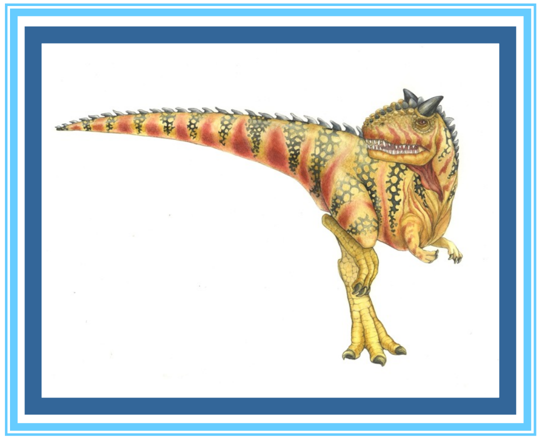
Chapter 6: Synchronization Tools



Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Monitors

Objectives

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

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
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}

Example of Read – Modify - Write

- When we make a flight reservation with an airline, we
  - bring up the seat map and look for available seats (read),
  - we pick a seat to reserve (modify),
  - and change the seat status to unavailable in the seat map (write).
- A bad potential scenario can happen as follows:
  - Two customers simultaneously bring up seat map of the same flight.
  - Both customers pick the same seat, say 9C.
  - Both customers change the status of seat 9C to unavailable in the seat map.
- After the sequence, both customers logically conclude that they are now exclusive owners of seat 9C. (Unpleasant situation!)

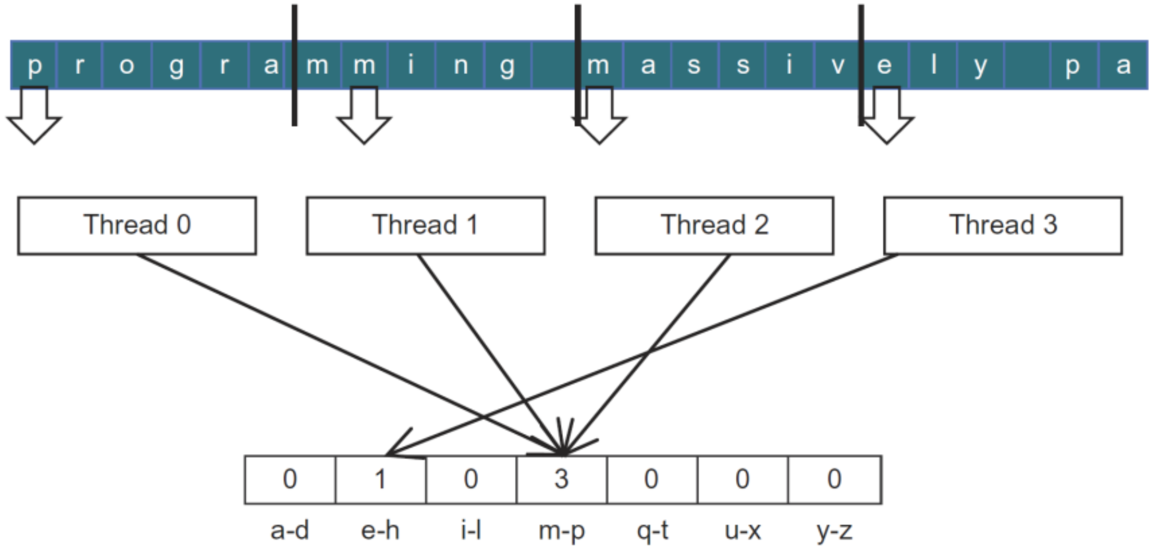
Read – Modify - Write

Observation: Threads 0, 1, and 2 all need to update the same counter (m-p), which is a conflict referred to as output interference.

One must understand the concepts of **race conditions** and **atomic operations** in order to safely handle such output interferences in the parallel code.

An increment to an interval counter in the *histo[]* array is an update, or read-modify-write, operation on a memory location.

- ➢Reading the memory location (read)
- ➢Adding one to the read content (modify)
- ➢Writing the new value back to the memory location (write)

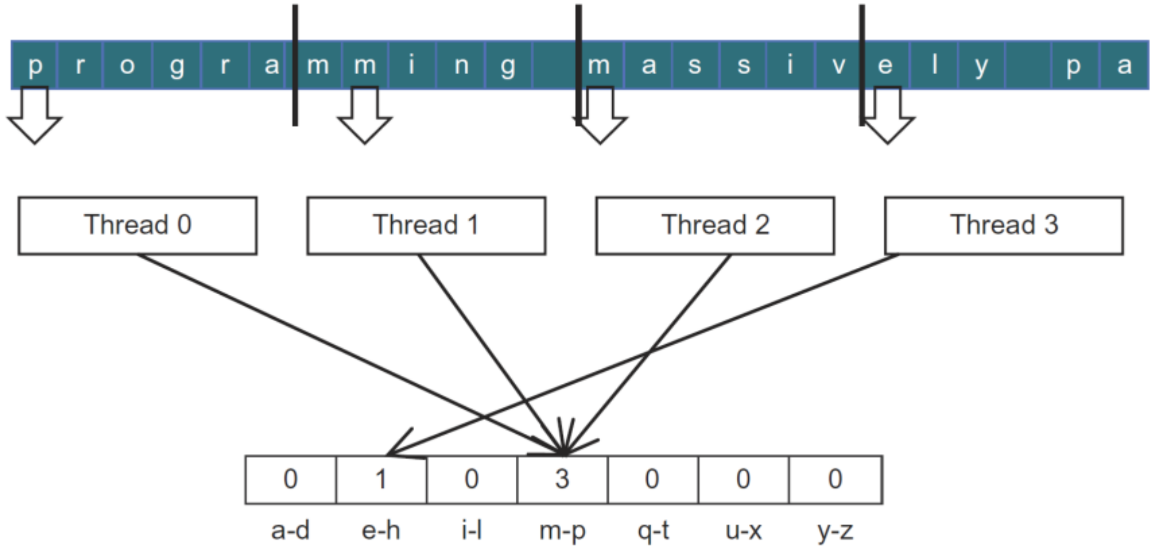

p r o g r a m m i n g m a s s i v e l y p a
Thread 0
Thread 1
Thread 2
Thread 3
0 1 0 3 0 0 0
a-d e-h i-l m-p q-t u-x y-z

Race Condition

The undesirable outcomes in the example are caused by a phenomenon called ***race condition***, where the outcome of two or more simultaneous update operations varies depending on the relative timing of the operations involved. Some outcomes are correct and some are incorrect.

Back to parallel histogram computation, race condition also happen when two threads attempt to update the same histo[] element.

```
thread1:Old ← Mem[x]        thread2:Old ← Mem[x]
        New ← Old + 1               New ← Old + 1
        Mem[x] ← New                Mem[x] ← New
```



p r o g r a m m i n g m a s s i v e l y p a
Thread 0
Thread 1
Thread 2
Thread 3
0 1 0 3 0 0 0
a-d e-h i-l m-p q-t u-x y-z

Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence (**Correct!**)

Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence (**Correct!**)

Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

Thread 1 Old = 0

Thread 2 Old = 0

Mem[x] = 1 after the sequence (**Wrong!**)

Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Thread 1 Old = 0

Thread 2 Old = 0

Mem[x] = 1 after the sequence (**Wrong!**)

Atomic Operations – Ensure Correct Outcomes

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

OR

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

We can eliminate such variations by preventing the interleaving of operation sequences of Thread 1 and Thread 2. Such timing constraints can be enforced with the use of atomic operations.

An atomic operation on a memory location is an operation that performs a read-modify-write sequence on the memory location in such a way that no other read-modify-write sequence to the location can overlap with it.That is, the read, modify, and write parts of the operation **form an indivisible unit**, hence the name atomic operation.

In practice, atomic operations are realized with hardware support to lock out other threads from operating on the same location until the current operation is complete. It is important to remember that atomic operations do not force particular thread execution orderings. Thread 1 can run either ahead of or behind Thread 2.

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process 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);
```

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, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Critical-Section Handling in OS

Two approaches 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 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 $\mathbf{P_i}$ 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

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 not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

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

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

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

Solution using compare_and_swap

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

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

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
- 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 this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

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

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

Mutex/Lock

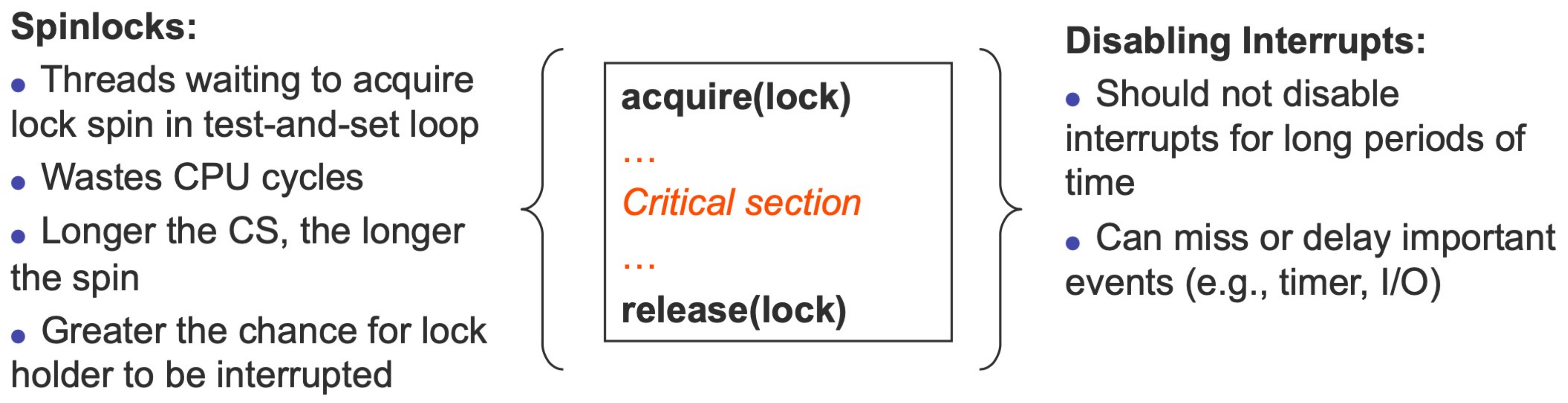
- We looked at using locks to provide mutual exclusion

As we said in one of our previous lessons, a mutex makes it possible to ensure that only one thread at a time has access to the object.

- Locks work, but they have some drawbacks when critical sections are long

- Spinlocks (while loop and test condition) – inefficient (If a thread is spinning on a lock, then the thread holding the lock cannot make progress)

- Disabling interrupts – can miss or delay important events



Spinlocks:

- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted

```
acquire(lock)
...
Critical section
...
release(lock)
```

Disabling Interrupts:

- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)

Semaphore

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

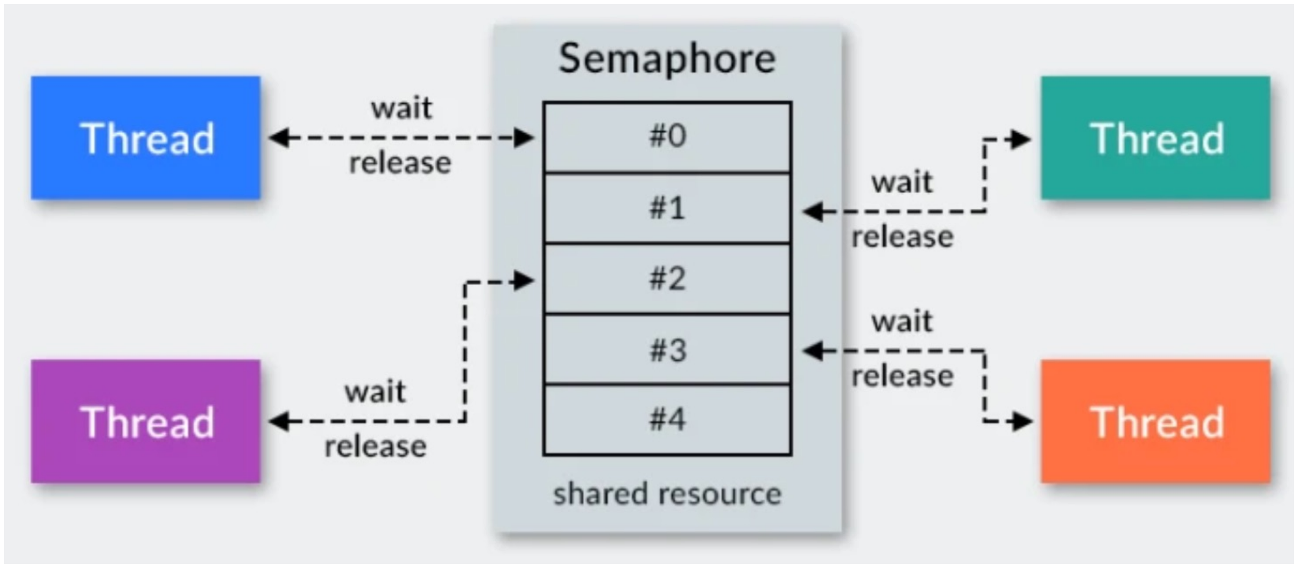
```
wait(S) {
     while (S <= 0)
        ; // busy wait
     S--;
}
```

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

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

Semaphores come in two types

- Mutex semaphore
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore
  - Represents a resource with many units available, or a resource that allows certain kinds of unsynchronized concurrent access (e.g., reading)
  - Multiple threads can pass the semaphore



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

Semaphores Implementation

```
wait (S) {
  Disable interrupts;
  while (S->value == 0) {
    enqueue(S->q, current_thread);
    thread_sleep(current_thread);
  }
  S->value = S->value – 1;
  Enable interrupts;
}
```

```
signal (S) {
  Disable interrupts;
  thread = dequeue(S->q);
  thread_start(thread);
  S->value = S->value + 1;
  Enable interrupts;
}
```

- thread_sleep() assumes interrupts are disabled
  - Note that interrupts are disabled only to enter/leave critical section

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

Blocking in Semaphores

Associated with each semaphore is a queue of waiting processes.

Semaphores support two operations:

wait (semaphore): decrement, block until semaphore is open

When wait() is called by a thread:

- If semaphore is open,
  - thread continues
- If semaphore is closed,
  - thread blocks on queue

signal (semaphore): increment, allow another thread to enter

Then signal() opens the semaphore:

- If a thread is waiting on the queue, the thread is unblocked
- If no threads are waiting on the queue, the signal is remembered for the next thread. In other words, signal() has "history".
- This "history" is a counter.

Semaphore Implementation with no Busy waiting

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

Implementation with no Busy waiting (Cont.)

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

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

Semaphores in Java

Semaphores in Java are represented by the `Semaphore` class.

When creating semaphore objects, we can use the following constructors:

```
Semaphore(int permits)
Semaphore(int permits, boolean fair)
```

We pass the following to the constructor:

- `int permits` — the initial and maximum value of the counter. In other words, this parameter determines how many threads can simultaneously access the shared resource;
- `boolean fair` — establishes the order in which threads will gain access. If `fair` is true, then access is granted to waiting threads in the order in which they requested it. If it is false, then the order is determined by the thread scheduler.

Semaphore Summary

- You may have noticed some similarities between a mutex and a semaphore. Indeed, they have the same mission: to synchronize access to some resource. The only difference is that an object's mutex can be acquired by only one thread at a time, while in the case of a semaphore, which uses a thread counter, several threads can access the resource simultaneously.
- However, they have some drawbacks
  - They are essentially shared global variables
    - » Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
  - No control or guarantee of proper usage. Sometimes hard to use and prone to bugs (incorrect use of semaphore operations, signal…wait)
- Another approach: Use programming language support

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes, so we need additional synchronization mechanisms, condition variables.

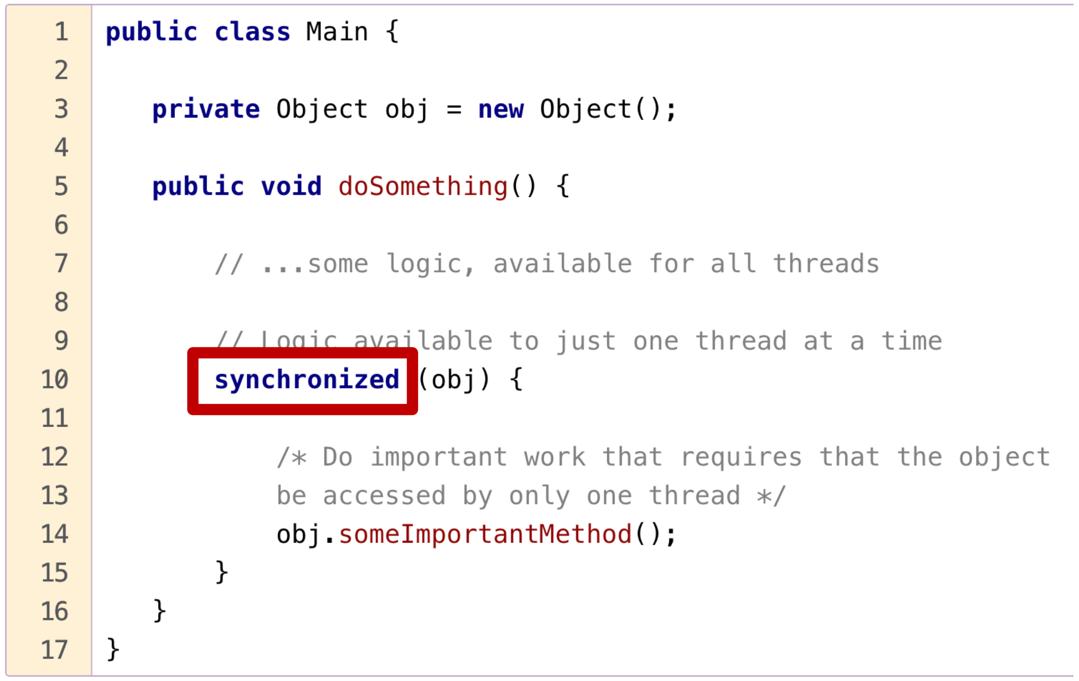
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

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

Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent procedure invocations
- A monitor protects its data from unstructured access
  - It guarantees that threads accessing its data through its procedures interact only in legitimate ways
- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is "in the monitor")
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks. So the monitor has a waiting queue.
  - If a thread leaves a monitor blocks, another one can enter

A monitor is a chunk of code that is "invisible" to the programmer.

- In the code block marked with the **synchronized** keyword, the mutex of our **obj** object is acquired.
- Great, we can acquire the lock, but how exactly is the "protection" provided? When we see the word synchronized, what prevents the other threads from entering the block?
- The protection comes from a monitor! The compiler converts the **synchronized** keyword into several special pieces of code.



```java
public class Main {

   private Object obj = new Object();

   public void doSomething() {

       // ...some logic, available for all threads

       // Logic available to just one thread at a time
       synchronized (obj) {

           /* Do important work that requires that the object
           be accessed by only one thread */
           obj.someImportantMethod();
       }
   }
}
```

Java uses the synchronized keyword to represent a monitor

```
public class Main {

    private Object obj = new Object();

    public void doSomething() throws InterruptedException {

        // ...some logic, available for all threads

        // Logic available to just one thread at a time:

        /* as long as the object's mutex is busy,
        all the other threads (except the one that acquired it) are put to sleep */
        while (obj.getMutex().isBusy()) {
            Thread.sleep(1);
        }

        // Mark the object's mutex as busy
        obj.getMutex().isBusy() = true;

        /* Do important work that requires that the object
        be accessed by only one thread */
        obj.someImportantMethod();

        // Free the object's mutex
        obj.getMutex().isBusy() = false;
    }
```

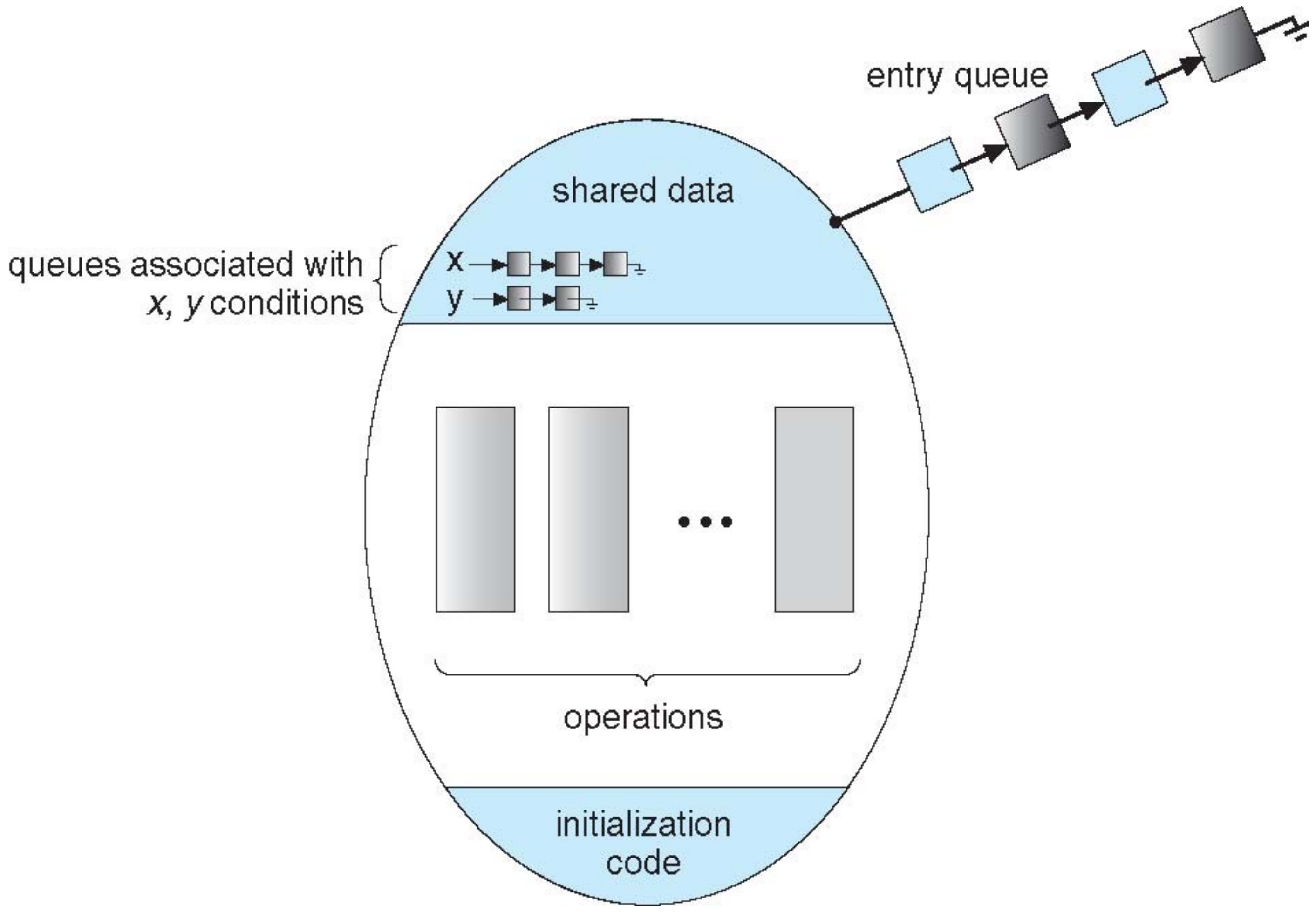
Signal Semantics

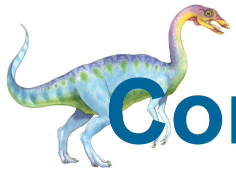
- There are two flavors of monitors that differ in the scheduling semantics of signal()
- Hoare monitors (original)
  - signal() immediately switches from the caller to a waiting thread
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling
- Mesa monitors (Mesa, Java)
  - signal() places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter runs again
    - Returning from wait() is only a hint that something changed
    - Must recheck conditional case

Quick Summary

- Semaphores
  - wait()/signal() implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use
- Monitors
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
    - » Only one thread can execute within a monitor at a time
  - Relies upon high-level language support
- Condition variables
  - Used by threads as a synchronization point to wait for events
  - Inside monitors (compiler adds the code), or outside with locks
  - A monitor is “just like” a module whose state includes a condition variable and a lock. With condition variables, the module methods may wait and signal on independent conditions.

Monitor with Condition Variables





Conditional Variable Implementation (I)

The *condition variable* is a synchronization primative that provides a queue for threads waiting for a resource. A thread tests to see if the resource is available. If it is available, it uses it. Otherwise it adds itself to the queue of threads waiting for the resource. When a thread has finished with a resource, it wakes up exactly one thread from the queue (or none, if the queue is empty). In the case of a sharable resource, a broadcast can be sent to wake up all sleeping threads.

Proper use of condition variables provides for safe access both to the queue and to test the resource even with concurrency. The implementation of condition variables involves several mutexes.

Condition variables support three operations:

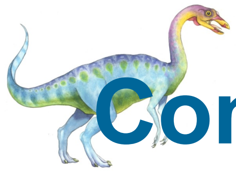
- wait – add calling thread to the queue and put it to sleep
- signal – remove a thread form the queue and wake it up
- broadcast – remove and wake–up all threads on the queue

When using condition variables, an additional mutex must be used to protect the critical sections of code that test the lock or change the locks state.

The following code illustrates a typical use of condition variables to acquire a resource. Notes that both the mutex mx and the condition variable cv are passed into the wait function.

If you examine the implementation of wait below, you will find that the wait function atomically releases the mutex and puts the thread to sleep. After the thread is signalled and wakes up, it reacquires the resource. This is to prevent a *lost wake–up.* This situation is discussed in the section describing the implementation of condition variables.

```
spin_lock s;

GetLock (condition cv, mutex mx)
{
  mutex_acquire (mx);
  while (LOCKED)
    wait (c, mx);

  lock=LOCKED;
  mutex_release (mx);
}


ReleaseLock (condition cv, mutex mx)
{
  mutex_acquire (mx);
    lock = UNLOCKED;
    signal (cv);
  mutex_release (mx);
}
```



Conditional Variable Implementation (II)

This is just one implementation of condition variables, others are possible.

Data Structure

The condition variable data structure contains a double–linked list to use as a queue. It also contains a semaphore to protect operations on this queue. This semaphore should be a spin–lock since it will only be held for very short periods of time.

```
struct condition {
  proc next;  /* doubly linked list implementation of */
  proc prev;  /* queue for blocked threads */
  mutex mx; /*protects queue */
};
```

wait()

The wait() operation adds a thread to the list and then puts it to sleep. The mutex that protects the critical section in the calling function is passed as a parameter to wait(). This allows wait to atomically release the mutex and put the process to sleep.

If this operation is not atomic and a context switch occurs after the release_mutex (mx) and before the thread goes to sleep, it is possible that a process will signal before the process goes to sleep. When the waiting() process is restored to execution, it will enter the sleep queue, but the message to wake it up will be forever gone.

```
void wait (condition *c, mutex *mx)
{
  mutex_acquire(&c->listLock);  /* protect the queue */
  enqueue (&c->next, &c->prev, thr_self()); /* enqueue */
  mutex_release (&c->listLock); /* we're done with the list */

  /* The suspend and release_mutex() operation should be atomic */
  mutex_acquire (mx);
  thr_suspend (self);  /* Sleep 'til someone wakes us */

  mutex_release (mx);

  return;
}
```

Conditional Variable Implementation (III)

signal()

The signal() operation gets the next thread from the queue and wakes it up. If the queue is empty, it does nothing.

```
void signal (condition *c)
{
  thread_id tid;

  mutex_acquire (c->listlock); /* protect the queue */
  tid = dequeue(&c->next, &c->prev);
  mutex_release (listLock);

  if (tid>0)
    thr_continue (tid);

  return;
}
```

broadcast()

The broadcast operation wakes up every thread waiting for a particular resource. This generally makes sense only with sharable resources. Perhaps a writer just completed so all of the readers can be awakened.

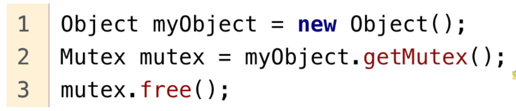
```
void broadcast (condition *c)
{
  thread_id tid;

  mutex_acquire (c->listLock); /* protect the queue */
  while (&c->next) /* queue is not empty */
  {
    tid = dequeue(&c->next, &c->prev); /* wake one */
    thr_continue (tid); /* Make it runnable */
  }
  mutex_release (c->listLock); /* done with the queue */
}
```

Monitors and Java

- A lock and condition variable are in every Java object
- No explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object's monitor
  - A thread enters an object's monitor by
- Executing a method declared "synchronized"
  - Can mix synchronized/unsynchronized methods in same class
  - Supports finer-grained locking than an entire procedure
- Every object can be treated as a condition variable
- Object::notify() has similar semantics as Condition::signal()

Semaphores and Monitors

- Spinlocks and disabling interrupts are useful only for very short and simple critical sections
  - Wasteful otherwise
  - These primitives are “primitive” – don’t do anything besides mutual exclusion
- Instead, we want synchronization mechanisms that
  - Block waiters
  - Leave interrupts enabled inside the critical section
- So we mainly use spinlocks as primitives to build higher-level synchronization constructs (Semaphores and Monitors).
  - Semaphores: binary (mutex) and counting
  - Monitors: mutexes and condition variables
- General speaking, the lock state cannot be controlled directly. Java has no mechanism that would let you explicitly take an object, get its mutex, and assign the desired status.



```java
Object myObject = new Object();
Mutex mutex = myObject.getMutex();
mutex.free();
```

End of Chapter 6