronization involves waiting, should "sleep" if wait for a long time

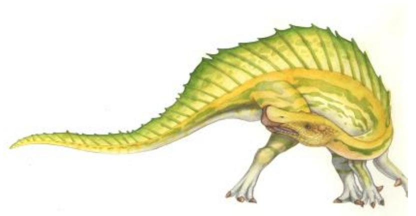
Synchronization Hardware

- Modern OS also provide other special atomic hardware instructions
    - Atomic = non-interruptible
  - Either test a memory word and set a value – **`Test_and_Set`**
  - Or swap contents of two memory words – **`Compare_and_Swap`**

test_and_set Instruction

- Definition:

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



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);
```

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;
}
```

Solution using compare_and_swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

- 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);
```

Sketch Proof

- **Mutual-exclusion**: P_i enters its critical section only if either `waiting[i]==false` or `key==false`. The value of `key` can become `false` only if `test_and_set()` is executed. Only the first process to execute `test_and_set()` will find `key==false`; all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, thus maintaining the mutual-exclusion requirement.

- **Progress**: since a process existing its critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.

- **Bounded-waiting**; when a process leaves its critical section, it scans the array `waiting` in cyclic order {`i+1, i+2, …,n-1,0,1,…i-1`}. It designates the first process in this ordering that is in the entry section (`waiting[j]==true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within `n-1` turns.

Semaphore

- Semaphore ***S*** – non-negative integer variable, kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**, and **`P()`** stands for "**`proberen`**" (to test) and **`V()`** stands for "**`verhogen`**" (to increment) in Dutch
- It is critical that semaphore operations are executed **atomically**. We have to guarantee that no more than one process can execute **`wait()`** and **`signal()`** operations on the same semaphore at the same time.
- The semaphore can only be accessed via two indivisible (atomic) operations

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

Semaphore Usage

- Counting semaphore – An integer value can range over an unrestricted domain
  - Counting semaphore can be used to control access to a given resource consisting of a finite number of instances; semaphore value is initialized to the number of resource available
- Binary semaphore – integer value can range only between `0` and `1`
  - This behaves similar to mutex locks

- This can also be used to solve various synchronization problems
- Consider P_1 and P_2 that shares a common semaphore **`synch`**, initialized to `0`; it require S_1 to happen before S_2

```
P1:
   S1;
   signal(synch);
P2:
   wait(synch);
   S2;
```

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 on the queue
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue
- Semaphore values may become negative, whereas this value can never be negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes currently waiting on the semaphore.

Semaphore Implementation with no Busy waiting (Cont.)

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

Noticing that

- Increment and decrement are done before checking the semaphore value

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 (details in Chapter 7)
- 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);
```

- Under a rare case, **deadlock** can occur, extremely difficult to debug
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue, in which it is suspended. For instance, if we remove processes from the queue associated with a semaphore using LIFO (last-in, first-out) order or based on certain priorities.

Priority Inversion

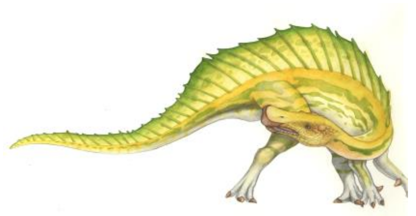
- **Priority Inversion** – A scheduling problem can arise when a lower-priority process holds a lock needed by a higher-priority process
  - This situation becomes more complicated if the low-priority process is preempted in favour of another process with a higher priority
- Consider three processes – `L, M` and `H`, whose priorities follow the order `L<M<H`.
  - Assume that process `H` requires resource `R`, which is currently being accessed by process `L`. Usually process `H` would wait for process `L` to finish using resource `R`. Now suppose `M` becomes runnable, thereby preempting process `L`. Indirectly, a process with a lower priority (`M`) has affected how long process `H` must wait for process `L` to relinquish resource `R`. This problem is known as **priority inversion**
- **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 resource. When they are finished, their priorities revert to their original values.
  - In the above example, process `L` would inherit the priority of process `H` temporarily, thereby preventing process `M` from preempting its execution. Process `L` relinquish its priority to its original value after finishing using resource `R`. Once resource `R` is available, process `H`, - not process `M` – would run next.

Classical Problems of Synchronization

- Classical problems of synchronization
  - 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



Bounded Buffer Problem (Cont.)

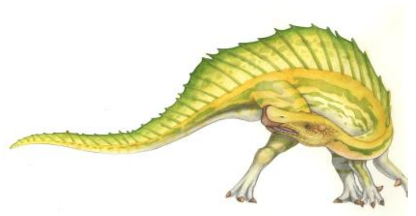
- The structure of the producer process

```
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 the data set at the same time
  - Only one single writer can access shared data at a time
- Several variations of how readers and writers are treated – all involve different priorities.
- The simplest solution, referred to as the first readers-writers problem, requires that no reader be kept waiting unless a writer has already gained access to the shared data
- 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);
```

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);
```

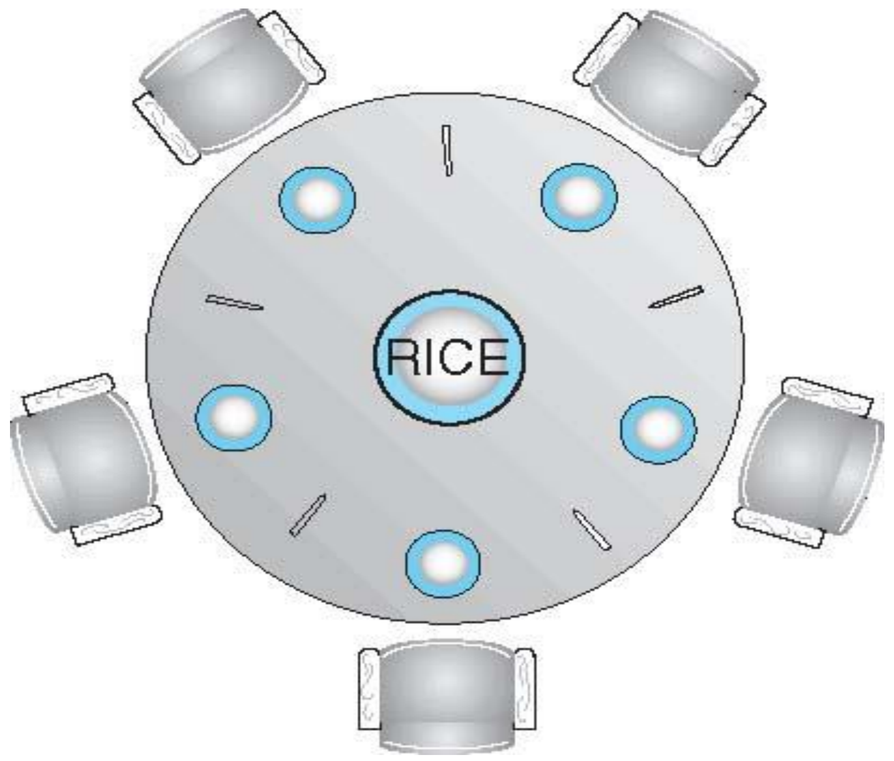
Explanations

- `rw_mutex` controls the access to shared data (critical section) for writers, and the first reader. The last reader leaving the critical section also has to release this lock
- `mutex` controls the access of readers to the shared variable `count`
- Writers wait on `rw_mutex`, first reader yet gain access to the critical section also waits on `rw_mutex`. All subsequent readers yet gain access wait on `mutex`

Readers-Writers Problem Variations

- First variation – no reader kept waiting unless a writer has gained access to use shared object. This can result in starvation for writers, thus (significantly) delay the update of the object.
- Second variation – once a writer is ready, it needs to perform update asap. In another word, if a writer is waiting to access the object (implying that there are readers reading at the moment), no new readers may start reading, i.e., they must wait after the writer updates the object. Of course those reader(s) inside will finish reading the object while the writer waits outside
- This can get far more complicated with multiple writers waiting outside (in sequence), readers require to access "corrected" version of the updates
- Both may have starvation problem leading to even more variations
- The problem can be solved (at least partially) by kernel providing reader-writer locks, in which multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process can acquire the reader-writer lock for writing.

Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- They do not interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from the bowl
  - Need both chopsticks 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

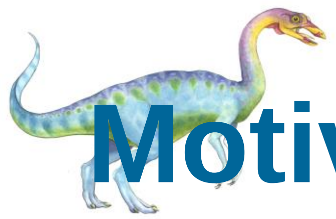
- The structure of Philosopher **i**:

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

                // eat

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

             // think

} while (TRUE);
```

- This guarantees that no two neighbours are eating simultaneously.
- What is the problem with this algorithm? – Deadlock
  - Suppose all five philosophers become hungry at the same time, and each grabs its left chopstick … while waiting for its right chopstick

Problems with Semaphores

- Semaphores provides a convenient and effective mechanism for process synchronization. Using them incorrectly, however, can result in timing errors that are difficult to detect, since such errors happen rarely only if particular execution sequences take place, and these sequences do not always occur – extremely difficult to debug
- Incorrect use of semaphore operations:
  - `signal (mutex)   ….   wait (mutex)`
  - `wait (mutex)   …   wait (mutex)`
  - Omitting of `wait(mutex)` or `signal(mutex)` (or both)
- Deadlock and starvation



Motivation for Monitor and Condition Variables

- The problem with semaphores are that they are dual purpose, which are used for both mutex and scheduling constraints
  - The bounded buffer example, **`mutex`** is a lock, while **`full`** and **`empty`** are really condition variables.

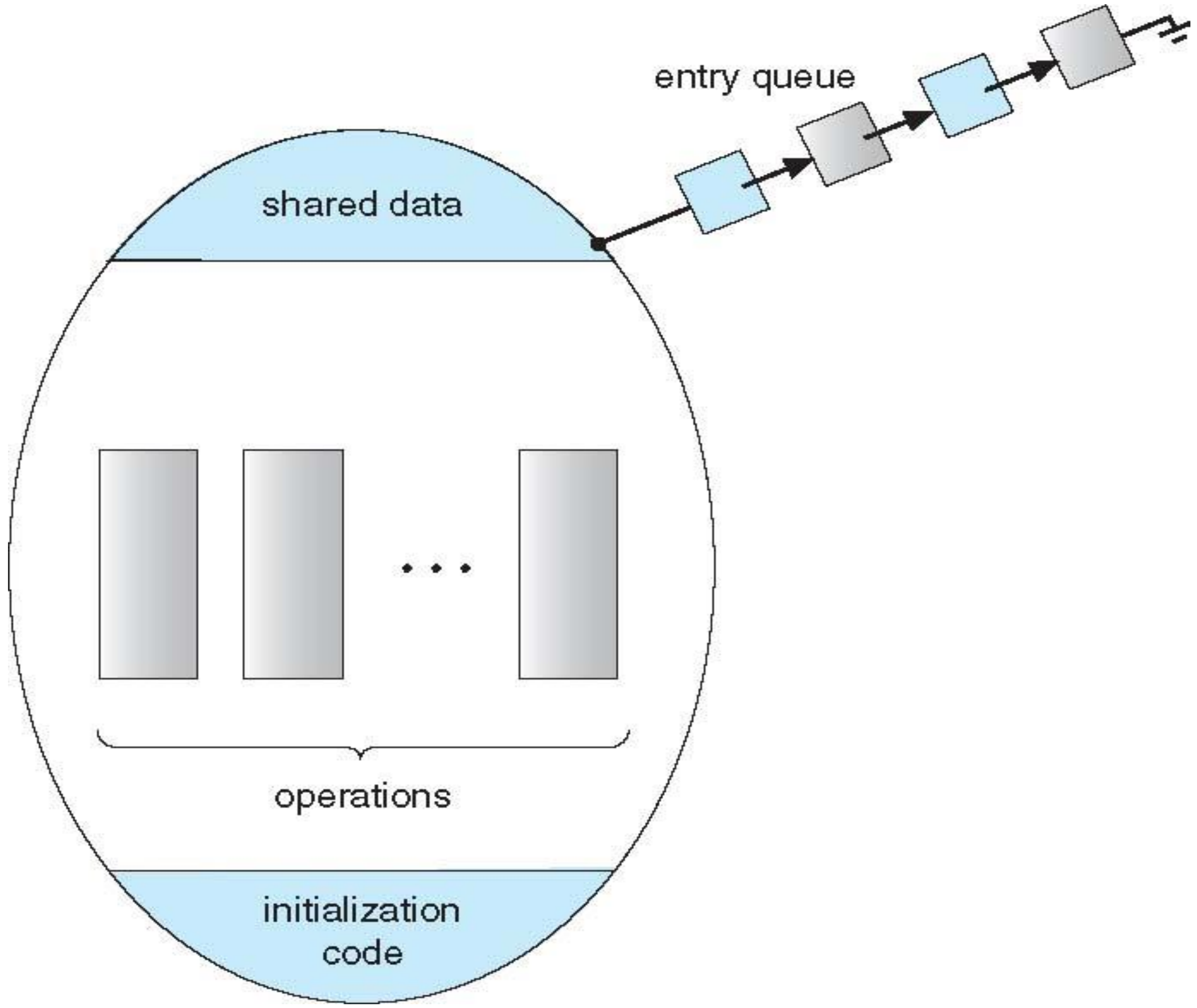
- Better idea: Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
- Definition of **Monitor**: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- An abstract data type or ADT, encapsulates data with a set of functions or procedures to operate on the data.
- The internal variables only are accessible by the codes within the procedure
- Only one process may be active within the monitor at a time – mutual exclusion (lock)

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

    procedure  Pn  (…) {……}

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

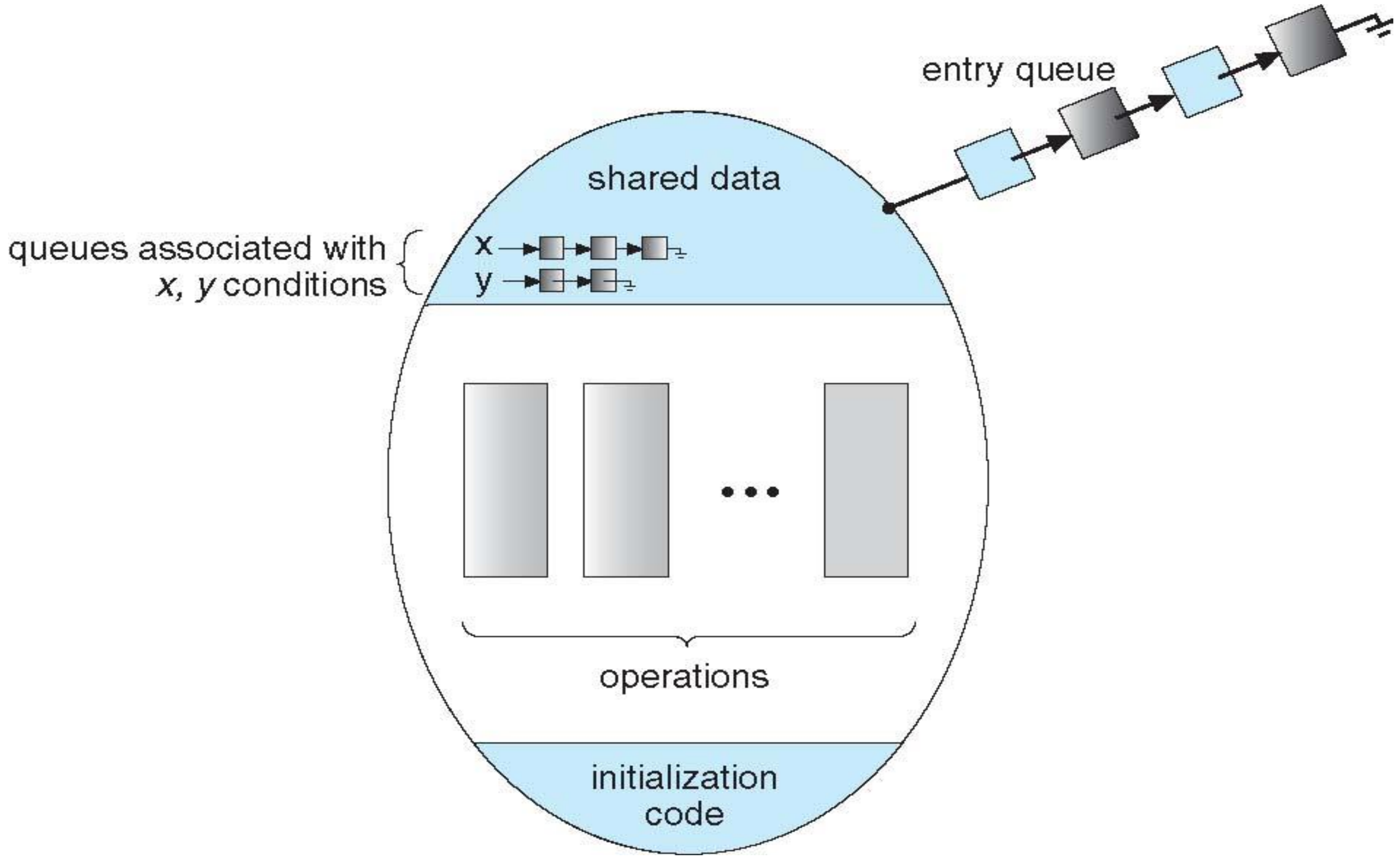
Schematic View of a Monitor



Condition Variables

- Condition variables x, y;
- Queue of threads waiting for something inside a Critical Section
  - This makes it possible to go to sleep (wait) inside a critical section by automatically and atomically releasing the lock of the monitor when it goes to sleep
  - This is different from semaphore in which it can not wait inside a critical section
- Two operations on a condition variable:
  - **`x.wait()`** – a process that invokes this operation (inside a monitor) is suspended – at the same time it has to release the monitor so others can enter
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`** earlier
    - If no **`x.wait()`** on the variable (queue is empty), then it has no effect on the variable.
    - This is different with **`signal()`** operation on a semaphore, which always increments the value of the semaphore
    - The process wakes up, will join the entry queue to wait for its turn to enter monitor (there is a lock); in another word, it can not jump back into a monitor directly

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`**, with Q in **`x.wait()`** state, what should happen next?
  - If Q is resumed, then P must wait, since they can not be inside the monitor simultaneously

- Options include
  - **Hoare monitors - signal and wait.** P gives up monitor and waits until Q leaves monitor or waits for another condition
  - **Mesa monitors - signal and continue.** P keeps the monitor and continues execution, Q wakes up and is placed on the entry queue.

- Both have pros and cons – language implementer can decide

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
- This solution ensures that a philosopher either has acquired two chopsticks or does not have any. In another word, this avoids the situation where a philosopher holds one chopstick while waiting for another.

Monitor Implementation Using Semaphores

- Variables

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

- A signalling process must wait until the resumed process either leaves or waits, then the signalling process can use **`next`** (representing the entry queue, and initialized to **`0`**) to suspend itself. An integer variable **`next_count`** is used to count the number of processes suspended on **`next`**

- Each external function ***F*** will be replaced by

```
wait(mutex);
    …
   body of F;


          …
if (next_count > 0)
   signal(next)   /* the process giving up monitor by letting another enter */
else
   signal(mutex); /* the process releases the monitor */
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

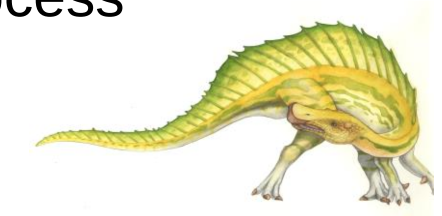
- For each condition variable ***x***, we have:

```
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);  /* the process giving up monitor by letting another enter */
else
      signal(mutex); /* the process releases the monitor */
wait(x_sem);
x_count--;
```

- The process checks if there are other processes waiting to enter the monitor (**`next_count`**), if there is, let one of them enter; otherwise it relinquishes the monitor. After that, it suspends itself by **`wait(x_sem)`**. The decrement **`x_count--`** (counting the number of processes suspended on this condition variable) will be executed when it is waked up later by another process



Monitor Implementation (Cont.)

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

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

- If there is no process waiting on condition `x`, `x.signal()` has not effect
- A process (if any) waked up from `x_sem`, will join the entry queue (`next`) to wait for its turn to enter the monitor, instead of entering the monitor immediately
- The process after waking up a process waiting on `x_sem`, will need to give up the monitor, and join the entry queue (`next`) to wait for its next turn to enter the monitor, signal-and wait option

Synchronization Examples

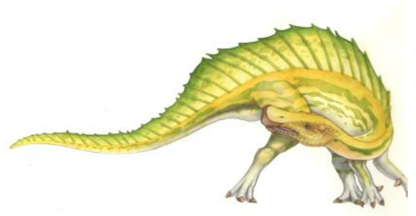
- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutex for efficiency when protecting data from short code segments, usually less than a few hundred instructions
  - Starts as a standard semaphore implemented as a spinlock in a multiprocessor system
  - If lock held, and by a thread running on another CPU, spins to wait for the lock to become available
  - If lock held by a 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. These are used to protect data that are frequently accessed, but usually in a read-only manner. The readers-writer locks are relatively expensive to implement.

Windows Synchronization

- The kernel uses interrupt masks to protect access to global resources on uni-processor systems
- The kernel uses spinlocks in multiprocessor systems
  - For efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock
- For thread synchronization outside the kernel, Windows provides dispatcher objects, threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers
  - Events are similar to condition variables; they may notify a waiting thread when a desired condition occurs
  - Timers are used to notify one or more thread that a specified amount of time has 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 kernel
- Linux provides:
  - semaphores
  - 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, which is available for programmers at the user level and is not part of any particular kernel.
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

End of Chapter 6