CECS 326
Operating Systems

Synchronization Tools & Examples

(Chapters 6&7. Operating system Concepts by Silberschatz, Galvin and Gagne)

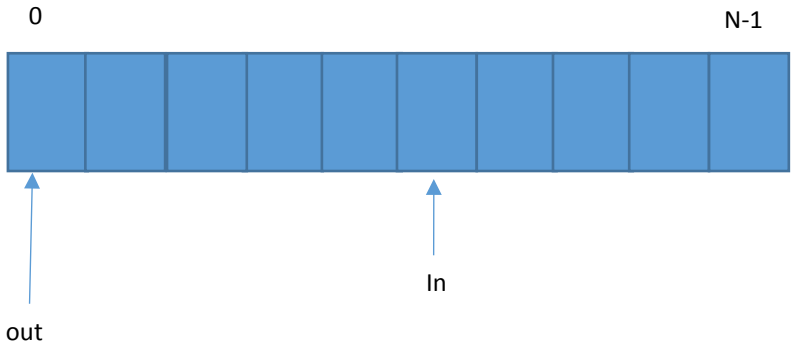
Background

- Processes can execute concurrently
- Process executes instructions one after another, but may be interrupted at any time between instruction execution
- Concurrent access to shared data may lead to race condition, resulting in data inconsistency
- Maintaining data consistency requires mechanisms to ensure orderly execution of cooperating processes
- Illustration of the problem:
  - Previous solution to the producer-consumer problem leaves one slot unused to enable distinction between the conditions of full vs. empty buffer. This deficiency may be overcome by using an integer counter that keeps track of the number of filled buffer slots. The counter is set to 0 initially, incremented by the producer after each data it places in the buffer and decremented by the consumer after it removes a data from it.

Bounded-Buffer for Producer/Consumer Problem

- Producer

```
item next_produced;
while (true) {
    /* produce an item in next_produced */
    while (((in + 1)%BUFFER_SIZE) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1)%BUFFER_SIZE;
```

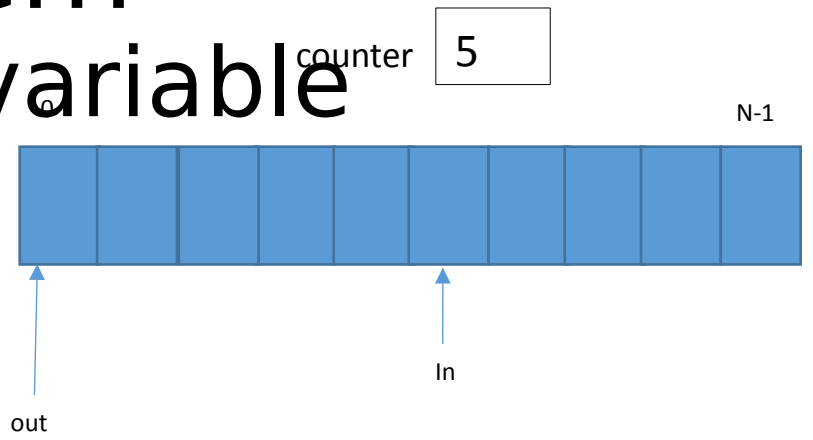


- Consumer

```
item next_consumed;
while (true) {
    while (in == out)
        ; /buffer empty, do nothing */
    next_consumed = buffer[out];
    out = (out + 1)%BUFFER_SIZE;
    /* consume the item in next_consumed */
}
```

- Solution is correct with one producer and one consumer, but can only use BUFFER_SIZE-1 slots in buffer

Producer-Consumer Problem -
Bounded Buffer with a counter variable



Producer

```
while (true) {
   /* produce an item in
next_produced */
   while
(counter==BUFFER_SIZE);
  /* do nothing */
   buffer[in]=next_produced;
   in=(in+1)%BUFFER_SIZE;
   counter++;
}
```

Consumer

```
while (true) {
   while (counter==0);
  /* do nothing */
   next_consumed=buffer[out];
   out = (out+1)% BUFFER_SIZE;
   counter--;
   /* consume the item in
next_consumed */
}
```

Race Condition

- counter++ could be implemented as
  (INC1) register1 = counter
  (INC2) register1 = register1 + 1
  (INC3) counter = register1
- counter–- could be implemented as
  (DEC1) register2 = counter
  (DEC2) register2 = register2 - 1
  (DEC3) counter = register2
- Consider the following execution interleaving with counter = 5 initially:

Producer execute INC1	(counter = 5, register1 = 5)
Producer execute INC2	(counter = 5, register1 = 6)
Consumer execute DEC1	(counter = 5, register2 = 5)
Consumer execute DEC2	(counter = 5, register2 = 4)
Producer execute INC3	(counter = 6, register1 = 6)
Consumer execute DEC3	(counter = 4, register2 = 4)

  - Value of counter is erroneous (correct value is 5)

Critical Section (CS)

- Cause of inconsistency in shared data *counter* is that it is manipulated concurrently by two processes
- Process requires mutually exclusive access to shared data to ensure integrity
- Section of code where such exclusive access is required is called Critical Section
- Critical sections in the producer-consumer implementation:
  - All statements where *counter* is referenced
- Critical section problem is to design protocol to ensure that when one process is in its critical section, no other may be in its critical section

Requirements of Solution to Critical-Section Problem

- Mutual Exclusion – If process P_i is executing in its critical section, then no other process can be executing in their critical sections
- 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
- 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

Simple Algorithm (1) for Two Processes

- Given processes P_0 and P_1
- Use a shared variable “turn” to control which process can enter critical section
- Initialize turn to 0

P_0: P_1:

```
while (turn != 0) while (turn != 1)
   /* do nothing */ ;          /* do nothing */ ;
<critical section><critical section>
turn = 1; turn = 0;
```

- Problem with solution?

Simple Algorithm (2) for Two Processes

- Given processes P_0 and P_1
- Use shared Boolean array "flag" to control which process can enter critical section
- Initialize both flag[0] and flag[1] to false

P_0:P_1:

```
flag[0] = true;flag[1] = true;
while (flag[1]){ while (flag[0]){
   flag[0] = false;    flag[1] = false;
   /* delay */     /* delay */
   flag[0] = true;     flag[1] = true;
} }
<critical section> <critical section>
flag[0] = false; flag[1] = false;
```

- Problem with solution?

Peterson's Algorithm for Two Processes

- Given processes P_0 and P_1
- Use two shared variables
  - int turn /* indicates whose turn it is to enter CS */
  - Boolean flag[2] /* flag[i] = true indicates that process is ready to enter CS */

P_i:

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j) /* do nothing */ ;
    <critical section>
    flag[i] = false;
    <remainder section>
} while (true);
```

- Provable that the three CS requirements are met:
  - Mutual exclusion is preserved (P_i enters CS only if either flag[j]= false or turn=i)
  - Progress requirement is satisfied
  - Bounded waiting requirement is met

Critical Section Handling in OS

- At any given time, many kernel-mode processes may be active in the OS
- These processes may need to manipulate shared data structures (e.g., ready queue, wait queues for various resources, file descriptors) and hence are subject to possible race conditions
- Two approaches are used to handle critical sections in OS:
  - Preemptive kernel – allows preemption of process when running in kernel mode
  - Non-preemptive kernel – a kernel-mode process runs until it exits kernel mode, blocks, or voluntarily yields CPU (essentially free of race conditions)

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code
- Could disable interrupts
  - Currently running code would execute without preemption
  - Won't work unless on a uniprocessor system, or all CPUs are disabled of interrupts at the same time. Too inefficient.
- Modern machines provide special atomic hardware instructions that can be used to implement a "locking" mechanism
  - Atomic means non-interruptible
  - A "test_and_set" instruction to test memory word and set value
  - A "swap" instruction to swap contents of two memory words

A Simple Idea For N Processes

- Given processes P_0, P_1,and P_{n-1}
- Use a Boolean shared variable flag to indicate status of shared resource
  - flag = true mean shared resource is available, OK to enter CS
  - flag = false means shared resource is in use, DO NOT enter CS
  - flag is initialized to true

P_i:

```
do {
   while (!flag); /* keep testing, stay out until true */
   flag = false;  /* set flag to block out other processes */
   <critical section>
   flag = true;   /* reset flag */
   <remainder section>
} while (true);
```

- Solution does not enforce mutual exclusion. Two or more processes may find flag = true, exit while loop, set flag to false, and enter CS

A Simple Idea For N Processes (cont.)

- Problem with this solution
  - Operations for testing value of flag and setting it to a specific value can be interrupted
  - For this idea to work, these operations must be atomic (indivisible)
  - A “test_and_set” instruction is designed to meet this need

Definition:

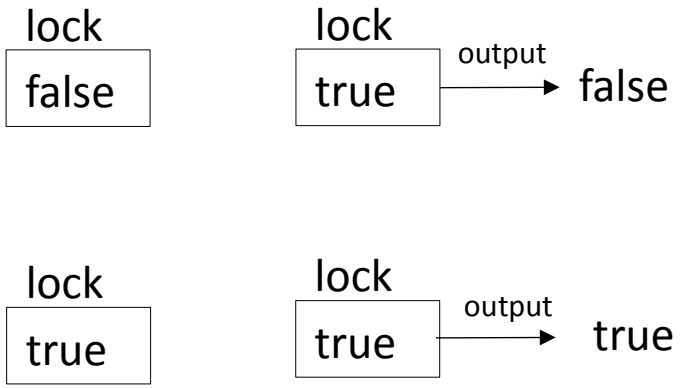
```
Boolean test_and_set (Boolean *target){
    Boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

- Executed atomically
- Returns the original value of passed parameter
- Set new value of passed parameter to true

Solution Using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution

```
do {
    while (test_and_set(&lock)); /* condition not
        right, stay out, keep checking */
    < critical section >
    lock = false;
    < remainder section >
} while (true);
```

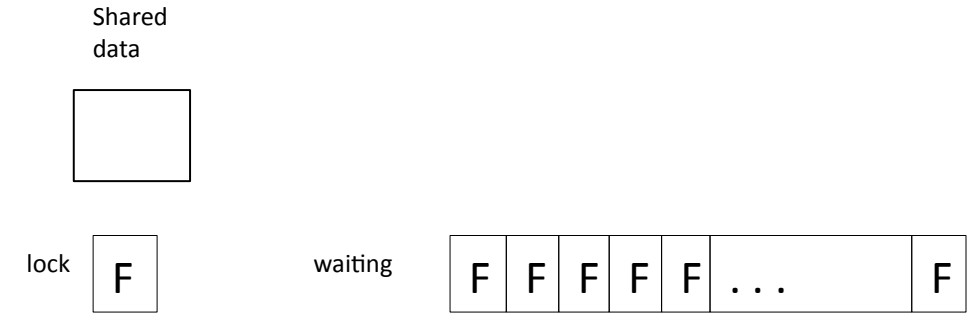


Bounded-waiting Mutual Exclusion with test_and_set()

- Given processes P_0, P_1, … and P_{n-1} ; lock is a shared variable initialized to false

Code for P_i**:**

```
do {
   waiting[i] = true;
   key = true;  /* key is a local variable */
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   < critical section >
   j = (i + 1) % n;
   while ((j != i) && !waiting[j]) /* find next waiting
      j = (j + 1) % n;   /* process & pass lock to it */
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   < remainder section >
} while (true);
```



P_0: key=

P_1: key=

P_2: key=

P_3: key=

P_4: key=

.

.

Code for P_i:

```
do {
   waiting[i] = true;
   key = true;   /* key a local
variable*/
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   < critical section >
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n; /* find next
waiting process & pass lock to it */
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   < remainder section >
} while (true);
```

Mutex Lock

- Previous solution is complicated and generally inaccessible to application programmers
- OS designers build software tools to solve Critical Section (CS) problem
- The simplest is mutex lock
  - CS is protected using acquire() and release(), which are defined based on a Boolean variable that indicates if lock is available or not
  - Calls to acquire() and release() must be atomic, usually implemented via hardware atomic instructions
  - This solution requires busy waiting. This lock is therefore called a spinlock.

acquire() and release()

- A mutex lock has a Boolean variable ***available*** whose value indicates if the lock is available or not.
- If the lock is available, a call to acquire() succeeds and the lock then becomes unavailable.
- A process that attempts to acquire an unavailable lock is blocked until the lock is released.

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}

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

- Processes use mutex lock as follows

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

Semaphore

- Synchronization tool that provides more sophisticated ways than mutex locks for processes to synchronize their activities
- Semaphore S
- Can only be accessed via two indivisible (atomic) operations wait() and signal() (originally called P() and V())
- Definition involving busy waiting, where S is an integer variable

```
wait(S) {
    while (S <= 0)
        ; // busy wait when S is not positive
    S--;
}
signal(S) {
    S++;
}
```

Semaphore Usage

- Can solve critical section problem, e.g., Processes P_i, $1 \le i \le n$, need to manipulate some shared data, therefore mutual exclusion is required
  - Create a semaphore "mutex" initialized to 1

P_i:

```
wait (mutex);
Critical Section
signal(mutex);
```

- Can also solve various other synchronization problems
- For example, consider processes P_1 and P_2 that require S_1 to happen before S_2 :
  - Create a semaphore "synch" initialized to 0

P_1:

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

P_2:

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

Semaphore Implementation

- Must guarantee that no two processes can execute wait() and signal() on the same semaphore at the same time
- Thus, implementation of these operations becomes a critical section problem where the wait and signal code are critical sections
- When a process is executing a long CS, other processes block themselves out of their CS in busy waiting occupying CPU non-productively. It is therefore desirable to have an implementation with no busy waiting.

Semaphore Implementation with No Busy Waiting

- Each semaphore consists of an integer and a queue, defined as

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

- To eliminate busy waiting, the following two operations are needed:
  - sleep – suspend the process invoking the operation
  - wakeup – resume the execution of a suspended process

Semaphore Implementation with No Busy Waiting

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

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

Classical Problems of Synchronization

- Problems that have different synchronization requirements:
  - Bounded-buffer problem
  - Readers and writers problem
  - Dining philosophers problem

Bounded-Buffer Problem

- A buffer with n slots, each can hold one data item
- Semaphore solution:
  - Semaphore **mutex** initialized to 1
  - Semaphore **full** initialized to 0
  - Semaphore **empty** initialized to n

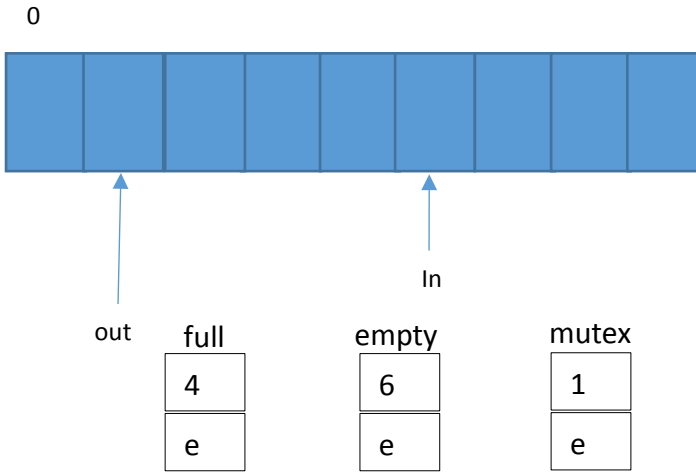
Bounded-Buffer Problem - Semaphore Solution

- Producer process

```
do {
        ...
    /* produce an item in
next_produced */
     ...
    wait(empty);
    wait(mutex);
     ...
    /* add next_produced to
the buffer */
     ...
    signal(mutex);
    signal(full);
} while (true);
```

- Consumer process

```
do {
    wait(full);
    wait(mutex);
     ...
    /* remove an item from
buffer to next_consumed */
     ...
    signal(mutex);
    signal(empty);
     ...
    /* consume the item in
next_consumed */
     ...
} 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
- Issues to address
  - Allow multiple readers to read at the same time
  - Only one single writer can access at a time
- Several variations of how readers and writers are considered in providing their access, all involve some form of priorities
- Shared data
  - Data set
  - Semaphore rw_mutex initialized to 1
  - Semaphore mutex initialized to 1
  - Integer read_count initialized to 0

Readers-Writers Problem

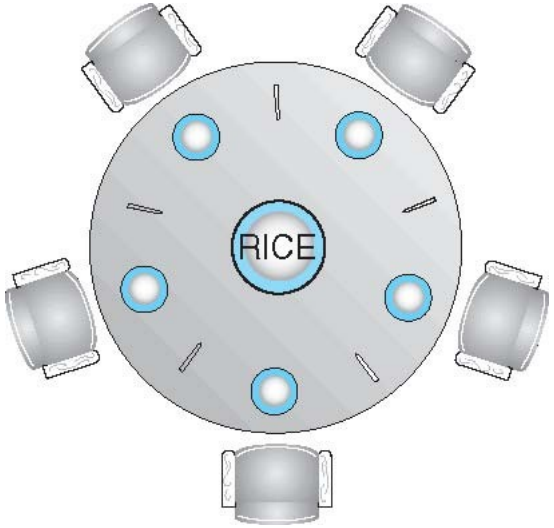
- Reader

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
       ...
    /* reading is performed */
       ...
    wait(mutex);
    read count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

- Writer

```
do {
    wait(rw_mutex);
       ...
    /* writing is
 performed */
       ...
    signal(rw_mutex);
} while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- No interaction with neighbors. Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl. Need both to eat, then release both when done
- Shared data, for the case with 5 philosophers
  - Bowl of rice (data set)
  - Semaphore chopsticks[5], all initialized to 1

Dining-Philosophers Problem

- Philosopher i

```
do {
    //  think

    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );
        //  eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
} while (TRUE);
```

- Problem with this solution?

Dining-Philosophers Problem

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up the chopsticks only if both are available (i.e., picking must be done in a critical section)
  - Use an asymmetric solution – an odd-numbered philosopher picks up first the left chopstick and then the right chopstick, and even-numbered philosopher picks up first the right chopstick and then the left chopstick

Problems with Semaphores

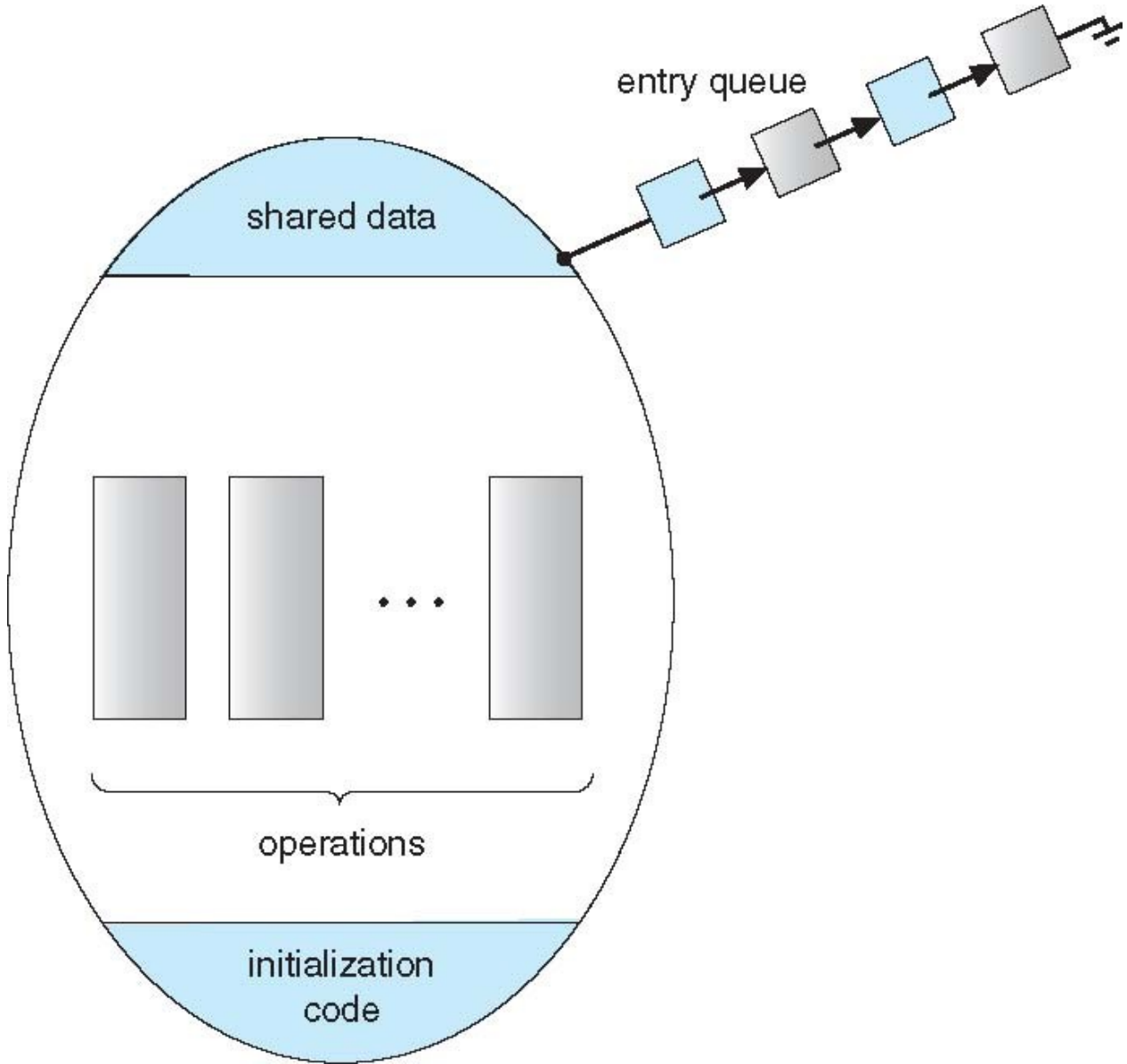
- Incorrect use of semaphore operations
  - signal(mutex) … wait (mutex)
  - wait(mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) or both
- Deadlock and starvation are possible

Monitors

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

```
Monitor monitor-name
{
   // shared variable declarations
   procedure P1 ( … ) { … }
   procedure P2 ( … ) { … }
   Initialization code
}
```

Schematic View of a Monitor



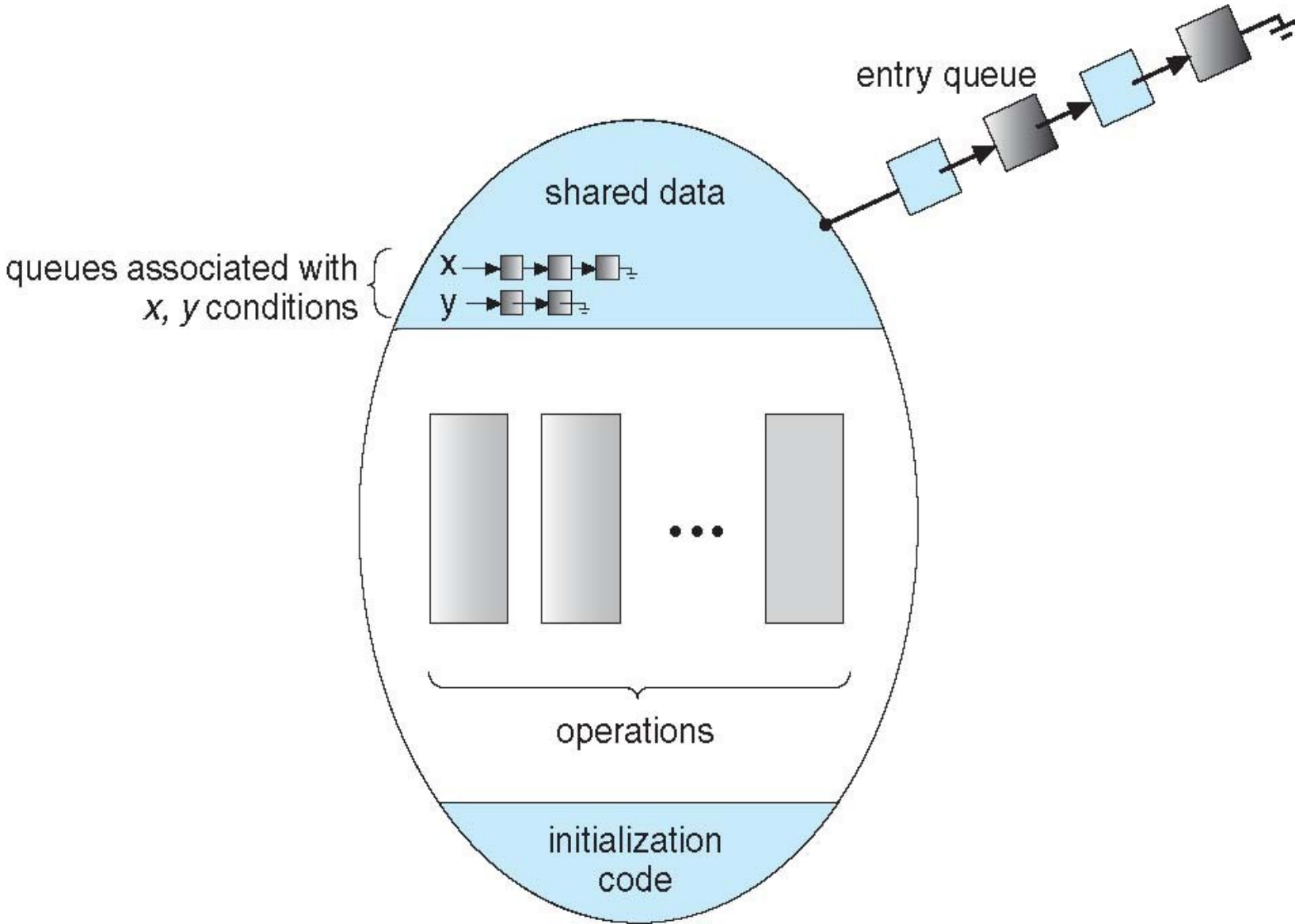
Condition Variables

- Declaration

  condition x, y;

- Two operations are available on a condition variable x:
  - x.wait() – a process that invokes this operation is suspended until x.signal()
  - x.signal() – resumes one of the processes (if any) that invoked x.wait(). No effect if no x.wait() on the variable x.

Schematic View of Monitor with Condition Variables



Choice of Implementation of Condition Variables

- If process P invokes x.signal(), and process Q is suspended in x.wait(), what should happen next?
  - Issue: both P and Q cannot execute concurrently. If Q resumed then P must wait
- Options include
  - Signal and wait – P waits until Q either leaves the monitor or it waits for another condition
  - Signal and continue – Q waits until P either leaves the monitor or it waits for another condition.
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING) state [5] ;
    condition self [5];

    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, enable them to eat
         // if hungry and waiting
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
```

Solution to Dining Philosophers (cont.)

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

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

Solution to Dining Philosophers (cont.)

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

```
DiningPhilosophers.pickup(i);
    EAT
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Dining Philosophers Problem - A Scenario

5 philosophers.

state

0	1	2	3	4
THINKING	THINKING	THINKING	THINKING	THINKING

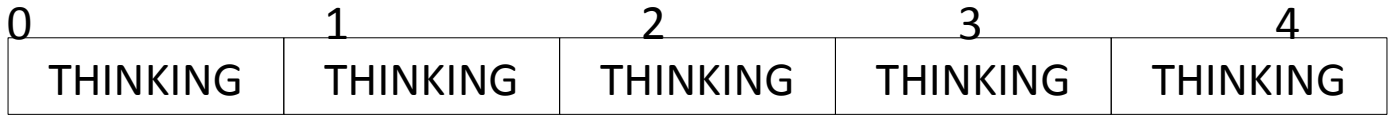
self

0	1	2	3	4
--	--	--	--	--

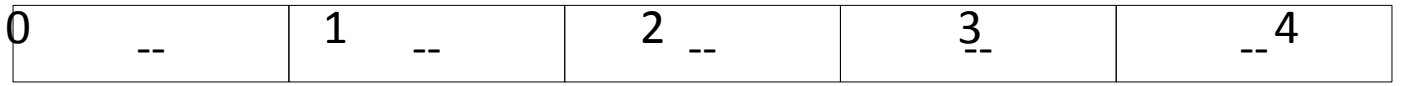
Note: self is an array of condition variables, each consisting of a queue, initialized to empty (symbol – indicates empty)

Dining Philosophers Problem – Scenario (cont)

Philosopher 2 invokes DiningPhilosophers.pickup(2):



state



self

Note: self is an array of condition variables, each consisting of a queue, initialized to empty (symbol – indicates empty)

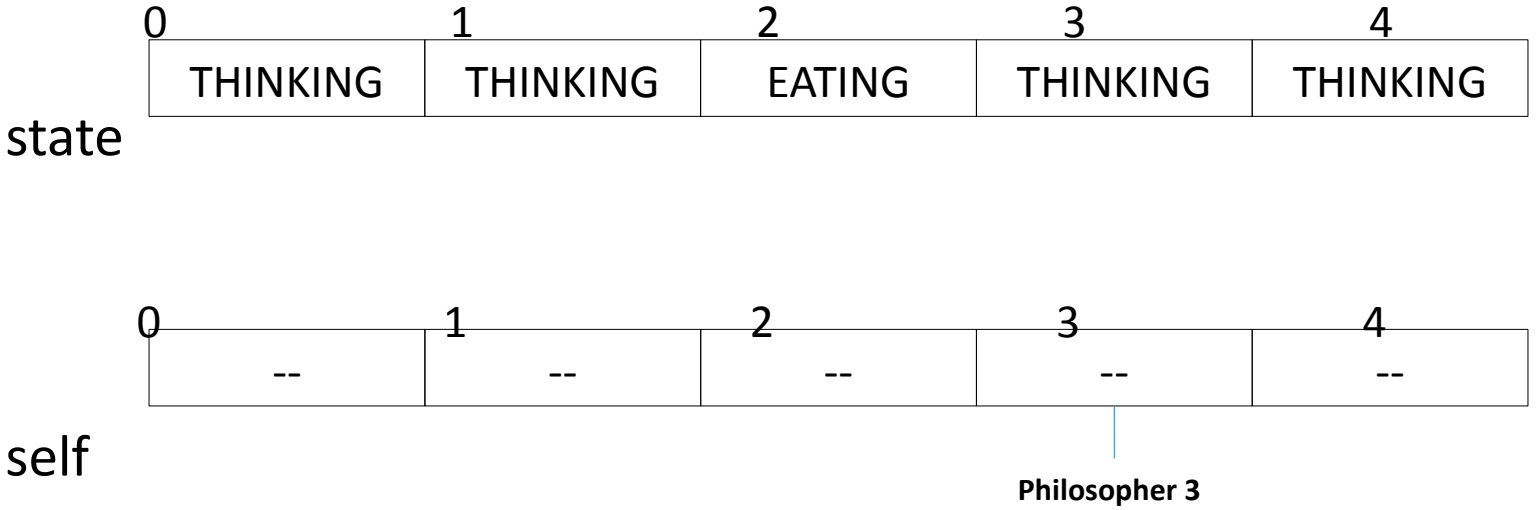
```
void pickup (int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING) self[i].wait;
}

void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors, enable them
    // to eat if hungry and waiting
    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 ;
         self[i].signal() ;
    }
}
```

Dining Philosophers Problem – Scenario (cont)

As Philosopher 2 is EATING, Philosopher 3 invokes DiningPhilosophers.pickup(3):



Note: self is an array of condition variables, each consisting of a queue, initialized to empty (symbol – indicates empty)

```
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, enable them
    // to eat if hungry and waiting
    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 ;
         self[i].signal() ;
    }
}
```

Dining Philosophers Problem – Scenario (cont)

Philosopher 2 invokes DiningPhilosophers.putdown(2):

state

self

Note: self is an array of condition variables, each consisting of a queue, initialized to empty (symbol – indicates empty)

```
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, enable them
    // to eat if hungry and waiting
    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 ;
         self[i].signal() ;
    }
}
```

Monitor Solution to Producer/Consumer Problem

```
monitor bounded-buffer
{
    message B[N]; // message: a user-defined type
            // N a predefined constant
    int in, out, count;
    condition nonfull, nonempty;

    void place( message x) {
        if (count == N) nonfull.wait;
        B[in] = x;
        in = (in + 1) % N;
        count = count + 1;
        nonempty.signal;
    }
```

Solution to Producer/Consumer (cont)

```
    void remove( message* x ) {
     if (count == 0) nonempty.wait;
        x = B[out];
        out = (out + 1) % N;
        count = count - 1;
        nonfull.signal;
    }

    initialization_code() {
        in = 0;
        out = 0;
        count = 0;
    }
}
```

Solution to Producer/Consumer (cont)

- Each producer invokes place() operation as follows:

```
. . .
produce message m;
bounded-buffer.place (m);
. . .
```

- Each consumer invokes remove() operation as follows:

```
. . .
bounded-buffer.remove(m);
consume message m;
. . .
```

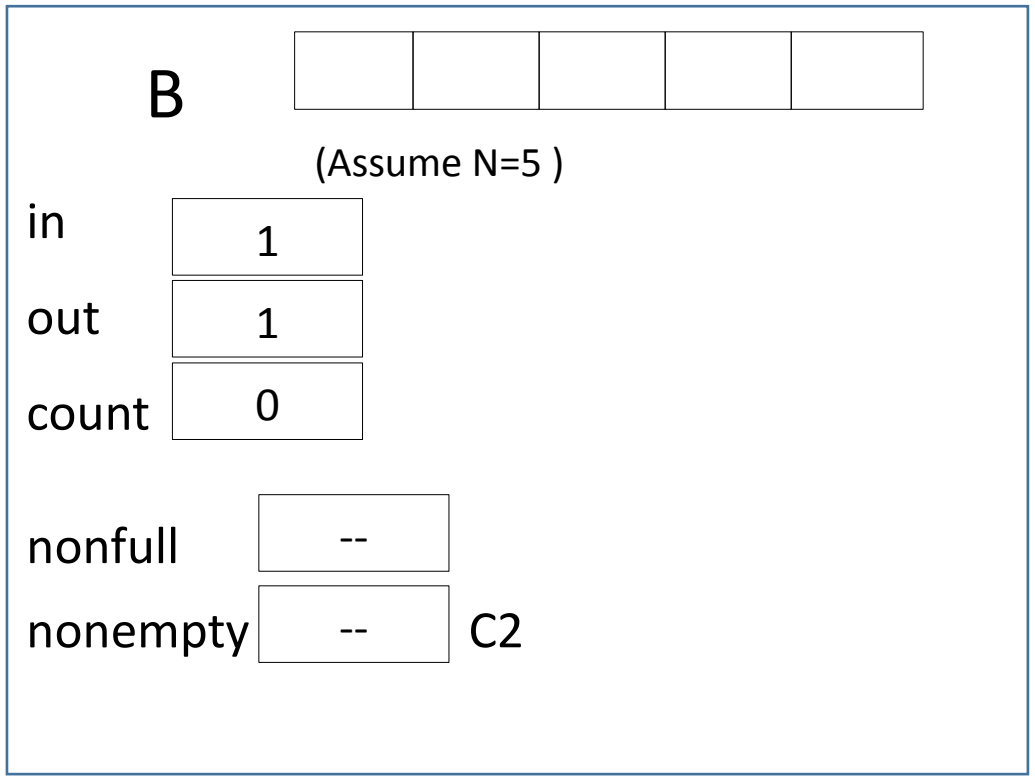
Monitor Solution to Producer/Consumer – A Scenario

Scenario

C1 calls bounded-buffer.remove(m)

C2 calls bounded-buffer.remove(m)

P1 calls bounded-buffer.place(m)




```
void place( message x) {
     if (count == N) nonfull.wait;
     B[in] = x;
     in = (in + 1) % N;
     count = count + 1;
     nonempty.signal;
}

void remove( message* x ) {
     if (count == 0) nonempty.wait;
     x = B[out];
     out = (out + 1) % N;
     count = count - 1;
     nonfull.signal;
}
```

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() is executed, which should be resumed?
- FCFS frequently not adequate
- Conditional-wait construct of the form x.wait(c)
  - Where c is priority number
  - Processes with lowest number (highest priority) is scheduled next

Single Resource Allocation Problem

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t);
. . .
Access the resource
. . .
R.release;
```

where R is an instance of type ResourceAllocator

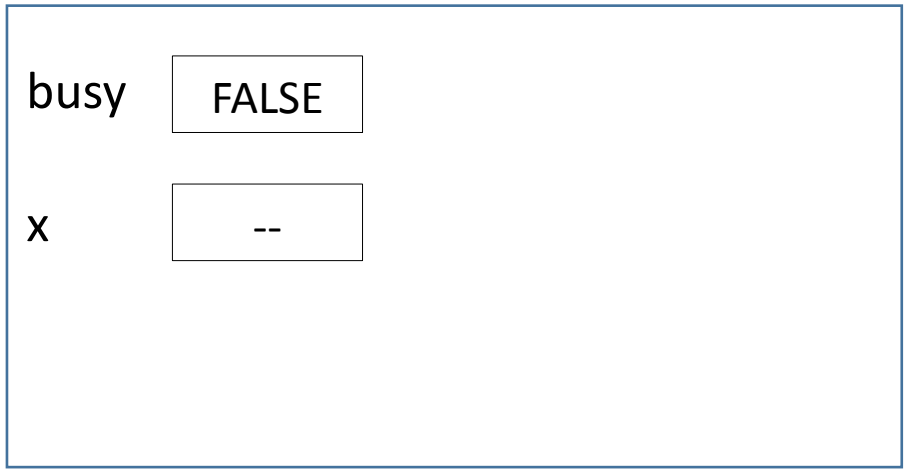
A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
 if (busy)
 x.wait(time); // processes queue in shortest
                          // job first order
 busy = TRUE;
   }
   void release() {
 busy = FALSE;
 x.signal();
   }
   initialization code() {
       busy = FALSE;
   }
}
```

Monitor Solution to Resource Allocation – A Scenario

Scenario

P1 calls ResourceAllocator.acquire(5)
P2 calls ResourceAllocator.acquire(20)
P3 calls ResourceAllocator.acquire(10)
P1 calls ResourceAllocator.release()




```
void acquire(int time) {
     if (busy)
        x.wait(time); // processes queue in
                      // in shortest job first
   // order
     busy = TRUE;
}

void release() {
    busy = FALSE;
    x.signal();
}
```

Synchronization Example - Linux

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections. Version 2.6 and later, fully preemptive
- Linux provides:
  - Atomic integers
  - Mutex locks
  - Semaphores
  - Spinlocks
  - Reader-writer versions of both semaphore and spinlock
- On single-CPU system, spinlocks are replaced by enabling and disabling kernel preemption