CIIC 4050 / ICOM5007
Operating Systems

Lecture 6

Process Synchronization

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we want to provide a solution to the consumer-producer problem with a buffer of limited capacity.
- We can do so by having an integer `counter` that keeps track of the number of full buffers.

Background

- Initially, `counter` = 0.
- It is incremented by the producer after it places a new item and is decremented by the consumer after it consumes an item.
- If `counter` == 0, the buffer is empty.
- If `counter` == `BUFFER_SIZE`, the buffer is full.

- When several process operate on shared data concurrently and the outcome of the execution depends on the particular order in which the access takes place, we have what is called a **race condition**.

Producer-Consumer

Producer:

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

   while (counter == BUFFER_SIZE) ;
       /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer:

```
while (true) {
      while (counter == 0)
             ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
       counter--;

      /* consume the item in next consumed */
}
```

Producer-Consumer

Producer:

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

   while (counter == BUFFER_SIZE) ;
       /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer:

```
while (true) {
      while (counter == 0)
             ; /* do nothing */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;

      /* consume the item in next consumed */
}
```

Race Condition

Race Condition

- The behavior of software execution where the output is dependent on the sequence or timing of other uncontrollable events.
- It becomes a bug when events do not happen in the order the programmer intended .
- Present in shared data items and collaborative processes or threads

Race Condition

- `counter++` is not an atomic operation, it could be implemented as

```
reg1 = counter
reg1 = reg1 + 1
counter = reg1
```

- `counter--` could be implemented as

```
reg2 = counter
reg2 = reg2 - 1
counter = reg2
```

Race Condition

- Consider this execution interleaving with "count = 5" initially:
  - S0: producer `reg1 = counter` `{reg1 = 5}`
    S1: producer `reg1 = reg1 + 1` `{reg1 = 6}`
    S2: consumer `reg2 = counter` `{reg2 = 5}`
    S3: consumer `reg2 = reg2 - 1` `{reg2 = 4}`
    S4: producer `counter= reg1` `{counter= 6}`
    S5: consumer `counter= reg2` `{counter= 4}`

Critical Section

- Assume several processes, each containing a particular portion of code in which at most a fixed number (usually one) of them can can be executing concurrently.
- Those restricted portions of code are called **critical sections**.

Critical Section

- Note that there might be critical sections in different process which do not conflict with each other.
- A critical section is so only with respect to a conflicting critical section of another processes (or processes).

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 that conflict with the one P_i is in.
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

Solution to Critical-Section Problem

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

Peterson's Solution:

- A software-based solution to CS Problem
  - Not guaranteed to work correctly in modern computer architectures…
  - Assume two processes: P1 and P2
- 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]`

Peterson's Solution:

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

Algorithm for Process P_i

- Assume both process execute the following code
  - For P_1, values for i and j are: i=1, j=2.
  - For P_2, values for i and j are: i=2, j=1.

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);  // busy wait
… critical section
flag[i] = FALSE;
… remainder section
```

Synchronization Hardware

- Software-only based solutions to the critical-section problem
- May not work in modern CPU architectures
- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - May cause other problems, such as hanging the whole system…

Synchronization Hardware

- Generally too inefficient on multiprocessor systems
  - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptable
  - Either `test` memory word and `set` value
  - Or `swap` contents of two memory words

Using Locks to Solve Critical-section Problem

- Any solution to critical-section problem requires some sort of `lock`.
- The idea is: each set of mutually conflicting critical section is protected by a lock.
- The process that acquires the lock first is the only one allowed to enter critical section protected by the lock.
- Process holding a lock must release it after exit from critical section.

Using Locks to Solve Critical-section Problem

- Protect a set of mutually conflicting critical sections that may exist in different processes.
- Require processes to acquire protecting lock before entering a particular critical section (to begin execution of part of it):

```
acquire lock
… critical section
release lock
```

- Locks can be implemented by combining hw and sw operations…

`TestAndSet` Instruction

- **`TestAndSet`** instruction: Atomically sets a target variable to **`true`** and returns its current value…

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

Solution using `TestAndSet`

- Shared boolean variable `lock`, initialized to **`false`**.
- Solution:

```
while ( TestAndSet (&lock ));   // do nothing
        …  critical section
lock = FALSE;
        … remainder section
```

`Swap` Instruction

- Atomic **`swap`** instruction: atomically switch the value in two storage locations.

```
void swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp:
}
```

Solution using `Swap`

- Shared **`boolean`** variable **`lock`** initialized to **`false`**
- Each process has a local **`boolean`** variable **`key`**
- Solution:

```
key = TRUE;
while ( key == TRUE)
   Swap (&lock, &key );
… critical section
lock = FALSE;
… remainder section
```

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

Mutex Locks

- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock is therefore called a **spinlock**

`acquire()` and `release()`

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

`acquire()` and `release()`

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

Semaphore

- Tool that does not require busy waiting. Less complicated…
- Semaphore a data type such as:

```
typedef struct {
   int value;     // current value of
                  // the semaphore
} Semaphore;
```

Semaphore

- Two standard operations modify `S:` `wait(&S)` and `signal(&S)`
  - Originally called `P()` and `V()`
  - Also called `down()` and `up()`
  - Or `aquire()` and `release()`
- Can only be accessed via these two indivisible (atomic) operations

Semaphore

```
–wait(Semaphore *S) {
    while (S->value <= 0);
       // no-op … busy waiting
       (S->value)--;
 }
–signal(Semaphore *S) {
    (S->value)++;
 }
```

Semaphore

```
–wait(Semaphore *S) {
    while (S->value <= 0);
       // no-op … busy waiting
       (S->value)--;
 }
–signal(Semaphore *S) {
    (S->value)++;
 }
```

Note: the atomicity here assumes that if a process checks the *while* expression and determines that it is `false`, it proceeds with the decrement operation right away without being interrupted.

Semaphores as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - can be simpler to implement
  - similar to **`mutex locks`**

Using Semaphores

- Consider P_1 and P_2 that require S_1 to happen before S_2
- Create semaphore "**synch**" initialized to 0

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

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

Binary Semaphores for Synchronization

- Provides mutual exclusion:
- Semaphore is shared by multiple processes

```
Semaphore mutex;
//initialized to 1 => binary
```

- All processes use mutex to control entrance to critical section:

```
wait (&mutex);
…critical section
signal (&mutex);
…remainder section
```

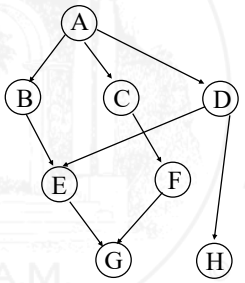
Counting Semaphores for Synchronization

Dependency Graph

- directed graph
- nodes represent processes of threads
- edge (a, b) implies that the execution of a needs to finish before b can initiate its

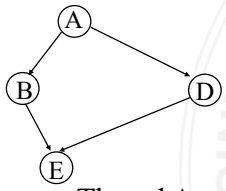
To enforce the right order, but at the same time allowing as much concurrency as possible, we can use semaphores…

Example:



Possible execution order:
- A, D, H, B, C, F, E, G
- A, B, C, D, E, H, F, G
- …

Counting Semaphores for Synchronization



Assume shared semaphores:

```
Semaphore BS = new Semaphore(0),
          DS = new Semaphore(0),
          ES = new Semaphore(-1);
```

Thread A:

```
… A's work
BS.signal();
DS.signal();
```

Thread B:

```
BS.wait();
… B's work
ES.signal();
```

Thread D:

```
DS.wait();
… D's work
ES.signal();
```

Thread E:

```
ES.wait();
… E's work
```

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** or **`signal()`** on 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
- Main disadvantage of semaphore definition is: **busy waiting**.

Semaphore Implementation

- Busy waiting is a problem in multiprogramming systems: **CPU waste**.
- Spinlocks are not bad at all if the waiting time is small, specially, if smaller (or expected to be) than context switch time.
  - Usually employed in multiprocessor systems.
- Semaphores implementation should eliminate busy waiting.
  - By blocking any process trying to acquire an assigned semaphore.
  - Waking it up when the semaphore is released.

Semaphore without Busy Waiting

- One alternative to remove busy waiting is to **block** processes whose `wait()` operation fail.
- A blocked process on a particular semaphore is waken up when `signal()` is called
- We can now define a semaphore as suggested by defining and object type containing two internal fields:
  - An integer value - the value of the process
  - A queue of PCPs - the waiting queue of processes

Semaphore without Busy Waiting

- In addition to the semaphore's standard operations, it has the following internal operations:
  - `block` – blocks a process. The process is removed from the ready queue.
  - `wakeup` – removes one of processes in the waiting queue of the semaphore and places it in the ready queue.

Semaphore without Busy Waiting

```
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

- Inappropriate use of semaphores is a common source of deadlocks.
- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Deadlock and Starvation

- Let S and Q be two semaphores initialized to 1 - this **may lead to deadlock**

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Priority Inversion Problem

- Let P1, P2, and P3 be three process such that:
  prty(P1) < prty(P2) < prty(P3)
- Assume P1 is running, P2 is blocked, and P3 is waiting for some resource (lock) held by P1.
- P2 wakes up and goes to ready queue.
- Since P2 has higher priority than P1, P1 is preempted for P2 to enter into Running state.
- P2, who has lower priority than P3 is causing P3 to wait longer for some resource that P2 does not hold.
- The problem occurs only if there are more than two different priorities.

Priority Inversion Problem (2)

- **Solution**: Use **priority-inheritance protocol**

- Whenever a set of processes is waiting for some resource being held by another process `P` of lower priority, `P` will temporarily get the same priority as the highest among the waiting processes.

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

- These problems are classical for testing different synchronization tools.
- We shall now study about this problems and possible solutions.

Bounded-Buffer Problem

We have seen this problem before…
- let's study a semaphore-based solution:

- N buffers, each can hold one item (or a buffer with N slots)
  - N is the capacity of the whole buffer or the max number of items that it can hold simultaneously.
- Semaphore mutex initialized to the value 1
  - binary semaphore used to provide mutual exclusion for accesses to the pool.

Bounded-Buffer Problem

- Semaphore full initialized to the value 0
  - counting semaphore
  - counts # of occupied slots (# of items) in the buffer
  - used to block not-allowed operations when the buffer is **full**
- Semaphore empty initialized to the value N
  - counting semaphore
  - counts # of empty (or available) slots in the buffer
  - used to block not-allowed operations when the buffer is **empty**.

Bounded Buffer Problem

- Shared Data Structures:
  - `int n;`
  - `semaphore mutex = 1;`
  - `semaphore empty = n;`
  - `semaphore full = 0;`

Producer process/thread:

```
/* Producers call this method
   */
do {
    wait(empty);
    wait(mutex);
    /* add produced to the
   buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

Consumer process/thread:

```
/* Consumers call this method */
do {
    wait(full);
    wait(mutex);
    /* remove item from the
   buffer */
    signal(mutex);
    signal(empty);
   /* consume item in next-
   consumed */
} while (true)
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.

Readers-Writers Problem

- Different variations:

(1) no reader shall be kept waiting unless a writer has already obtained

permission to use the shared data object

- *may cause starvation to writers*

(2) once a writer wants to write (is ready), that writer perform its write as soon as possible

- *may cause starvation to readers*

Readers-Writers Version (1) Solution

- Shared Data Structures

```
int read_count = 0;     /* Number of readers currently active */

semaphore mutex = 1;    /* Binary semaphore to control updates to */
                        /* variable read_count */

semaphore rw_mutex = 1; /* Binary semaphore to control that   */
                        /* no writer operates if some other   */
                        /* thread is already operating in it. */
```

Readers-Writers Version (1)

```
/* Reader Structure */
do {
  wait(mutex);
  /* the first reader indicates
   that the database is being read
   */
  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);
```

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

Dining-Philosophers Problem

- Five philosophers spend their lives thinking and eating rice.
- There are only 5 chopstick on the table, one between every two philosophers.
- In order to eat, a philosopher needs to get hold of the two chopsticks that are closest to her.
- Yes, they need to share the chopsticks.
- A philosopher cannot pick up a chopstick that is already taken by a neighbor.

Shared data:

- the five chopsticks
- the rice
- (infinite source of read-only data)

Dining-Philosophers A Possible Solution

```
Semaphore chopstick[] = new Semaphore[5];
for (int i=0; i<5; i++)
   chopstick[i] = new Semaphore(1);

// each Philosopher is a different thread, executing:

while (true) {    /* get left chopstick */
   wait (chopstick[i]);
   /* get right chopstick */
   wait (chopstick[(i+1)%5]);
   eating();
   /* return left chopstick */
   signal (chopstick[i]);
   // return right chopstick
   signal (chopstick[(i+1)%5]);
}
```

Dining-Philosophers A Possible Solution

```
Semaphore chopstick[] = new Semaphore[5];
for (int i=0; i<5; i++)
   chopstick[i] = new Semaphore(1);

// each Philosopher is a different thread, executing:

while (true) {    /* get left chopstick */
   wait (chopstick[i]);
   /* get right chopstick */
   wait (chopstick[(i+1)%5]);
   eating();
   /* return left chopstick */
   signal (chopstick[i]);
   // return right chopstick
   signal (chopstick[(i+1)%5]);
}
```

WARNING
This solution has a potential deadlock situation...

Dining-Philosophers Problem Algorithm

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Semaphores' Limitations

- Although semaphores may provide an effective mechanism for synchronization, it is easy to use them incorrectly
- The programming language cannot help in blocking several possible errors
- Incorrect use of semaphore operations:
  - omit `wait` or `signal` statements
  - incorrect order of wait & signal statements.
  - may cause violation of mutual exclusion, etc.
  - may also cause deadlocks
- Better language constructs are needed

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
  - we can think of it as a Java class…
- Only one process is allowed to be active within the monitor at a time

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

Monitors

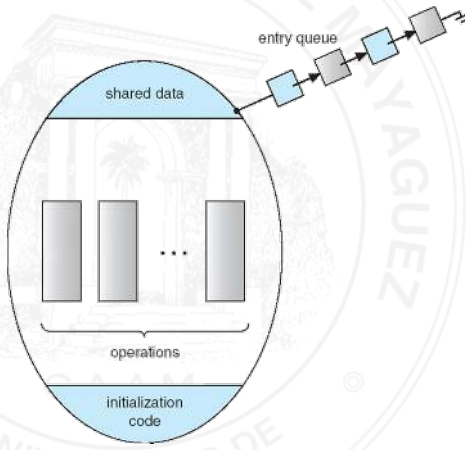
```
monitor monitor-name  {

   // shared variables
   procedure P1 (…) { …. }
            …

   procedure Pn (…) {……}

   Initialization code ( ….)
   { …. }

}
```



Condition Variables

- Monitor construct ensures only one process at the time can be active within the monitor (executing part of it).
- However, this is not powerful enough and extra synchronization mechanisms are needed.
  - One such mechanism are **condition variables**.
  - A new data type is required: Condition
- `Condition x, y; //declares two condition variables`

Condition Variables

- Two operations on a condition variable:
  - `x.wait ()` – a process that invokes the operation is suspended.
  - `x.signal ()` – resumes one of the processes (if any) that is currently waiting because it has invoked `x.wait ()`.
  - Has no effect if no process is waiting on x at the moment.

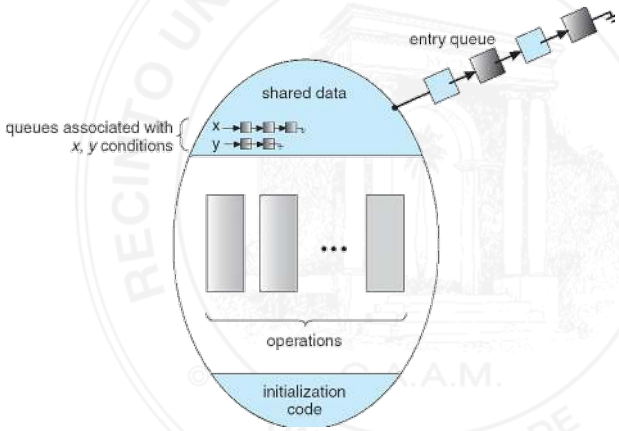
Condition Variables (cont)

- If a process P executes `x.signal()` and there is another process Q waiting on `x`, what should happened?
- Should P and Q both continue simultaneously within the monitor?
- Two possibilities:
  - **Signal and wait**: P either waits until Q leaves the monitor or for another condition.
  - **Signal and continue**: Q either waits until P leaves the monitor or for another condition.

Condition Variables (cont)

- A compromise between the two alternatives is needed.
- Some languages implement the following policy:
  - When P executes **`signal()`** operation, it immediately leaves the monitor and Q is immediately resumed…

Monitor with Condition Variables



Schematic view of a monitor and condition variables.

Solution to Dining Philosophers

This solution **has the problem of possible starvation**… but it is a good start…

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

void pickup_chopsticks (int i) {
   state[i] = HUNGRY;
   test(i);
   if (state[i] != EATING)
      self[i].wait();
}
void putdown_chopsticks (int i){
   state[i] = THINKING;
   /* test left & right */
   test((i + 4) % 5);
   test((i + 1) % 5);
}

void test (int i) {
   if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
      state[i] = EATING ;
      self[i].signal () ;
   }
}
initialization_code() {
   for (int i = 0; i < 5; i++)
      state[i] = THINKING;
}
```

Solution to Dining Philosophers

- Each philosopher *i* will invoke the operations **`pickup_chopsticks()`** and **`putdown_chopsticks()`** in the following sequence:

```
DiningPhilosophers.pickup_chopsticks(i);
EAT
DiningPhilosophers.putdown_chopsticks(i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Process must execute `wait(mutex)` before entering the monitor and must execute `signal(mutex)` after leaving the monitor.
- Variables

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

Monitor Implementation Using Semaphores

- Each procedure ***F*** will be replaced by

```
wait(mutex);
        …
   body of F;
        …
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

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

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation `x.wait()` can be implemented

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

Monitor Implementation (Cont.)

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

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

Resuming Processes within a Monitor

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

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

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

- Where R is an instance of type `ResourceAllocator`

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
      if (busy)
         x.wait(time);
         busy = TRUE;
   }
   void release() {
      busy = FALSE;
      x.signal();
   }
initialization code() {
    busy = FALSE;
   }
}
```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- pthreads

Solaris Synchronization

- Variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Adaptive mutexes for efficiency when protecting data from short code segments
  - locks can behave as spinlocks or as blocking-wakeup
  - selection depends on number of cores or if lock's owner is blocked or if it is active

Solaris Synchronization (2)

- Condition variables and readers-writers locks when longer sections of code need access to data
  - readers-writers locks allow multiple reader threads simultaneously while only one writer…
- Turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - these are queues of waiting threads…

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes, semaphores, timers
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks **(on uniprocessor, enable/disable kernel preemption)**
  - atomic integers
  - reader-writer versions of both (spinlocks, semaphores)

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

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

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

OpenMP

- OpenMP (multi-processing) is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
 {
   #pragma omp critical
   {
      count += value
   }
 }
```

- Code within **`#pragma omp critical`** treated as a critical section and performed atomically.

Functional Programming Languages

- Functional PLs offer a different paradigm than procedural PLs
  - they do not maintain state.
- Variables treated as immutable
  - cannot change state once they have been assigned a value.
- Increasing interest in functional languages
  - Erlang and Scala for their approach in handling data races, actor concurrency model, immutable objects, etc.