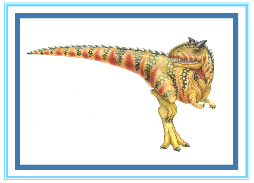
Chapter 6: Process Synchronization



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

- 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
  - Processes may be interrupted at any time, partially completing execution
- Concurrent access to 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:
  Suppose that we want to provide a solution to the Producer-Consumer 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 each time by the producer after it produces an item and places in the buffer and is decremented each time by the consumer after it consumes an item in the buffer.

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
```

- `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 $\{p_0, p_1, \ldots\ p_{n-1}\}$

- Each process has a **Critical Section** segment of code, during which
  - A process may be changing common variables, updating table, writing file, and etc.
  - We need to ensure when one process is in **Critical Section**, no other may be in its critical section

- **Critical section problem** is to design protocol(s) 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 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, 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(s), the selection of the process(es) 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
    - 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 if 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 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 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

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 – 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**)

Synchronization Hardware

- Many systems provide hardware support for critical section code

- All solutions below based on the 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

- 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:
 }
```

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);
```

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. 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`, 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.

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**
- 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);
```

Semaphore

- Semaphore ***S*** – integer variable
- Two standard operations modify ***S***: `wait()` and `signal()`
  - Originally called `P()` and `V()`
- 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. This is a critical section problem
  - Disable interrupts in a single-processor system would work, but more complicated in a multiprocessor system
- 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** – 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**
- Can implement a counting semaphore ***S*** as a binary semaphore
- Can 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 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
- Semaphore values may be 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 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);
   }
}
```

Deadlock and Starvation

- **Deadlock** – two or 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

```
          P0                  P1
     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. For instance, if we remove processes from the queue associated with a semaphore using LIFO (last-in, first-out) order.

Priority Inversion

- **Priority Inversion** – A scheduling problem 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 the shared data at a time
- Several variations of how readers and writers are treated – all involve priorities. The **simplest** solution, referred 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);
```

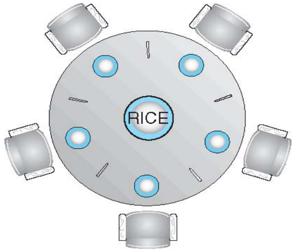
Readers-Writers Problem (Cont.)

- 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);

Readers-Writers Problem Variations

- First variation – no reader kept waiting unless writer has gained access to use shared object
- Second variation – once writer is ready, it performs write 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).
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing **reader-writer locks**, in which multiple processes are permitted to concurrently acquire a reader-writer lock in red 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 can result in timing errors that are difficult to detect, since such errors happen only if particular execution sequences take place, and these sequences do not always occur

- 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

Monitors

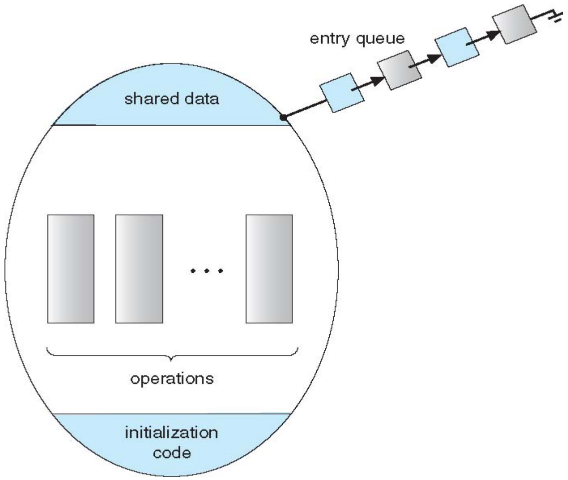
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- An abstract data type or ADT, encapsulates data with a set of functions to operate on the data.
- The internal variables only are accessible by code within the procedure
- Only one process may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

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

    procedure Pn (…) {……}

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

Schematic View of a Monitor

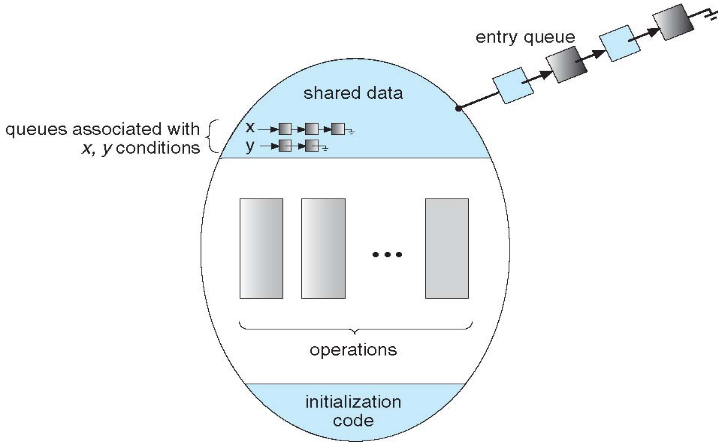


Condition Variables

- condition x, y;
- Two operations 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

---

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
  - **Signal and wait** – P either waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q either waits until P leaves the monitor or 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

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;
```

- A signalling process must wait until the resumed process either leaves or waits, then signalling processes can use `next` (initialized to `0`) to suspend themselves. 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 monitor by letting another enter */
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--;
```

- 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 variable `x_count--` 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
- 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
- This implementation is applicable to the definitions of monitors given by both Hoare and Brinch-Hansen

---

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, 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 uniprocessor 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 a condition variable; 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