Monitors & Deadlocks

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

Contents

Concept of Monitor

- Various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.
  - All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward.
  - If this sequence is not observed, two processes may be in their critical sections simultaneously.
  - It's difficult to detect these errors since they happen only if particular execution sequences take place and these sequences do not always occur.
  - This situation may be caused by an honest programming error or an uncooperative programmer.
- Example 1 and example 2 show us the various difficulties that may result. Note that these difficulties will arise even if a single process is not well behaved.

Concept of Monitor

- Example 1.
  - Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
    ...
critical section
    ...
wait(mutex);
```

  - In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.
  - This error may be discovered only if several processes are simultaneously active in their critical sections. Note that this situation may not always be reproducible.

Concept of Monitor

- Example 2.
  - Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes:

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

  - In this case, a deadlock will occur.
- To deal with such errors, researchers have developed high-level language constructs.
  - The *monitor type* is a fundamental high-level synchronization construct that provides a convenient and effective mechanism for process synchronization.
    - found in many concurrent programming languages

Concept of Monitor

- Monitor Usage
  - A *monitor type* is an Abstract Data Type (ADT) that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor.
  - The monitor type also declares the variables whose values define the state of an instance of that ADT, along with the bodies of functions that operate on those variables.
  - Layout of a monitor

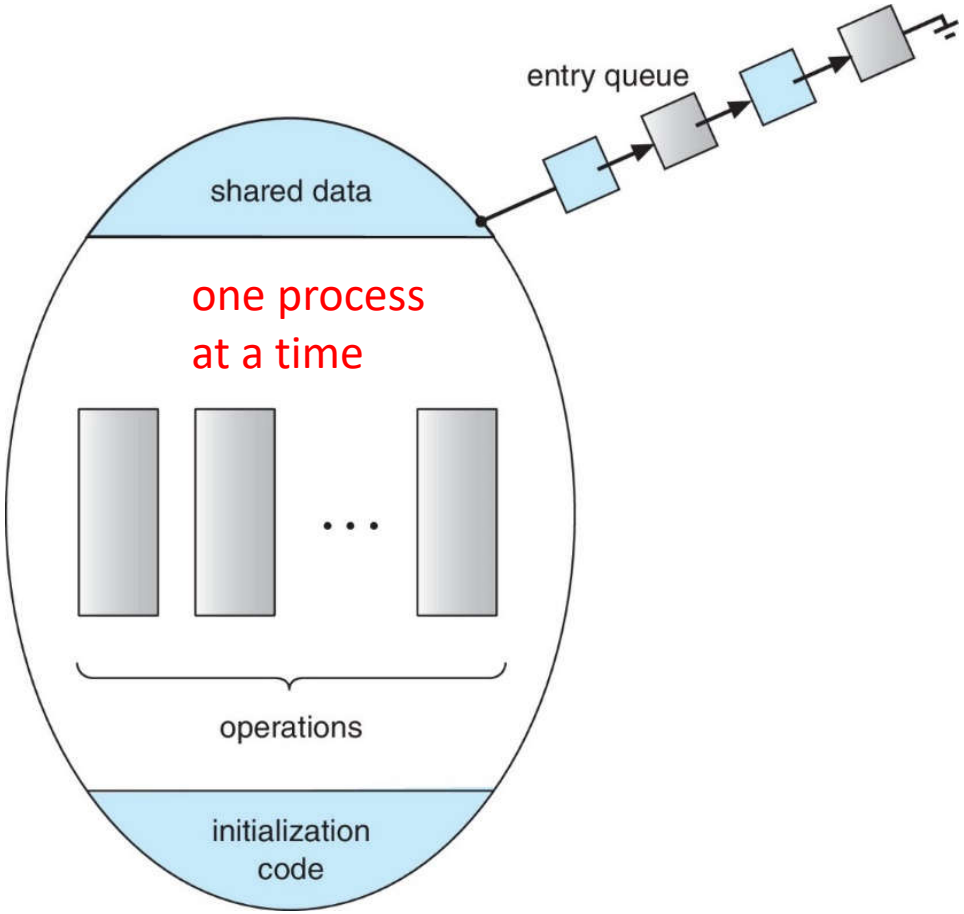
```
monitor monitor-name
{
    shared variable declarations
        ...
    function P1 ( … ) { … }
    function P2 ( … ) { … }
        …
    function Pn ( … ) { … }

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

Concept of Monitor

- Monitor Type
  - *Sharing*: A monitor is shared by concurrent processes.
  - *Encapsulation & Security:* The representation of a monitor cannot be used directly by the various processes.
    - Any function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
    - The local variables of a monitor can be accessed by only the local functions.
  - *Mutual Exclusion*: The monitor construct ensures that only one process at a time is active within the monitor.
    - Programmer does not need to code this synchronization constraint explicitly.
      - Hence, shared data are protected by placing them in the monitor.
      - The monitor locks shared data on process entry.
  - A monitor can be implemented by semaphores.

Concept of Monitor

- Monitor Type
  - Schematic view of a monitor



Monitor Conditions

- Condition Type
  - The monitor is not sufficiently powerful for modeling some synchronization schemes.
  - The *condition construct* provides additional synchronization mechanisms for implementation of process synchronization done by programmers.
    - Variables of type condition:

      ```
      condition x, y;
      ```

    - Condition variables are local to the monitor (accessible only within the monitor).
  - The only operations that can be invoked on a condition variable are conditional wait cwait and conditional signal csignal executed on an associated queue. For example, the conditional wait operation

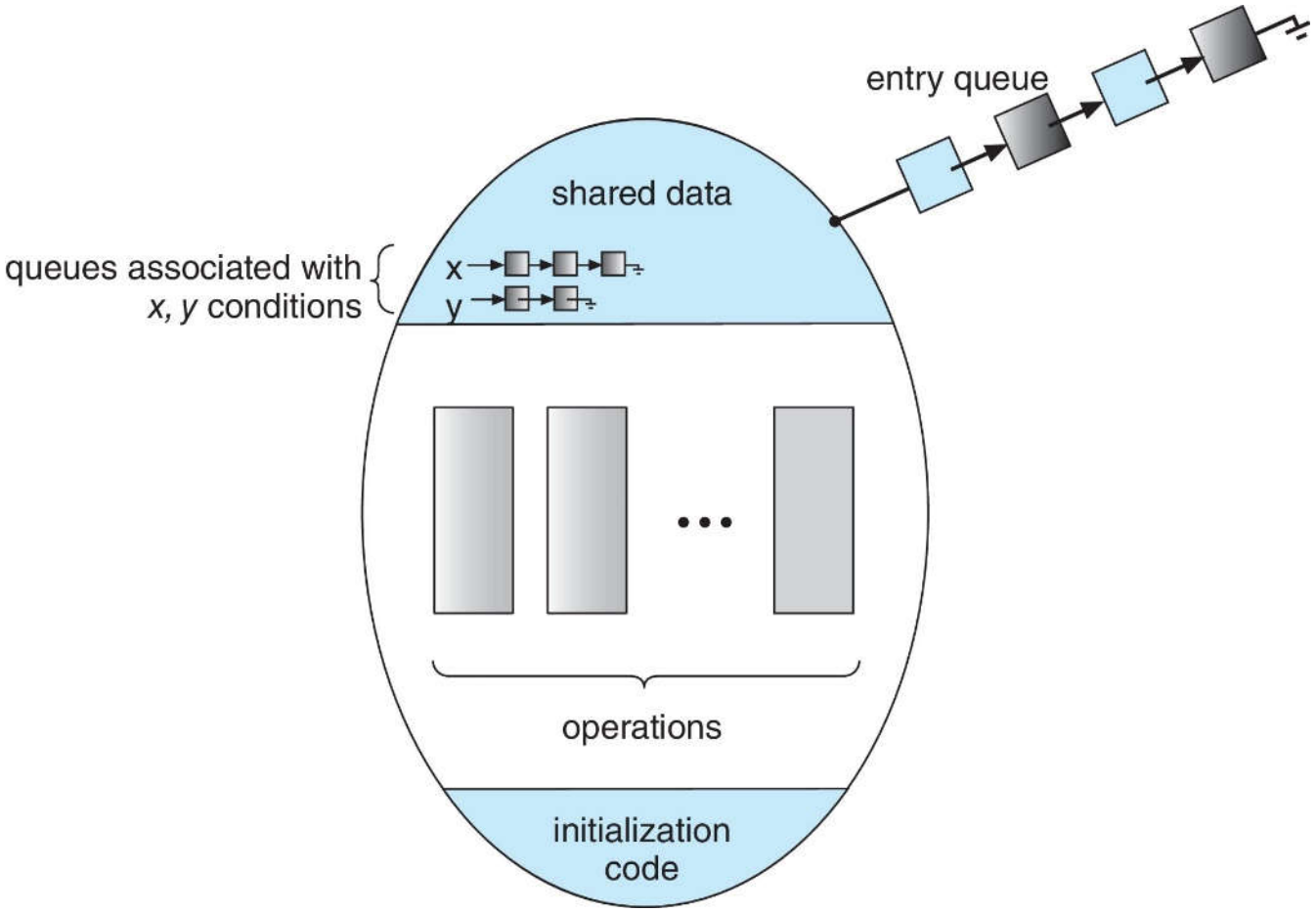
    ```
    x.wait();
    ```

    means that the process invoking this operation is suspended until another process invokes

    ```
    x.signal();
    ```

Monitor Conditions

- Monitor with Condition Variables
  - Awaiting processes are either in the entrance queue or in a condition queue.

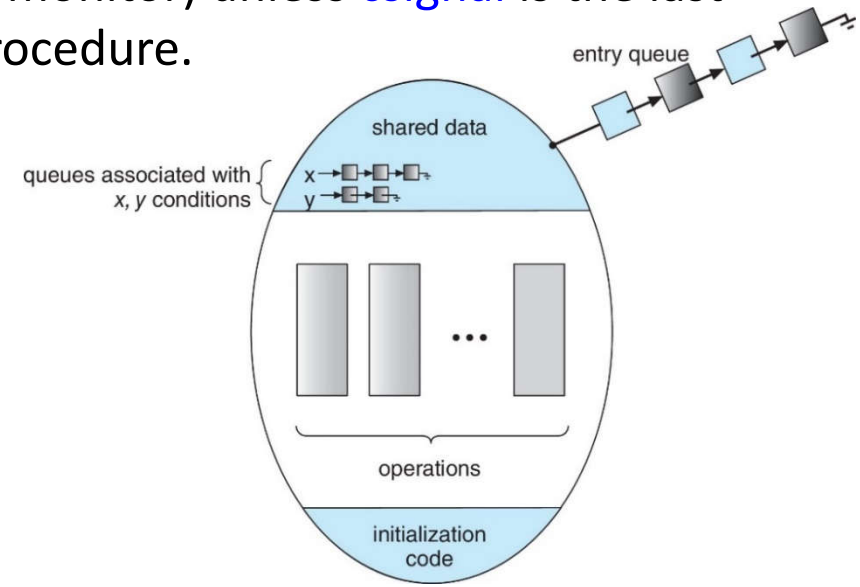


Monitor Conditions

- cwait and csignal Operations
  - Process synchronization is done by the programmer by using condition variables
    - A process may need to wait for the conditions before executing in the monitor.
  - The cwait operation x.wait(); or cwait(x);
    - x.wait() means that the process invoking this operation is suspended/blocked until another process invokes x.signal().
  - The csignal operation x.signal(); or csignal(x);
    - x.signal() resumes exactly one suspended process that invoked x.wait(). If no process is suspended, then the csignal operation has *no effect*.
    - Contrast this operation with the normal signal(mutex) operation associated with semaphores, which always affects the state of the semaphore by mutex.value++.
  - We may need an additional integer counter x_count to describe the length of the waiting queue for condition x (a condition queue).

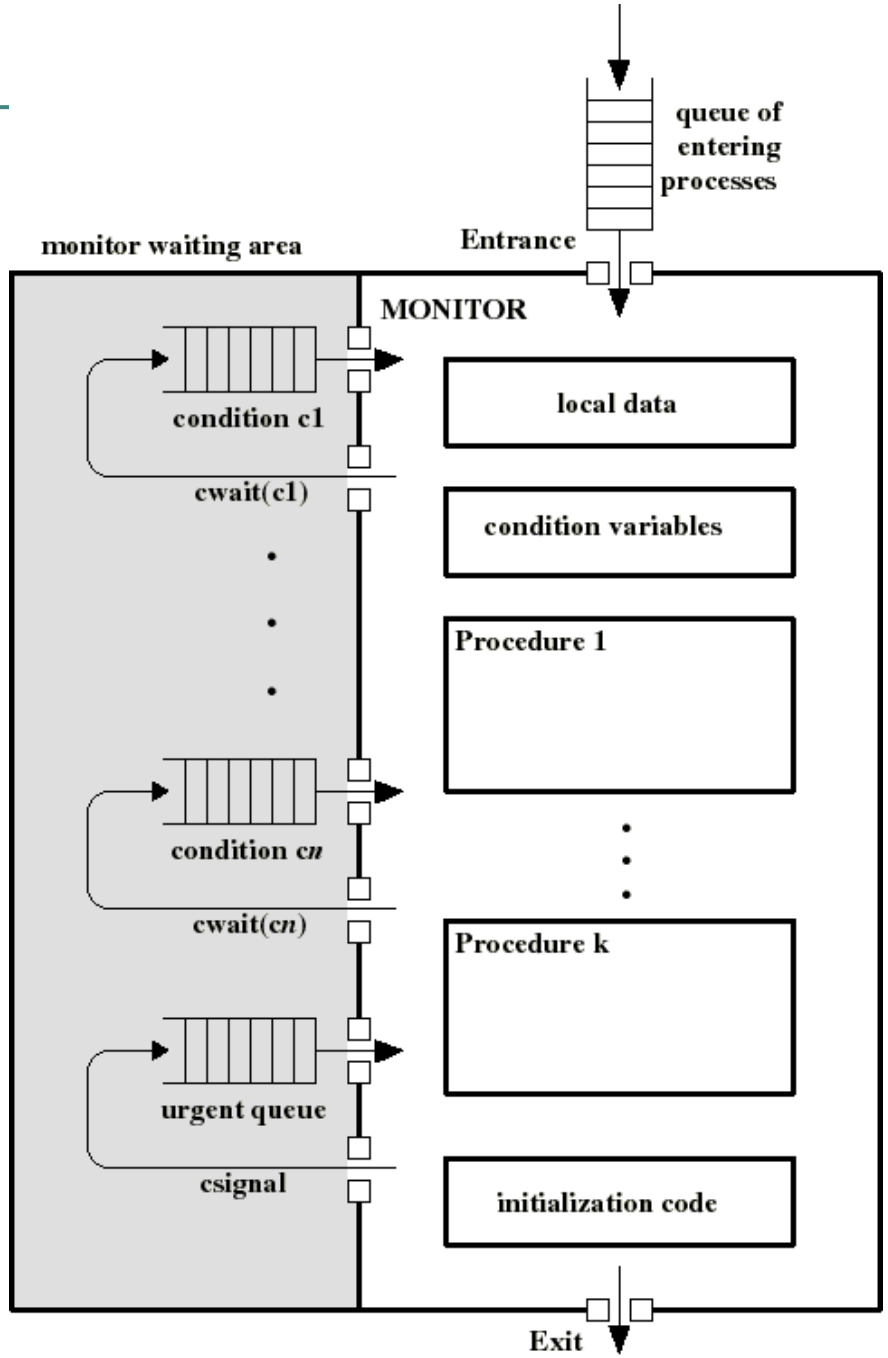
Monitor Conditions

- cwait and csignal Operations
  - Awaiting processes are either in the entrance queue or in a condition queue.
  - A process puts itself into condition cn queue by issuing cwait(cn).
  - csignal(cn) brings one process in condition cn queue into the monitor.
    - Remember that the monitor is mutual exclusive, hence csignal(cn) blocks the calling process and puts it in the *urgent queue* (waiting to enter the monitor) unless csignal is the last operation of the monitor procedure.



Monitor Conditions

- cwait and csignal Operations – One Possible Dynamics
  - Awaiting processes are either in the entrance queue or in a condition queue.
  - A process puts itself into condition cn queue by issuing cwait(cn).
  - csignal(cn) brings one process in condition cn queue into the monitor.
    - Remember that the monitor is mutual exclusive, hence csignal(cn) blocks the calling process and puts it in the *urgent queue* (waiting to enter the monitor) unless csignal is the last operation of the monitor procedure.



Monitor Conditions

- cwait and csignal Operations
  - Suppose that for a condition x, process P invokes x.signal(), and process Q is suspended in x.wait(). What should happen next?
    - The monitor is mutual exclusive and both Q and P cannot execute in parallel. If Q is resumed, then P must wait.
  - Options include: *(Tony Hoare & Per B. Hansen*, 1974)
    - *Signal-and-wait* – P either waits until Q leaves the monitor or waits for another condition. (*Hoare* scheme)
      - P signals and waits in the urgent queue or wait for another condition. Q resumes.
    - *Signal-and-continue* – Q either waits until P leaves the monitor or waits for another condition.
      - P signals and continues. Q waits in the urgent queue.
      - Positive: P was already executing in the monitor. It seems reasonable that let P continue.
      - Negative: By the time Q is resumed from urgent queue, the logical condition x that Q was waiting for may no longer hold. This may cause a system overhead.

Monitor Conditions

- cwait and csignal Operations
  - Suppose that for a condition x, process P invokes x.signal(), and process Q is suspended in x.wait(). What should happen next?
    - The monitor is mutual exclusive and both Q and P cannot execute in parallel. If Q is resumed, then P must wait.
  - Options include: *(Tony Hoare & Per B. Hansen*, 1974)
    - Improved *Signal-and-wait* (*Hansen*)
      - signal() is the latest statement before leave the monitor.
      - When P signals, it leaves the monitor immediately.
      - Q is immediately resumed.
    - These options have pros and cons (各有利弊) – language implementer can decide.
  - Concurrent Pascal
    - P executing signal() immediately leaves the monitor. Q is resumed.
  - Implemented in other languages including Mesa, C#, Java.

Examples Using Monitor

- Solving Producer-Consumer Problem with Monitor
  - Synchronization is now confined within the monitor.
  - append() and take() are procedures/functions within the monitor
    - They are the only means by which Producer or Consumer can access the buffer.
  - Producer:

```
repeat
    produce v;
    append(v);
forever
```

  - Consumer:

```
repeat
    take(v);
    consume v;
forever
```

Examples Using Monitor

- Solving Bounded-Buffer Producer-Consumer Problem with Monitor
  - needs to hold the buffer:
    - buffer: array[n] of items;
  - needs two condition variables:
    - notfull: csignal(notfull) indicates that the buffer is not full.
    - notempty: csignal(notempty) indicates that the buffer is not empty.
  - needs buffer pointers and counts:
    - nextin: points to next item to be appended.
    - nextout: points to next item to be taken.
    - count: holds the number of items in buffer.

Examples Using Monitor

- Solving Bounded-Buffer Producer-Consumer Problem with Monitor
  - Shared data and conditions:

```
struct items array[n];
int nextin = 0, nextout = 0, count = 0;
condition notfull, notempty;
```

  - Procedure append(v):

```
if (count == n) cwait(notfull);
buffer[nextin] = v;
nextin = (nextin + 1) mod n;
count++;
csignal(notempty);
```

  - Procedure take(v):

```
if (count == 0) cwait(notempty);
v = buffer[nextout];
nextout = (nextout + 1) mod n;
count--;
csignal(notfull);
```

Examples Using Monitor

- Solving Bounded-Buffer Producer-Consumer Problem with Monitor
  - in PASCAL

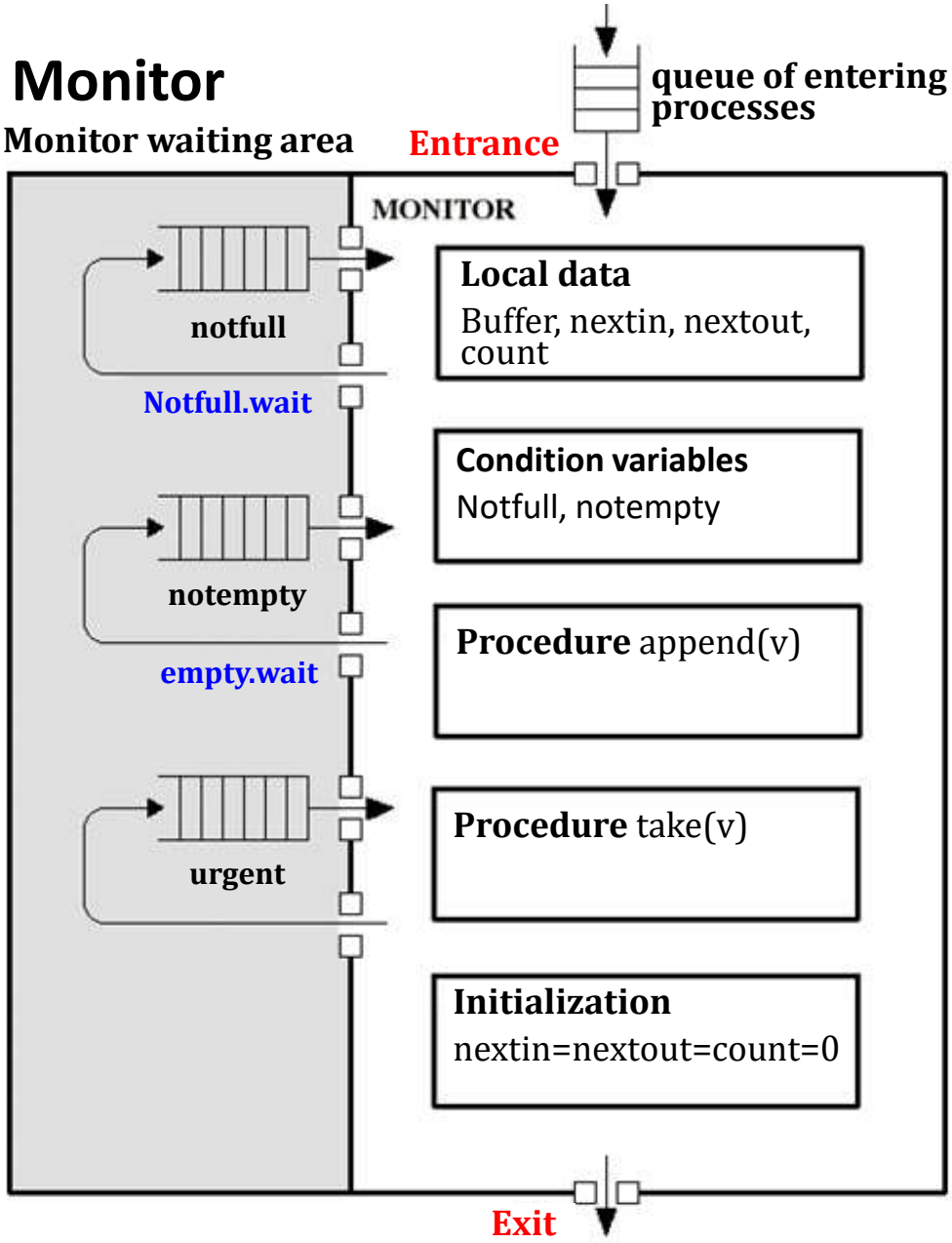
```
Type pc = monitor
Var buffer: array[0,..,n-1] of items;
    notfull, notempty: condition;
    nextin, nextout, count: integer; /* initially 0 */
Procedure entry append(v);
  Begin
    if (count >= n) then notfull.wait;
    buffer[nextin]:= v;
    nextin:= (nextin + 1) mod n;
    count:= count + 1;
    notempty.signal;
  End;
Procedure entry take(v);
  Begin
    if (count = 0) notempty.wait;
    v:= buffer[nextout];
    nextout:= (nextout + 1) mod n;
    count:= count – 1;
    notfull.signal;
  End;
```

Examples Using Monitor

- Solving Bounded-Buffer Producer-Consumer Problem with Monitor
  - in PASCAL

```
Begin
  nextin:= nextout:= count:= 0;
End.

Producer:
  Begin
    repeat
      produce an item v;
      pc.append(v);
    until false
  End;

Consumer:
  Begin
    repeat
      pc.take(v);
      consume the item v;
    until false
  End;
```

Examples Using Monitor



Examples Using Monitor

- Solving Dining-Philosophers Problem with Monitor
  - A deadlock-free solution to the dining-philosophers problem is illustrated. The restriction is that a philosopher may pick up her chopsticks only if both of them are available. An enum data structure is designed to distinguish among three states in which we may find a philosopher:

```
enum {THINKING, HUNGRY, EATING} state[5];
```

  - A HUNGRY Philosopher i (i = 0 .. 4) can set her state to EATING

```
state[i] = EATING;
```

    only if her two neighbors are not eating:

```
(state[(i + 4) % 5] != EATING) &&
(state[(i + 1) % 5] != EATING)
```

  - We also need to declare

```
condition self[5];
```

    - This allows philosopher i to delay herself when she is HUNGRY but is unable to obtain the chopsticks she needs.

Examples Using Monitor

- Solving Dining-Philosophers Problem with Monitor

```
1.  monitor DiningPhilosophers
2.  {
3.      enum { THINKING; HUNGRY, EATING } state [5];
4.      condition self [5];

5.      void pickup(int i) {
6.          state[i] = HUNGRY;
7.          test(i);
8.          if (state[i] != EATING)
9.              self[i].wait();
10.     }

11.     void putdown(int i) {
12.         state[i] = THINKING;
13.         test((i + 4) % 5); /* test left neighbor */
14.         test((i + 1) % 5); /* test right neighbor */
15.     }
```

Examples Using Monitor

- Solving Dining-Philosophers Problem with Monitor

```
16.     void test(int i) {
17.         if ((state[(i + 4) % 5] != EATING) &&
18.                         (state[i] == HUNGRY) &&
19.               (state[(i + 1) % 5] != EATING) ) {
20.             state[i] = EATING;
21.             self[i].signal();
22.         }
23.     }

24.     initialization_code() {
25.         for (int i = 0; i < 5; i++)
26.             state[i] = THINKING;
27.     }
28. }
```

Examples Using Monitor

- Solving Dining-Philosophers Problem with Monitor
  - Example.
    - Let state[2] = EATING, state[3] = THINKING, state[4] = EATING
    - Concurrent process:

```
pickup(3); putdown(2); putdown(4);
```

    - One possible time Interleaving

Pro(2) #Line	state[2]	Pro(3) #Line	state[3]	Pro(4) #Line	state[4]
	EATING		THINKING		EATING
		6	HUNGRY		
		7	waiting		
12	THINKING				
13					
14					
	continuing			12	THINKING
			EATING	13	
			resuming	14	continuing

Examples Using Monitor

- Solving Dining-Philosophers Problem with Monitor
  - Each philosopher i invokes the operations pickup() and putdown() as follows:

```
DiningPhilosophers.pickup(i);
  /* eating */
DiningPhilosophers.putdown(i);
```

  - The solution ensures that no two neighbors are eating simultaneously .
  - This is a deadlock-free solution, but starvation is possible.

Monitor Implementation

- Monitor Implementation Using Semaphores
  - Shared data:
    - mutex: binary semaphore, initialized to 1.
      - provided for each monitor to ensure mutual exclusion.
    - next: binary semaphore, initialized to 0.
      - *Hoare* scheme is used by default. Processes can suspend themselves by using signal(next).
    - next_count: integer variable, initialized to 0.
      - provided to count the number of processes suspended on next.

Monitor Implementation

- Monitor Implementation Using Semaphores
  - Shared data:

```
semaphore mutex;  /* initially to 1 */
semaphore next;   /* initially to 0 */
int next_count = 0;
```

  - Each external procedure F will be replaced by

```
wait(mutex);
body of F
if (next_count > 0)
    signal(next);
  /* F gives up signaling mutex, but awakening one
     process suspended on next */
else
  /* no process suspended on next, F signals mutex */
    signal(mutex);
```

Monitor Implementation

- Monitor Implementation Using Semaphores
  - More shared data for each condition variable x:

```
semaphore x_sem;  /* initially to 0 */
int x_count = 0;  /* number of x-waiting processes */
```

  - Implementation of x.wait():

```
x_count++;
if (next_count > 0)
    signal(next);  /* wakeup a process in urgent queue */
else
    signal(mutex); /* open the entrance of the monitor */
wait(x_sem);       /* join in the condition x queue */
x_count--;       /* retreat from the condition x queue */
```

  - Implementation of x.signal():

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);  /* wakeup a process in condition x
                       queue */
    wait(next);     /* join in the urgent queue (Hoare)*/
    next_count--;   /* retreat from the urgent queue */
}                   /* otherwise x.signal does nothing */
```

Resuming Processes within a Monitor

- Resuming Processes within a Monitor
  - If several processes are suspended on condition x, and x.signal() is executed by some process, which of them should be resumed next?
  - Without the simple FCFS solution, the *conditional-wait* construct can be used, of the form

    ```
    x.wait(c);
    ```

    where c is called a *priority number*.
  - When a process invokes x.wait(c) and suspended, the priority number c is stored with the suspended process.
  - When x.signal() is executed, the process with the smallest priority number is resumed next.

Resuming Processes within a Monitor

- Example: A Monitor to Allocate Single Resource
  - 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.

Resuming Processes within a Monitor

- Example: 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;
    }
}
```

Resuming Processes within a Monitor

- Example: A Monitor to Allocate Single Resource
  - Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:
    - A process might access a resource without first gaining access permission to the resource.
    - A process might never release a resource once it has been granted access to the resource.
    - A process might attempt to release a resource that it never requested.
    - A process might request the same resource twice (without first releasing the resource).

System Model

- Resources and Instances
  - A system consists of a finite number of resources to be distributed among a number of competing processes.
  - The resources are partitioned into several types, each consisting of some number of identical instances.
    - Memory space
    - CPU cycles
    - Files
    - I/O devices: printers, DVD drives, …
  - If a process requests an instance of a resource type, the allocation of *any instance* of the type will satisfy the request.
  - A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.
  - The number of resources requested may not exceed the total number of resources available in the system.

System Model

- Sequence of Resource Utilization
  - *Request.* The process requests the resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
  - *Use.* The process can operate on the resource.
  - *Release.* The process releases the resource.
- Resource Management
  - A system table – It records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated.
  - A waiting queue - If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.
- Deadlock
  - A set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

Deadlock Characterization

- Necessary Conditions
  - A deadlock situation can arise if the following four conditions hold simultaneously in a system:

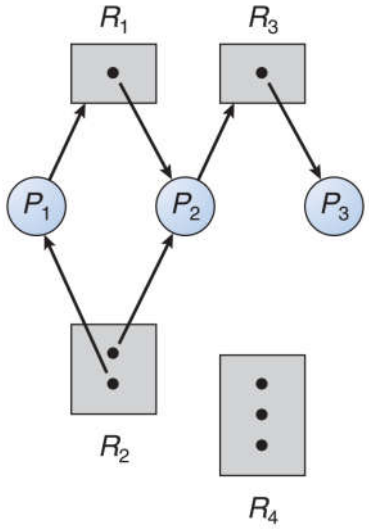
1. Mutual exclusion (互斥). At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. Hold and wait (保持等待). A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. No preemption (非抢占). Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. Circular wait (循环等待). A set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0. (It implies the hold-and-wait condition)

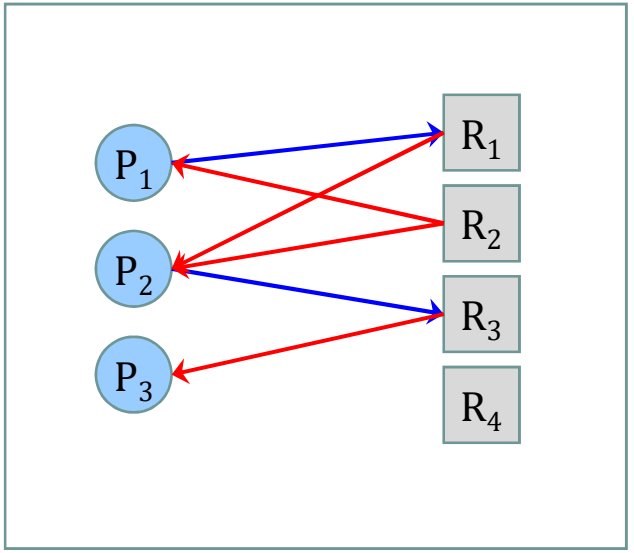
Deadlock Characterization

- Resource-Allocation Graph
  - Deadlocks can be described more precisely in terms of a directed bipartite graph called a *system resource-allocation graph*.
  - This graph consists of a set of vertices V and a set of edges E. The set V is partitioned into two different types of nodes:
    - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the active processes in the system, and
    - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
  - A directed edge from process P_i to resource type R_j is a *request edge*, denoted by $P_i \rightarrow R_j$; it signifies that process P_i has requested an instance of resource type R_j and is currently waiting for that resource.
  - A directed edge from resource type R_j to process P_i is an *assignment edge*, denoted by $R_j \rightarrow P_i$; it signifies that an instance of resource type R_j has been allocated to process P_i.

Deadlock Characterization

- Resource-Allocation Graph
  - Example of a resource-allocation graph
    - R_1 and R_3 both have only one instance.
    - R_2 has two instances.
    - R_4 has three instances.
    - P_1 is holding an instance of R_2.
    - P_1 is waiting an instance of R_1.
    - P_2 is holding an instance of R_1.
    - P_2 is holding an instance of R_2.
    - P_2 is waiting an instance of R_3.
    - P_3 is holding an instance of R_3.





Deadlock Characterization

- Resource-Allocation Graph
  - A cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.
    - Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle involving only a set of resource types, each of which has only *a single instance*, then a deadlock has occurred. Each process involved in the cycle is deadlocked.

Deadlock Characterization

Resource-Allocation Graph

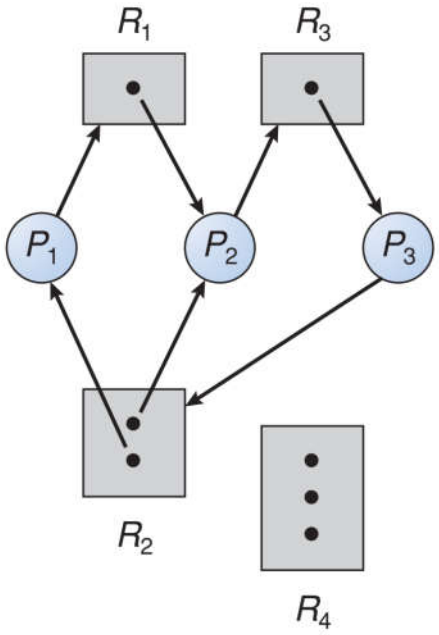
Example.

- A resource-allocation graph with a *deadlock*
  - Two minimal cycles exist in the system:

    $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

    $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

  - Processes P_1, P_2, and P_3 are deadlocked. Process P_2 is waiting for the resource R_3, which is held by process P_3. Process P_3 is waiting for either process P_1 or process P_2 to release resource R_2. In addition, process P_1 is waiting for process P_2 to release resource R_1.

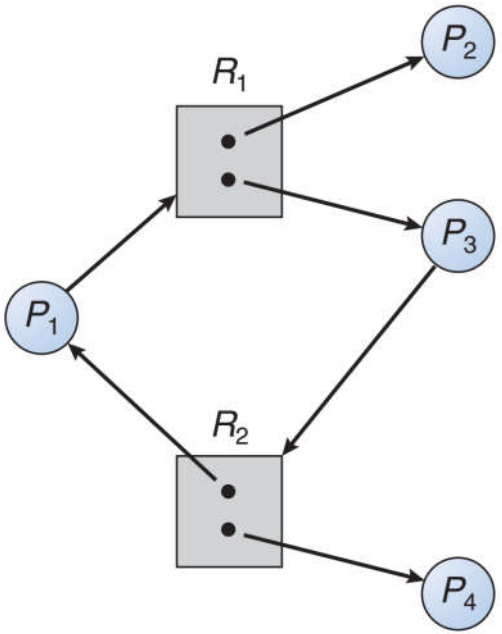


Deadlock Characterization

- Resource-Allocation Graph
  - Example.
    - A resource-allocation graph with a cycle but *no deadlock*
      - A cycles exists in the system:

        $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

      - There is no deadlock. Observe that process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle. Same thing happens when process P_2 releases its instance of resource type R_1.



Deadlock Characterization

- Methods for Handling Deadlocks
  - The deadlock problem may be dealt with in one of three ways:
    - M1.
      - Use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlocked state.
    - M2.
      - Allow the system to enter a deadlocked state, detect it, and recover.
    - M3.
      - Ignore the problem altogether and pretend that deadlocks never occur in the system.

Deadlock Characterization

- Methods for Handling Deadlocks
  - M1. Preventing and Avoiding
    - To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock avoidance scheme.
    - *Deadlock prevention* provides a set of methods for ensuring that at least one of the four necessary conditions cannot hold.
      - These methods prevent deadlocks by constraining how requests for resources can be made.
    - *Deadlock avoidance* requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime.

Deadlock Characterization

- Methods for Handling Deadlocks
  - M2. Detecting and Recovering
    - If a system does not employ either a deadlock prevention or a deadlock avoidance algorithm, then a deadlock situation may arise.
    - In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the occurring deadlock.

Deadlock Characterization

- Methods for Handling Deadlocks
  - M3. Ignoring
    - Although ignoring deadlocks may not seem to be a viable approach to the deadlock problem, it is nevertheless used in most operating systems.
    - In many systems, deadlocks occur infrequently (say, once per year); thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods, which must be used constantly.
    - A system may be in a frozen state but not in a deadlocked state.
      - e.g., a real-time process running at the highest priority, or any process running on a non-preemptive scheduler.
      - A system in frozen will not return control to the operating system. It must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

The Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type. the *Banker's Algorithm* is a *deadlock-avoidance* algorithm applicable to such a system but is less efficient than the resource-allocation graph scheme.
- When a new thread enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the thread must wait until some other thread releases enough resources.

The Banker's Algorithm

- Global Data Structures
  - Suppose there are *m* types of resources in the system and *n* processes requiring resources.

```
int available[m];
int work[m]; /* pretend available resources */
```

  - The vector `available` indicates the number of available resources of each type. `available[j] == k` means *k* instances of resource type R_j are available.

```
int max[n][m];
```

  - The matrix `max` defines the maximum demand of each thread. `max[i][j] == k` means thread T_i may request *at most k* instances of resource type R_j.

```
int allocation[n][m] = {0};
```

  - The matrix `allocation` defines the number of resources of each type currently allocated to each thread. `allocation[i][j] == k` means thread T_i is currently allocated *k* instances of resource R_j.

The Banker's Algorithm

- Global Data Structures
  - Suppose there are *m* types of resources in the system and *n* processes requiring resources.

```
int need[n][m] = {0};
```

  - The matrix `need` indicates the remaining resource need of each thread. `need[i][j] == k` means thread T_i may need *k* more instances of resource type R_j to complete its task.
  - Note that

```
need[i][j] == max[i][j] - allocation[i][j].
```

- Safe State and Unsafe State
  - The state of the system reflects the current allocation of resources to processes. Thus, the state consists of the vector `available`, and the two matrices, `need` and `allocation`. A *safe state* is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock (i.e., all of the processes can be run to completion).

The Banker's Algorithm

- Definition
  - Let X and Y be two vectors of length m.
    - X ≤ Y if and only if X[i] ≤ Y[i] for all i = 1, 2, ..., m.
    - In addition, Y < X if Y ≤ X and Y ≠ X.
    - We need $O(m)$ to decide whether X ≤ Y.
- Notation
  - For a matrix M[n][m], we use M[i] to denote the i-th row vector of matrix M.

The Banker's Algorithm

- Safety Algorithm – safetycheck().
  - Data Structures

```
#define TRUE 0
#define FALSE -1
int finish[n] = {FALSE};
int safe_seq[n];
```

  - Procedure:

```
for (i = 1; i <= n; i++) { /* we need to label n processes */
    for (j = 0; j < n; j++) { /* select one process */
        if (finish[j] == FALSE) { /* Tj has not been tested */
            if (need[j] <= work) { /* Tj can finish its work */
                work = work + allocation[j];
                    /* after Tj released its resource(s) */
                finish[j] = TRUE; /* Tj is tested */
                break;
            } /* else select the next process to test */
        }
    }
}
if (i > n)
    return EXIT_SECCESS;
else
    return EXIT_FAILURE;
```

$O(m)$ is needed to decide whether `need[j] <= work`
The complexity of safetycheck() is $O(n^2 m)$

The Banker's Algorithm

- Resource-Request Algorithm.
  - Data Structures

```
int request[m]; /* request for resources by thread Ti */
```

  - Procedure:

```
input request from process Ti
if (request > need)
    exit(1); /* Ti has exceeded its maximum claim */
if (request > available)
    exit(1); /* resources are unavailable */

  /* pretending allocation */
work = available - request;
allocation[i] = allocation[i] + request;
need[i] = need[i] - request;

ret = safetycheck();
return ret;
  /* roll back the allocation if EXIT_FAILURE */
```

- Examples.
  - Check the textbooks of *A. Silberschatz* and *W. Stallings*.

The Banker's Algorithm

- Limitations of Deadlock Avoidance
  - The Banker's Algorithm needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making it impossible to implement the Banker's algorithm. Also, it is unrealistic to assume that the number of processes is static since in most systems the number of processes varies dynamically. Moreover, the requirement that a process will eventually release all its resources (when the process terminates) is sufficient for the correctness of the algorithm, however it is not sufficient for a practical system. Waiting for hours (or even days) for resources to be released is usually not acceptable.

The Banker's Algorithm

- Limitations of Deadlock Avoidance
  - The deadlock avoidance strategy does not predict deadlock with certainty; it merely anticipates the possibility of deadlock and assures that there is never such a possibility.
  - Deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes, as in deadlock detection, and is less restrictive than deadlock prevention. However, it does have a number of restrictions on its use:
    - The maximum resource requirement for each process must be stated in advance.
    - The processes under consideration must be independent; that is, the order in which they execute must be unconstrained by any synchronization requirements.
    - There must be a fixed number of resources to allocate.
    - No process may exit while holding resources.

The Banker's Algorithm

- alg.17-1-bankers-5.c