Process Synchronization

Background

- Cooperation is required for several reasons like: Information sharing, Computation speedup, Modularity, Convenience
- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

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. (~ Deadlock)

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. (~ Starvation)

Requirements for Solution

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its non-critical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

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
        remainder 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) do nothing;
        critical section
    turn = j;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

- This solution guarantees mutual exclusion
- Drawback 1: processes must strictly alternate
- Drawback 2: if one processes fails other process is permanently blocked
- This problem arises due to fact that it stores name of the process that may enter critical section rather than the process state

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]) do nothing;
    critical section
    flag [i] = false;
        remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

- This approach Satisfies mutual exclusion
- This approach may lead to dead lock

What is wrong with this implementation ?

- A process sets its state without knowing the state of other. Dead lock occurs because each process can insist on its right to enter critical section

Algorithm 3 (Peterson's Solution)

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

```
do {
    flag [i]:= true;
    turn = j;
    while (flag [j] and turn = j) do nothing;
        critical section
    flag [i] = false;
        remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Bakery Algorithm

Critical section for n processes

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

Bakery Algorithm

- Notation <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0,\ldots, n - 1$
- 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]) do no-op ;
            while ((number[j] != 0) && ((number[j],j) < (number[i],i)))
            do no-op ;
    }
      critical section
    number[i] = 0;
      remainder section
} while (1);
```

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - ➔A process runs until it invokes an operating-system service or until it is interrupted
  - ➔Disabling interrupts guarantees mutual exclusion
  - ➔Processor is limited in its ability to interleave programs
  - ➔Multiprocessing
    - disabling interrupts on one processor will not guarantee mutual exclusion

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - ➔Performed in a single instruction cycle (Atomic)
  - ➔Not subject to interference from other instructions
  - ➔Test and Set
  - ➔Swap

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)) do no-op;
        critical section
    lock = false;
        remainder section
}
```

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);
```

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;

- Process P_i

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

- **Key is a local boolean variable**

Mutual Exclusion Machine Instructions

- Advantages
  - ➔Applicable to any number of processes on either a single processor or multiple processors. It is simple and therefore easy to verify
  - ➔It can be used to support multiple critical sections

Mutual Exclusion Machine Instructions

- Disadvantages
  - ➔Busy-waiting consumes processor time
  - ➔Starvation is possible when a process leaves a critical section and more than one process is waiting.

Semaphores (OS Support)

- Semaphore is a variable that has an integer value
  - ➔May be initialized to a nonnegative number
  - ➔Wait operation decrements the semaphore value
  - ➔Signal operation increments semaphore value

- Wait and signal operations cannot be interrupted
- If a process is waiting for a signal, it is suspended until that signal is sent
- Queue is used to hold processes waiting on the semaphore

Semaphores

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

wait (*S*):

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

***S*--;**

---

signal (*S*):

***S*++;**

- ***In the above definition the value of semaphore is never negative***

Critical Section of *N* Processes

- Shared data:

  semaphore mutex; ***//***initially *mutex* = 1

- Process *Pi:*

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

- ***SPINLOCKS (Busy Waiting):* Useful only if waiting period is small and hence may save a context switch**

Avoiding Busy Waiting

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 are 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);
}
```

- **Value of semaphore can be negative and represents the number of processes waiting on it**

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
$\vdots$	$\vdots$
A	*wait*(*flag*)
signal(*flag*)	*B*

- Semaphore operations are 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);
}
```

- **Value of semaphore can be negative and represents the number of processes waiting on it**

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.

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.

Classical Problems of Synchronization

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

Bounded-buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1
- **Buffer size is n**
- **Mutex provides exclusive access to the buffer**
- **Consumers wait on full**
- **Producers wait on empty**

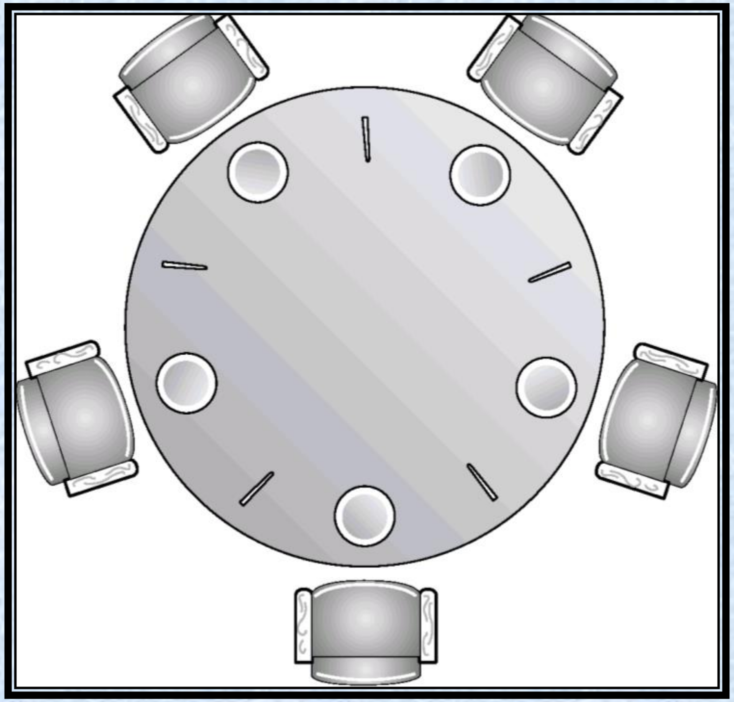
Bounded-buffer Problem: Producer Process

```
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);
```

Dining-philosophers Problem



- Shared data

  semaphore chopstick[5];

Initially all values are 1

- To start eating, a philosopher needs two chopsticks
- A philosopher may pick up only one chopstick at a time
- After eating, the philosopher releases both the chopsticks

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);
```

The solution is not deadlock free!!

➔All philosophers pick up left chopsticks!!

➔Allow at most 4 philosophers to be sitting on the table

➔Allow a philosopher to pick chopsticks only if both are available

➔An *odd* philosopher picks up first the left and then the right chopstick while a *even* philosopher does the reverse

Implementation

- Semaphore operations are 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);
}
```

- **Value of semaphore can be negative and represents the number of processes waiting on it**

Readers-Writers Problem

- Two readers can access the shared data item simultaneously.
- A writer requires exclusive access
- No reader should wait unless a writer is already in critical section
- Writers may starve
- Shared data

  var *mutex*, *wrt*: *semaphore* (=1);

  readcount : *integer* (=0);
- *wrt* is common to both readers and writers. It functions as mutual exclusion semaphore for writers.It is also used by first and last reader that enters or exits the CS.
- *readcount* keeps track of how many readers are currently accessing the object.
- *mutex* provides mutual exclusion for updating *readcount*.

Readers-Writers Problem

- Writer process

wait(*wrt*);

…

writing is performed

…

signal(*wrt*);

Readers-Writers Problem (Cont.)

- 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):
```

Monitors

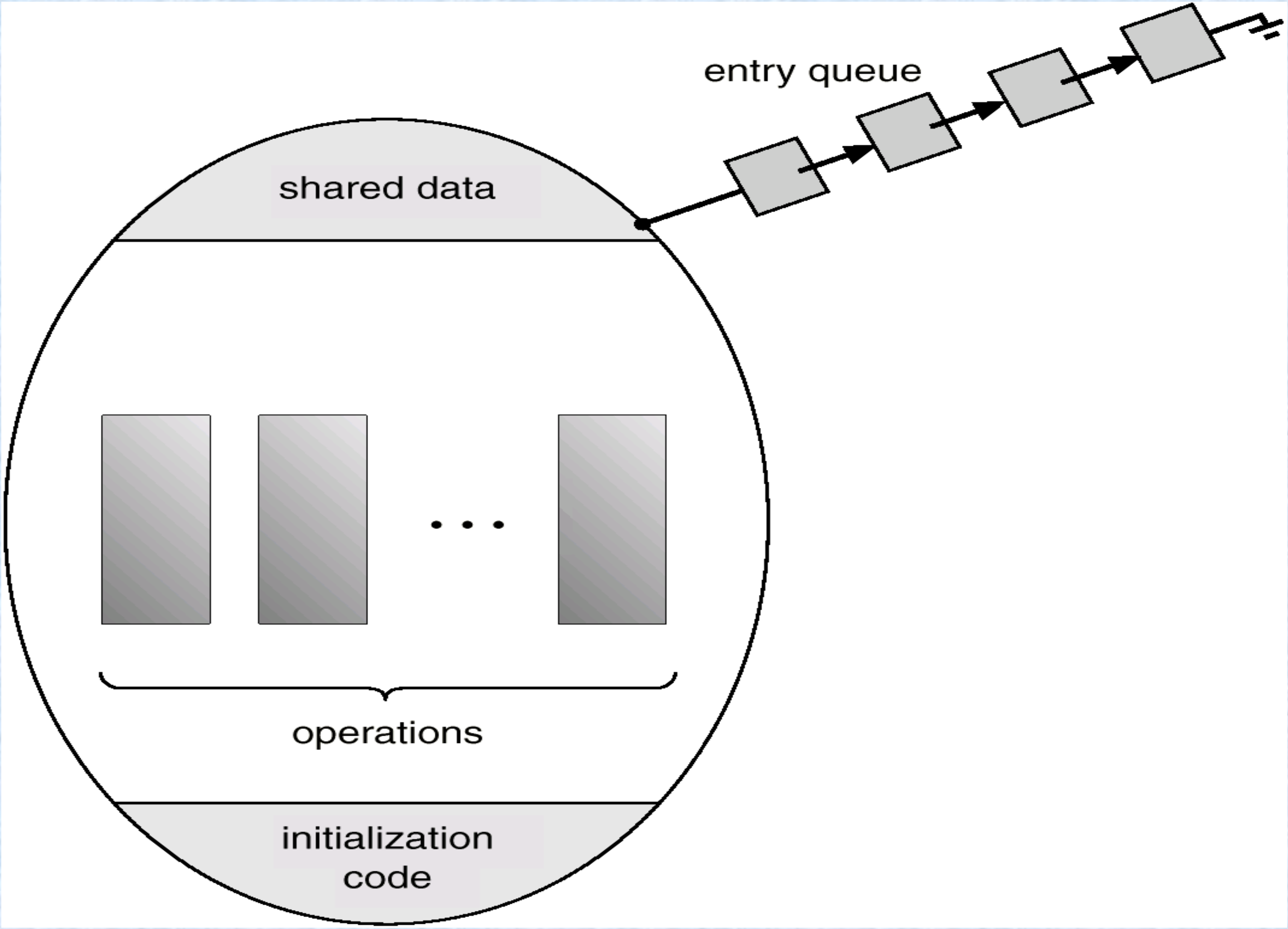
- Monitor is a software module
- Chief characteristics
  - ➔Local data variables are accessible only by the monitor
  - ➔Process enters monitor by invoking one of its procedures
  - ➔Only one process may be executing in the monitor at a time

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 Pn (…);
        begin…end;
    begin
        initialization code
    end
```

Schematic view of a monitor



Monitors (Cont.)

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

  var *x, y: condition*

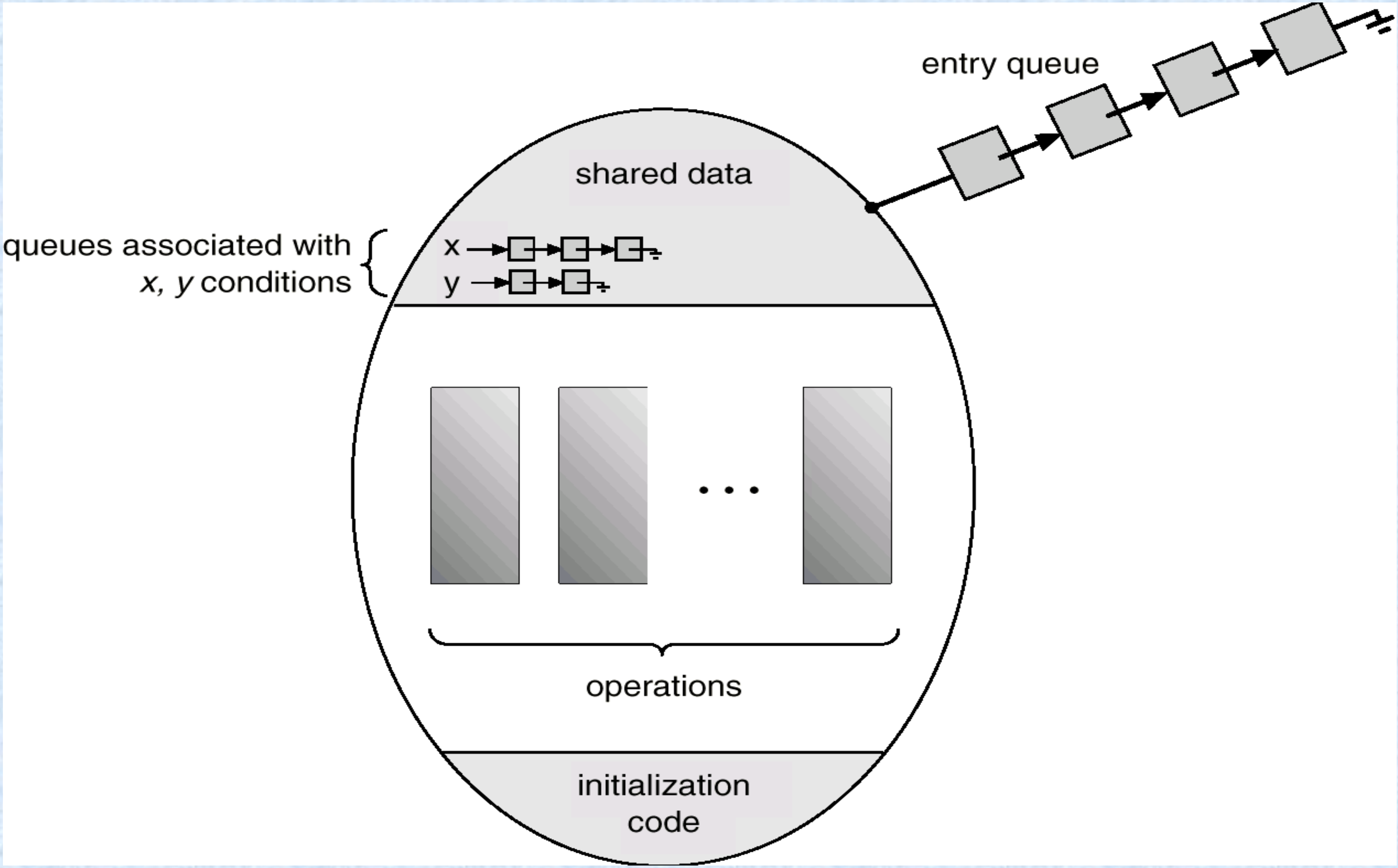
- Condition variable can only be used with the operations *wait* and *signal*.
  - The operation

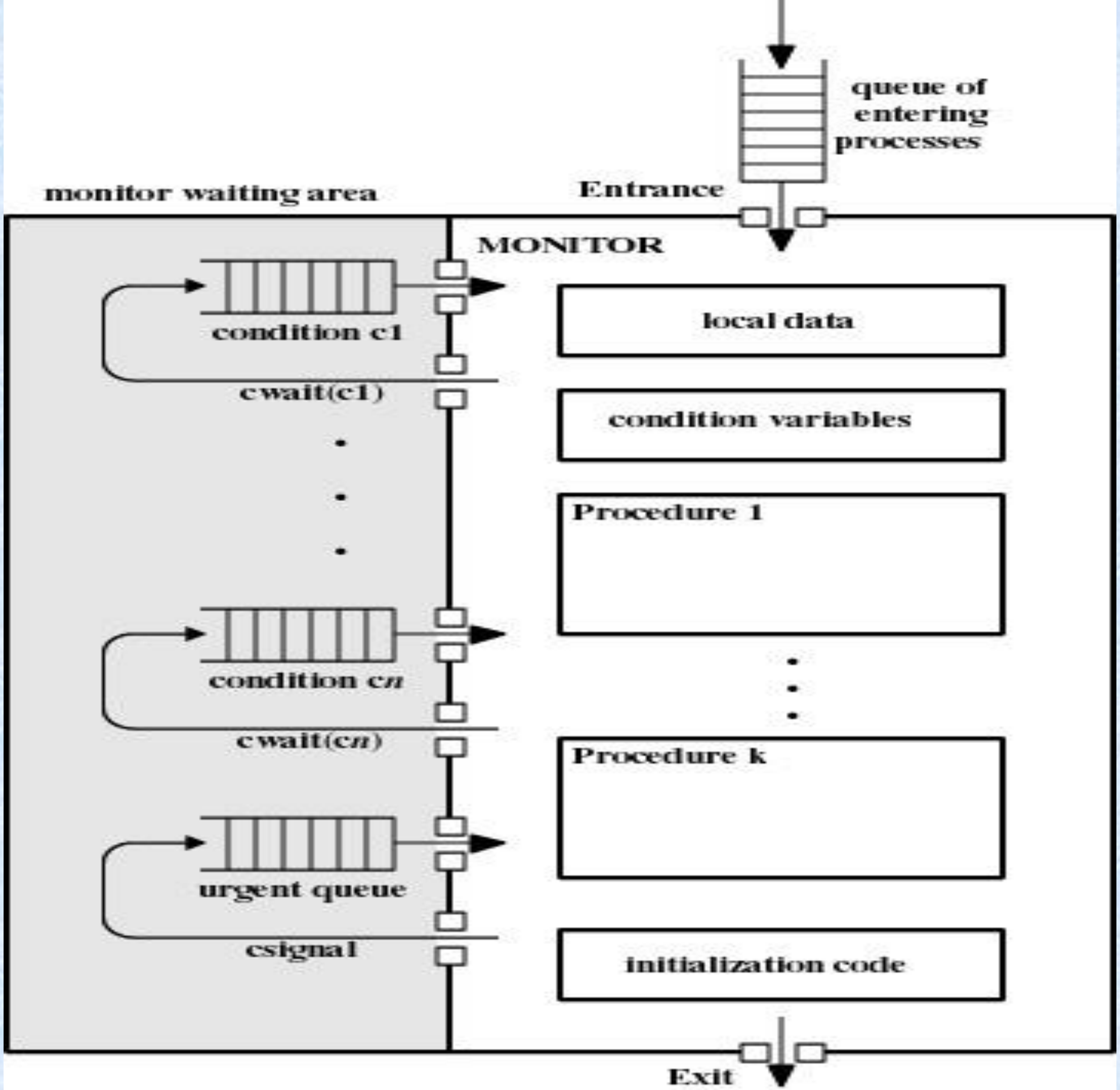
    x.wait;

    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.

Monitor with condition variables




queue of entering processes
monitor waiting area
Entrance
MONITOR
condition c1
cwait(c1)
condition cn
cwait(cn)
urgent queue
csignal
local data
condition variables
Procedure 1
Procedure k
initialization code
Exit

Dining Philosophers Example

```
type dining-philosophers = monitor
   var state : array [0..4] of :(thinking, hungry, eating);
   var self : array [0..4] of condition;
   procedure  pickup (i: 0..4);
       begin
             state[i] := hungry;
             test (i);
             if state[i] ≠ eating then self[i].wait;
       end;

   procedure  putdown (i: 0..4);
       begin
             state[i] := thinking;
             test (i+4 mod 5);
             test (i+1 mod 5);
       end;
```

Dining Philosophers (Cont.)

```
procedure test(i: 0..4);
    begin
        if state[i+4 mod 5] ≠ eating
            and state[i] = hungry
            and state[i+1 mod 5] ] ≠ eating
            then begin
                    state[i] := eating;
                    self[i].signal;
            end;

        end;
```

```
    begin
        for i := 0 to 4
            do state[i] := thinking;
    end.
```

Philosopher *i* must invoke the operations *pickup* and *putdown* in the following sequence:

dp.pickup(i)

…..

Eat

…..

dp.putdown(i)

Monitor Implementation Using Semaphores

- Variables

  var *mutex: semaphore* (init = 1)
  next: semaphore (init = 0)
  next-count: integer (init = 0)

- Each external procedure *F* will be replaced by

  wait(*mutex*);
  …
  body of *F*;
  …
  if next-count > 0
  then signal(next)
  else signal(mutex);

- Mutual exclusion within a monitor is ensured.

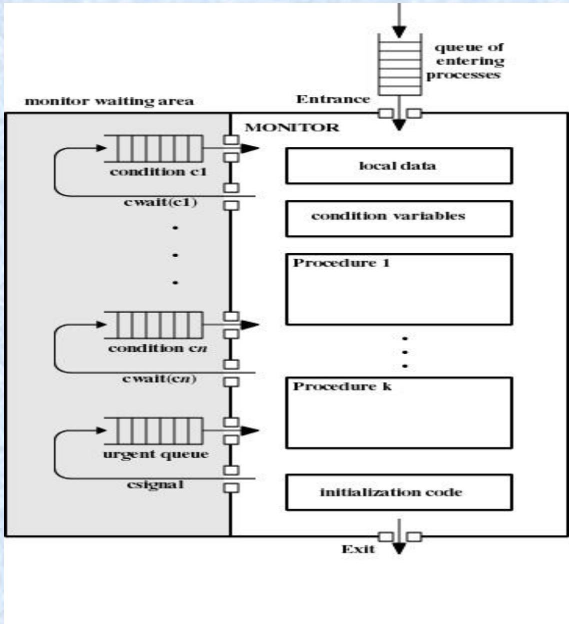


Figure 5.21 Structure of a Monitor

Monitor Implementation (Cont.)

- For each condition variable x, we have:

  var *x-sem*: *semaphore* (init = 0)

  x-count: integer (init = 0)

- The operation x.wait can be implemented as:

```
x-count := x-count + 1;
if next-count >0
    then signal(next)
    else signal(mutex);
wait(x-sem);
x-count := x-count – 1;
```

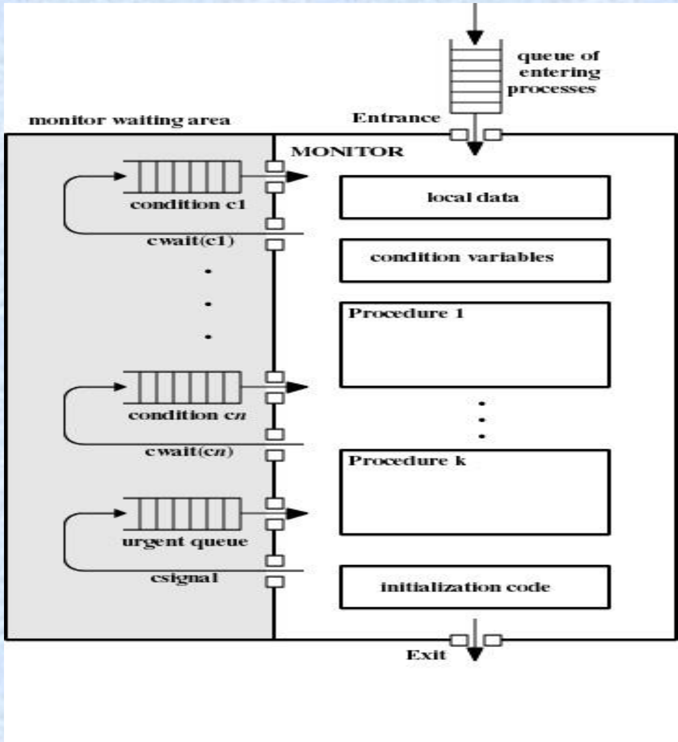


Figure 5.21 Structure of a Monitor

Monitor Implementation (Cont.)

- The operation x.signal can be implemented as:

if *x-count* > 0

then begin

next-count := *next-count* + 1;

signal(x-sem);

wait(next);

next-count := *next-count* – 1;

end;

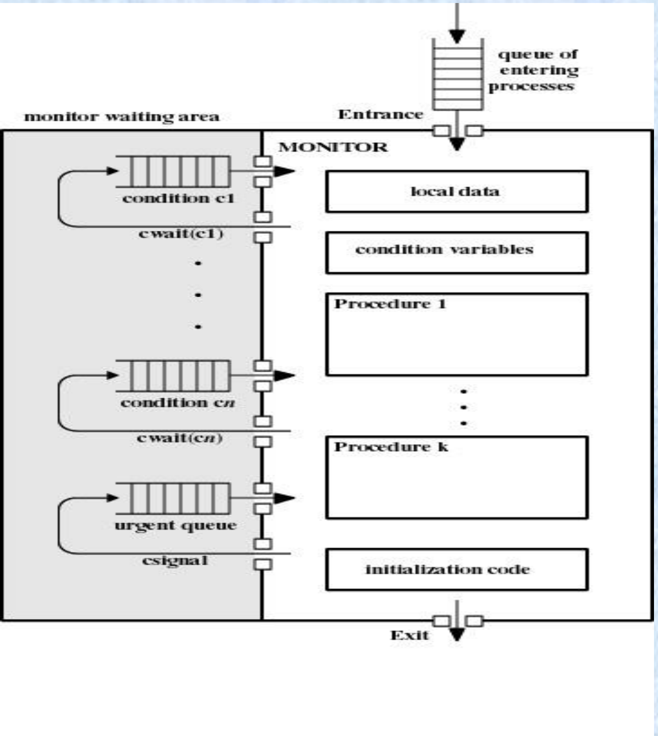


Figure 5.21 Structure of a Monitor

Monitor Implementation (Cont.)

- *Conditional-wait* construct: *x.wait(c);*
  - *c* – integer expression evaluated when the wait operation is executed.
  - value of *c* (*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.