BILKENT UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING
CS342 OPERATING SYSTEMS

LECTURE 6
PROCESS SYNCHRONIZATION
(CHAPTER 6)

Dr. İbrahim Körpeoğlu
http://www.cs.bilkent.edu.tr/~korpe

References

- *The slides here are adapted/modified from the textbook and its slides: Operating System Concepts, Silberschatz et al., 7th & 8th editions, Wiley.*

REFERENCES

- Operating System Concepts, 7th and 8th editions, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum, 3rd edition, 2009.

Outline

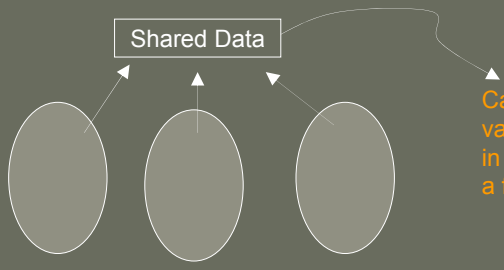
- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples from operating systems

Objectives

- 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

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

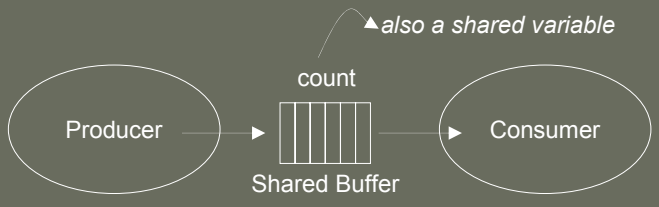


Can be a shared memory variable, a global variable in a multi-thread program or a file; or a kernel variable

Concurrent Threads or Processes

Producer Consumer Problem Revisited

- 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 count that keeps track of the number of full buffers. Initially, count 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.



at most BUFFER_SIZE items

Producer and Consumer Code

PRODUCER

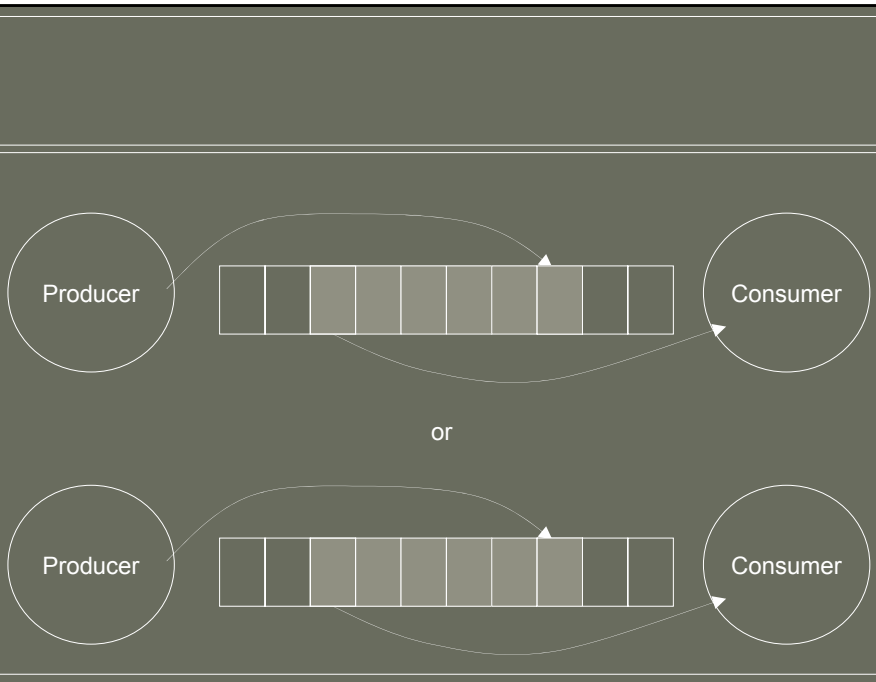
```
while (true) {
        /*  produce an item and
            put in nextProduced  */

        while (count == BUFFER_SIZE)
                ; // do nothing

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

CONSUMER

```
while (true)  {
        while (count == 0)
                ; // do nothing

        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;

        /*  consume the item
        in nextConsumed */
}
```


a possible Problem: race condition

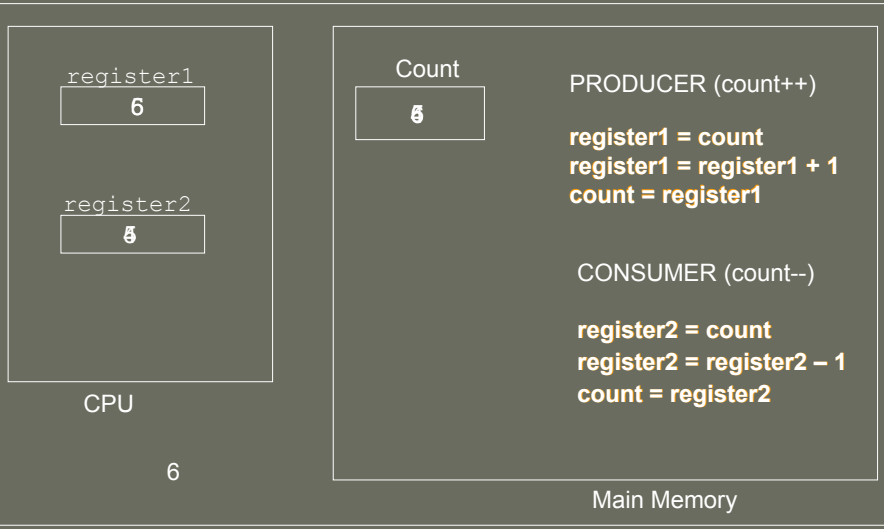
- Assume we had 5 items in the buffer
- Then:
  - Assume producer has just produced a new item and put it into buffer is about to increment the count.
  - Assume the consumer has just retrieved an item from buffer and is about the decrement the count.
  - Namely: Assume producer and consumer is now about to execute count++ and count– statements.



Race Condition

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2

Race Condition

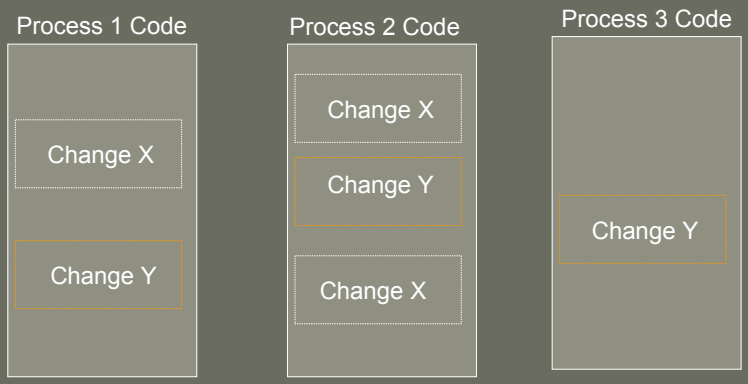


Interleaved Execution sequence

- Consider this execution interleaving with "count = 5" initially:
  - S0: producer execute register1 = count {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = count {register2 = 5}
  - S3: consumer execute register2 = register2 - 1 {register2 = 4}
  - S4: producer execute count = register1 {count = 6 }
  - S5: consumer execute count = register2 {count = 4}

Programs and critical sections

- The part of the program (process) that is accessing and changing shared data is called its critical section



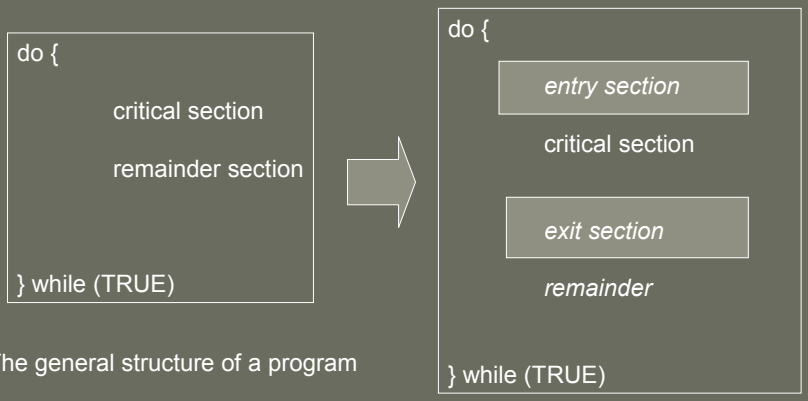
Assuming X and Y are shared data.

Program lifetime and its structure

- Considering a process:
  - It may be executing critical section code from time to time
  - It may be executing non critical section code (remainder section) other times.
- We should not allow more than one process to be in their critical regions where they are manipulating the *same* shared data.

Structuring Programs

- The general way to do that is:



The general structure of a program

Entry section will allow only one process to enter and execute critical section code.

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

Applications and Kernel

- Multiprocess applications sharing a file or shared memory segment may face critical section problems.
- Multithreaded applications sharing global variables may also face critical section problems.
- Similarly, kernel itself may face critical section problem. It is also a program. It may have critical sections.

Kernel Critical Sections

- While kernel is executing a function x(), a hardware interrupt may arrive and interrupt handler h() can be run. Make sure that interrupt handler h() and x() do not access the same kernel global variable. Otherwise race condition may happen.
- While a process is running in user mode, it may call a system call s(). Then kernel starts running function s(). CPU is executing in kernel mode now. We say the process is now running in kernel mode (even though kernel code is running).
- While a process X is running in kernel mode, it may or may not be pre-empted. It preemptive kernels, the process running in kernel mode can be preempted and a new process may start running. In non-preemptive kernels, the process running in kernel mode is not preempted unless it blocks or returns to user mode.

Kernel Critical Sections

- In a preemptive kernel, a process X running in kernel mode may be suspended (preempted) at an arbitrary (unsafe) time. It may be in the middle of updating a kernel variable or data structure at that moment. Then a new process Y may run and it may also call a system call. Then, process Y starts running in kernel mode and may also try update the same kernel variable or data structure (execute the critical section code of kernel). We can have a race condition if kernel is not synchronized.
- Therefore, we need solve synchronization and critical section problem for the kernel itself as well. The same problem appears there as well.

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE 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 (1)
```

entry section

exit section

Two processes executing concurrently

PROCESS 0

```
do {
        flag[0] = TRUE;
        turn = 1;
        while (flag[1] && turn == 1);
                critical section
        flag[0] = FALSE;
                remainder section
} while (1)
```

PROCESS 1

```
do {
        flag[1] = TRUE;
        turn = 0;
        while (flag[0] && turn == 0);
                critical section
        flag[1] = FALSE;
                remainder section
} while (1)
```

Shared Variables

flag 0 1

turn

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Use lock variables?
  - Can be source of race conditions?
  - Hardware can provide extra and more complex instructions to avoid race conditions

Solution to Critical-section Problem Using Locks

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

Only one process can acquire lock. Others has to wait (or busy loop)

Atomic Hardware Instructions

- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptible
  - Either test memory word and set value (TestAndSet)
  - Or swap contents of two memory words (Swap)

TestAndSet Instruction

- Is a machine/assembly instruction.
- Need to program in assembly to use. Hence Entry section code should be programmed in assembly
- But here we provide definition of it using a high level language code.

Definition of TestAndSet Instruction

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

Solution using TestAndSet

- Shared boolean variable lock, initialized to false.

Solution:

```
do {
    while ( TestAndSet (&lock ))
          ;   // do nothing

    //    critical section

    lock = FALSE;

    //      remainder section

} while (TRUE);
```

entry section

exit_section

In assembly

```
entry_section:
        TestAndSet   REGISTER, LOCK;
        CMP REGISTER, #0
        JNE  entry_section;
        RET
```

entry section code

```
exit_section:
        move LOCK, #0
        RET
```

exit section code

```
main:
        ..
        call entry_section;
        execute criticial region;
        call exit_section;
```

Swap Instruction

- Is a machine/assembly instruction. Intel 80x86 architecture has an XCHG instruction
- Need to program in assembly to use. Hence Entry section code should be programmed in assembly
- But here we provide definition of it using a high level language code.

Definition of Swap Instruction

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


Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

```
do {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

        //    critical section

        lock = FALSE;

        //      remainder section

} while (TRUE);
```

- TestAndSet and Swap provides mutual exclusion: 1st property satisfied
- But, Bounded Waiting property, 3rd property, may not be satisfied.
- A process X may be waiting, but we can have the other process Y going into the critical region repeatedly

Bounded-waiting Mutual Exclusion with TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&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);
```

entry section code

exit section code

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S*: integer variable
  shared, and can be a kernel variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
  - Also called down() and up()
  - Semaphores can only be accessed via these two indivisible (atomic) operations;
  - They can be implemented as system calls by kernel. Kernel makes sure they are indivisible.
- Less complicated entry and exit sections when semaphores are used

Semaphore Operations: Meaning

- wait (S): indivisible (until calling process is blocked)
  - if S is positive (S > 0), decrement S and return.
    will not cause the process to block.)
  - If S is not positive, then the calling process is put to sleep (blocked), until someone does a signal and this process is selected to wakeup.
- signal (S): indivisible (never blocks the calling process)
  - If there is one or more processes sleeping on S, then one process is selected and waken up, and signal returns.
  - If there is no process sleeping, then S is simply incremented by 1 and signal returns.

Semaphore as General Synchronization Tool

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
  - Binary semaphores provides mutual exclusion; can be used for the critical section problem.
- Counting semaphore – integer value can range over an unrestricted domain
  - Can be used for other synchronization problems; for example for resource allocation.

Usage

- Binary semaphores (mutexes) can be used to solve critical section problems.
- A semaphore variable (lets say mutex) can be shared by N processes, and initialized to 1.
- Each process is structured as follows:

```
do {
      wait (mutex);
       // Critical Section
      signal (mutex);
      // remainder section
} while (TRUE);
```

usage: mutual exclusion

Process 0

```
do {
      wait (mutex);
        // Critical Section
      signal (mutex);
      // remainder section
} while (TRUE);
```

Process 1

```
do {
      wait (mutex);
      // Critical Section
      signal (mutex);
      // remainder section
} while (TRUE);
```

Kernel

```
wait() {…}        signal() {…}
Semaphore mutex;    //  initialized to 1
```


usage: other synchronization problems

P0

```
…
S1;
….
```

P1

```
…
S2;
….
```

Assume we definitely want to have S1 executed before S2.

Solution:

```
semaphore x  = 0; // initialized to 0
```

P0

```
…
S1;
signal (x);
….
```

P1

```
…
wait (x);
S2;
….
```

Uses of Semaphore: synchronization

Buffer is an array of BUF_SIZE Cells (at most BUF_SIZE items can be put)

Producer

```
do {
        // produce item
        …
        put item into buffer
        ..
        signal (Full_Cells);

} while (TRUE);
```

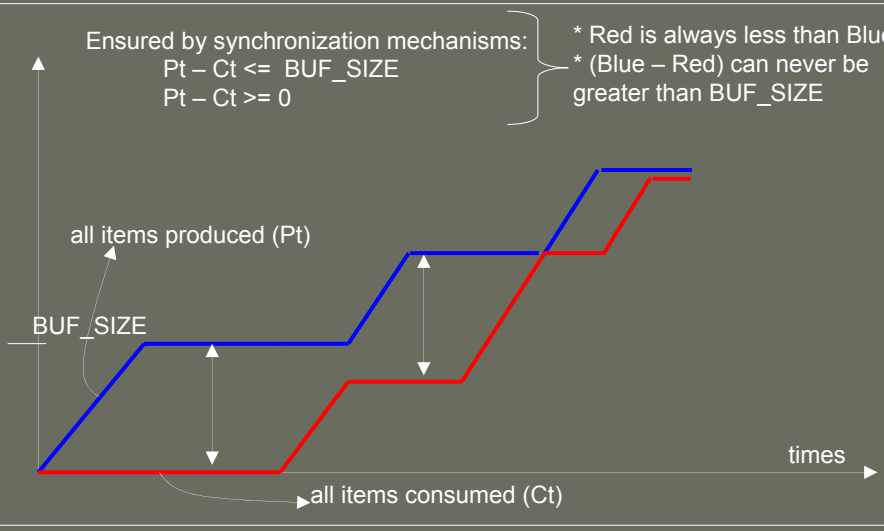
Consumer

```
do {
        wait (Full_Cells);
        ….
        remove item from buffer
        ..
        …
} while (TRUE);
```

Kernel

```
wait() {…}          signal() {…}

Semaphore Full_Cells = 0;    //  initialized to 0
```


Consumer/Producer is Synchronized



usage: resource allocation

- Assume we have a resource that has 5 instances. A process that needs that type of resource will need to use one instance. We can allow at most 5 process concurrently using these 5 resource instances. Another process (processes) that want the resource need to block. How can we code those processes?
- Solution:

one of the processes creates and initializes a semaphore to 5.

```
semaphore  x = 5; // semaphore to access resource
```

```
wait (x);
…
….use one instance
of the resource…
…
signal (x);
```

Each process has to be coded in this manner.

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.
- Kernel can guarantee this.

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


Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

Implementation of wait:

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

Implementation of signal:

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

        }
}
```


Kernel Implementing wait and signal

- The wait and signal operations must be atomic. The integer value is updated. No two process should update at the same time.
  How can the kernel ensure that? It can NOT use semaphores to implement semaphores.
- Implementation of these operations in kernel becomes the critical section problem where the wait and signal code are placed in the critical section. How can ensure two processes will not execute at the same time in wait or signal?
  - 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 busy waiting is not a good solution for applications. But, for short kernel critical sections, it may be acceptable in multi-CPU systems.

Deadlock and Starvation

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

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

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- 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

The structure of the producer process

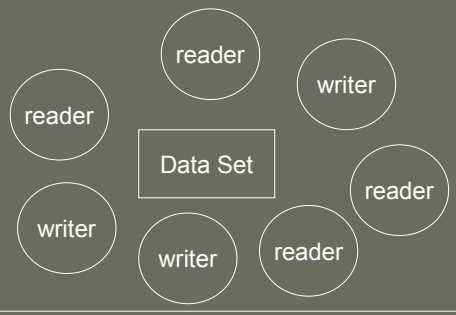
```
do  {
            //   produce an item in nextp

             wait (empty);
             wait (mutex);

             //  add the item to the  buffer

             signal (mutex);
             signal (full);

} while (TRUE);
```

The structure of the consumer process

```
do {
             wait (full);
             wait (mutex);

            //  remove an item from
            //  buffer to nextc

            signal (mutex);
            signal (empty);

            //  consume the item in nextc
} while (TRUE);
```

Readers-Writers Problem

- A data set 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 to read at the same time. Only one single writer can access the shared data at the same time



Readers-Writers Problem

- Shared Data
  - Data set
  - Integer readcount initialized to 0
    - Number of readers reading the data at the moment
  - Semaphore mutex initialized to 1
    - Protects the readcount variable (multiple readers may try to modify it)
  - Semaphore wrt initialized to 1
    - Protects the data set (either writer or reader(s) should access data at a time)

Readers-Writers Problem (Cont.)

The structure of a writer process

```
do {
        wait (wrt) ;
        //    writing is performed
        signal (wrt) ;
} while (TRUE);
```

The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex);

        // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
                signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```


Dining-Philosophers Problem

Assume a philosopher needs two forks to eat. Forks are like resources.
While a philosopher is holding a fork, another one can not have it.

Dining-Philosophers Problem

- Is not a real problem
- But lots of real resource allocation problems look like this. If we can solve this problem effectively and efficiently, we can also solve the real problems.
- From a satisfactory solution:
  - We want to have concurrency: two philosophers that are not sitting next to each other on the table should be able to eat concurrently.
  - We don't want deadlock: waiting for each other indefinitely.
  - We don't want starvation: ne philosopher waits forever.

Dining-Philosophers Problem (Cont.)

Semaphore chopstick [5] initialized to 1

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

        // eat

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

        // think
} while (TRUE);
```

This solution provides concurrency but may result in deadlock.

Problems with Semaphores

Incorrect use of semaphore operations:

signal (mutex) …. wait (mutex)

wait (mutex) … wait (mutex)

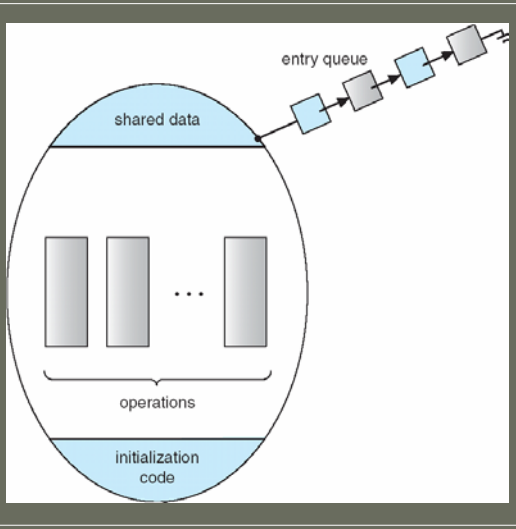
Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
        // shared variable declarations

        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}

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

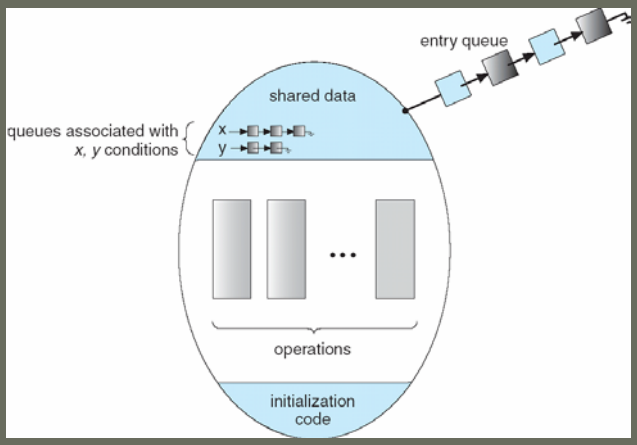
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Condition Variables

- Condition variables are not semaphores. They are different even though they look similar.
  - A condition variable does not count: have no associated integer.
  - A signal on a condition variable x is lost (not saved for future use) if there is no process waiting (blocked) on the condition variable x.
  - The wait() operation on a condition variable x will always cause the caller of wait to block.
  - The signal() operation on a condition variable will wake up a sleep process on the condition variable, if any. It as no effect if there is nobody sleeping.

Monitor Solution to Dining Philosophers

```
monitor DP {
   enum { THINKING;
            HUNGRY, EATING) state [5] ;
   condition  cond [5];

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

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

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

} /* end of monitor */
```


Solution to Dining Philosophers (cont)

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

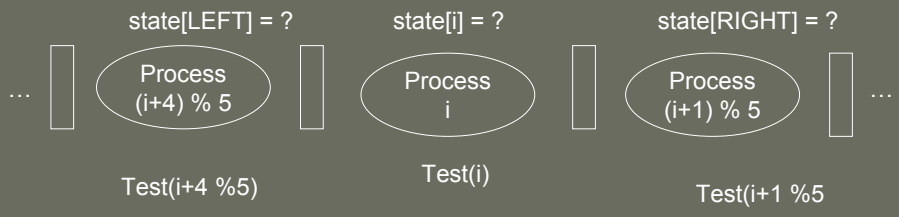
Philosopher i

```
…
DP DiningPhilosophers;
….
while (1)
        THINK…

        DiningPhilosophters.pickup (i);

        EAT /* use resource(s) */

        DiningPhilosophers.putdown (i);

        THINK…
}
```

Monitor Solution to Dining Philosophers

#define LEFT (i+4)%5
#define RIGHT (i+1)%5

THINKING?
HUNGRY?
EATING?



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1); allows only one process to be active
semaphore next;   // (initially  = 0);  causes signaler to sleep
int next-count = 0;     /* num sleepers since they signalled */
```

- 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 Using Semaphores

- Condition variables: how do we implement them?
- Assume the following strategy is implementing regarding who will run after a signal() is issued on a condition variable:
  - "The process that calls signal() on a condition variable is blocked. It can not be waken up if there is somebody running inside the monitor".
- Some programming languages require the process calling signal to quit monitor by having the signal() call as the last statement of a monitor procedure.
  - Such a strategy can be implemented in a more easy way.

Monitor Implementation Using Semaphores

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

```
semaphore x_sem; // (initially = 0); causes caller of wait to sleep
int x-count = 0; // number of sleeper on condition
```

The operation x.wait can be implemented as:

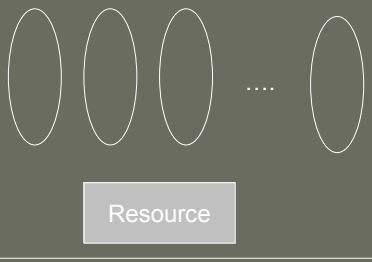
```
x-count++;
if (next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x-count--;
```

The operation x.signal can be implemented as:

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

A Monitor to Allocate Single Resource

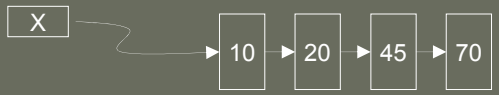
- Now we illustrate how monitors can be used to allocate a resource to one of several processes.
- We would like to apply a priority based allocation. The process that will use the resource for the shortest amount of time will get the resource first if there are other processes that want the resource.



A Monitor to Allocate Single Resource

- Assume we have condition variable implementation that can enqueue sleeping processes with respect to a priority specified as a parameter to wait() call.
  - cond x;
  - …
  - x.wait (priority);

Queue of sleeping processes waiting on condition x



priority could be the time-duration to use the resource

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;
        }
}
```


A Monitor to Allocate Single Resource

Process 1	Process 2		Process N
ResourceAllocator RA; RA.acquire(10); … ….use resource… …. RA.release();	ResourceAllocator RA; RA.acquire(30); … ….use resource… …. RA.release();	…	ResourceAllocator RA; RA.acquire(25); … ….use resource… …. RA.release();

Each process should use resource between acquire() and release() calls.

Spin Locks

- Kernel uses to protect *short* critical regions (a few instructions) on multi-processor systems.
- Assume we have a process A running in CPU 1 and holding a spin lock and executing the critical region touching to some shared data.
- Assume at the same, another process B running in CPU 2 would like run a critical region touching to the same shared data.
- B can wait on a semaphore, but this will cause B to sleep (a context switch is needed; costly operation). However, critical section of A is short; It would be better if B would busy wait for a while; then the lock would be available.
- Spin Locks are doing this. B can use a Spin Lock to wait (busy wait) until A will leave the critical region and releases the Spin Lock. Since critical region is short, B will not wait much.

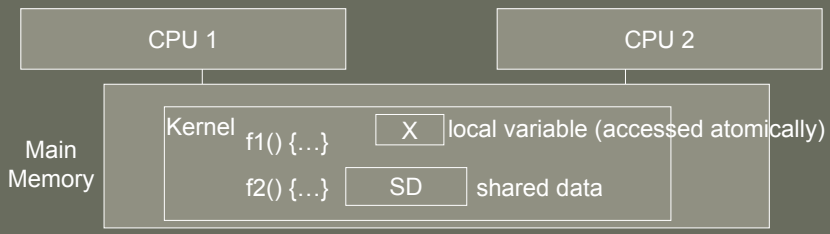
Spin Locks

Process A running in kernel mode (i.e. executing kernel code shown)

```
f1() {…
  acquire_spin_lock_(X);
  …//critical region….
  …touch to SD (shared data);
  release_spin_lock(X);
}
```

Process B running in kernel mode (i.e. executing kernel code shown)

```
f2() {…
  acquire_spin_lock_(X);
  …//critical region….
  …touch to SD (shared data);
  release_spin_lock(X); …
}
```



Spin Locks

- a spin lock can be acquired after busy waiting.
- Remember the TestAndSet or Swap hardware instructions that are atomic even on multi-processor systems. They can be used to implement the busy-wait acquisition code of spin locks.
- While process A is in the critical region, executing on CPU 1 and having the lock (X set to 1), process A may be spinning on a while loop on CPU 2, waiting for the lock to be become available (i.e. waiting X to become 0). As soon as process A releases the lock (sets X to 0), process B can get the lock (test and set X), and enter the critical region.

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Windows XP 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 either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

End of lecture