

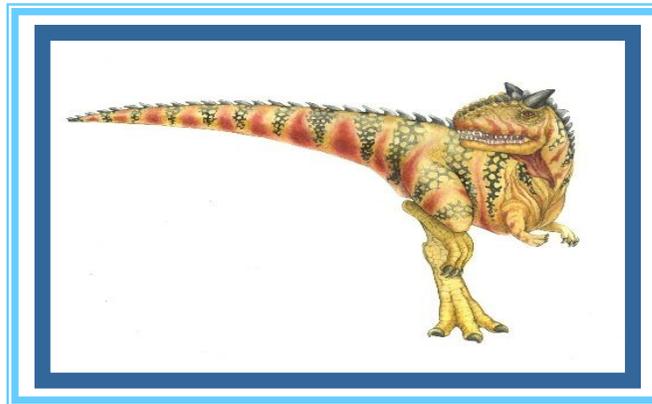
CSMC 412

Operating Systems

Prof. Ashok K Agrawala

Set 6

Synchronization Tools



Synchronization Tools

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

Objectives

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

Background

- Processes can execute concurrently
 - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Systems = Objects + Activities

- **Safety** is a property of **objects**, and groups of objects, that participate across multiple activities.
 - Can be a concern at many different levels: objects, composites, components, subsystems, hosts, ...
- **Liveness** is a property of **activities**, and groups of activities, that span across multiple objects.
 - Levels: Messages, call chains, threads, sessions, scenarios, scripts workflows, use cases, transactions, data flows, mobile computations, ...

Violating Safety

- Data can be shared by threads
 - Scheduler can interleave or overlap threads arbitrarily
 - Can lead to *interference*
 - Storage corruption (e.g. a *data race/race condition*)
 - Violation of representation invariant
 - Violation of a protocol (e.g. *A* occurs before *B*)

How does this apply to OSs?

- Any resource that is shared could be accessed inappropriately
 - Shared memory
 - Kernel threads
 - Processes (shared memory set up by kernel)
 - Shared resources
 - Printer, Video screen, Network card, ...
- OS must protect shared resources
 - And provide processes a means to protect their own abstractions

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 */  
}
```

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state cnt = 0

Start: both threads ready to run. Each will increment the global count.

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state cnt = 0

y = 0



*T1 executes, grabbing
the global counter value into y.*

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state cnt = 0

y = 0



y = 0

T1 is pre-empted. T2 executes, grabbing the global counter value into y.

Data Race Example

```
static int cnt = 0;  
t1.run() {  
    int y = cnt;  
    cnt = y + 1;  
}  
t2.run() {  
    int y = cnt;  
    cnt = y + 1;  
}
```

Shared state **cnt = 1**

y = 0



y = 0

T2 executes, storing the incremented cnt value.

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state **cnt = 1**

$y = 0$



$y = 0$

*T2 completes. T1
executes again, storing the
old counter value (1) rather
than the new one (2)!*

But When I Run it Again?

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state cnt = 0

Start: both threads ready to run. Each will increment the global count.

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state cnt = 0

y = 0



*T1 executes, grabbing
the global counter value into y.*

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

$y = 0$

Shared state **cnt = 1**



T1 executes again, storing the counter value

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state cnt = 1

y = 0



y = 1

*T1 finishes. T2 executes,
grabbing the global
counter value into y.*

Data Race Example

```
static int cnt = 0;
t1.run() {
    int y = cnt;
    cnt = y + 1;
}
t2.run() {
    int y = cnt;
    cnt = y + 1;
}
```

Shared state **cnt = 2**

$y = 0$



$y = 1$

T2 executes, storing the incremented cnt value.

What happened?

- In the first example, **t1** was preempted after it read the counter but before it stored the new value.
 - Depends on the idea of an *atomic action*
 - Violated an object invariant
- A particular way in which the execution of two threads is interleaved is called a *schedule*. We want to prevent this undesirable schedule.
- Undesirable schedules can be hard to reproduce, and so hard to debug.

Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{counter = 4}

Question

- If you run a program with a race condition, will you always get an unexpected result?
 - No! It depends on the scheduler
 - ...and on the other threads/processes/etc that are running on the same CPU
- Race conditions are hard to find

Disabling Interrupts

- Doesn't work for multiprocessors
- Doesn't permit different groups of critical sections

Synchronization

```
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);
void run() {
    Mutex_Lock (&lock);
    int y = cnt;
    cnt = y + 1;
    Mutex_Unlock (&lock);
}
```

***Lock**, for protecting
The shared state*

***Acquires the lock**;
Only succeeds if not
held by another
thread*

Releases the lock

Java-style synchronized block

```
static int cnt = 0;
struct Mutex lock;
Mutex_Init(&lock);
void run() {
    synchronized (lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

***Lock**, for protecting
The shared state*

***Acquires the lock**;
Only succeeds if not
held by another
thread*

Releases the lock

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 0



T1 acquires the lock

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 0

y = 0



T1 reads cnt into y

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

$y = 0$

Shared state $cnt = 0$



*T1 is pre-empted.
T2 attempts to
acquire the lock but fails
because it's held by
T1, so it blocks*

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

$y = 0$

Shared state **cnt = 1**



*T1 runs, assigning
to cnt*

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

y = 0

Shared state cnt = 1



*T1 releases the lock
and terminates*

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

y = 0

Shared state cnt = 1



*T2 now can acquire
the lock.*

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

Shared state cnt = 1

y = 0



T2 reads cnt into y.

y = 1

Applying synchronization

```
int cnt = 0;
t1.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
t2.run() {
    synchronized(lock) {
        int y = cnt;
        cnt = y + 1;
    }
}
```

y = 0

Shared state **cnt = 2**



*T2 assigns cnt,
then releases the lock*

y = 1

Mutexes (locks)

- Only one thread can “acquire” a mutex
 - Other threads block until they can acquire it
 - Used for implementing [critical sections](#)
- A critical section is a piece of code that should not be interleaved with code from another thread
 - Executed [atomically](#)
- We'll look at other ways to implement critical sections later ...

Mutex Policies

- What if a thread already holds the mutex it's trying to acquire?
 - Re-entrant mutexes: The thread can reacquire the same lock many times. Lock is released when object unlocked the corresponding number of times
 - This is the case for Java
 - Non-reentrant: Deadlock! (defined soon.)
 - This is the case in GeekOS
- What happens if a thread is killed while holding a mutex? Or if it just forgets to release it
 - Could lead to deadlock

Java Synchronized statement

- **synchronized (obj) { statements }**
- Obtains the lock on **obj** before executing statements in block
 - **obj** can be any Object
- Releases the lock when the statement block completes
 - Either normally, or due to a return, break, or exception being thrown in the block
- Can't forget to release the lock!

Synchronization not a Panacea

- Two threads can block on locks held by the other; this is called *deadlock*

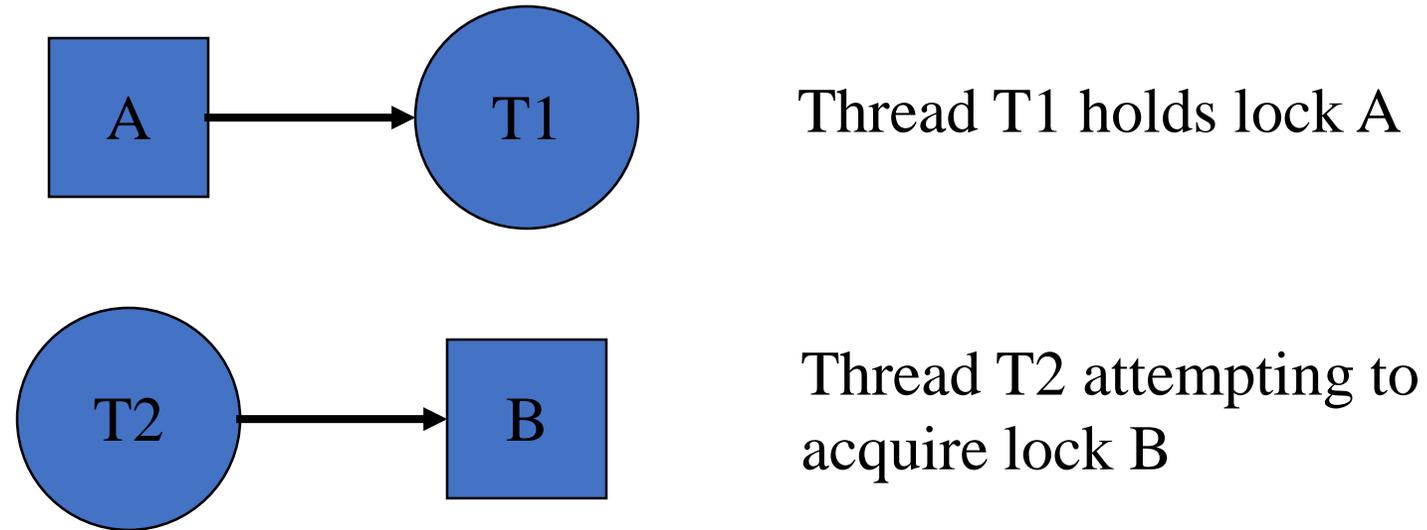
```
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}
```

```
T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```

Deadlock

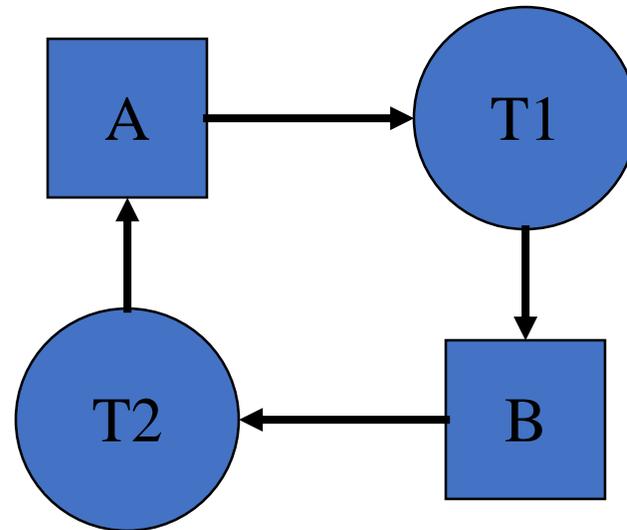
- Quite possible to create code that deadlocks
 - Thread 1 holds lock on **A**
 - Thread 2 holds lock on **B**
 - Thread 1 is trying to acquire a lock on **B**
 - Thread 2 is trying to acquire a lock on **A**
 - Deadlock!
- Not easy to detect when deadlock has occurred
 - other than by the fact that nothing is happening

Deadlock: Wait graphs



Deadlock occurs when there is a cycle in the graph

Wait graph example



T1 holds lock on **A**

T2 holds lock on **B**

T1 is trying to acquire a lock on **B**

T2 is trying to acquire a lock on **A**

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the N processes

Two-task Solution

- Two tasks, T_0 and T_1 (T_i and T_j)
- Three solutions presented.

Algorithm 1

- Threads share a common integer variable **turn**
 - **Turn** takes values 0 and 1
- Initialize **turn** to 0
- Entry Section for thread i
 - If **turn**== i , thread i is allowed to proceed, else yield
- Exit Section for thread i
 - **turn**== $(1-i)$

Algorithm 1

- Satisfies mutual exclusion but not progress.
 - Processes are forced to enter their critical sections alternately.
 - One process not in its critical section thus prevents the other from entering its critical section.

Algorithm 2

- Boolean flags to indicate thread's interest in entering critical section

- Entry Code

```
if (t == 0) {  
    flag0 = true;  
    while(flag1 == true)  
        Thread.yield();  
}  
else {  
    flag1 = true;  
    while (flag0 == true)  
        Thread.yield();  
}
```

- Exit Code

```
if (t == 0)  
    flag0 = false;  
else  
    flag1 = false;
```

- Initialize

- Both flags to false

Algorithm 2

- Satisfies mutual exclusion, but not progress requirement.
 - Both processes can end up setting their flag[] variable to true, and thus neither process enters its critical section!

Algorithm 3 Peterson's Solution

- Combine ideas from 1 and 2

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 requirements are met:
 1. Mutual exclusion is preserved
 - P_i enters CS only if:
either **flag[j] = false** or **turn = i**
 2. Progress requirement is satisfied
 3. Bounded-waiting requirement is met

Algorithm 3

- Meets all three requirements; solves the critical-section problem for two processes.
 - One process is always guaranteed to get into its critical section.
 - Processes are forced to take turns when they both want to get in.

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- Notation \leq lexicographical order (ticket #, process id #)
 - $(a,b) < c,d$ if $a < c$ or if $a = c$ and $b < d$
 - $\max(a_0, \dots, a_{n-1})$ is a number, k , such that $k \geq a_i$ for $i = 0, \dots, n - 1$
- Shared data

boolean choosing[n];

int number[n];

Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
do {  
    choosing[i] = true;  
    number[i] = max(number[0], number[1], ..., number [n - 1])+1;  
    choosing[i] = false;  
    for (j = 0; j < n; j++) {  
        while (choosing[j]) ;  
        while ((number[j] != 0) && (number[j,j] < number[i,i])) ;  
    }  
    critical section  
    number[i] = 0;  
    remainder section  
} while (1);
```

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - Essentially free of race conditions in kernel mode

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - **Atomic** = non-interruptible
 - Either test memory word and set value
 - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

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

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

Solution using test_and_set()

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

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

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

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

Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

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

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

Mutex Locks

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

acquire() and release()

- ```
acquire() {
 while (!available)
 ; /* busy wait */
 available = false;;
}
```
- ```
release() {  
    available = true;  
}
```
- ```
do {
 acquire lock
 critical section
 release lock
 remainder section
} while (true);
```

# Semaphore

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

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

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

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

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

# Information Implications of Semaphore

- A process has synch points
  - To go past a synch point certain conditions must be true
    - Conditions depend not only on ME but other processes also
    - Have to confirm that the conditions are true before proceeding, else have to wait.
- P(S) – Wait (S)
  - If can complete this operation
    - Inform others through changed value of S
    - Proceed past the synch point
  - If can not complete
    - Wait for the event when S becomes  $>0$
- V(S) – Signal (S)
  - Inform others that I have gone past a synch point.

# Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider  $P_1$  and  $P_2$  that require  $S_1$  to happen before  $S_2$

Create a semaphore “**synch**” initialized to 0

**P1:**

```
S1;
signal (synch) ;
```

**P2:**

```
wait (synch);
S2;
```

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

# Semaphore 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
  - Also known as mutex locks
- Can implement a counting semaphore  $S$  as a binary semaphore
- Provides mutual exclusion

Semaphore  $S$ ; // initialized to 1

```
P(S);
criticalSection();
V(S);
```

# Implementing $S$ as a Binary Semaphore

- Data structures:

**binary-semaphore S1, S2;**

**int C;**

- Initialization:

**S1 = 1**

**S2 = 0**

**C = initial value of semaphore S**

# Implementing S

- *wait* operation

```
wait(S1);
C--;
if (C < 0) {
 signal(S1);
 wait(S2);
}
signal(S1);
```

- *signal* operation

```
wait(S1);
C ++;
if (C <= 0)
 signal(S2);
else
 signal(S1);
```

# 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
- ```
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);
    }
}
```

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
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>...</code>	<code>...</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

- **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](#)

Problems with Semaphores

- 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

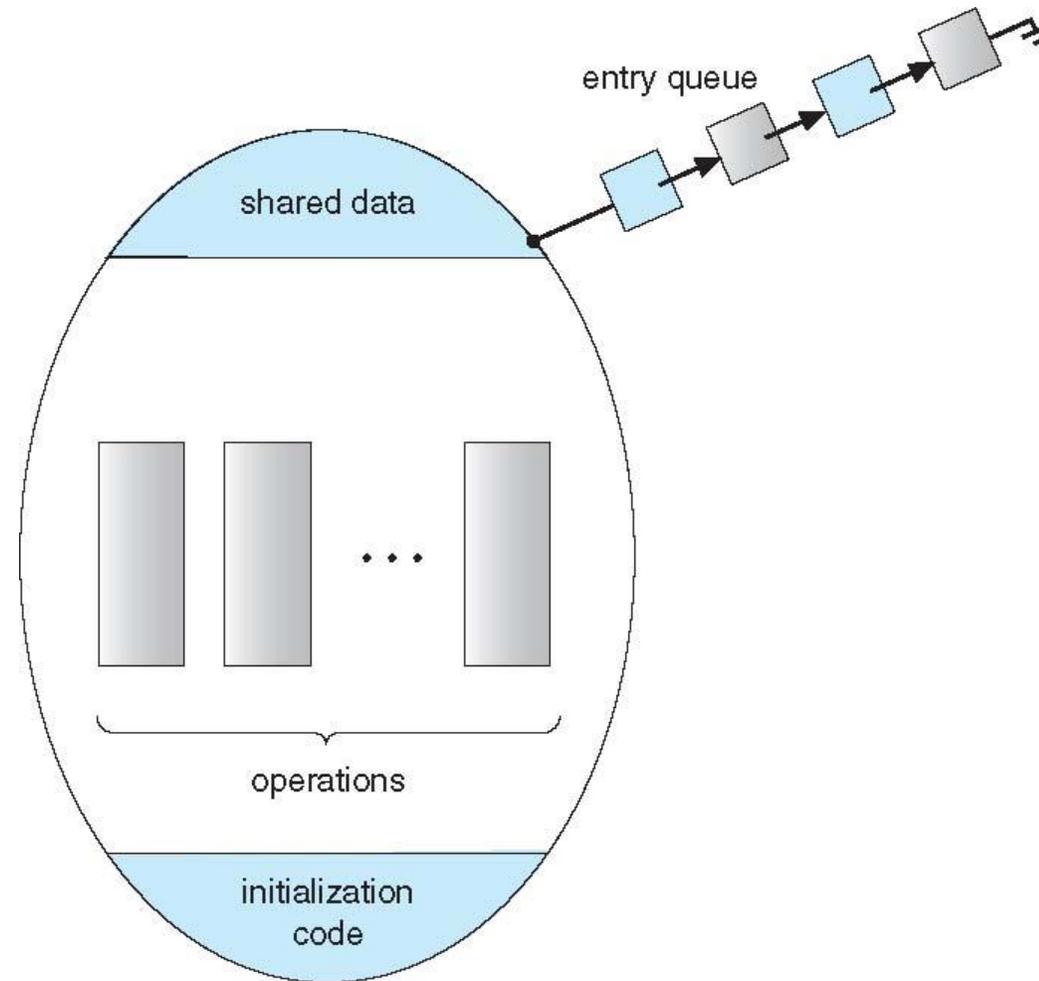
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization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

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

    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
}
```

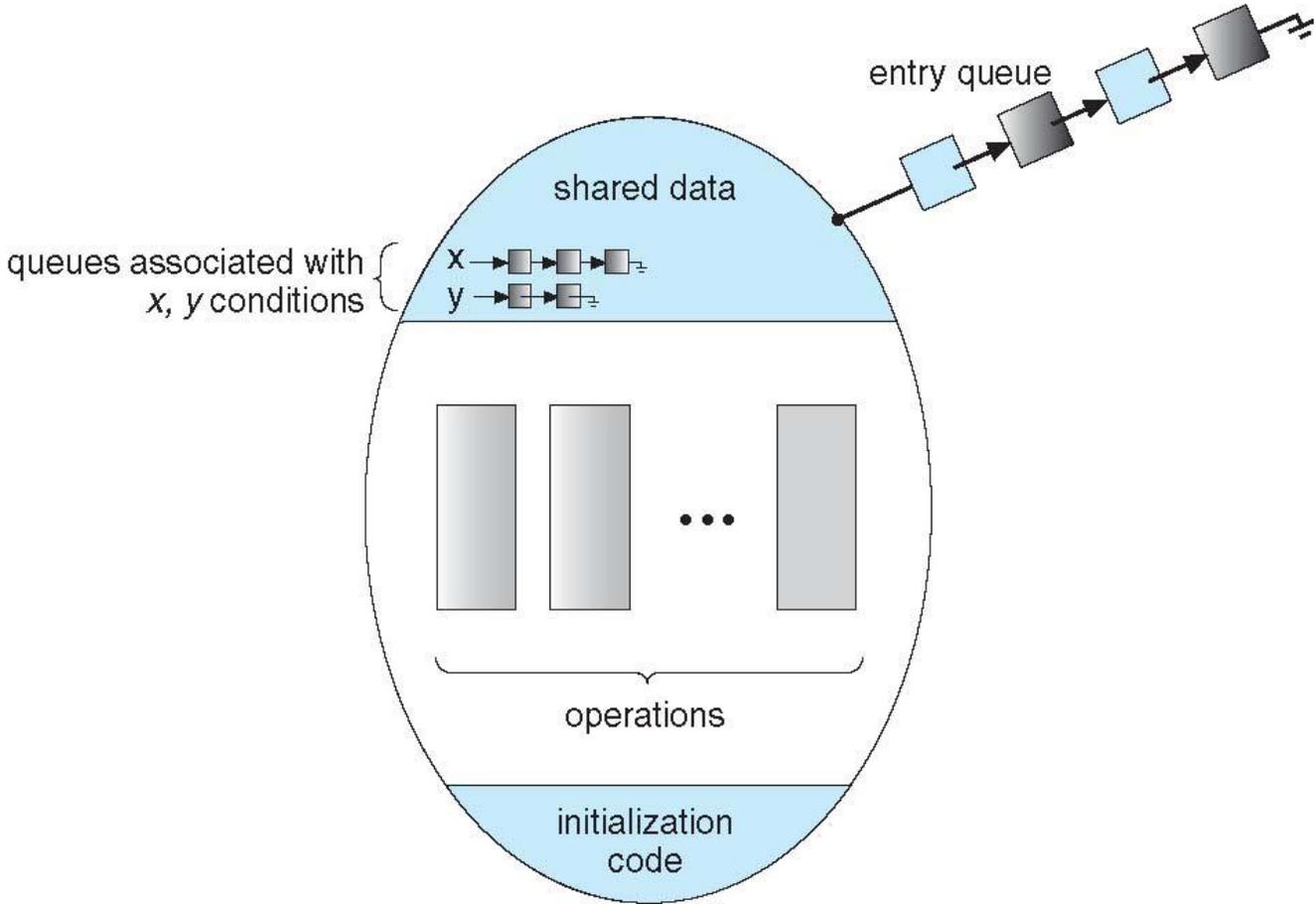
Schematic view of a Monitor



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



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 parallel. 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 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:

```
0)      semaphore x_sem; // (initially =  
        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 $x.\text{signal}()$ executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form $x.\text{wait}(c)$
 - 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 resource ;
```

```
...
```

```
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;
    }
}
```

End of Chapter 6

