OperAting Systems
Cs3500 – CH - 7

Prof. Sukhendu Das Deptt. of Computer Science and Engg., IIT Madras, Chennai – 600036.

Email: sdas@cse.iitm.ac.in
URL: http://www.cse.iitm.ac.in/~vplab/os.html

Sept. – 2022.

Synchronization Tools

Outline

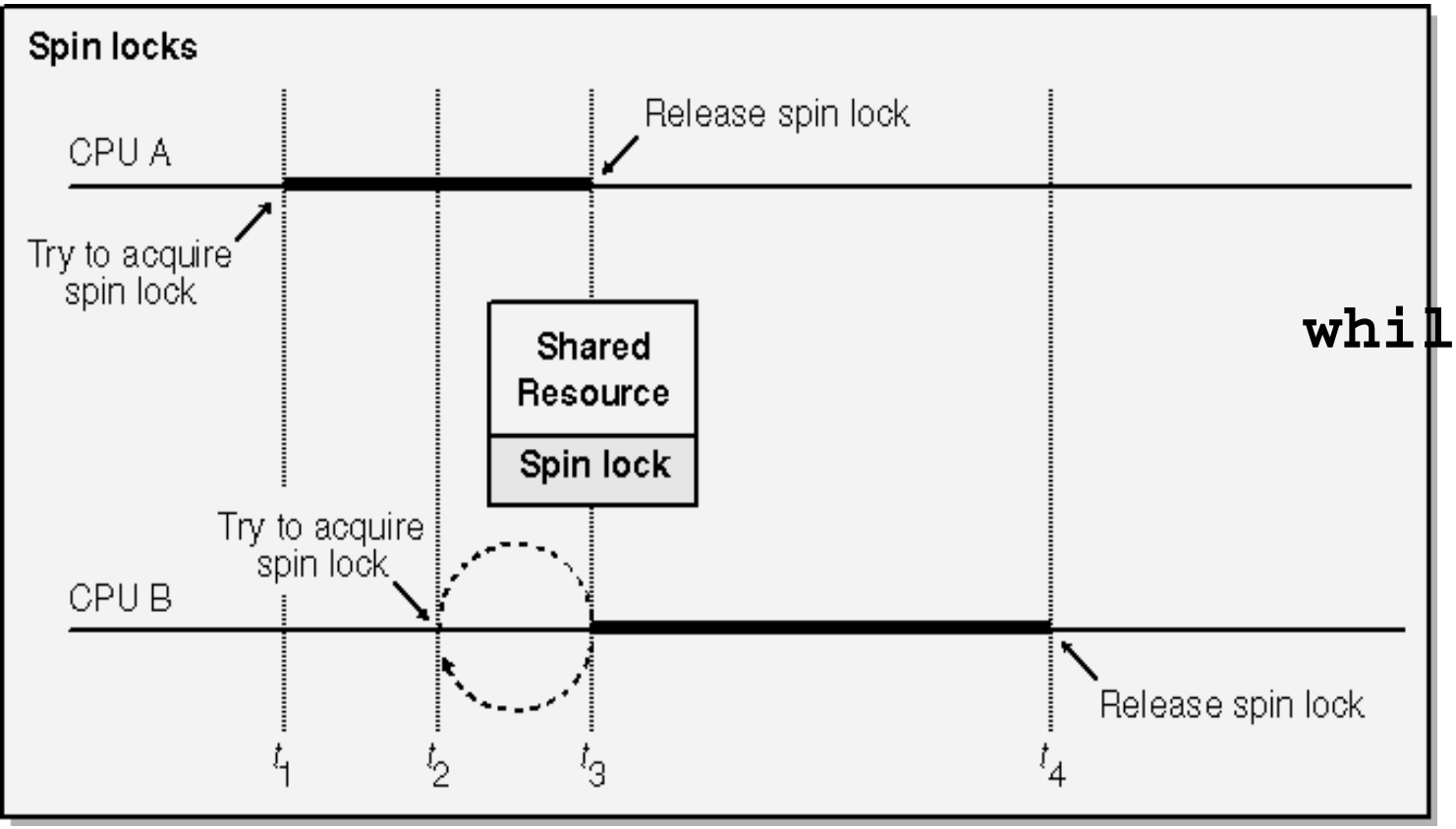
- **Mutex Locks**
- **Semaphores**
- **Monitors**
- **Liveness**
- **Bounded-Buffer Problem**
- **Readers and Writers Problem**
- **Dining-Philosophers Problem**

Mutex Locks

- Hardware based solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is **mutex lock**
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First `acquire()` a lock
  - Then `release()` the lock
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires busy waiting
  - This lock therefore called a spinlock

In software engineering, a *spinlock* is a lock that causes a thread trying to acquire it to simply wait in a loop while repeatedly checking whether the lock is available. Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting

Solution to CS Problem Using Mutex Locks




```
while (true) {
        acquire lock

            critical section

        release lock

    remainder section
                    }
```

The definition of acquire() is as follows:

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

The definition of release() is as follows:

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

Calls to either acquire(), or release() must be performed atomically.

Exercise: mutex locks can be implemented using the CAS operation

```
monitor class Account {
   private int balance := 0
   invariant balance >= 0

   public method boolean withdraw(int amount)
      precondition amount >= 0
   {
      if balance < amount {
         return false
      } else {
         balance := balance - amount
         return true
      }
   }
    public method deposit(int amount)
      precondition amount >= 0
   {
      balance := balance + amount
  }
}
```

```
class Account {
   private lock myLock
   private int balance := 0
   invariant balance >= 0

   public method boolean withdraw(int amount)
     precondition amount >= 0
   {
      myLock.acquire()
      try {
         if balance < amount {
            return false
         } else {
            balance := balance - amount
            return true
         }
      } finally {
         myLock.release()
     }
   }
```

```
public method deposit(int amount)
   precondition amount >= 0
{
   myLock.acquire()
   try {
      balance := balance + amount
    }    finally {
   myLock.release()
               }
}
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes 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++;
}
```

Semaphore Types

- 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 implement a counting semaphore ***S*** as a binary semaphore ← *Exercise*
- With semaphores we can solve various synchronization problems

Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to **1**

```
wait(mutex);
   CS
signal(mutex);
```

    - Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
      - Create a semaphore "**synch**" initialized to 0

```
P1:
   S1;
   signal(synch);

P2:
   wait(synch);
   S2;
```

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*

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

- Waiting queue

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

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);
                    }
}
```

The sleep() operation suspends the process that invokes it.

The wakeup(P) operation resumes (waiting to ready state) the execution of a suspended process P.

These two operations are provided by the operating system as basic system calls.

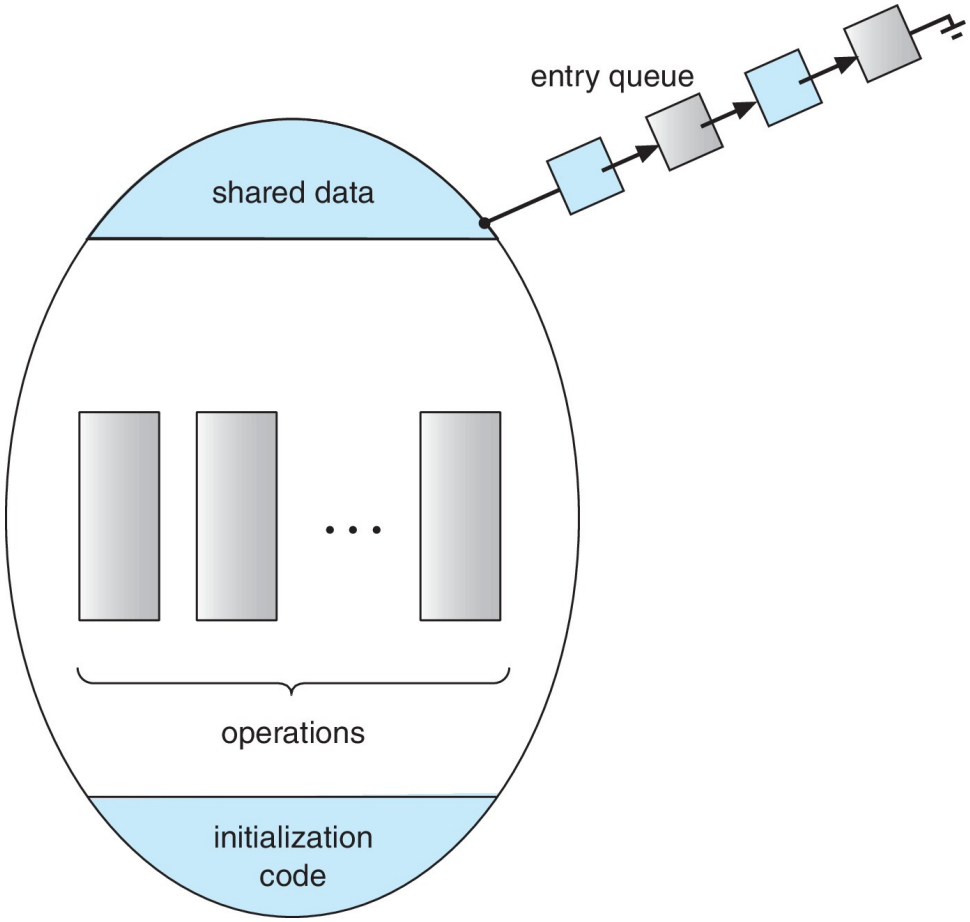
Problems with Semaphores

- Incorrect use of semaphore operations:
  - **signal(mutex) …. wait(mutex)**
  - **wait(mutex) … wait(mutex)**
  - Omitting of **wait (mutex)** and/or **signal (mutex)**
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.
- Difficult to debug
- Threads could communicate using semaphores too.

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*
- Pseudocode syntax of a monitor:

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

        procedure P2 (…) { …. }

        procedure Pn (…) {……}

   initialization code (…) { … }
}
```



Schematic view of a Monitor

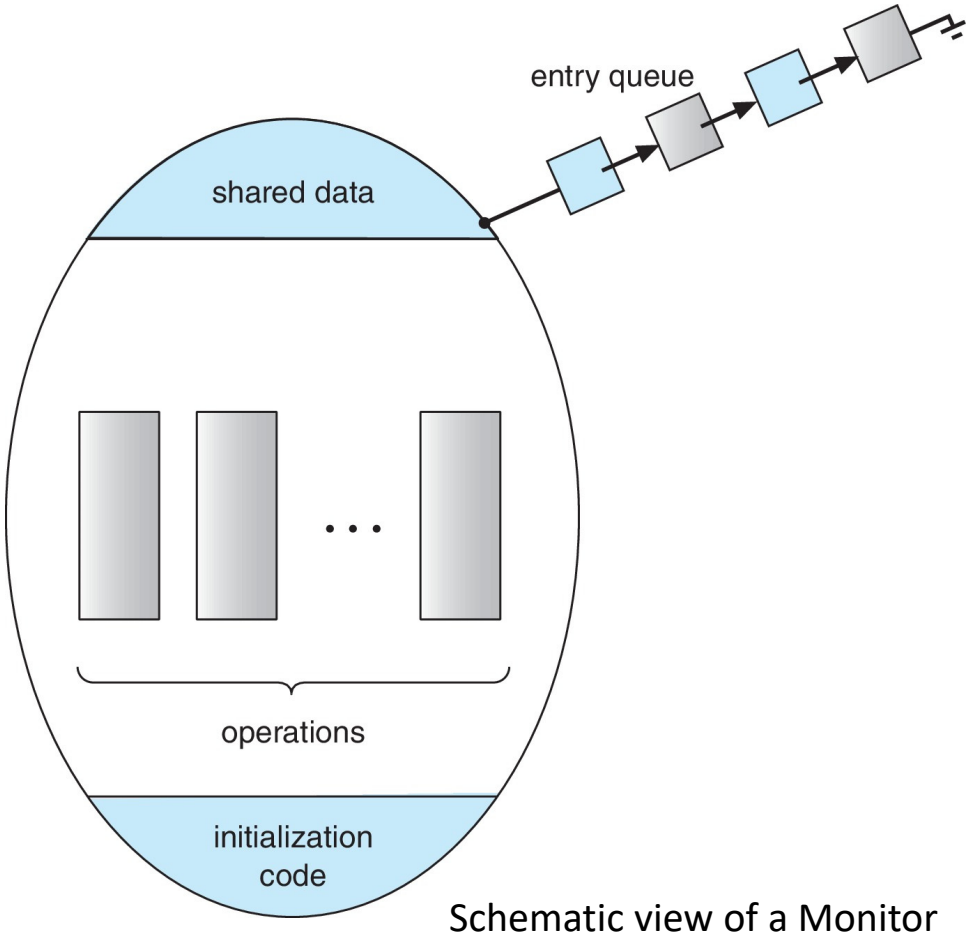
Types of Wait Queues

Two types of Wait Qs in a monitor

- Entry to monitor; a Q of threads waiting to obtain Mutex so that they can enter;

- CVs – each *condition variable* has a Q of threads, waiting in the associated condition

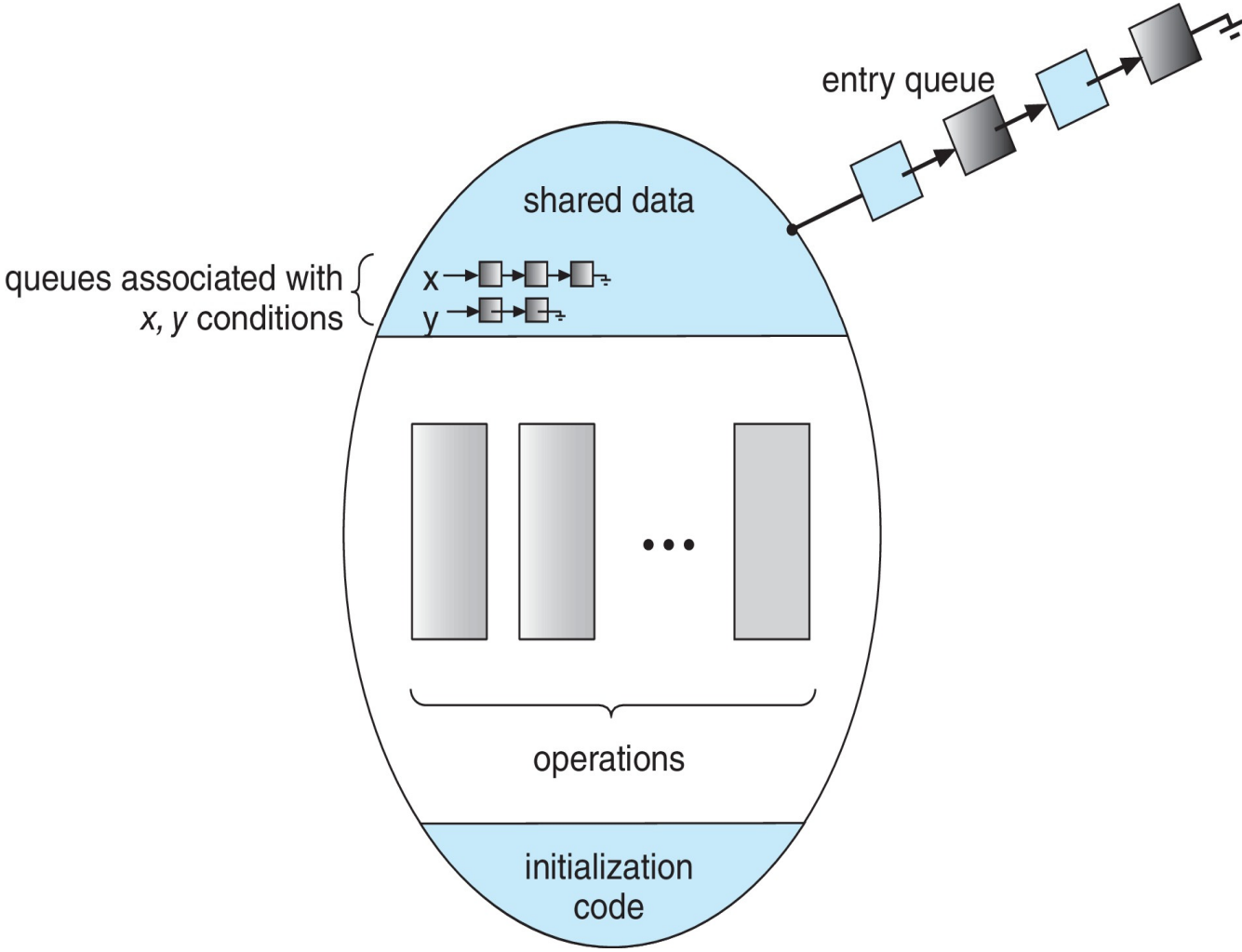
```
Monitor EventTracker { // Trivial Monitor example only
    int numburgers = 0;
    condition hungrycustomer;

    void customerenter() {
        while (numburgers == 0)
        hungrycustomer.wait();
        numburgers -= 1
        }

    void produceburger() {
        ++numburgers;
        hungrycustomer.signal();
        }
```



Schematic view of a Monitor

Condition Variables

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



Monitor with Condition Variables

Usage of Condition Variable Example

- Consider P_1 and P_2 that that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
  - One condition variable "x" initialized to 0
  - One Boolean variable "done"
  - **F1:**

```
S1;
done = true;
x.signal();
```

  - **F2:**

```
if done = false
   x.wait()
S2;
```

```
acquire(m); // Acquire this monitor's lock.
while (!p) { // While the condition/predicate/assertion that we are waiting for is not true...
        wait(m, cv); // Wait on this monitor's lock and condition variable.
                    }
// ... Critical section of code goes here ...

        signal(cv2);
         // cv2 might be the same as cv or different.

release(m); // Release this monitor's lock.
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure ***P*** is replaced by

```
wait(mutex);
    ...
body of P;
    ...
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation Using Semaphores - II

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)  -    signaling processes can use "next"
                                           to suspend themselves
int next_count = 0; // number of processes waiting inside the monitor
```

- Each function ***P*** will be replaced by

```
        wait(mutex);
            ...
body of P;
            ...
        if (next_count > 0)
          signal(next)
        else
          signal(mutex);
```

Logic: A process must execute wait(mutex) before entering the monitor,
and must execute signal(mutex) after leaving the monitor

- Mutual exclusion within a monitor is ensured.
- Use of the signal-and-wait scheme in implementation. Since a signalling process must wait until the resumed process either leaves or waits, an additional binary semaphore, next, is introduced, initialized to 0.

Implementation (Mon + Semaph) – Condition Variables

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

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

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

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

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

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

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

a significant improvement in efficiency is possible – left as exercise

Resuming Processes within a Monitor

- If several processes queued on condition variable **x**, and **x.signal()** is executed, which process should be resumed?

- FCFS frequently not adequate

- Use the **conditional-wait** construct of the form
  x.wait(c)
  where:
  - **c** is an integer (called the priority number) expression; evaluated on "wait" call;

  - The process with lowest number (highest priority) is scheduled next, on x.signal

Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
- The process with the *shortest time* is allocated the resource first

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

- Where, R is an instance of type **ResourceAllocator**
- Where, **t** is the maximum time a process plans to use the resource

A Monitor to Allocate A 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 Monitor (Cont.)

- Usage:
  acquire
  ...
  release

- Incorrect use of monitor operations (more hazards in next slide)
  release() ... acquire()
  acquire() ... acquire())
  Omitting of **acquire()** and/or **release()**

Must inspect all the programs that make use of the *ResourceAllocator* monitor and its managed resource. We must check two conditions to establish the correctness of this system.

First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols.

- Usage:
  acquire
  ...
  release

Unfortunately, the monitor concept 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.

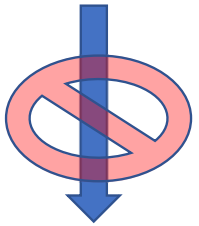
• A process might request the same resource twice (without first releasing the resource).

Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.
- Waiting indefinitely violates the progress and bounded-waiting criteria discussed at the beginning of this chapter.
- **Liveness** refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a liveness failure

- **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);`



- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P_0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are **deadlocked**.

A set of processes is in a ***deadlocked state*** when every process in the set is waiting for an event that can be caused only by another process in the set. The "events" with which we are mainly concerned here are the acquisition and release of resources such as mutex locks and semaphores.

Liveness

- Other forms of deadlock:
- **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
  - Solved 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 involved/used.

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes:

1. **Bounded-Buffer Problem**
2. **Readers and Writers Problem**
3. **Dining-Philosophers Problem**

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **mutex** initialized to the value **1**
- Semaphore **full** initialized to the value **0**
- Semaphore **empty** initialized to the value **n**

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
     ...
   /* produce an item in next_produced
*/
     ...
   wait(empty);
   wait(mutex);
     ...
   /* add next produced to the buffer
*/
     ...
   signal(mutex);
   signal(full);
}
```

- The structure of the consumer process

```
while (true) {
   wait(full);
   wait(mutex);
     ...
  /* remove an item from buffer to
next_consumed */
     ...
   signal(mutex);
   signal(empty);
     ...
   /* consume the item in next
consumed */
     ...
}
```

Readers-Writers Problem

- A data set (D/B) 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 (only) to read at the same time; no issues
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities

- Shared Data
  - →Data set
  - →Semaphore **mutex** initialized to 1
  - → Semaphore **rw_mutex** initialized to 1
  - → Integer **read_count** initialized to 0

- The structure of a writer process

```
while (true) {
    wait(rw_mutex);
        …
    /* writing is performed */
        …
    signal(rw_mutex);
}
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

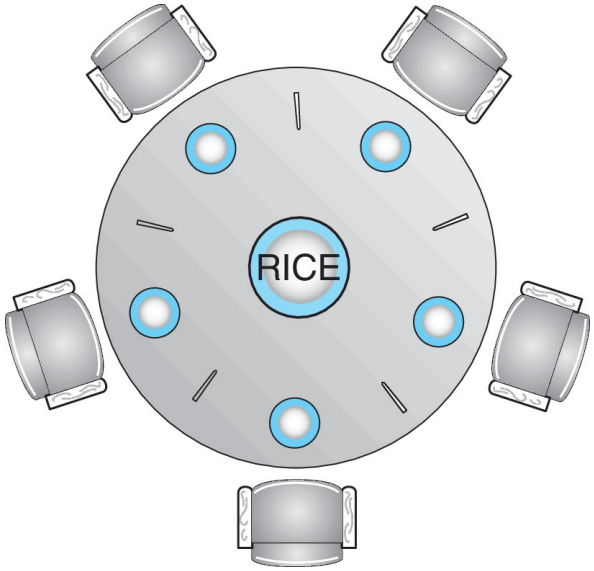
```
while (true){
        wait(mutex);
    read_count++;
    if (read_count == 1) /* first reader */
        wait(rw_mutex);
        signal(mutex);
    …
    /* reading is performed */
    …
        wait(mutex);
    read_count--;
    if (read_count == 0) /* last reader */
        signal(rw_mutex);
        signal(mutex);
}
```

Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the “First reader-writer” problem.
- The “Second reader-writer” problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
- Both the first and second may result in starvation. leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowel of rice in the middle.
- They spend their lives alternating thinking and eating.
- They do not 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, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

Semaphore Solution:

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

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

    /* eat for awhile */

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

    /* think for awhile */

}
```

Phil – 1, 2,…,5
All concurrently operate

Who gets a chance ??

- What is the problem with this algorithm? – MUTEX OK ?

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
        enum {THINKING; HUNGRY, EATING} state [5];
        condition self [5]; //Monitor usage

        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;
           }
}
```

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).

Solution to Dining Philosophers (Cont.)

Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation.

This solution ensures that no two neighbours are eating simultaneously and that no deadlocks will occur. However, it is possible for a philosopher to starve to death. Left as an exercise for you.

- 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 << -- proof is left as exercise – Soln later

Self – study;

- POSIX synchronization (kernel level)

- Transactional memory (STM, HTM);

- OpenMP criticial constructs

- Functional Prog. Languages (Erlang, Scala)

-