VIT
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SWE3001-Operating Systems

Prepared By

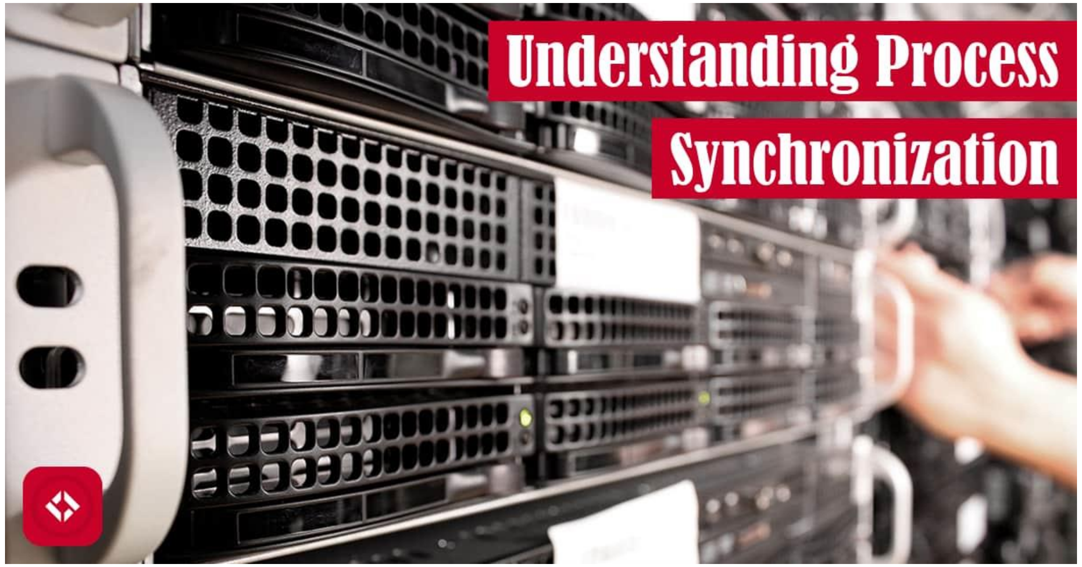
Dr. L. Mary Shamala

Assistant Professor

SCOPE/VIT

Module 3: Process Synchronization

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



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:

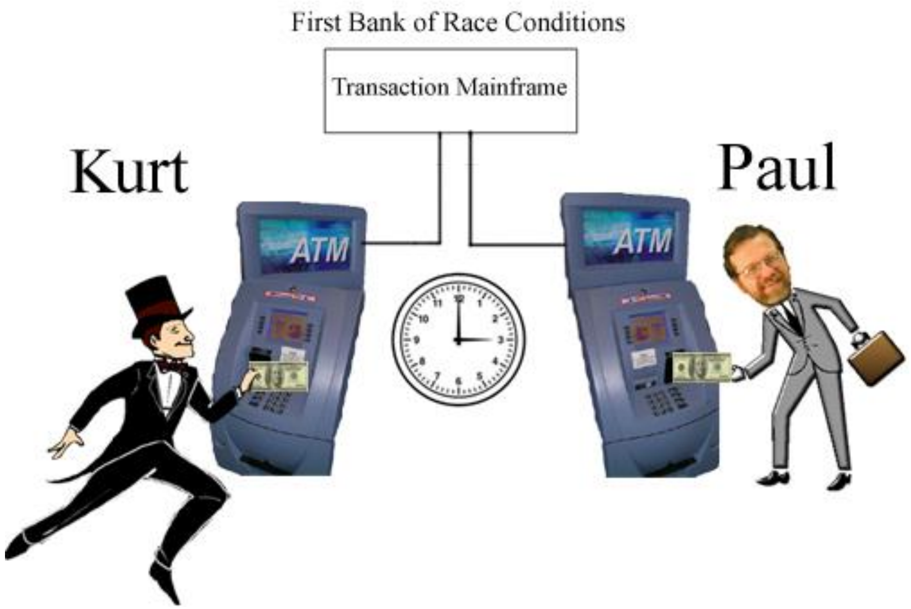
T0: producer execute `register1 = counter` {register1 = 5}
T1: producer execute `register1 = register1 + 1` {register1 = 6}
T2: consumer execute `register2 = counter` {register2 = 5}
T3: consumer execute `register2 = register2 - 1` {register2 = 4}
T4: producer execute `counter = register1` {counter = 6 }
T5: consumer execute `counter = register2` {counter = 4}

Race Condition…

- We arrived at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition, ensure that only one process at a time can be manipulating the variable counter.
- To make such a guarantee, we require that the processes be synchronized in some way

Why Process Synchronization is important?

- To avoid inconsistent state
- To prevent race condition



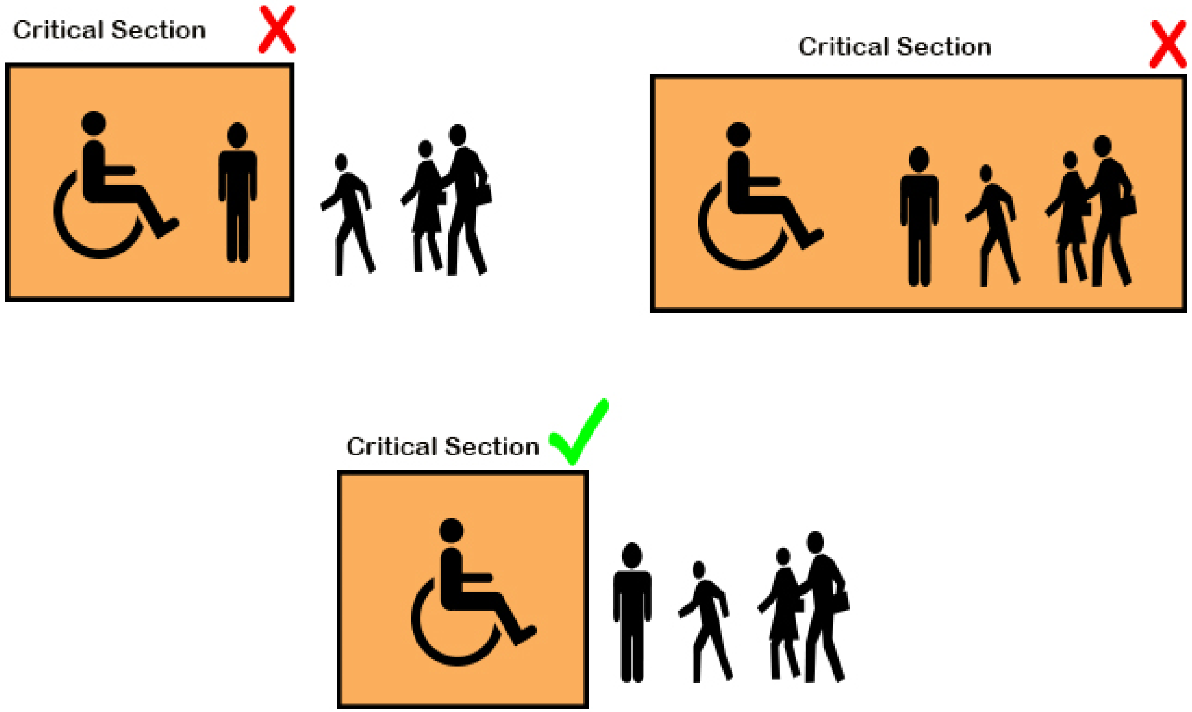
The Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing 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);
```



Critical Section…

- The main blocks of process are:
  - **Entry Section –** To enter the critical section code, a process must request permission. Entry Section code implements this request.
  - **Critical Section –** This is the segment of code where process changes common variables, updates a table, writes to a file and so on. When 1 process is executing in its critical section, no other process is allowed to execute in its critical section.
  - **Exit Section –** After the critical section is executed , this is followed by exit section code which marks the end of critical section code.
  - **Remainder Section –** The remaining code (other parts of the code which are not in the Critical or Exit sections) of the process is known as remaining section.

Solution to Critical-Section Problem: Requirements

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 are used to handle critical sections in operating systems:
  1. **Preemptive** – allows preemption of the process when running in kernel mode
     - Preemptive kernels are especially difficult to design for SMP architecture
     - A preemptive kernel may be more responsive
     - More suitable for real-time programming
  2. **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
     - Essentially free of race conditions in kernel mode

Solution to Critical-Section Problem

- Peterson's Solution
- Synchronization Hardware
  - Test and Set
  - Compare and Swap
- Mutex Locks
- Semaphores
- Monitors

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = `*`true`*** implies that process P_i is ready!

Algorithm for Process P_i

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

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

- Limitations
  - Two process solution
  - No guarantee to work on modern computer architectures

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- test and set() instruction
  - Or swap contents of two memory words- compare and swap() instruction

test_and_set Lock

- A hardware solution to the synchronization problem
- There is a **shared lock variable** that can take either of the two values:0 or 1
- Before entering into the critical section, a process inquires about the lock
- If it is locked, it keeps on waiting till it becomes free
- If it is not locked, it takes the lock and executes the critical section

`test_and_set` instruction

Definition of **`test_and_set`** instruction:

```
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` Lock

- 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

- Solution that satisfies all the critical section requirements
- The common data structures are

  boolean waiting[n];
  boolean lock;

```
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 (short for mutual exclusion) 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

acquire() and release()

Definition of acquire()

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

Definition of acquire()

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

Solution to Critical-section Problem Using Mutex Locks

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

Mutex Locks…

- Disadvantage
  - This solution requires **busy waiting-** While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire()
  - This lock is therefore called a **spinlock** because the process "spins" while waiting for the lock to become available.
  - Continual looping is a problem in real multiprogramming system-Busy waiting wastes CPU cycle
- Advantage of spinlock
  - No context switch is required when a process must wait on a lock
  - Spinlocks are useful if locks are held for short times
  - Often employed on multiprocessor systems

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore is a simple integer(non-negative) variable and is shared between processes
- This variable is used to solve critical section problems & to achieve process synchronization in multiprocessing environment
- A **Semaphore** ***S*** is an integer variable that can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**

Semaphore…

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

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

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

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

Semaphore Usage

- Types of semaphores

1. **Counting semaphore** – integer value can range over an unrestricted domain
2. **Binary semaphore**
   - integer value can range only between 0 and 1
   - Same as a **mutex lock** as they are locks that provide mutual exclusion

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances
- Can solve various synchronization problems

Semaphore Usage

- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**`synch`**" initialized to 0

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

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

Semaphore Implementation

- The definitions of the wait() and signal() semaphore operations requires **busy waiting**
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively
- This type of semaphore is also called **spinlock** because the process "spins" while waiting for the lock
- To overcome the need for busy waiting, modify the definition of the wait() and signal() operations as follows:
  - When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait
    - Rather than engaging in busy waiting, the process can block itself
    - The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
  - Then control is transferred to the CPU scheduler, which selects another process to execute.

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 Definition

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

Implementation with no Busy waiting (Cont.)

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


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

Semaphore Implementation …

- It is critical that semaphore operations be executed atomically.
- 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
- Note that not completely eliminated busy waiting with this definition of the wait() and signal() operations
  - moved busy waiting from the entry section to the critical sections of application programs.
  - limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short
- Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time.
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

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

- **Starvation** – **indefinite blocking:** A process may never be removed from the semaphore queue in which it is suspended

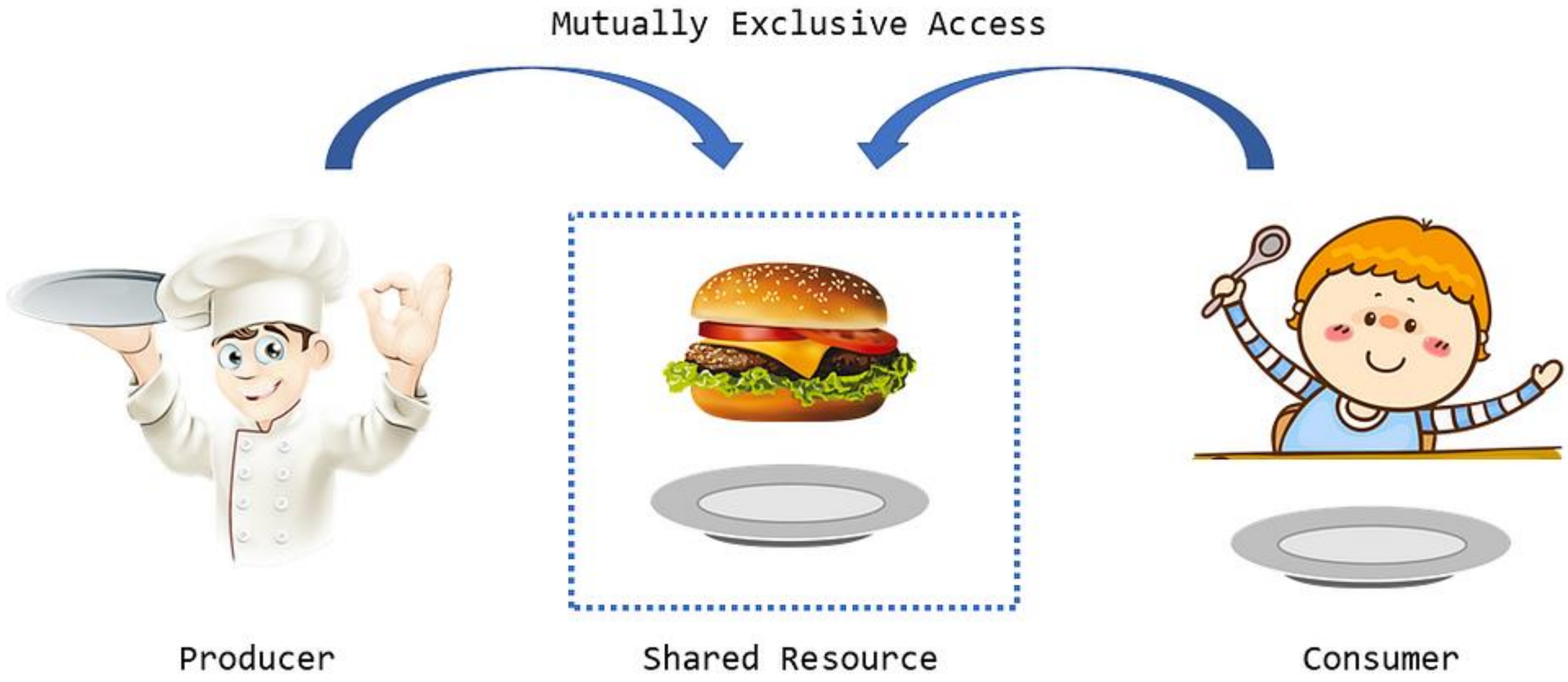
Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
- It occurs only in systems with more than two priorities
- One solution is to have only two priorities.
- General purpose OS Solves the problem via **priority-inheritance protocol**
  - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
  - When they are finished, their priorities revert to their original values

Classical Problems of Synchronization

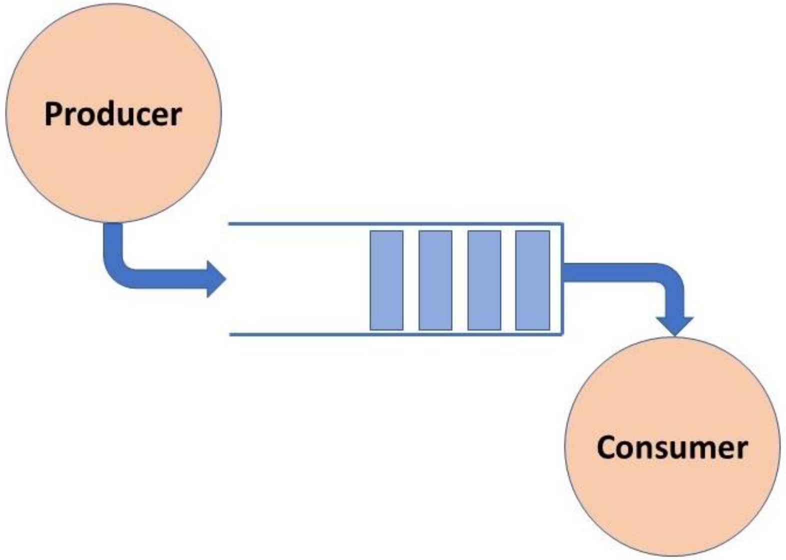
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
  - Sleeping Barber Problem
  - Cigarette Smokers Problem

Producer-Consumer Problem



Bounded-Buffer Problem

- Producer process produce data/item.
- Consumer consume the data/item produced by the Producer.
- Both producer and consumer share a common **buffer** consisting of n slots (each can hold one item)
- Requirements for Process synchronization
  - The Producer process must not produce an item if the shared buffer is full.
  - The Consumer process must not consume an item if the shared buffer is empty.
  - Access to the shared buffer must be mutually exclusive



Bounded-Buffer Problem

- To solve the Producer-Consumer problem 3 semaphores are used:
  - Semaphore **`mutex`** provides mutual exclusion for accesses to the buffer pool
  - Semaphore **`full`** count the number of full buffers
  - Semaphore **`empty`** count the number of empty buffers
- The producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

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);
```

```
buffer[in] = next_produced;
in = (in + 1) % n;
```

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);
```

```
next_consumed=buffer[out];
out = (out+ 1) % n;
```

Readers-Writers Problem

- A database is shared among a number of concurrent processes
  - **Readers** – only read the database; they do ***not*** perform any updates
  - **Writers** – can both read and write the database
- **Problem**
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
  - To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
- Several variations of how readers and writers are considered – all involve some form of priorities
  - ***First*** readers –writers problem – no reader kept waiting unless writer has permission to use shared object
  - ***Second*** readers –writers problem – once writer is ready, it performs the write ASAP
  - Both may have starvation leading to even more variations

Solution to First Readers-Writers Problem (Cont.)

- Shared Data by reader processes
  - Dataset
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0
- **`rw_mutex`**
  - Common to both reader and writer processes
  - Functions as a mutual exclusion semaphore for the writers.
  - Also used by the first or last reader that enters or exits the critical section
- **`mutex`** is used to ensure mutual exclusion when the variable read_count is updated.
- **`read_count`** keeps track of the number of processes currently reading the dataset

Solution to First Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex); // writer wants to enter critical section
            ...
        /* writing is performed */
            ...
       signal(rw_mutex); //writer leaves
 } while (true);
```

Solution to First Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
     wait(mutex); // reader wants to read
     readcount++;    //the number of readers is increased by 1
     if (readcount == 1)
       wait(rw_mutex); //ensure no writer can enter critical section if there is even one reader
    signal(mutex); //other readers can enter while this current reader is inside critical section
         ...
      /* reading is performed by current reader*/
         ...
    wait(mutex);
     readcount--;    // a reader wants to leave
     if (readcount == 0) //no reader is left in the critical section
    signal(rw_mutex); //writers can enter
    signal(mutex); //reader leaves
 } while (true);
```

Readers-Writers Problem…

- Problem is solved on some systems by kernel providing **reader-writer locks**
- Acquiring a reader–writer lock, specifying the mode of the lock: either read or write access.
- When a process wishes only to read shared data, it requests the reader–writer lock in read mode.
- A process wishing to modify the shared data must request the lock in write mode.
- Multiple processes are permitted to concurrently acquire a reader–writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.
- Reader–writer locks are most useful in the following situations:
  - In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
  - In applications that have more readers than writers.

Dining-Philosophers Problem

- The dining-philosophers problem is considered a classic synchronization problem
- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore `chopstick[5]` initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher ***i***:

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

               //  eat

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

                //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

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

Problems with Semaphores

- Incorrect use of semaphores can result in timing errors that are difficult to detect
- 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 are possible.

Monitors

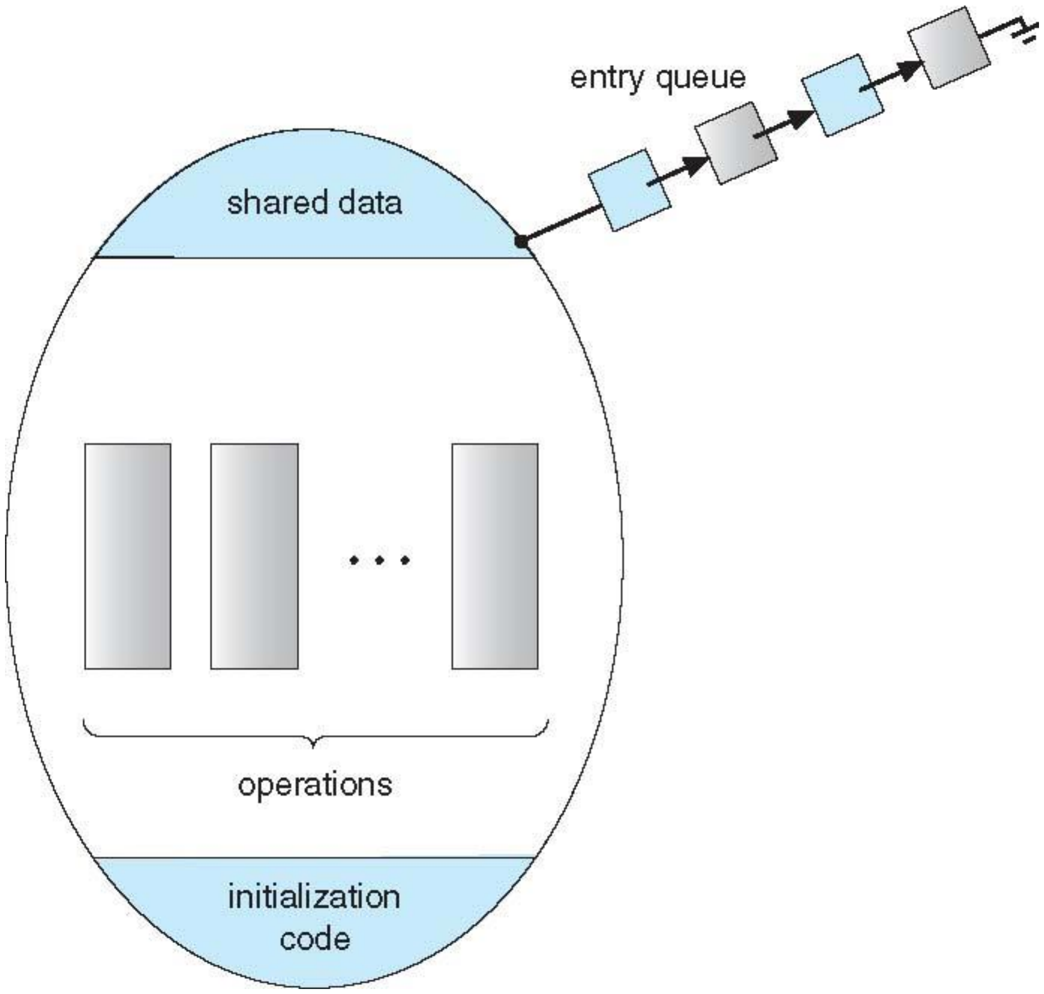
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *An Abstract data type* that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor.
- The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables
  - internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

Monitor Usage

- The syntax of a monitor type

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
  procedure P2 (…) { …. }
  …
  procedure Pn (…) {……}

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

Schematic view of a Monitor

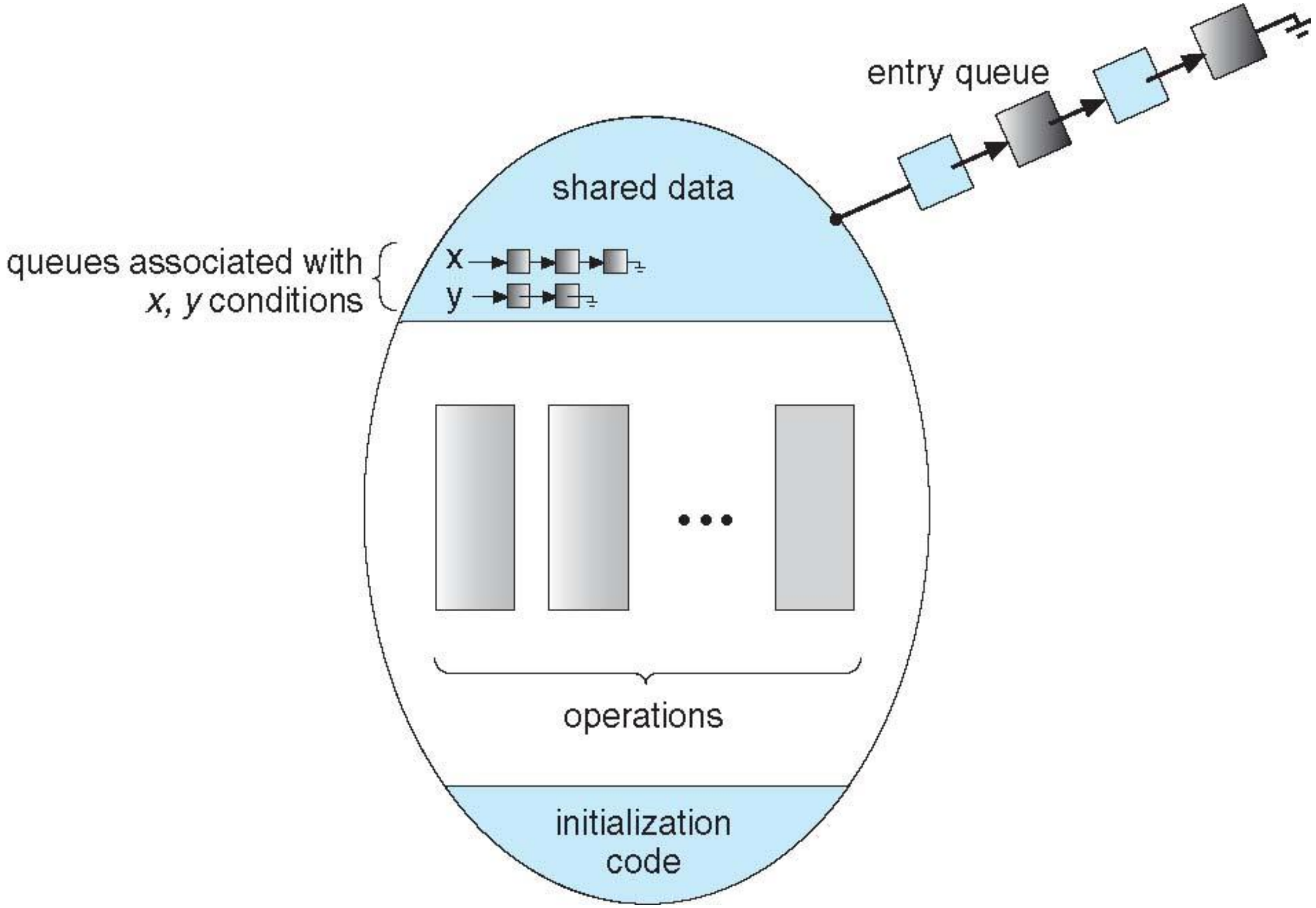


- Monitor construct ensures that only one process at a time is active within the monitor.
- Programmer does not need to code this synchronization constraint explicitly.
- But not powerful enough to model some synchronization schemes
- Need to define additional synchronization mechanisms

Condition Variables

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

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in the language Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including C#, Java

Monitor Solution to Dining Philosophers Problem

- This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher.
- Data structure: **enum {THINKING, HUNGRY, EATING } state[5];**
- Philosopher i can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING).
- Need to declare: **condition self[5];**

Monitor Solution to Dining Philosophers

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

  void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING) self[i].wait;
}
     void putdown (int i) {
         state[i] = THINKING;
                    // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
}
```

Solution to Dining Philosophers (Cont.)

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

     initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
     }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

          EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

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

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

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

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

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

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

- The operation x.wait can be implemented as:

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

Monitor Implementation (Cont.)

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

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

Resuming Processes within a Monitor

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

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;
  }
}
```

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
- A process that needs to access the resource in question must observe the following sequence:

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

- Where R is an instance of type `ResourceAllocator`

Issues in Monitors

- The monitor cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:
  - A process might access a resource without first gaining access permission to the resource
  - A process might never release a resource once it has been granted access to the resource.
  - A process might attempt to release a resource that it never requested.
  - A process might request the same resource twice (without first releasing the resource).
- Ensure correct use of higher-level programmer-defined operations

Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Atomic integers
  - Semaphores
  - Spinlocks
  - Reader-writer versions of both
- On single-cpu systems, spinlocks are replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Pthreads also provide semaphores though semaphores are not part of the Pthreads standard (belong to the POSIX SEM extension)
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Pthreads Synchronization-Mutex Locks

- A mutex lock is used to protect critical sections of code
- Pthreads uses the **pthread_mutex_t** data type for mutex locks.
- A mutex is created with the **pthread_mutex_ init()** function.
- Example usage

```
#include < pthread.h >
Pthread_mutex_t mutex;
/* create the mutex lock */
Pthread_ mutex_ init(&mutex,NULL);
```

Pthreads Synchronization…

- The mutex is acquired and released with the **pthread_mutex_lock()** and **pthread_mutex_unlock()** functions.
- Example

```
/* acquire the mutex lock */
Pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

- All mutex functions return a value of 0 with correct operation; if an error occurs, these functions return a nonzero error code.

Pthreads Synchronization-Semaphores

- Pthreads also provide semaphores (belong to the POSIX SEM extension)
- POSIX specifies two types of semaphores
  - **Named** - has an actual name in the file system and can be shared by multiple unrelated processe
  - **Unnamed**- can be used only by threads belonging to the same process.
- **Sem_init()** function for creating and initializing an unnamed semaphore

```
#include < semaphore.h >
Sem_t sem;
sem_init(&sem, 0, 1); /* Create the semaphore and initialize it to 1 */
```

Pthreads Synchronization-Semaphores…

- **Sem_init()** function for creating and initializing an unnamed semaphore
- The **sem_init()** function is passed three parameters:
  1. A pointer to the semaphore
  2. A flag indicating the level of sharing
  3. The semaphore's initial value
- Example

```
#include < semaphore.h >
sem-_t sem;
sem_init(&sem, 0, 1); /* Create the semaphore and initialize it to 1 */
```

Pthreads Synchronization-Semaphores…

- Pthreads names wait() and signal() operations as **sem_wait()** and **sem_post()** respectively.
- All semaphore functions return 0 when successful, and nonzero when an error condition occurs.
- Sample Code to protect a critical section using the semaphore

```
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

Summary

- Mutual exclusion ensures that a critical section of code is used by only one process or thread at a time.
- Computer hardware provides several operations that ensure mutual exclusion.
- Synchronization problems are used to test nearly every newly proposed synchronization scheme
- Mutex locks and semaphores are efficient tools
- Monitors provide a synchronization mechanism for sharing abstract data types.
- A condition variable provides a method by which a monitor function can block its execution until it is signaled to continue

Review Questions

- What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.
- Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems
- Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.
- Describe how the compare and swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

Review Questions…

- Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit(amount) and withdraw(amount) . These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit() . Describe how a race condition is possible and what might be done to prevent the race condition from occurring
- Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:
  - The lock is to be held for a short duration.
  - The lock is to be held for a long duration.
  - A thread may be put to sleep while holding the lock

Review Questions…

- Design an algorithm for a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (ticks). You may assume the existence of a real hardware clock that invokes a function tick() in your monitor at regular intervals