CISC 3320
C21c OS Tools for Synchronization

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

Outline

- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

OS Tools for Synchronization

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
  - Mutex lock
  - Semaphore (with/without busy-waiting)
  - Monitor

Mutex Locks

- Protect a critical section by
  - first acquire() a lock, then
  - release() the lock
- Calls to acquire() and release() must be atomic
- Boolean variable indicating if lock is available or not

Solution to Critical-section Problem Using Locks

```
while (true) {
        acquire(); /* acquire lock */

        critical section

        release(); /* release lock  */

        remainder section
}
```

Mutex Lock Definitions

- These two functions must be implemented atomically.

```
acquire() {
    while (!available)

       ; /* busy wait */

    available = false;

 }

release() {

    available = true;

}
```

Implementing Mutex Lock

- Both test-and-set and compare-and-swap can be used to implement these functions.
- How?

Mutex: Remark

- acquire() and release() usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires busy waiting
    - This type of mutex lock therefore called a spinlock

Questions?

- Concept of mutex lock
- Implementation of mutex lock
- Concept of spinlock
- Advantage of disadvantage of spinlock

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()
    - Sometimes also called down() and up() (often in Unix)

Definition: wait() and signal()

wait()/P()/down()

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

signal()/V()/up()

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

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

Solution using Semaphore

- Consider P1 and P2 that require S1 to happen before S2

```
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 without 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;
```

```
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);
   }
}
```

sleep() and wakeup(P)

- The sleep() operation suspends the process that invokes it.
- The wakeup(P) operation resumes 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

Questions?

- Definition Semaphore
- Implementation with or without busy-waiting
- Problems with semaphore

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

Syntax of a Monitor

- Pseudocode syntax of a monitor:

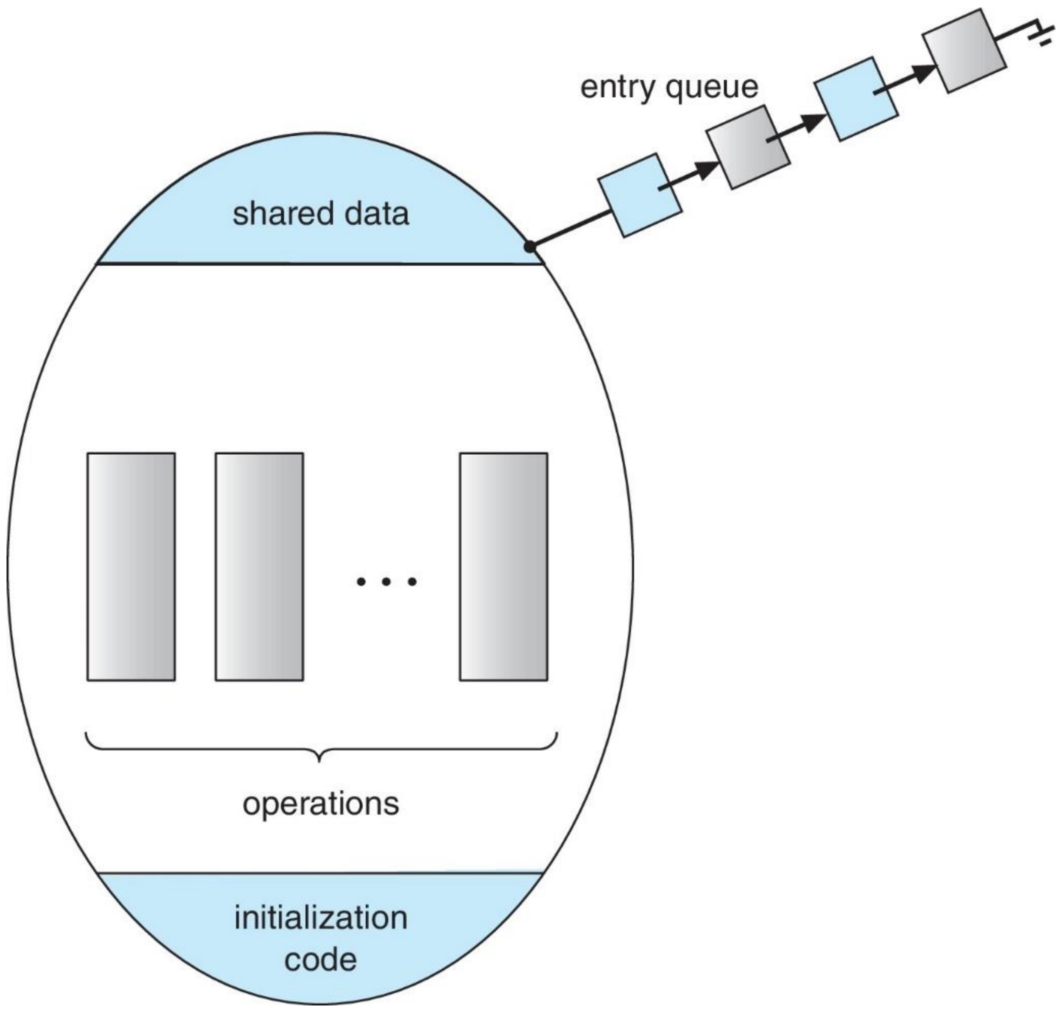
```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }


  function P2 (…) { …. }

  function Pn (…) {……}


  initialization code (…) { … }
}
```

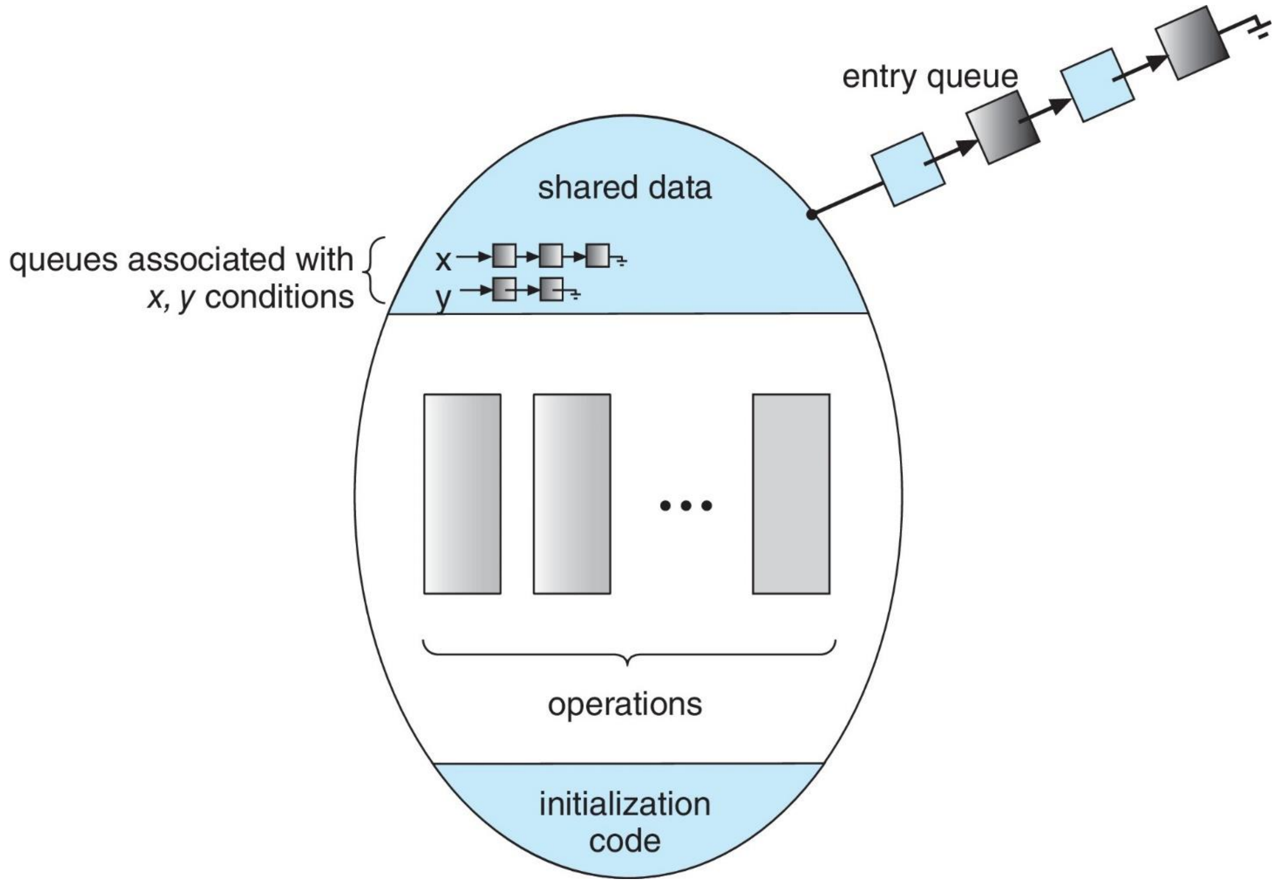
Schematic view of a Monitor



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



Choices of Condition Variables

- 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 Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Implementation Using Semaphores

- Variables

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

- Each function ***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--;
```

- 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 variable **`x`**, and **`x.signal()`** is executed, which process 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

Resuming Processes

- 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(t);
```

- 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;
        }
}
```

Questions?

- Mutex lock
- Semaphore (with/without busy-waiting)
- Monitor

Synchronization Issues

- Liveness
  - Deadlock
  - Starvation
  - Priority inversion

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

- **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

```
         P0                          P1

     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**.

Starvation

- 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
  - Solved via priority-inheritance protocol

Priority Inheritance Protocol

- Consider the scenario with three processes P1, P2, and P3. P1 has the highest priority, P2 the next highest, and P3 the lowest. Assume a resouce P3 is assigned a resource R that P1 wants. Thus, P1 must wait for P3 to finish using the resource. However, P2 becomes runnable and preempts P3. What has happened is that P2 - a process with a lower priority than P1 - has indirectly prevented P3 from gaining access to the resource.
- To prevent this from occurring, a priority inheritance protocol is used. This simply allows the priority of the highest thread waiting to access a shared resource to be assigned to the thread currently using the resource. Thus, the current owner of the resource is assigned the priority of the highest priority thread wishing to acquire the resource.

Questions?

- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation