

CSMC 412

Operating Systems Prof. Ashok K Agrawala

© 2004 Ashok Agrawala
Set 6
(chapter 7)

Announcements

- Project #2
 - ☞ Due October 1

- Reading
 - ☞ Chapter 7

Concurrency and Synchronization

- Concurrency
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000

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

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;
}
```

Shared state **cnt = 1**

y = 0



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.

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

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;    y = 0
    cnt = y + 1;
  }
}
```

Shared state cnt = 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;    y = 0
    cnt = y + 1;
  }
}
```

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;    y = 0
    cnt = y + 1;
  }
}
```

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;    y = 0
    cnt = y + 1;
  }
}
```

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;    y = 0
    cnt = y + 1;
  }
}
```

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;    y = 0
    cnt = y + 1;
  }
}
```

Shared state cnt = 1



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;    y = 0
    cnt = y + 1;
  }
}
```

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

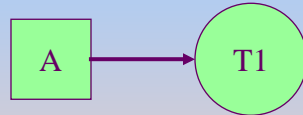
- Two threads can block on locks held by the other; this is called *deadlock*

```
Object A = new Object();
Object B = new Object();
T1.run() {
    synchronized (A) {
        synchronized (B) {
            ...
        }
    }
}
T2.run() {
    synchronized (B) {
        synchronized (A) {
            ...
        }
    }
}
```

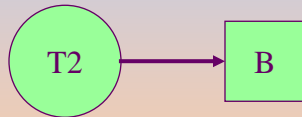
Deadlock

- 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



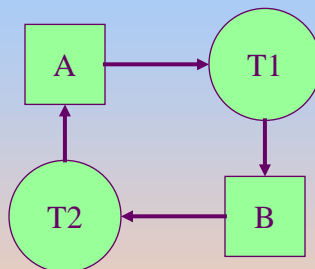
Thread T1 holds lock A



Thread T2 attempting to acquire lock B

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**

Key Ideas

- Multiple threads can run simultaneously
 - ↳ Either truly in parallel on a multiprocessor
 - ↳ Or can be scheduled on a single processor
 - ▢ A running thread can be pre-empted at any time
- Threads can share data
 - ↳ Need to prevent interference
 - ▢ Synchronization is one way, but not the only way
 - ↳ Overuse of synchronization can create deadlock
 - ▢ Violation of liveness

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.

Synchronization

- Let us revisit the issues of synchronization for the producer/consumer problem.
- Have N processes
 - ☞ There are Synch points in each process
 - ☞ Proceeding past a synch point requires some global conditions to be true.
 - ☞ Process must confirm that the conditions are true before it proceeds
- Global conditions
 - ☞ How to keep track of them
 - ☞ How to check them and update them
 - ☞ How to initiate an action based on them

Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Producer process

```
item nextProduced;

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

Bounded-Buffer

- Consumer process

```
item nextConsumed;

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

Bounded Buffer

- The statements

```
counter++;  
counter--;
```

must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

- The statement “**count++**” may be implemented in machine language as:

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

- The statement “**count--**” may be implemented as:

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

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded Buffer

- Assume **counter** is initially 5. One interleaving of statements is:

producer: **register1 = counter** (*register1 = 5*)
producer: **register1 = register1 + 1** (*register1 = 6*)
consumer: **register2 = counter** (*register2 = 5*)
consumer: **register2 = register2 - 1** (*register2 = 4*)
producer: **counter = register1** (*counter = 6*)
consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

Race Condition

- **Race condition:** The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

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.

Assumptions

- Some instructions are atomic
 - ☞ load, store, test instructions cannot be interrupted
- Hardware configuration can vary
 - ☞ Single or multiple processors

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {  
    entry section  
    critical section  
    exit section  
    reminder section  
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
 - ☞ `int turn;`
initially `turn = 0`
 - ☞ `turn = i` $\Rightarrow P_i$ can enter its critical section
- Process P_i

```
do {  
    while (turn != i);  
    critical section  
    turn = j;  
    reminder section  
} while (1);
```

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

- Shared variables
 - ☞ **boolean flag[2];**
initially **flag [0] = flag [1] = false.**
 - ☞ **flag [i] = true** $\Rightarrow P_i$ ready to enter its critical section
- Process P_i

```
do {  
    flag[i] := true;  
    while (flag[j]) ;  
    critical section  
    flag [i] = false;  
    remainder section  
} while (1);
```

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

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {  
    flag [i]:= true;  
    turn = j;  
    while (flag [j] and turn = j) ;  
    critical section  
    flag [i] = false;  
    remainder section  
} while (1);
```

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., 1,2,3,3,3,4,5...

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);
```

Synchronization Hardware

- Uniprocessor machine
 - Disable/enable interrupts:
Process P_i
do {
 while (!Interrupts_Enabled());
 Disable_Interrupts()
 critical section
 Enable_Interrupts()
 remainder section
}

Disabling Interrupts

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

Synchronization Hardware

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {  
    boolean rv = target;  
    target = true;  
  
    return rv;  
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Process P_i

```
do {  
    while (TestAndSet(lock)) ;  
    critical section  
    lock = false;  
    remainder section  
}
```

Synchronization Hardware

- Atomically swap two variables.

```
void Swap(boolean &a, boolean &b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

```
boolean lock;  
boolean waiting[n];
```

- Process P_i

```
do {  
    key = true;  
    while (key == true)  
        Swap(lock, key);  
    critical section  
    lock = false;  
    remainder section  
}
```

Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore S – non-negative integer variable
- can only be accessed via two indivisible (atomic) operations

P(S)

wait (S):

```
while  $S \leq 0$  do no-op;  
S--;
```

V(S)

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.

Critical Section of n Processes

- Shared data:
`semaphore mutex; //initially mutex = 1`

- Process P_i :

```
do {  
    wait(mutex);  
    critical section  
    signal(mutex);  
    remainder section  
} while (1);
```

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {  
    int value;  
    struct process *L;  
} semaphore;
```
- Assume two simple operations:
 - **block** suspends the process that invokes it.
 - **wakeup(P)** resumes the execution of a blocked process **P**.

Implementation

- Semaphore operations now defined as

```
wait(S):  
    S.value--;  
    if (S.value < 0) {  
        add this process to S.L;  
        block;  
    }  
  
signal(S):  
    S.value++;  
    if (S.value <= 0) {  
        remove a process P from S.L;  
        wakeup(P);  
    }
```

Semaphore as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore $flag$ initialized to 0
- Code:

```
       $P_i$                  $P_j$   
      ⋮                  ⋮  
      A                  wait(flag)  
      signal(flag)       B
```

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.

Two Types of Semaphores

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

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);
```

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Shared data

semaphore full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1

Bounded-Buffer Problem Producer Process

```
do {  
    ...  
    produce an item in nextp  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    add nextp to buffer  
    ...  
    signal(mutex);  
    signal(full);  
} while (1);
```

Bounded-Buffer Problem Consumer Process

```
do {  
    wait(full)  
    wait(mutex);  
    ...  
    remove an item from buffer to nextc  
    ...  
    signal(mutex);  
    signal(empty);  
    ...  
    consume the item in nextc  
    ...  
} while (1);
```

Readers-Writers Problem

- Shared data

semaphore mutex, wrt;

Initially

mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process

wait(wrt);

...

writing is performed

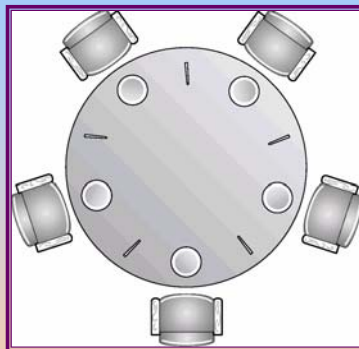
...

signal(wrt);

Readers-Writers Problem Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
    wait(rt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex);
```

Dining-Philosophers Problem



- Shared data

```
semaphore chopstick[5];
```

Initially all values are 1

Dining-Philosophers Problem

```
■ Philosopher i:  
do {  
    wait(chopstick[i])  
    wait(chopstick[(i+1) % 5])  
    ...  
    eat  
    ...  
    signal(chopstick[i]);  
    signal(chopstick[(i+1) % 5]);  
    ...  
    think  
    ...  
} while (1);
```

Critical Regions

- High-level synchronization construct
- A shared variable **v** of type **T**, is declared as:
v: shared T
- Variable **v** accessed only inside statement
region v when B do S

where **B** is a boolean expression.
- While statement **S** is being executed, no other process can access variable **v**.

Critical Regions

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression B is evaluated. If B is true, statement S is executed. If it is false, the process is delayed until B becomes true and no other process is in the region associated with v .

Example – Bounded Buffer

- Shared data:

```
struct buffer {  
    int pool[n];  
    int count, in, out;  
}
```

Bounded Buffer Producer Process

- Producer process inserts **nextp** into the shared buffer

```
region buffer when( count < n) {  
    pool[in] = nextp;  
    in:= (in+1) % n;  
    count++;  
}
```

Bounded Buffer Consumer Process

- Consumer process removes an item from the shared buffer and puts it in **nextc**

```
region buffer when (count > 0) {  
    nextc = pool[out];  
    out = (out+1) % n;  
    count--;  
}
```

Implementation region x when B do S

- Associate with the shared variable x , the following variables:
semaphore mutex, first-delay, second-delay;
int first-count, second-count;
- Mutually exclusive access to the critical section is provided by **mutex**.
- If a process cannot enter the critical section because the Boolean expression B is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate B .

Implementation

- Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively.
- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
- For an arbitrary queuing discipline, a more complicated implementation is required.

Monitors

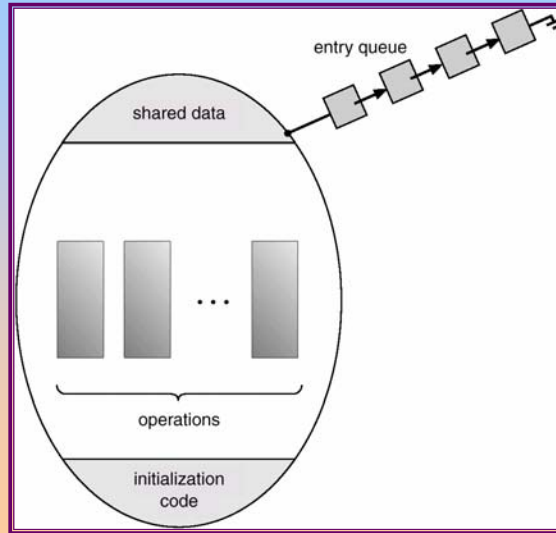
- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body P2 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}
```

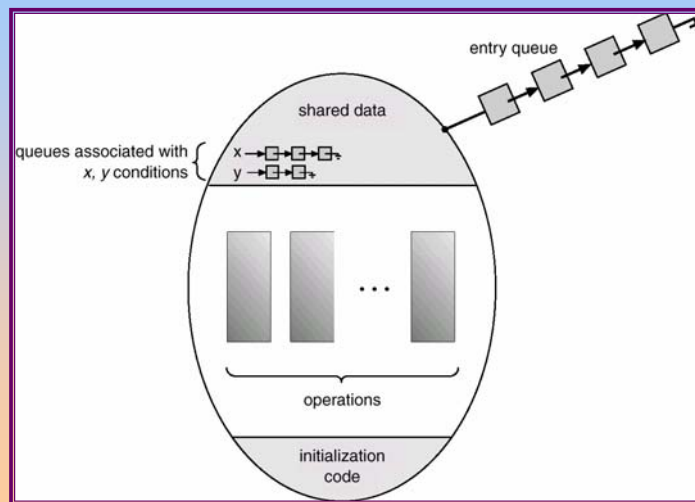
Monitors

- To allow a process to wait within the monitor, a **condition** variable must be declared, as
condition x, y;
- Condition variable can only be used with the operations **wait** and **signal**.
 - The operation
x.wait();
means that the process invoking this operation is suspended until another process invokes
x.signal();
 - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Schematic View of a Monitor



Monitor With Condition Variables



Dining Philosophers Example

```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)           // following slides
    void putdown(int i)         // following slides
    void test(int i)            // following slides
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```

Dining Philosophers

```
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);
}
```

Dining Philosophers

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

Monitor Implementation Using Semaphores

- Variables

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

- Each external 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

- 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

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

```
if (x-count > 0) {  
    next-count++;  
    signal(x-sem);  
    wait(next);  
    next-count--;  
}
```

Monitor Implementation

- *Conditional-wait* construct: **x.wait(c)**;
 - ☞ **c** – integer expression evaluated when the **wait** operation is executed.
 - ☞ value of **c** (a *priority number*) stored with the name of the process that is suspended.
 - ☞ when **x.signal** is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
 - ☞ User processes must always make their calls on the monitor in a correct sequence.
 - ☞ Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Solaris 2 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.
- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.
- Uses *turnstile*s to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Windows 2000 Synchronization

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