

ICS 143 - Principles of Operating Systems



Lecture Set 3 - Process Synchronization
Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

Outline



- The Critical Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors

Producer-Consumer Problem

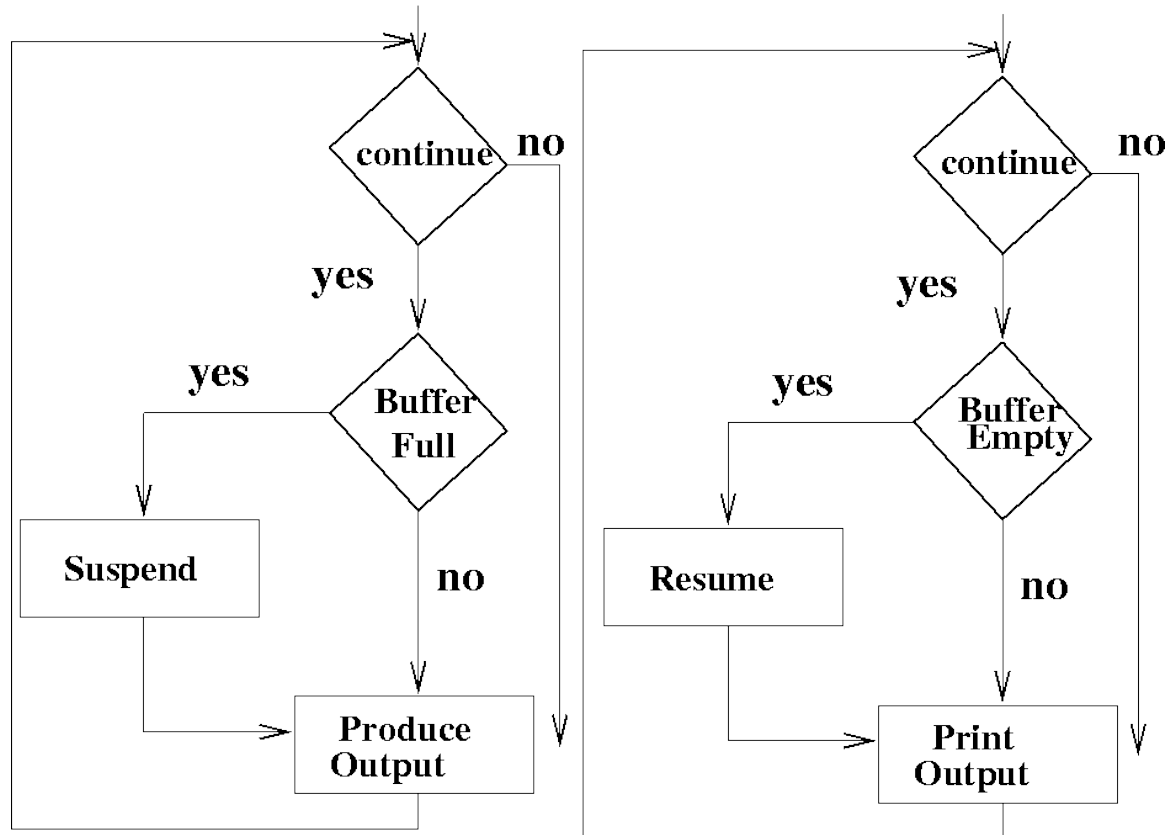


- Paradigm for cooperating processes;
 - producer process produces information that is consumed by a consumer process.
- We need buffer of items that can be filled by producer and emptied by consumer.
 - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
 - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
- Producer and Consumer must synchronize.

Producer-Consumer Problem

PRODUCER

CONSUMER



Bounded Buffer using IPC (messaging)



| Producer

repeat

...

produce an item in *nextp*;

...

send(*consumer*, *nextp*);

until false;

| Consumer

repeat

receive(*producer*, *nextc*);

...

consume item from *nextc*;

...

until false;

Bounded-buffer - Shared Memory Solution

■ Shared data

```
var n;  
type item = ....;  
var buffer: array[0..n-1] of item;  
in, out: 0..n-1;  
in := 0; out := 0; /* shared buffer = circular array */  
/* Buffer empty if in == out */  
/* Buffer full if (in+1) mod n == out */  
/* noop means 'do nothing' */
```

Bounded Buffer - Shared Memory Solution



- Producer process - creates filled buffers

repeat

...

produce an item in *nextp*

...

while $in+1 \bmod n = out$ **do** *noop*;

buffer[in] := nextp;

in := in+1 mod n;

until *false*;

Bounded Buffer - Shared Memory Solution



- Consumer process - Empties filled buffers

repeat

while *in = out* **do** *noop*;

nextc := buffer[out] ;

out := out + 1 mod n;

...

consume the next item in *nextc*

...

until *false*

Shared data



- 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 the bounded-buffer problem allows at most $(n-1)$ items in the buffer at the same time.

Bounded Buffer

- A solution that uses all N buffers is not that simple.
 - | Modify producer-consumer code by adding a variable *counter*, initialized to 0, incremented each time a new item is added to the buffer

- Shared data

```
type item = ....;  
var buffer: array[0.. $n$ -1] of item;  
in, out: 0.. $n$ -1;  
counter: 0.. $n$ ;  
in, out, counter := 0;
```

Bounded Buffer

- Producer process - creates filled buffers

repeat

...

produce an item in *nextp*

...

while *counter = n* **do** *noop*;

buffer[in] := nextp;

in := in+1 mod n;

counter := counter+1;

until *false*;

Bounded Buffer

- Consumer process - Empties filled buffers

```
repeat
    while counter = 0 do noop;
    nextc := buffer[out] ;
    out := out + 1 mod n;
    counter := counter - 1;
    ...
    consume the next item in nextc
until false;
```

- The statements

```
counter := counter + 1;
counter := counter - 1;
```

must be executed *atomically*.

- Atomic Operations

- An operation that runs to completion or not at all.

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 (**we expect count = 5 in the end too**):

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

Problem is at the lowest level

- If threads are working on separate data, scheduling doesn't matter:

Thread A

$x = 1;$

Thread B

$y = 2;$

- However, What about (Initially, $y = 12$):

Thread A

$x = 1;$

$x = y + 1;$

Thread B

$y = 2;$

$y = y * 2;$

- What are the possible values of x ?

- Or, what are the possible values of x below?

Thread A

$x = 1;$

Thread B

$x = 2;$

- x could be non-deterministic (1, 2??)



The Critical-Section Problem

■ N processes all competing to use shared data.

- Structure of process P_i --- Each process has a code segment, called the critical section, in which the shared data is accessed.

repeat

entry section /* enter critical section */

critical section /* access shared variables */

exit section /* leave critical section */

remainder section /* do other work */

until false

■ Problem

- Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution: Critical Section Problem - Requirements



I Mutual Exclusion

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

I Progress

- If no process is executing in its critical section and there exists 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.

I 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.

Solution: Critical Section

Problem - Requirements



- Assume that each process executes at a nonzero speed in the critical section. That is, assume that each process finishes executing the critical section once entered
- No assumption concerning relative speed of the n processes.
- Assume that a process can get stuck in its remainder section indefinitely, e.g., in a non-terminating while loop

Solution: Critical Section Problem -- Initial Attempt

- Only 2 processes, P0 and P1
- General structure of process P_i (P_j)
 - repeat**
 - entry section*
 - critical section*
 - exit section*
 - remainder section*
 - until false**
- Processes may share some common variables to synchronize their actions.

Algorithm 1

■ Shared Variables:

- | **var** *turn*: (0..1);
initially *turn* = 0;
- | *turn* = *i* \Rightarrow P_i can enter its critical section

■ Process P_i

repeat

while *turn* \neq *i* **do** *no-op*;

critical section

turn := *j*;

remainder section

until false

Satisfies mutual exclusion, but not progress.

Algorithm 1



- Satisfies mutual exclusion
 - The turn is equal to either i or j and hence one of P_i and P_j can enter the critical section
- Does not satisfy progress
 - Example: P_i finishes the critical section and then gets stuck indefinitely in its remainder section. Then P_j enters the critical section, finishes, and then finishes its remainder section. P_j then tries to enter the critical section again, but it cannot since turn was set to i by P_j in the previous iteration. Since P_i is stuck in the remainder section, turn will be equal to i indefinitely and P_j can't enter although it wants to. Hence no process is in the critical section and hence no progress.
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorithm 2

■ Shared Variables

- | **var** *flag*: **array** (0..1) **of** boolean;
initially *flag*[0] = *flag*[1] = false;
- | *flag*[*i*] = true \Rightarrow *P*_{*i*} ready to enter its critical section

■ Process *P*_{*i*}

repeat

flag[*i*] := true;

while *flag*[*j*] **do** no-op;

critical section

flag[*i*] := false;

remainder section

until false

▪

Algorithm 2



- Satisfies mutual exclusion
 - If P_i enters, then $\text{flag}[i] = \text{true}$, and hence P_j will not enter.
- Does not satisfy progress
 - Can block indefinitely.... Progress requirement not met
 - Example: There can be an interleaving of execution in which P_i and P_j both first set their flags to true and then both check the other process' flag. Therefore, both get stuck at the entry section
- We don't need to discuss/consider bounded wait when progress is not satisfied

Algorithm 3

■ Shared Variables

- | **var** *flag*: **array** (0..1) **of** boolean;
initially *flag*[0] = *flag*[1] = false;
- | *flag*[*i*] = *true* \Rightarrow *P*_{*i*} ready to enter its critical section

■ Process *P*_{*i*}

repeat

while *flag*[*j*] **do** *no-op*;
flag[*i*] := *true*;

critical section

flag[*i*] := *false*;

remainder section

until false

Algorithm 3



- ***Does not satisfy mutual exclusion***
 - Example: There can be an interleaving of execution in which both first check the other process' flag and see that it is false. Then they both enter the critical section.
- We don't need to discuss/consider progress and bounded wait when mutual exclusion is not satisfied

Algorithm 4

- Combined Shared Variables of algorithms 1 and 2
- Process P_i

repeat

flag[i] := true;

turn := j;

while (*flag[j] and turn=j*) **do** *no-op;*

critical section

flag[i] := false;

remainder section

until false

YES!!! Meets all three requirements, solves the critical section problem for 2 processes.

Also called "Peterson's solution"

Algorithm 4



- Satisfies mutual exclusion
 - If one process enters the critical section, it means that either the other process was not ready to enter or it was this process' turn to enter. In either case, the other process will not enter the critical section
- Satisfies progress
 - If one process exits the critical section, it sets its ready flag to false and hence the other process can enter. Moreover, there is no interleaving in the entry section that can block both.
- Satisfies bounded wait
 - If a process is waiting in the entry section, it will be able to enter at some point since the other process will either set its ready flag to false or will set to turn to this process.

Bakery Algorithm



■ Critical section for n processes

- | Before entering its critical section, process receives a number. Holder of the smallest number enters critical section.
- | If processes P_i and P_j receive the same number,
 - if $i \leq j$, 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. 1,2,3,3,3,3,4,4,5,5

Bakery Algorithm (cont.)

■ Notation -

■ Lexicographic order(ticket#, process id#)

| $(a,b) < (c,d)$ if $(a < c)$ or if $((a = c) \text{ and } (b < d))$

| $\max(a_0, \dots, a_{n-1})$ is a number, k , such that $k \geq a_i$
for $i = 0, \dots, n-1$

■ Shared Data

var *choosing*: **array** $[0..n-1]$ **of** *boolean*; (initialized to *false*)
number: **array** $[0..n-1]$ **of** *integer*; (initialized to 0)

Bakery Algorithm (cont.)

repeat

choosing[*i*] := *true*;

number[*i*] := max(*number*[0], *number*[1], ..., *number*[*n*-1]) + 1;

choosing[*i*] := *false*;

for *j* := 0 **to** *n*-1

do begin

while *choosing*[*j*] **do no-op**;

while *number*[*j*] <> 0

and (*number*[*j*], *j*) < (*number*[*i*], *i*) **do no-op**;

end;

critical section

number[*i*] := 0;

remainder section

until false;

Supporting Synchronization

<i>Programs</i>	<i>Shared Programs</i>
<i>Higher-level API</i>	<i>Locks Semaphores Monitors Send/Receive CCregions</i>
<i>Hardware</i>	<i>Load/Store Disable Ints Test&Set Comp&Swap</i>

- We are going to implement various synchronization primitives using atomic operations
 - Everything is pretty painful if only atomic primitives are load and store
 - Need to provide inherent support for synchronization at the hardware level
 - Need to provide primitives useful at software/user level

Hardware Solutions for Synchronization

- Load/store - Atomic Operations required for synchronization
 - Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!
- Mutual exclusion solutions presented depend on memory hardware having read/write cycle.
 - If multiple reads/writes could occur to the same memory location at the same time, this would not work.
 - Processors with caches but no cache coherency cannot use the solutions
- In general, it is impossible to build mutual exclusion without a primitive that provides some form of mutual exclusion.
 - How can this be done in the hardware???
 - How can this be simplified in software???

Synchronization Hardware

- Test and modify the content of a word atomically - **Test-and-set instruction**

```
function Test-and-Set (var target: boolean): boolean;  
  begin  
    Test-and-Set := target;  
    target := true;  
  end;
```

- Similarly “**SWAP**” instruction

Mutual Exclusion with Test-and-Set



- Shared data: var lock: boolean (initially false)

- Process P_i

repeat

while *Test-and-Set (lock)* do no-op;

critical section

lock := false;

remainder section

until false;

Bounded Waiting Mutual Exclusion with Test-and-Set

```
var  $j : 0..n-1$ ;  
    key : boolean;  
repeat  
    waiting [ $j$ ] := true; key := true;  
    while waiting[ $i$ ] and key do key := Test-and-Set(lock);  
    waiting [ $i$ ] := false;  
    critical section  
     $j := i+1 \bmod n$ ;  
    while ( $j \neq i$ ) and (not waiting[ $j$ ]) do  $j := j + 1 \bmod n$ ;  
    if  $j = i$  then lock := false;  
        else waiting[ $j$ ] := false;  
    remainder section  
until false;
```

Hardware Support: Other examples

- **swap (&address, register)** { /* x86 */
 temp = M[address];
 M[address] = register;
 register = temp;
}
- **compare&swap (&address, reg1, reg2)** { /* 68000 */
 if (reg1 == M[address]) {
 M[address] = reg2;
 return success;
 } else {
 return failure;
 }
}
- **load-linked&store conditional (&address)** {
 /* R4000, alpha */
 loop:
 ll r1, M[address];
 movi r2, 1; /* Can do arbitrary comp */
 sc r2, M[address];
 beqz r2, loop;
}

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;;`
 `}`

•Semantics of

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

•Critical section implementation

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

Semaphore

- Semaphore S - integer variable (non-negative)
 - used to represent number of abstract resources
- Can only be accessed via two indivisible (atomic) operations
 - $wait(S)$: **while** $S \leq 0$ **do** no-op
 $S := S - 1$;
 - $signal(S)$: $S := S + 1$;
 - P or $wait$ used to acquire a resource, waits for semaphore to become positive, then decrements it by 1
 - V or $signal$ releases a resource and increments the semaphore by 1, waking up a waiting P , if any
 - If P is performed on a $count \leq 0$, process must wait for V or the release of a resource.

$P()$: "**proberen**" (to test) ; $V()$ "**verhogen**" (to increment) in Dutch

Example: Critical Section for n Processes

I Shared variables

var *mutex*: semaphore
initially *mutex* = 1

I Process P_i

repeat

wait(mutex);

critical section

signal (mutex);

remainder section

until false

Semaphore as a General Synchronization Tool

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

P_i	P_j
:	:
:	:
:	:
A	$wait(flag)$
$signal(flag)$	B

Problem...

- **Locks** prevent conflicting actions on shared data
 - Lock before entering critical section and before accessing shared data
 - Unlock when leaving, after accessing shared data
 - Wait if locked
- All Synchronization involves waiting
 - **Busy Waiting**, uses CPU that others could use. This type of semaphore is called a *spinlock*.
 - Waiting thread may take cycles away from thread holding lock (no one wins!)
 - OK for short times since it prevents a context switch.
 - Priority Inversion: If busy-waiting thread has higher priority than thread holding lock \Rightarrow no progress!
 - Should *sleep* if waiting for a long time
- For longer runtimes, need to modify P and V so that processes can *block* and *resume*.

Semaphore Implementation

- Define a semaphore as a record

```
type semaphore = record  
    value: integer;  
    L: list of processes;  
end;
```

- Assume two simple operations

- | *block* suspends the process that invokes it.
- | *wakeup(P)* resumes the execution of a blocked process *P*.

Semaphore

Implementation(cont.)

- Semaphore operations are now defined as

wait (S): S.value := S.value -1;

if *S.value* < 0

then begin

add this process to *S.L*;

block;

end;

signal (S): S.value := S.value +1;

if *S.value* <= 0

then begin

remove a process *P* from *S.L*;

wakeup(P);

end;

Block/Resume Semaphore Implementation



- If process is blocked, enqueue PCB of process and call scheduler to run a different process.
- Semaphores are executed atomically;
 - | no two processes execute *wait* and *signal* at the same time.
 - | Mutex can be used to make sure that two processes do not change count at the same time.
 - If an interrupt occurs while mutex is held, it will result in a long delay.
 - Solution: Turn off interrupts during critical section.

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 semaphores initialized to 1

P_0
 $wait(S);$
 $wait(Q);$
 \vdots
 $signal(S);$
 $signal(Q);$

P_1
 $wait(Q);$
 $wait(S);$
 \vdots
 $signal(Q);$
 $signal(S);$

- 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; simpler to implement.
- Can implement a counting semaphore S as a binary semaphore.

Classical Problems of Synchronization



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

Bounded Buffer Problem

■ Shared data

```
type item = ....;  
var buffer: array[0..n-1] of item;  
full, empty, mutex : semaphore;  
nextp, nextc : item;  
full := 0; empty := n; mutex := 1;
```


Bounded Buffer Problem

- Producer process - creates filled buffers

repeat

...

produce an item in *nextp*

...

wait (empty);

wait (mutex);

...

add *nextp* to buffer

...

signal (mutex);

signal (full);

until *false;*

Bounded Buffer Problem

■ Consumer process - Empties filled buffers

repeat

wait (full);

wait (mutex);

...

remove an item from *buffer* to *nextc*

...

signal (mutex);

signal (empty);

...

consume the next item in *nextc*

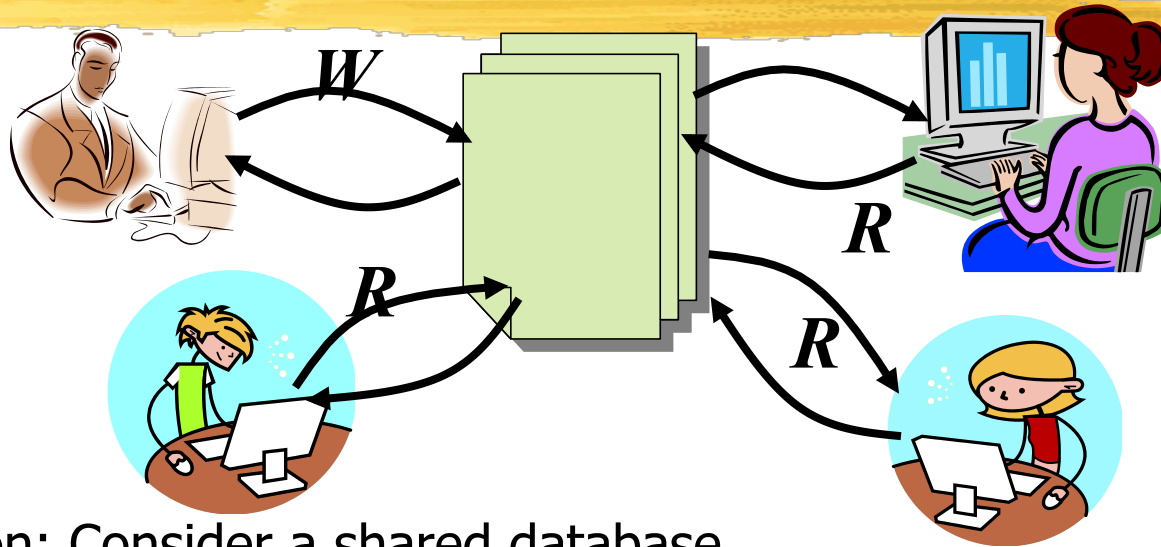
...

until *false;*

Discussion

- ASymmetry?
 - Producer does: $P(\text{empty}), V(\text{full})$
 - Consumer does: $P(\text{full}), V(\text{empty})$
- Is order of P's important?
 - Yes! Can cause deadlock
- Is order of V's important?
 - No, except that it might affect scheduling efficiency

Readers/Writers Problem



- Motivation: Consider a shared database
 - Two classes of users:
 - | Readers – never modify database
 - | Writers – read and modify database
 - Is using a single lock on the whole database sufficient?
 - | Like to have many readers at the same time
 - | Only one writer at a time

Readers-Writers Problem

■ Shared Data

```
var mutex, wrt: semaphore (=1);  
    readcount: integer (= 0);
```

■ Writer Process

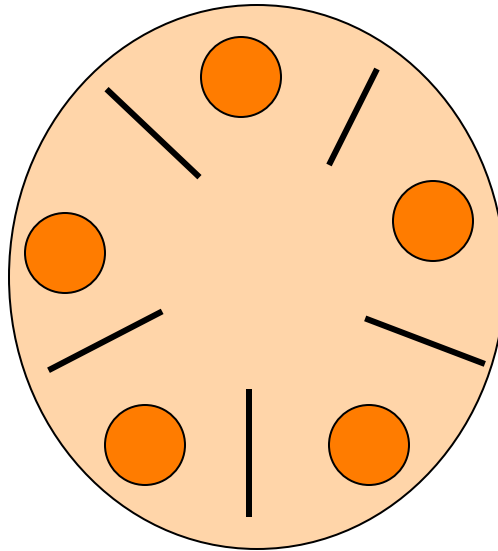
```
wait(wrt);  
    ...  
    writing is performed  
    ...  
signal(wrt);
```

Readers-Writers Problem

■ Reader process

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

Dining-Philosophers Problem



Shared Data

var chopstick: array [0..4] of semaphore (=1 initially);

Dining Philosophers Problem

I Philosopher i :

repeat

wait (chopstick[i]);

wait (chopstick[$i+1 \bmod 5$]);

...

eat

...

signal (chopstick[i]);

signal (chopstick[$i+1 \bmod 5$]);

...

think

...

until *false;*

Higher Level Synchronization

■ Timing errors are still possible with semaphores

| Example 1

```
    signal(mutex);  
        ...  
        critical region
```

```
    wait(mutex);
```

| Example 2

```
    wait(mutex);  
        ...  
        critical region
```

```
    wait(mutex);
```

| Example 3

```
    wait(mutex);  
        ...  
        critical region
```

```
    Forgot to signal
```

Motivation for Other Sync. Constructs



- Semaphores are a huge step up from loads and stores
 - Problem is that semaphores are dual purpose:
 - They are used for both mutex and scheduling constraints
 - Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?
- Idea: allow manipulation of a shared variable only when condition (if any) is met – **conditional critical region**
- Idea : Use *locks* for mutual exclusion and *condition variables* for scheduling constraints
 - **Monitor:** a lock (for mutual exclusion) and zero or more condition variables (for scheduling constraints) to manage concurrent access to shared data
 - Some languages like Java provide this natively

Conditional Critical Regions

- High-level synchronization construct
- A shared variable v of type T is declared as:
var v : shared T
- Variable v is accessed only inside statement
region v when B do \underline{S}
where B is a boolean expression.
While statement S is being executed, no other process
can access variable v .

Critical Regions (cont.)

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement,

region v **when** B **do** \underline{S}

- 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 variables

var *buffer*: **shared record**

pool: **array**[0..*n*-1] **of** *item*;

count, *in*, *out*: *integer*;

end;

■ Producer Process inserts *nextp* into the shared buffer

region *buffer* **when** *count* < *n*

do begin

pool[*in*] := *nextp*;

in := *in* + 1 **mod** *n*;

count := *count* + 1;

end;

Bounded Buffer Example

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

region *buffer* **when** *count* > 0

do begin

nextc := *pool*[*out*];

out := *out* + 1 **mod** *n*;

count := *count* - 1;

end;

Monitors

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

type *monitor-name* = **monitor**

variable declarations

procedure entry *P1* (...);

begin ... end;

procedure entry *P2* (...);

begin ... end;

⋮

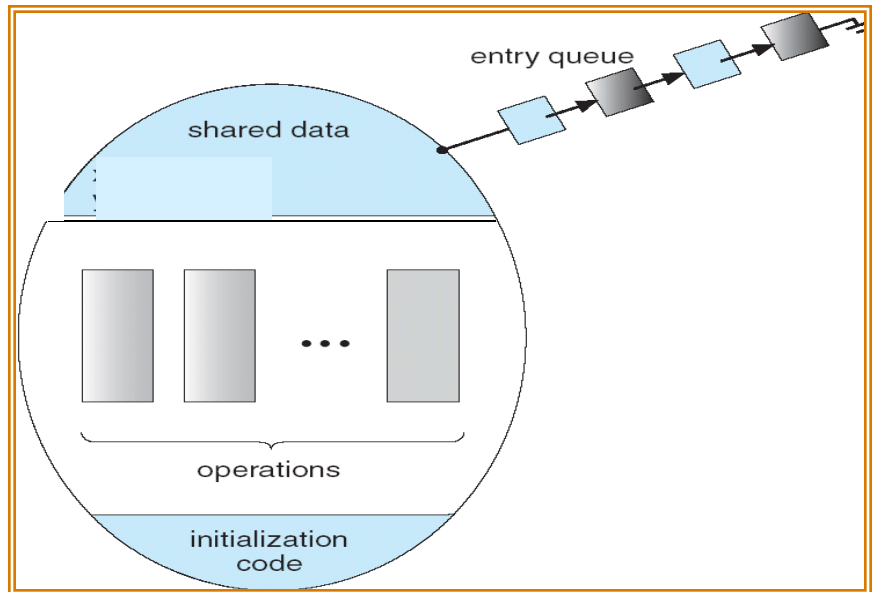
procedure entry *P_n*(...);

begin ... end;

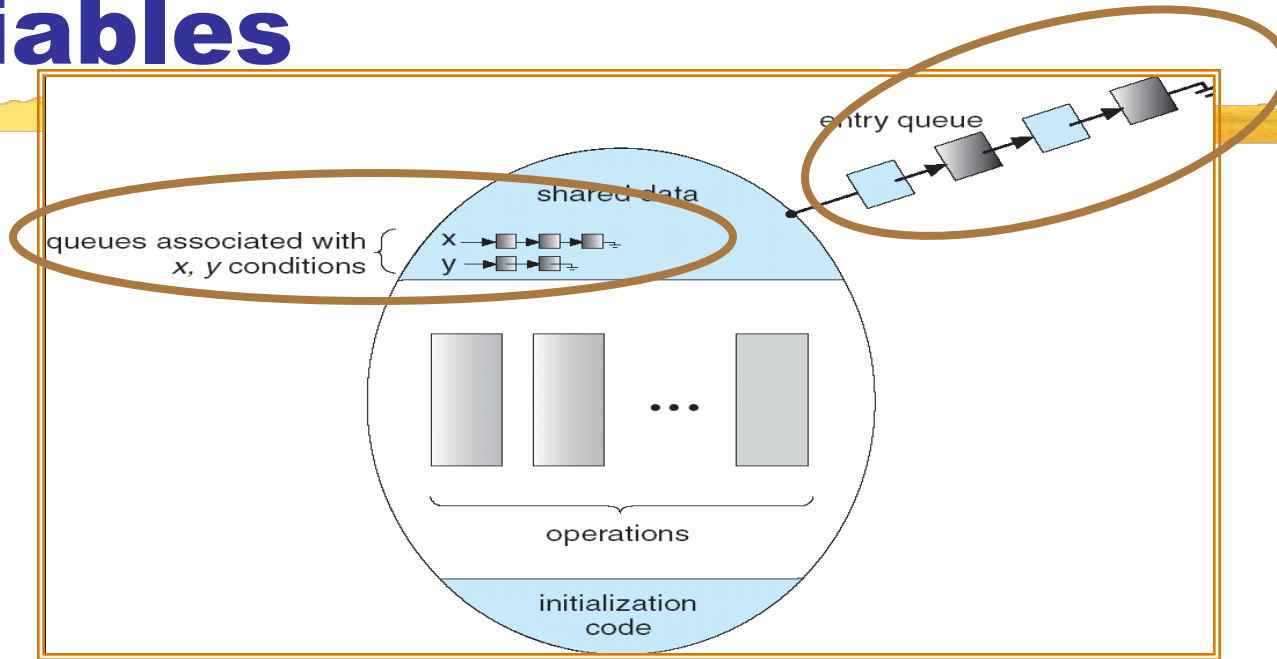
begin

initialization code

end.



Monitor with Condition Variables



- **Lock:** the lock provides mutual exclusion to shared data
 - Always acquire before accessing shared data structure
 - Always release after finishing with shared data
 - Lock initially free
- **Condition Variable:** a queue of threads waiting for something *inside* a critical section
 - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep

Monitors with condition variables

- To allow a process to wait within the monitor, a condition variable must be declared, as:

var *x,y: condition*

- | Condition variable can only be used within the operations *wait* and *signal*. Queue is associated with condition variable.
 - 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.

Dining Philosophers

```
type dining-philosophers= monitor
  var state: array[0..4] of (thinking, hungry, eating);
  var self: array[0..4] of condition;
  // condition where philosopher I can delay himself when hungry
  but // is unable to obtain chopstick(s)
  procedure entry pickup (i :0..4);
  begin
    state[i] := hungry;
    test(i); //test that your left and right neighbors are not eating
    if state [i] <> eating then self [i].wait;
  end;
  procedure entry putdown (i:0..4);
  begin
    state[i] := thinking;
    test (i + 4 mod 5 ); // signal one neighbor
    test (i + 1 mod 5 ); // signal other neighbor
  end;
```

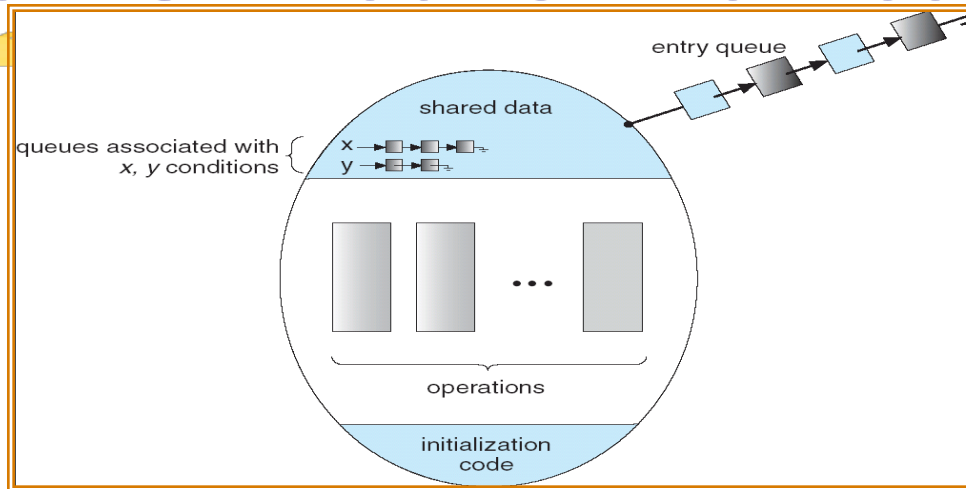
Dining Philosophers (cont.)

```
procedure test (k : 0..4);  
begin  
    if state [k + 4 mod 5] <> eating  
    and state [k] = hungry  
    and state [k + 1 mod 5] <> eating  
    then  
        begin  
            state[k] := eating;  
            self[k].signal;  
        end;  
    end;  
  
begin  
    for i := 0 to 4  
        do state[i] := thinking;  
    end;
```

Additional (extra) slides

A thick, horizontal yellow brushstroke underline that spans the width of the slide, positioned directly beneath the title text.

Mesa vs. Hoare monitors



■ Who proceeds next – signaler or waiter?

■ Hoare-style monitors(most textbooks):

- | Signaler gives lock, CPU to waiter; waiter runs immediately
- | Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again

■ Mesa-style monitors (most real operating systems):

- | Signaler keeps lock and processor
- | Waiter placed on ready queue with no special priority
- | Practically, need to check condition again after wait (condition may no longer be true!)

Implementing S (counting sem.) as a Binary Semaphore

■ Data Structures

```
var S1 : binary-semaphore;  
    S2 : binary-semaphore;  
    S3 : binary-semaphore;  
    C: integer;
```

■ Initialization

```
S1 = S3 = 1;  
S2 = 0;  
C = initial value of semaphore S;
```

Implementing S

Wait operation

```
wait(S3);  
wait(S1);  
C := C - 1;  
if C < 0  
then begin  
    signal(S1);  
    wait(S2);  
end  
else signal(S1);  
    signal(S3);
```

Signal operation

```
wait(S1);  
C := C + 1;  
if C <= 0 then signal(S2);  
    signal(S1);
```

Implementing Regions

■ Region x when B do S

*var mutex, first-delay, second-delay: semaphore;
first-count, second-count: integer;*

■ 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 queueing of processes for a semaphore.
- For an arbitrary queueing discipline, a more complicated implementation is required.

Implementing Regions

```
wait(mutex);
while not B
do begin  first-count := first-count + 1;
         if second-count > 0
           then signal (second-delay);
           else signal (mutex);
         wait(first-delay);
         first-count := first-count - 1;
         second-count := second-count + 1;
         if first-count > 0 then signal (first-delay)
                               else signal (second-delay);
         wait(second-delay);
         second-count := second-count - 1;
       end;
S;
if first-count > 0 then signal (first-delay);
else if second-count > 0
  then signal (second-delay);
  else signal (mutex);
```