Operating Systems: Internals and Design Principles

Chapter 6 Concurrency: Deadlock and Starvation

Seventh Edition
By William Stallings

Operating Systems: Internals and Design Principles

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone. Statute passed by the Kansas State Legislature, early in the 20th century.

—*A TREASURY OF RAILROAD FOLKLORE,*
B. A. Botkin and Alvin F. Harlow

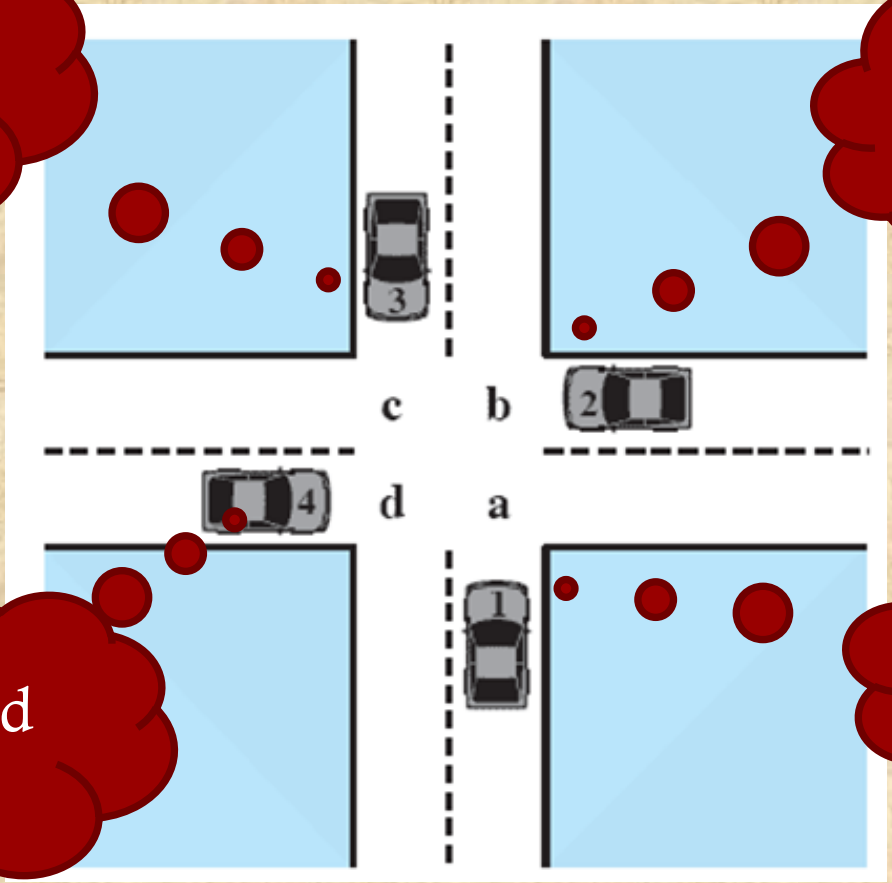
Deadlock

- The permanent blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent
- No efficient solution

Potential Deadlock

I need quad C and D

I need quad B and C



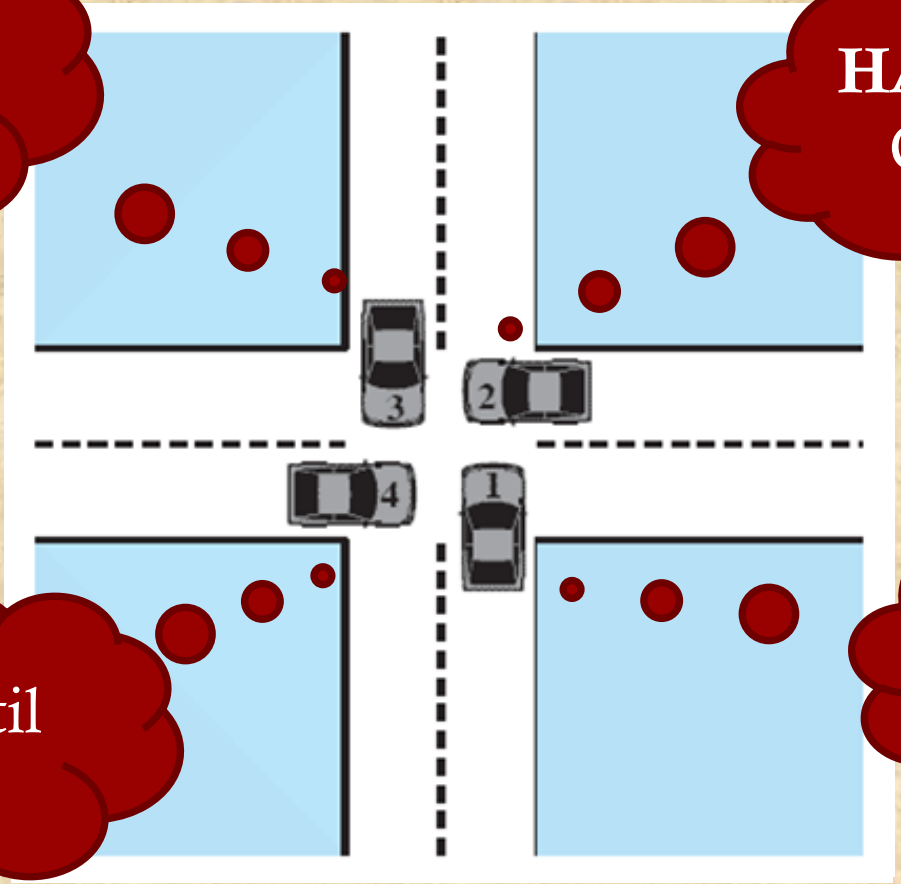
I need quad D and A

I need quad A and B

Actual Deadlock

HALT until
D is free

HALT until
C is free



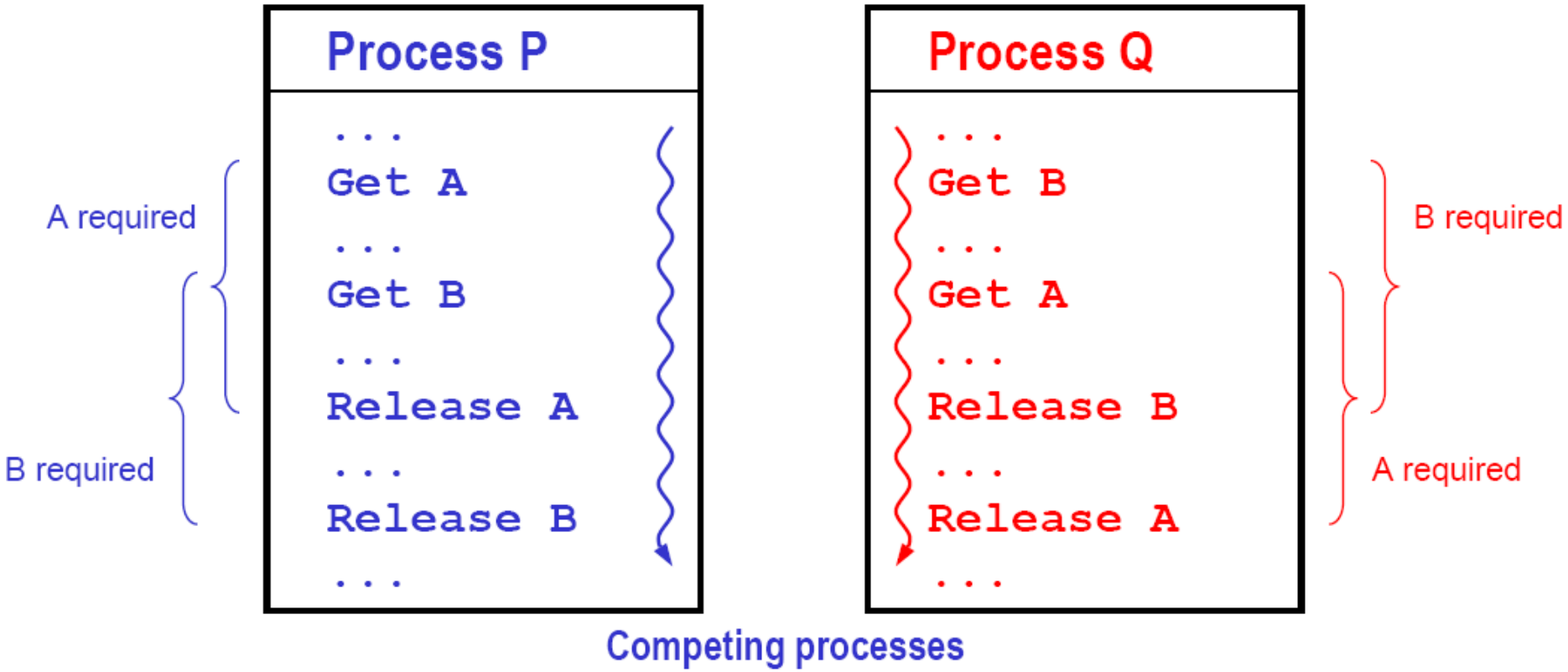
HALT until
A is free

HALT until
B is free

Deadlocks

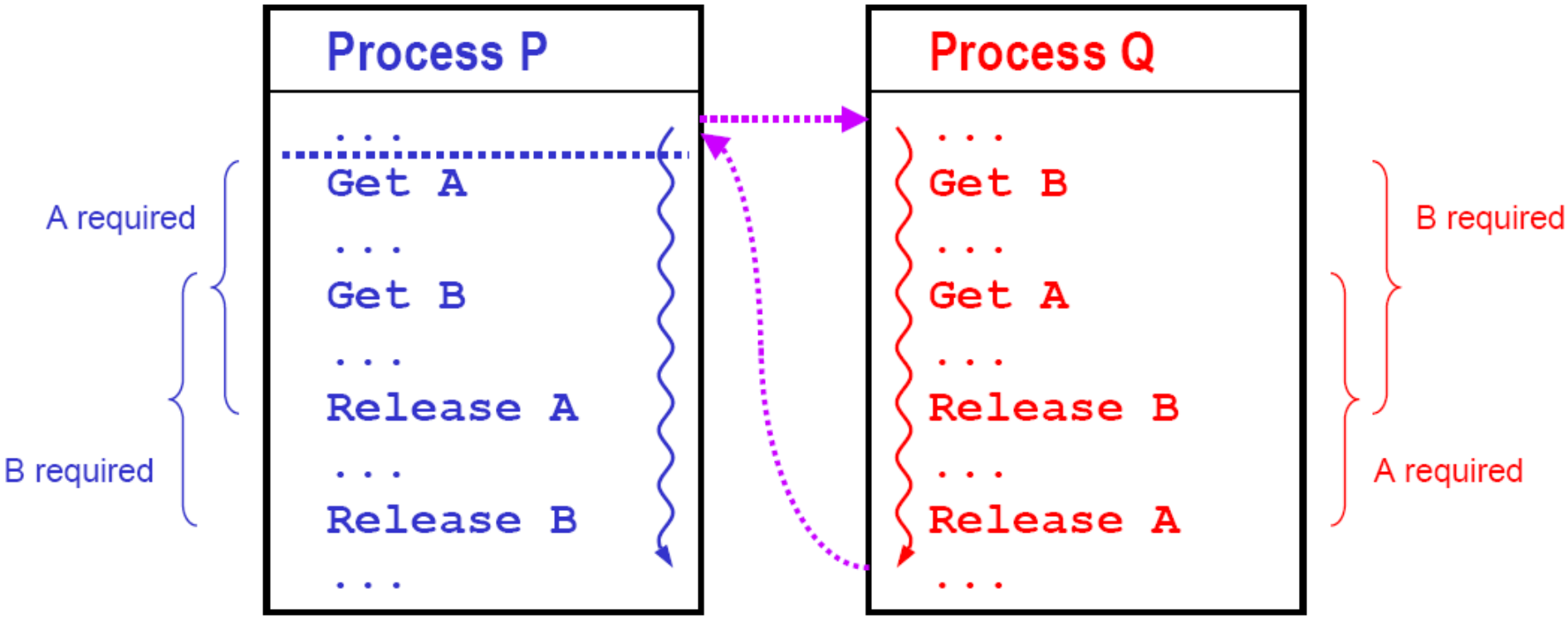
Illustration of a deadlock

- ✓ two processes, P and Q, compete for two resources, A and B
- ✓ each process needs exclusive use of each resource



Deadlocks

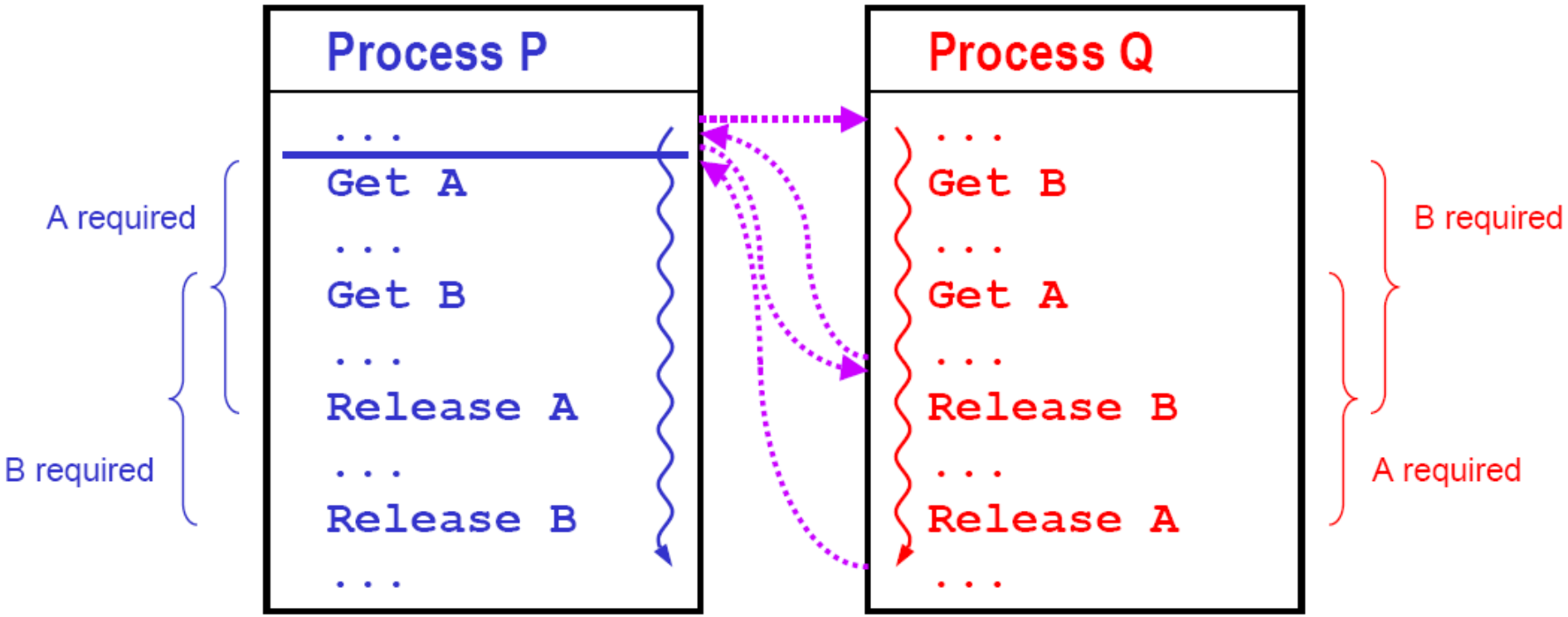
- **Illustration of a deadlock — scheduling path 1 ☺**
  - ✓ Q executes everything before P can ever get A
  - ✓ when P is ready, resources A and B are free and P can proceed



Happy scheduling 1

Deadlocks

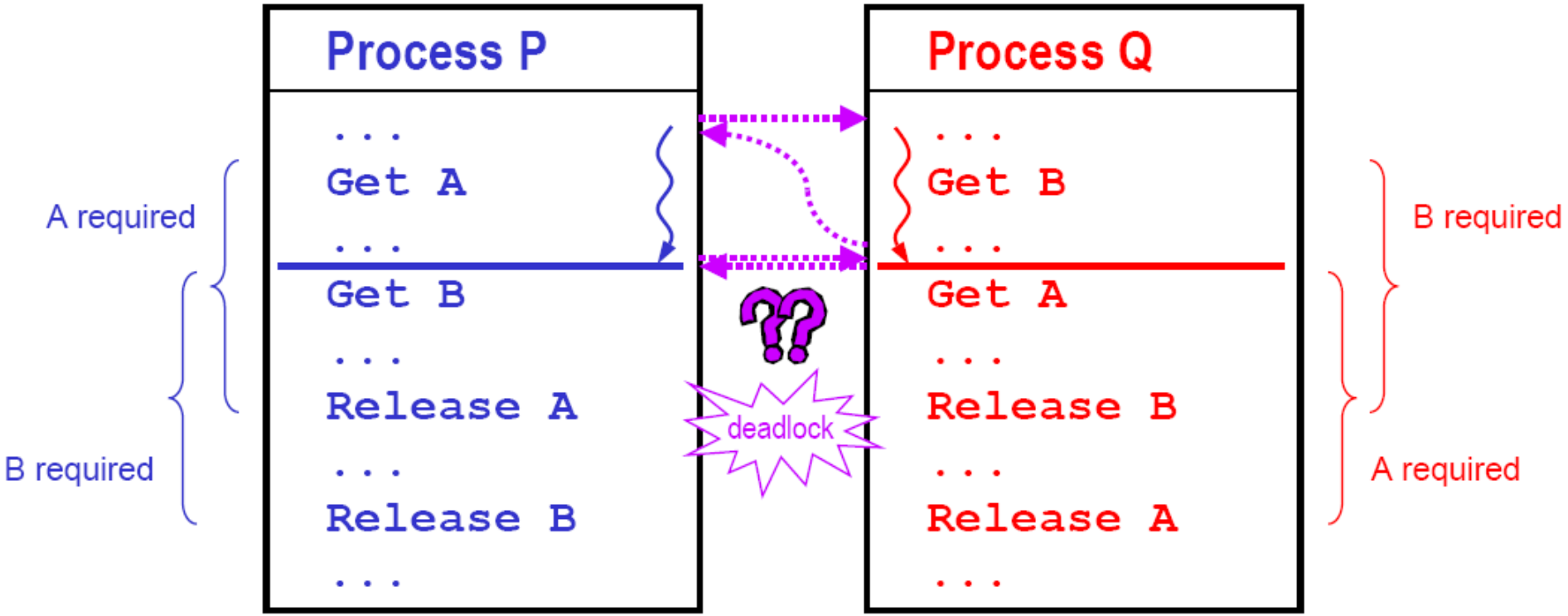
- **Illustration of a deadlock — scheduling path 2** ☺
  - ✓ Q gets B and A, then P is scheduled; P wants A but is blocked by A's mutex; so Q resumes and releases B and A; P can now go



Happy scheduling 2

Deadlocks

- **Illustration of a deadlock — scheduling path 3** ☹
  - ✓ Q gets only B, then P is scheduled and gets A; now both P and Q are blocked, each waiting for the other to release a resource



Bad scheduling → deadlock

Joint Progress Diagram

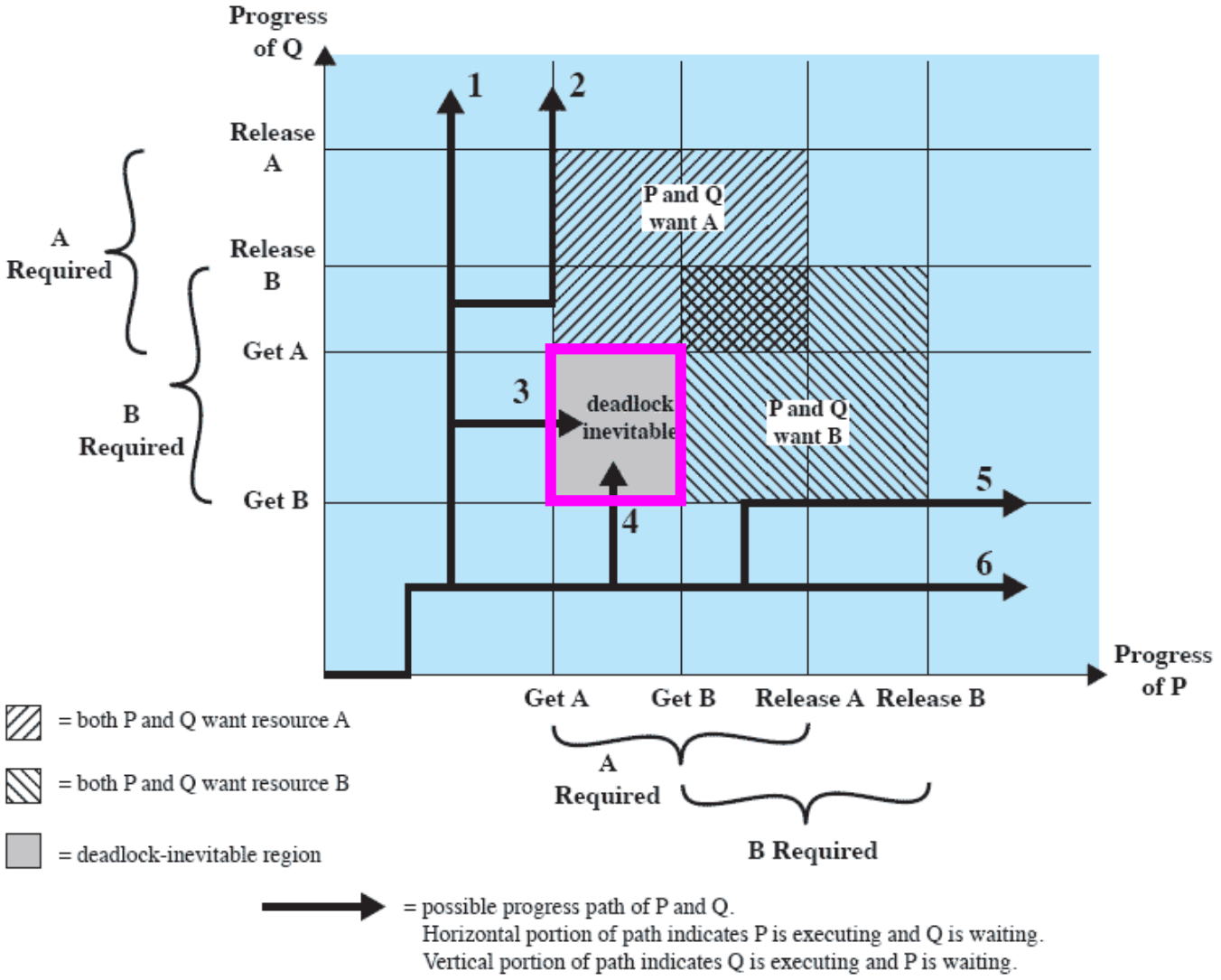
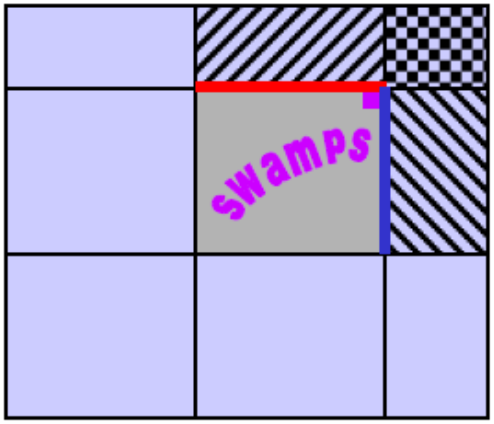


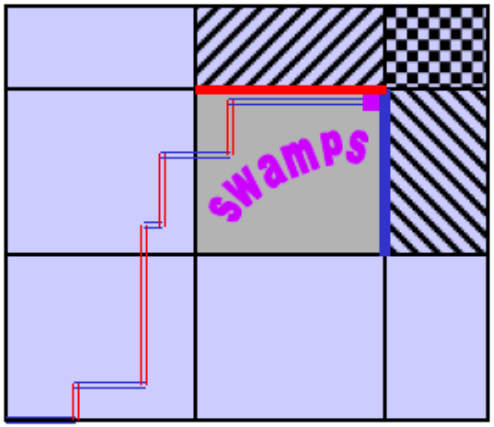
Figure 6.2 Example of Deadlock

Deadlocks

- **Deadlocks depend on the program and the scheduling**
  - ✓ **program design**
    - the order of the statements in the code creates the "landscape" of the joint progress diagram
    - this landscape may contain gray "swamp" areas leading to deadlock

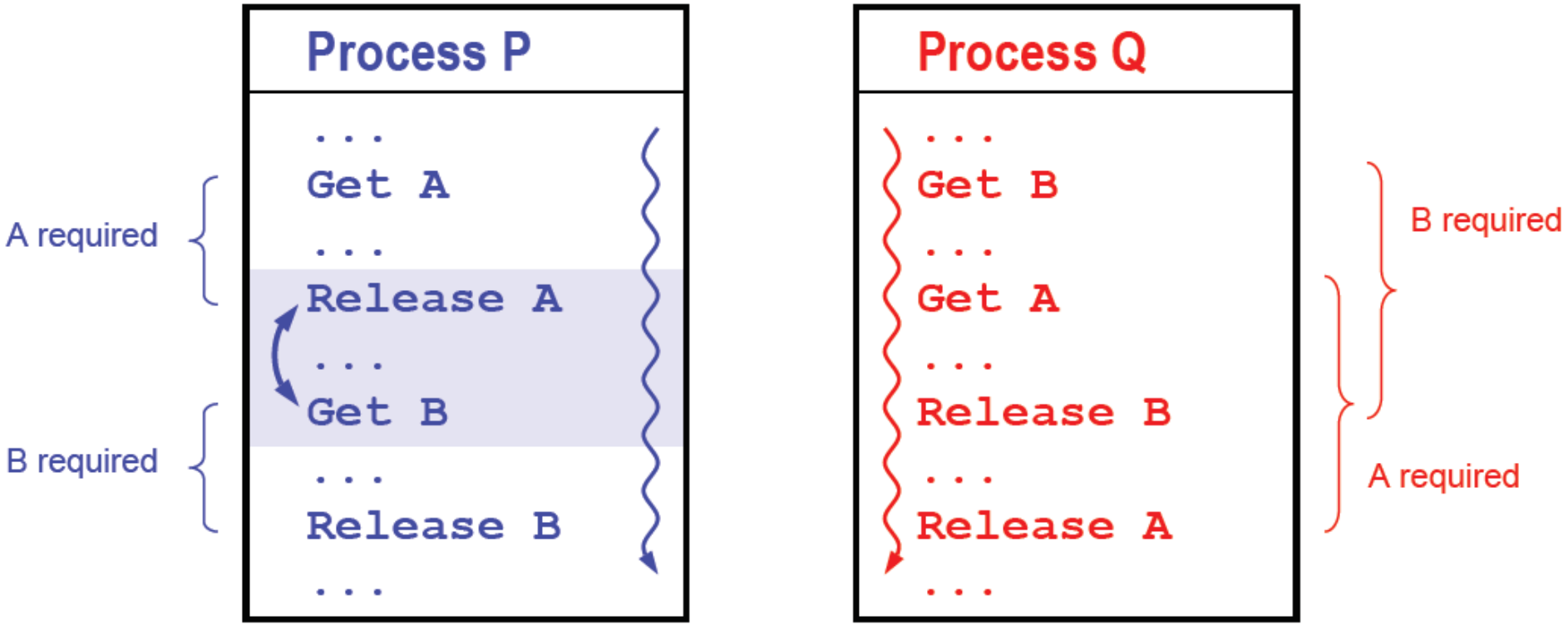


  - ✓ **scheduling condition**
    - the interleaved dynamics of multiple executions traces a "path" in this landscape
    - this path may sink in the swamps

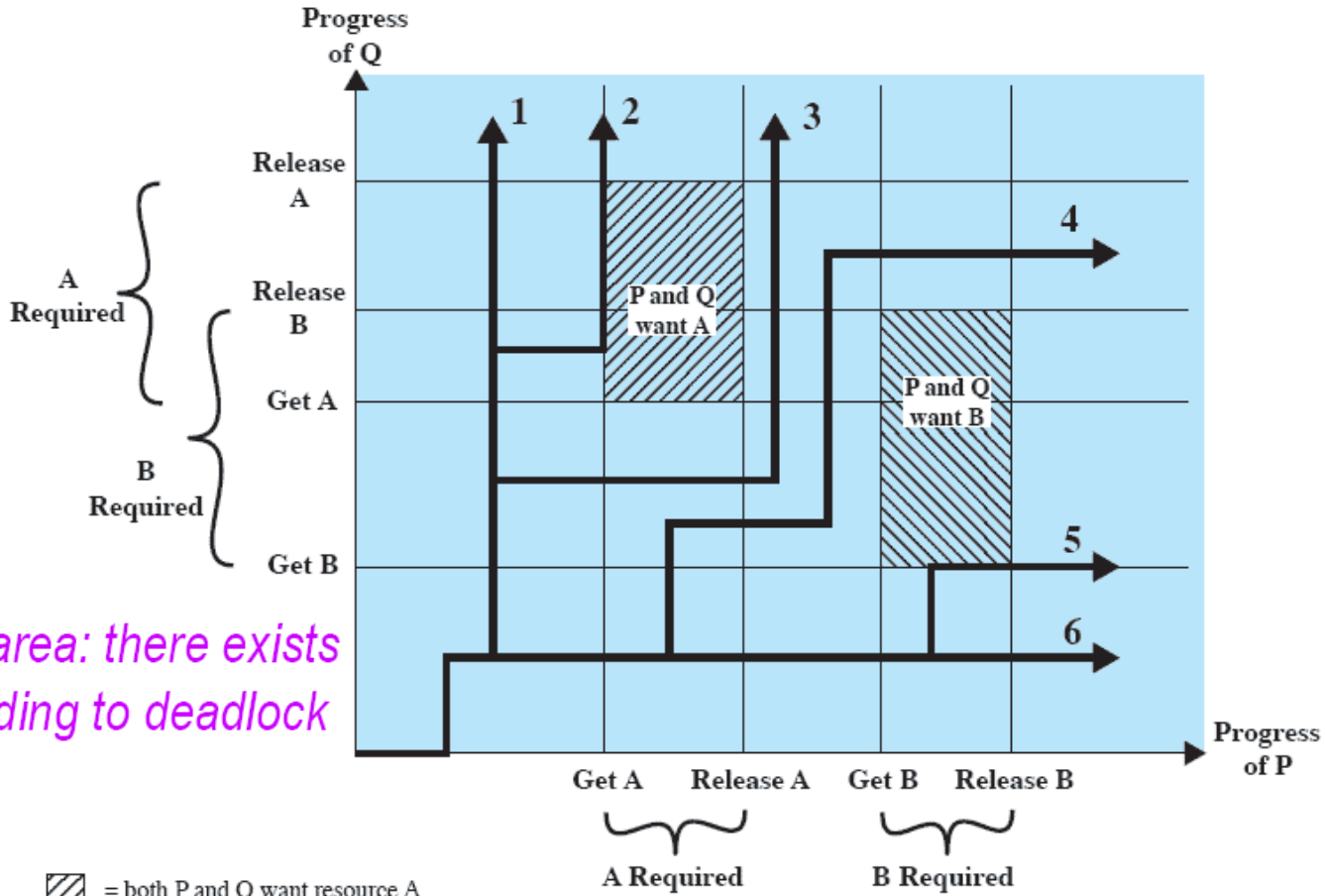


Deadlocks

- **Changing the program changes the landscape**
  - ✓ here, P releases A before getting B
  - ✓ deadlocks between P and Q are not possible anymore



No Deadlock Example



→ *no swamp area: there exists no path leading to deadlock*

Figure 6.3 Example of No Deadlock [BACO03]

Resource Categories

Reusable

- can be safely used by only one process at a time and is not depleted by that use
  - processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock may occur if each process holds one resource and requests the other

Consumable

- one that can be created (produced) and destroyed (consumed)
  - interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive message is blocking
- May take a rare combination of events to cause deadlock

Reusable Resources Example

- D enforces mutual exclusion on a disk file D
- T enforces mutual exclusion on a tape drive T

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4
Example of Two Processes Competing for Reusable Resources

Reusable Resources Example

- D enforces mutual exclusion on a disk file D
- T enforces mutual exclusion on a tape drive T

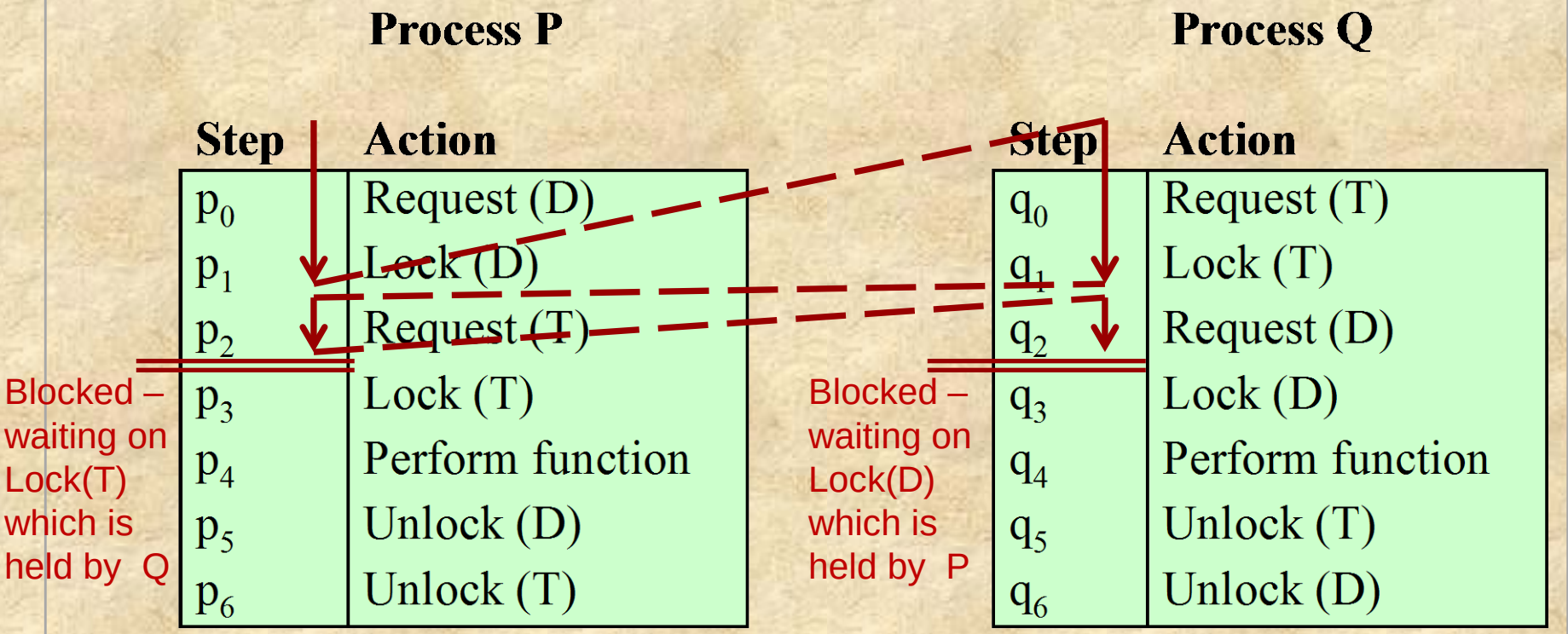


Figure 6.4
Example of Two Processes Competing for Reusable Resources

Example 2: Memory Request

- Space is available for allocation of 200Kbytes, and the following sequence of events occur:

P1	P2
. . . **Request 80 Kbytes;** . . . **Request 60 Kbytes;**	. . . **Request 70 Kbytes;** . . . **Request 80 Kbytes;**

- Deadlock occurs if both processes progress to their second request

Consumable Resources Deadlock

- Consider a pair of processes, in which each process attempts to receive a message from the other process and then send a message to the other process:

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

- Deadlock occurs if the Receive is blocking

Deadlock Detection, Prevention, and Avoidance

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Resource Allocation Graphs

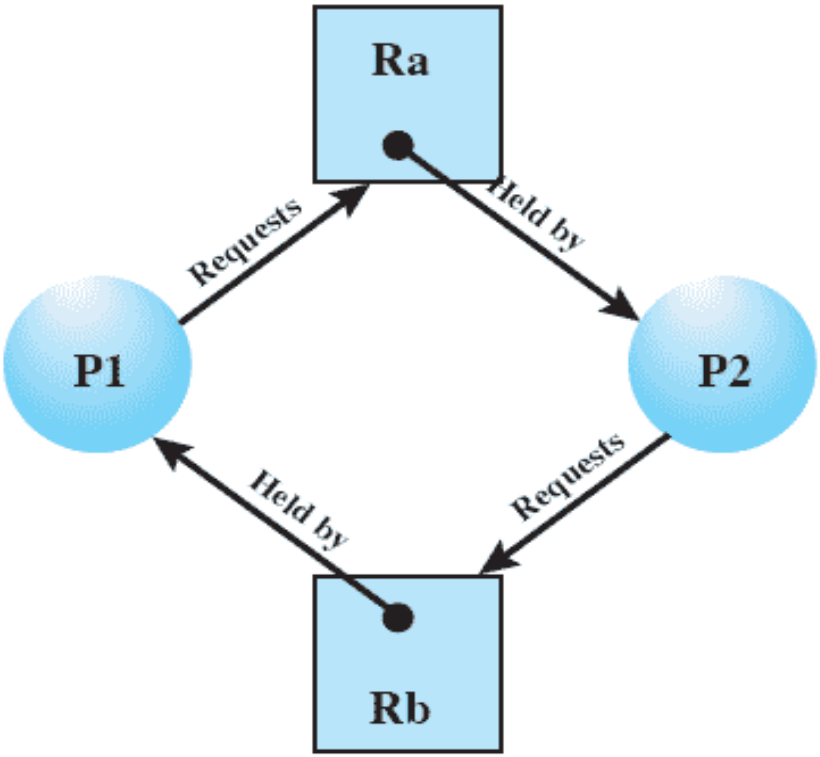
Deadlocks

- **Design conditions for deadlock (create the swamps)**
  1. **mutual exclusion** — the design contains protected critical regions; only one process at a time may use these
  2. **hold & wait** — the design is such that, while inside a critical region, a process may have to wait for another critical region
  3. **no resource preemption** — there must not be any hardware or O/S mechanism forcibly removing a process from its CR

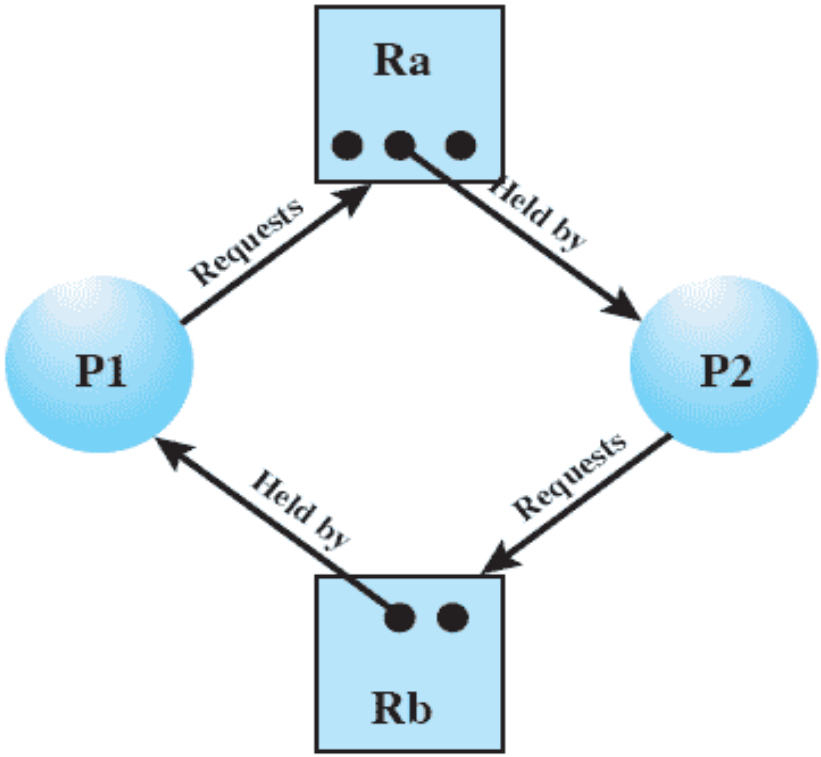
+ **Scheduling condition for deadlock (go to the swamps)**
  4. **circular wait** — two or more hold-&-wait's are happening in a circle: each process holds a resource needed by the next

= **Deadlock!**

Resource Allocation Graphs



(c) Circular wait



(d) No deadlock

Resource Allocation Graphs

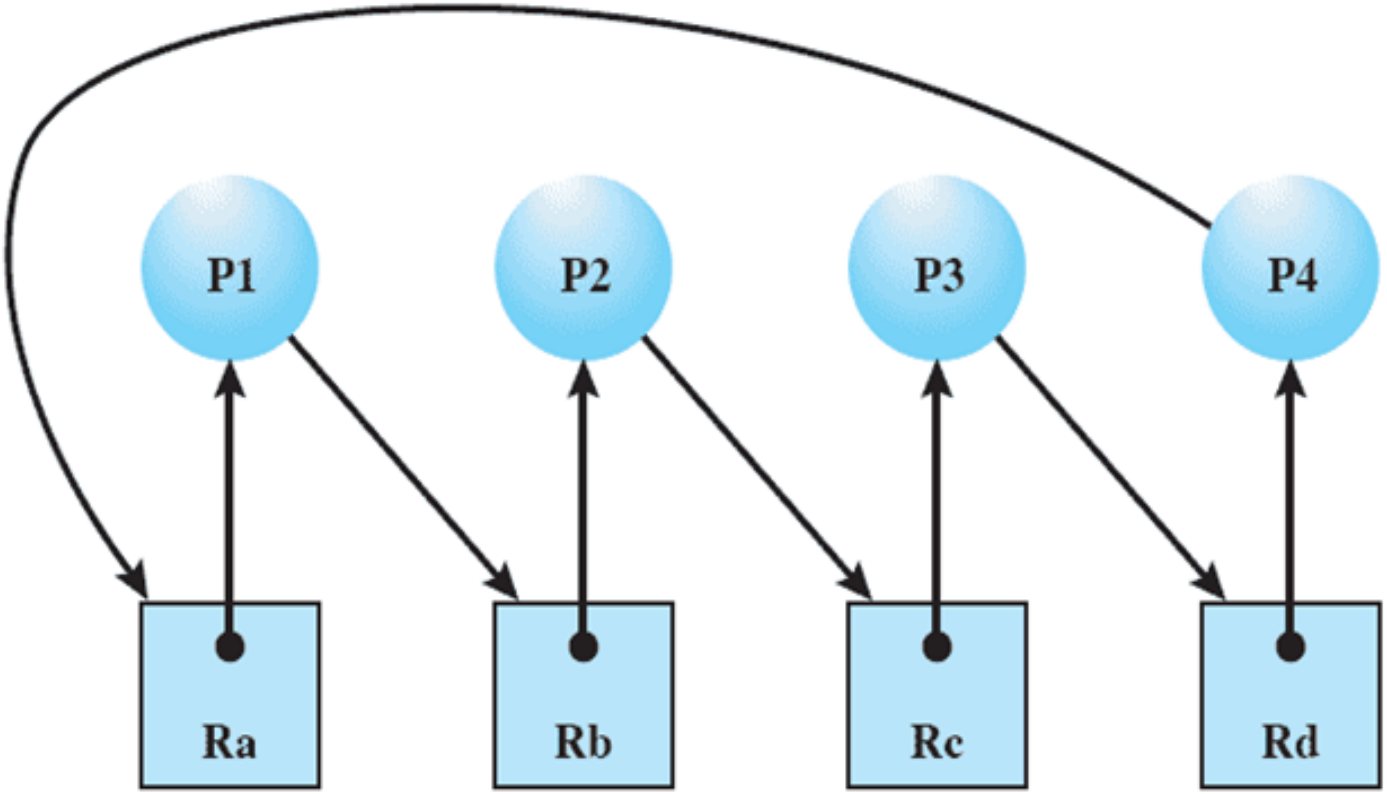
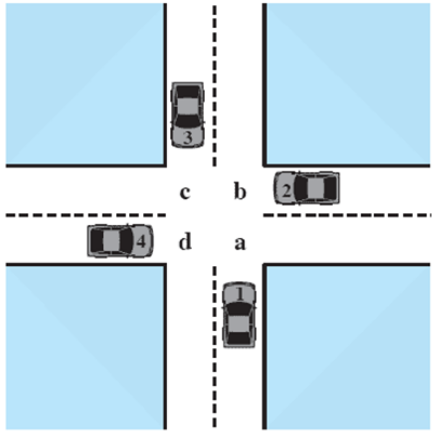


Figure 6.6 Resource Allocation Graph for Figure 6.1b



Possibility of Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it

Existence of Deadlock

Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
• only one process may use a resource at a time	• a process may hold allocated resources while awaiting assignment of others	• no resource can be forcibly removed from a process holding it	• a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Dealing with Deadlock

- Three general approaches exist for dealing with deadlock:

Prevent Deadlock – changing the rules

- adopt a policy that eliminates one of the conditions

Avoid Deadlock – optimizing the allocation

- make the appropriate dynamic choices based on the current state of resource allocation

Detect Deadlock – recovering after the fact

- attempt to detect the presence of deadlock and take action to recover

Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
  - Indirect
    - prevent the occurrence of one of the three necessary conditions
  - Direct
    - prevent the occurrence of a circular wait

Deadlock Condition Prevention

Mutual Exclusion

if access to a resource requires mutual exclusion then it must be supported by the OS

Not possible to disallow mutual exclusion

Deadlock Condition Prevention

Inefficient and impractical:

- defeats interleaving,
- creates long waits,
- cannot predict all resource needs

Hold and Wait

require that a process request **all** of its required resources at one time and blocking the process until all requests can be granted simultaneously

Deadlock Condition Prevention

No Preemption

Require that a process releases and requests again

if a process holding certain resources is denied a further request, that process must release its original resources and request them again

OS may preempt the second process and require it to release its resources

→ *ok*

Deadlock Condition Prevention

Inefficient again impractical:

- defeats interleaving,
- creates long waits,
- can deny resource access unnecessarily

Circular Wait

Define a linear ordering of resources and require that a process request **all** of its required resources in that specified order

Deadlock Prevention

- Deny circular wait
  - Associate an index with each resource so that a total ordering is imposed on the resources
  - Resource R_i preceeds R_j if $i < j$
  - Processes / threads must request resources in order
  - Deadlock could only happen if:
    - P_0 holds R_i and requests R_j
    - P_1 holds R_j and requests R_i
    - Implies that $i < j$, and $j < i$.
  - Deadlock is not possible

Deadlock Avoidance

- **Allow all conditions, but allocate wisely**
  - ✓ given a resource allocation request, a decision is made dynamically whether granting this request can potentially lead to a deadlock or not
    - do not start a process if its demands might lead to deadlock
    - do not grant an incremental resource request to a running process if this allocation might lead to deadlock

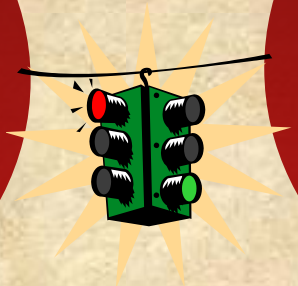
  - ✓ avoidance strategies requires knowledge of future process request (calculating "chess moves" ahead)

Two Approaches to Deadlock Avoidance

Deadlock Avoidance

Resource Allocation Denial

- do not grant an incremental resource request to a process if this allocation might lead to deadlock



Process Initiation Denial

- do not start a process if its demands might lead to deadlock

Deadlock Avoidance

Resources = **R** = $(R_1, R_2, \ldots, R_n)$	Total amount of each resource in the system
Available = **V** = $(V_1, V_2, \ldots, V_n)$	Total amount of each resource not allocated to any process
Claim = **C** = $\begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1n} \\ C_{21} & C_{22} & \cdots & C_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nn} \end{bmatrix}$	C_{ij} = requirement of process i for resource j
Allocation = A = $\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{bmatrix}$	A_{ij} = current allocation to process i of resource j

Deadlock Avoidance

- Process Initiation Denial

$$R_j = V_j + \sum_{i=1}^{n} A_{ij} \quad \text{for all } j$$

$$C_{ij} \le R_j \quad \text{for all } i, j$$

$$A_{ij} \le C_{ij} \quad \text{for all } i, j$$

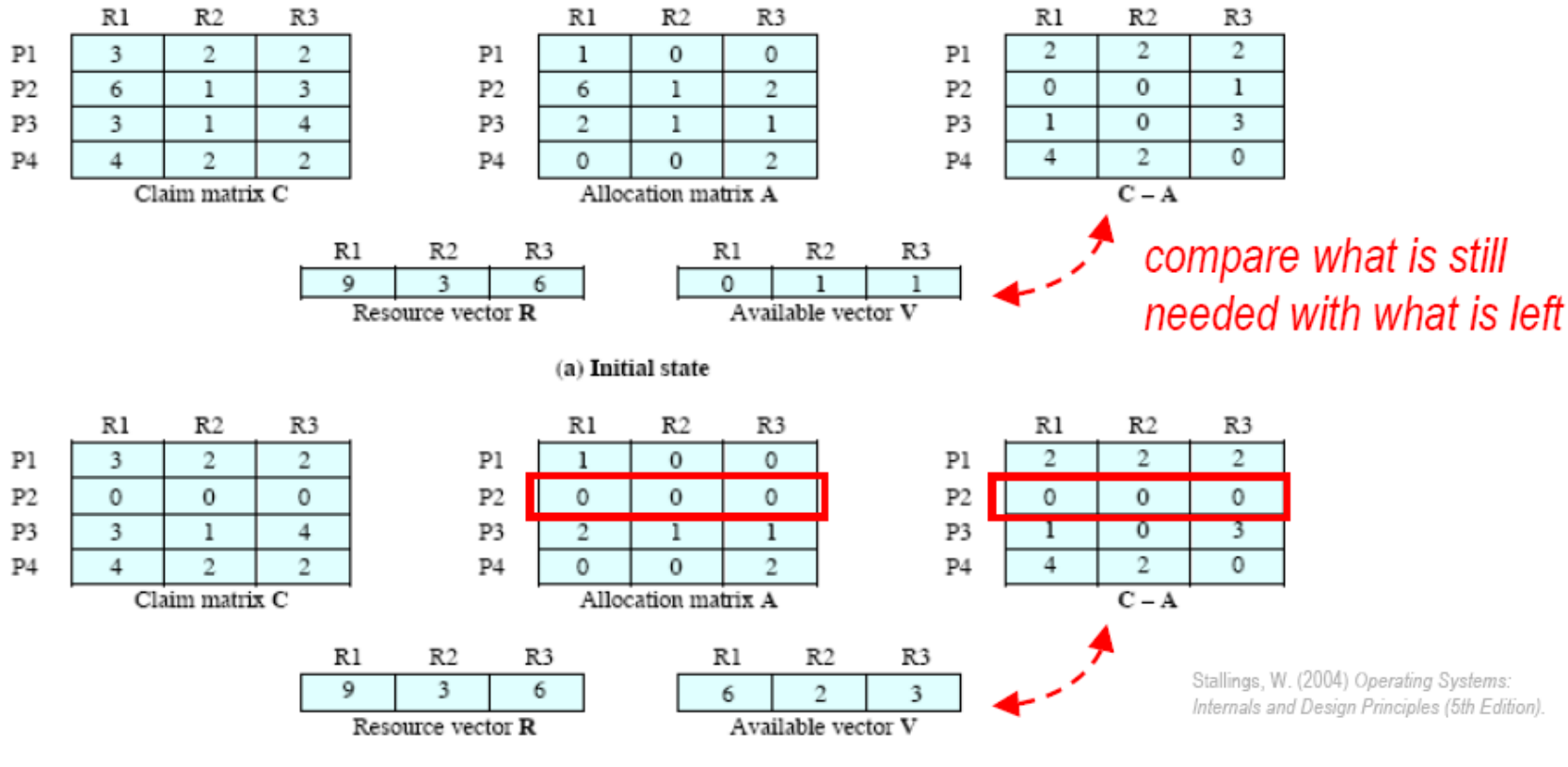
  - Refuse to start a new process if its resource requirements might lead to deadlock
  - Start a new process only if: $R_j \ge C_{(n+1)j} + \sum_{i=1}^{n} C_{ij} \quad \text{for all } j$

Resource Allocation Denial

- Referred to as the ***banker's algorithm***
- ***State*** of the system reflects the current allocation of resources to processes
- ***Safe state*** is one in which there is at least **one sequence of resource allocations to processes** that does **not** result in a deadlock
- ***Unsafe state*** is a state that is not safe
  - ✓ Analogy = banker refusing to grant a loan if funds are too low to grant more loans + uncertainty about how long a customer will take to repay

Determination of a Safe State

✓ can a process run to completion with the available resources?



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Determination of a safe state

Determination of a Safe State

- ✓ idea: refuse to allocate if it may result in deadlock

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

(c)

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d)

(d) P3 runs to completion

all could run to completion:
→ thus, (a) was a safe state

Determination of a safe state (cont'd)

Determination of a Safe State

P4 Now Runs to Completion

Thus, the state defined originally is a **safe state**

Determination of an Unsafe State

- ✓ idea: refuse to allocate if it may result in deadlock

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

(a) Initial state

(a) safe ← (a')

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b') unsafe

(b) P1 requests one unit each of R1 and R3

potential for deadlock (we don't know how long Ri will be kept)

→ thus, (b') is an unsafe state: don't allow (b') to

Determination of an unsafe state

Deadlock Avoidance Logic

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                           /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

Request of process i exceeds its claim

Request of process i exceeds availability

Assume request is satisfied and update allocation and availability vectors

Check if state is safe or unsafe

(b) resource alloc algorithm

Deadlock Avoidance Logic

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                          /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

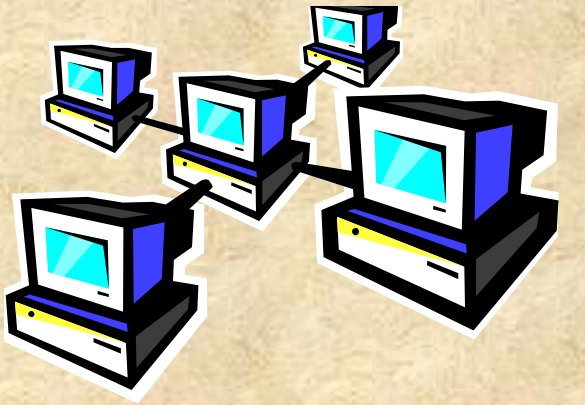
Figure 6.9 Deadlock Avoidance Logic

Banker's Algorithm

- Concept: ensure that the system of processes and resources is **always in a safe state**
- Mechanism: when a process makes a request for a set of resources
  - **Assume** that the request is granted
  - Update the system state accordingly
  - Determine if the result is a safe state
  - If so, grant he request; if not, block the process until it is safe to grant the request

Deadlock Avoidance Advantages

- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention



Deadlock Avoidance Restrictions

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

Deadlock Strategies

Deadlock prevention strategies are very conservative

- limit access to resources by imposing restrictions on processes

Deadlock **detection** strategies do the opposite

- resource requests are granted whenever possible

Deadlock Detection Algorithms

- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur

Advantages:

- it leads to early detection
- the algorithm is relatively simple

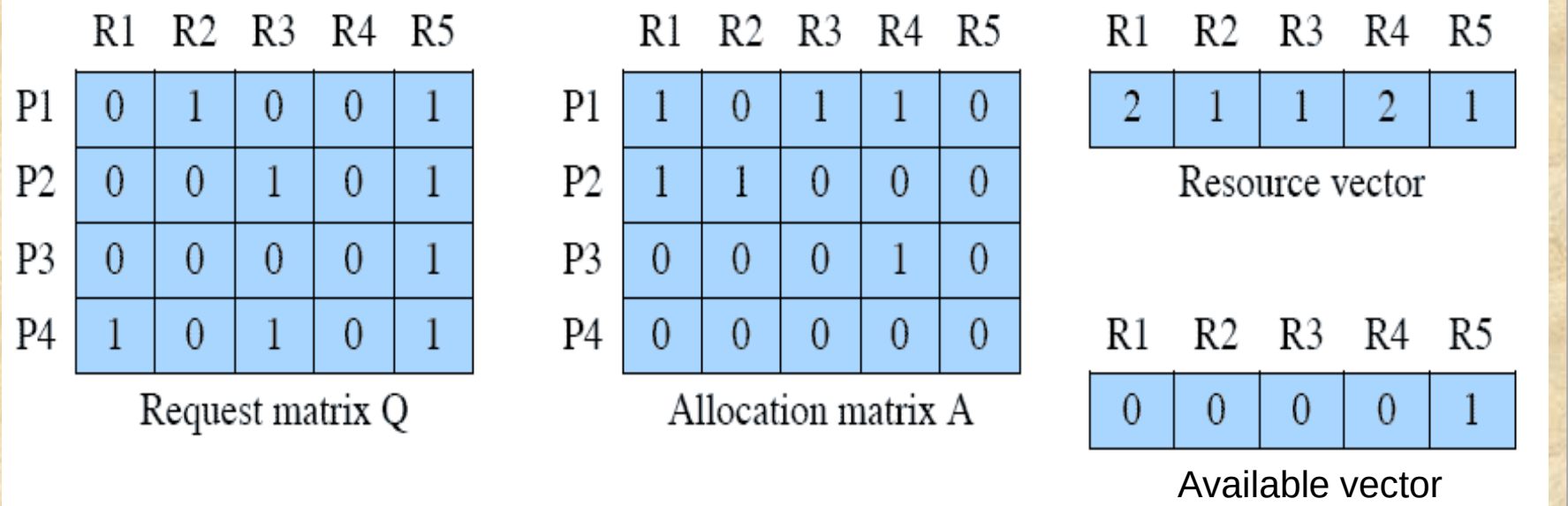
Disadvantage

- frequent checks consume considerable processor time

Deadlock Detection

- **Step 1:** Mark each process that has a row in the Allocation matrix of all 0s
- **Step 2:** Initialize temporary vector W to be equal to the Available vector
- **Step 3:** Find index i such that process i is currently unmarked and the ith row of Q is less than or equal to W. If no such row is found, then terminate.
- **Step 4:** for each row found in step 3, mark process i and add the corresponding row of the allocation matrix to W.
- **At the end, unmarked processes are deadlocked**

Deadlock Detection Algorithm



	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Figure 6.10 Example for Deadlock Detection

Mark P4

Set W = (0 0 0 0 1) the available vector

Select P3, row 3 in Q is less or equal to W

Mark P3

Original W A_3 New W

Set W = (0 0 0 0 1) + (0 0 0 1 0) = (0 0 0 1 1)

No more rows exist that satisfy step 3

Terminate

Processes P1 and P2 are not marked

Processes P1 and P2 are deadlocked

Recovery Strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- **Successively** abort deadlocked processes until deadlock no longer exists
- **Successively** preempt resources/processes until deadlock no longer exists

"**Successively**" means an order is followed: least amount of CPU time consumed, lowest priority, least total resources allocated so far, etc.

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

Deadlock Approaches

Integrated Deadlock Strategy

- Group resources into a number of different resource classes
- Use the linear ordering strategy (defined previously for prevention of circular wait) to prevent deadlocks between resource classes
- Within a given resource class, use the algorithm most appropriate for that class

Integrated Deadlock Strategy

- Example
  - Classes of resources
    1. Swappable space: blocks of memory on secondary storage for use in swapping processes
    2. Process resources: assignable devices, such as tape drives and files
    3. Main memory: assignable to processes in pages or segments
    4. Internal resources: such as I/O channels

Integrated Deadlock Strategy

Inside Each Resource Class

- Swappable Space
  - Require that all resources that may be used must be allocated at the same time (eliminates the hold and wait condition)
  - Reasonable if maximum storage requirements are known in advance (often the case)

Integrated Deadlock Strategy

Inside Each Resource Class

- Process Resources
  - Deadlock avoidance
    - Reasonable to expect that processes will declare in advance the resources they will require (such as files to open, tape drives to use, etc.)
  - Deadlock preventions could also be used in this class

Integrated Deadlock Strategy

Inside Each Resource Class

- Main Memory
  - Prevention by Preemption
    - Most effective strategy
    - When a process is preempted (i.e., it moves from running to ready, or running to blocked), swap the process to secondary storage (the swap space) to free the memory for use by another process (thus resolving any potential deadlocks)

Integrated Deadlock Strategy

Inside Each Resource Class

- Internal Resources
  - Resource Ordering (deny circular wait condition)

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time (mutual exclusion)
- No philosopher must starve to death (avoid deadlock and starvation)

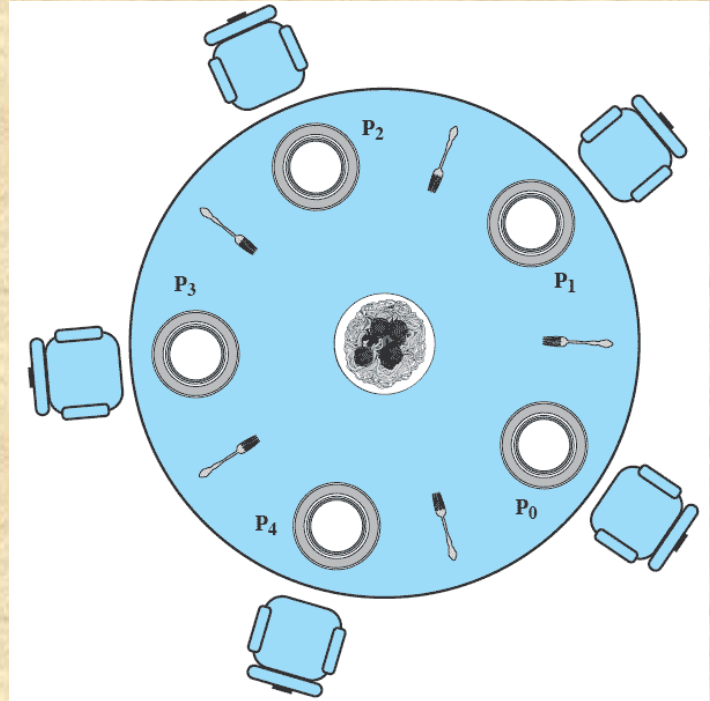


Figure 6.11 Dining Arrangement for Philosophers

Using Semaphores

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

Cont.

Problem

- What happens if all 5 philosophers get hungry at the same time and all pick up their left forks at the same time
- No one can pick up their right fork
- Deadlock

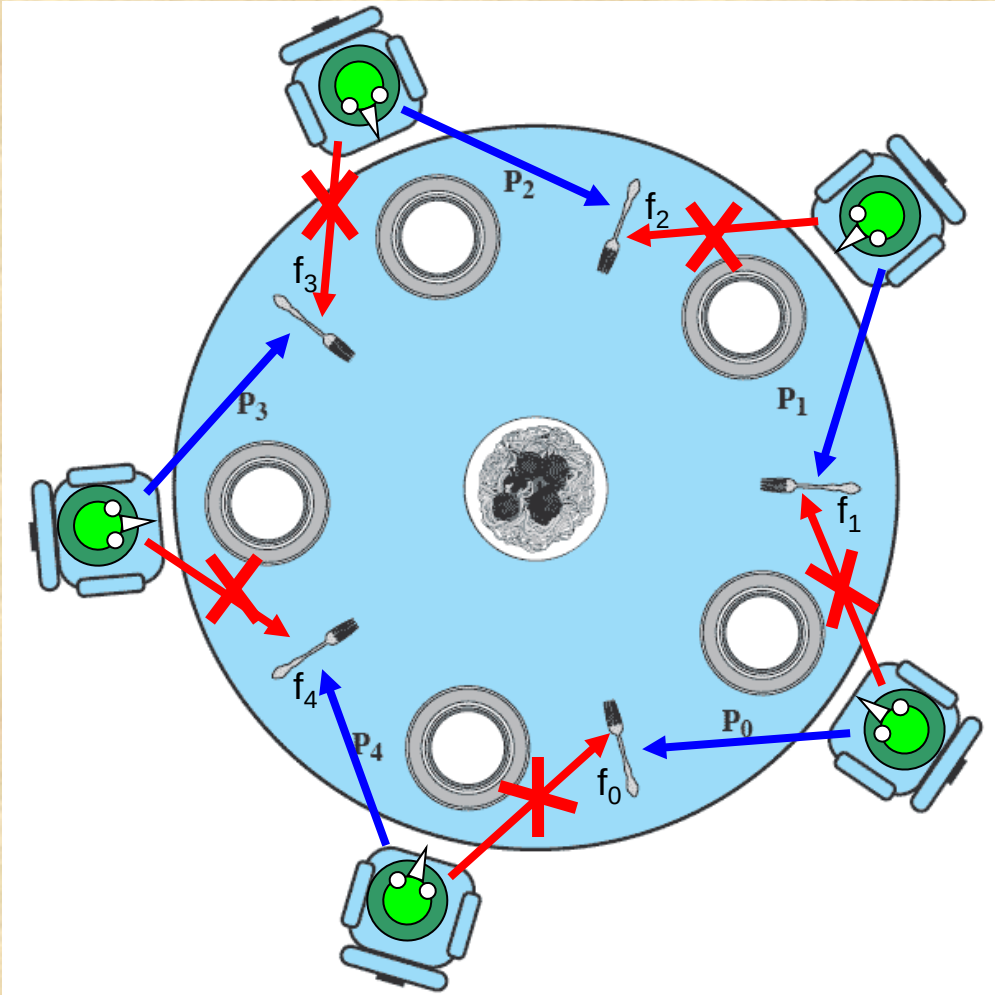


Figure 6.11 Dining Arrangement for Philosophers

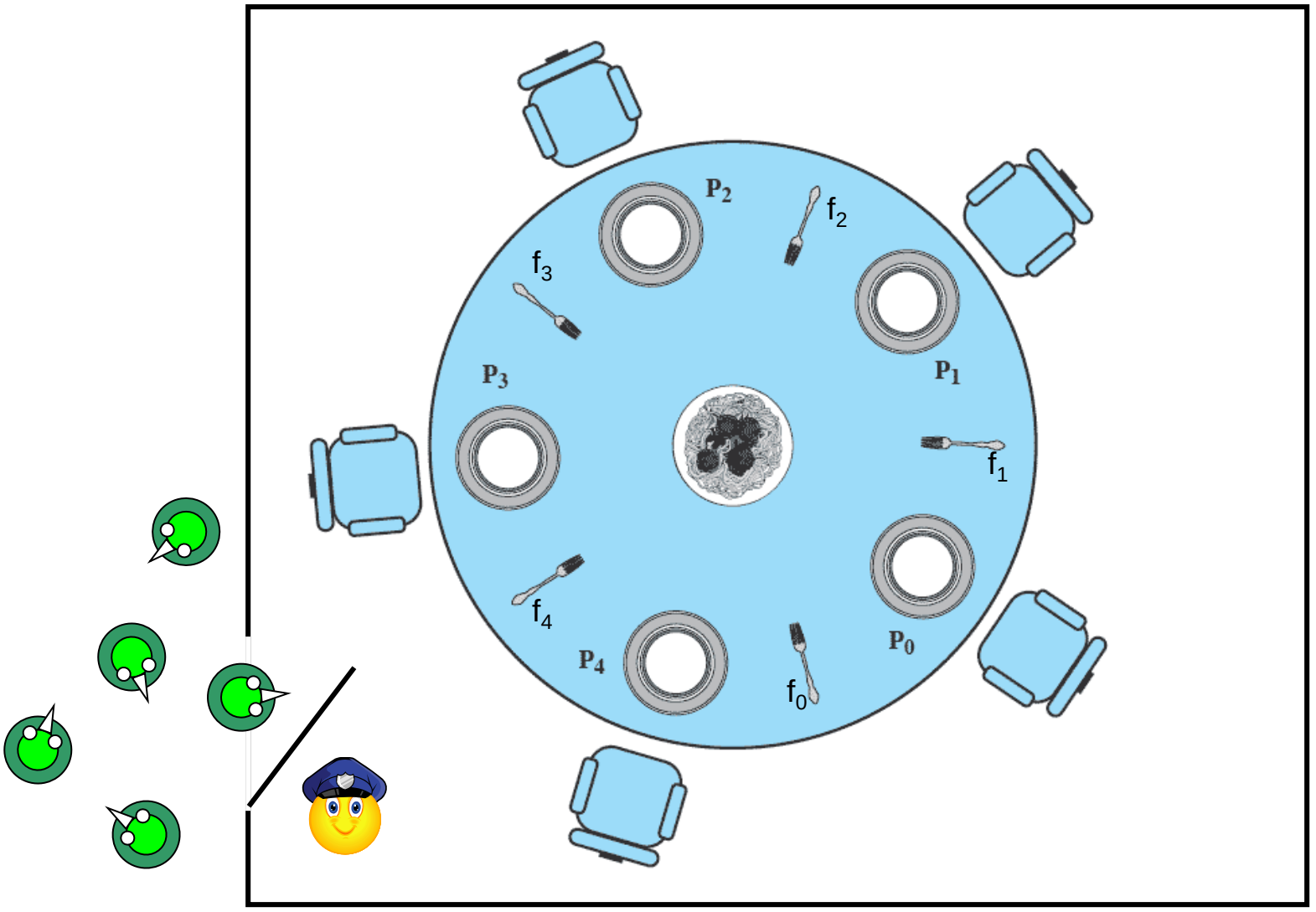
A Second Solution . . .

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

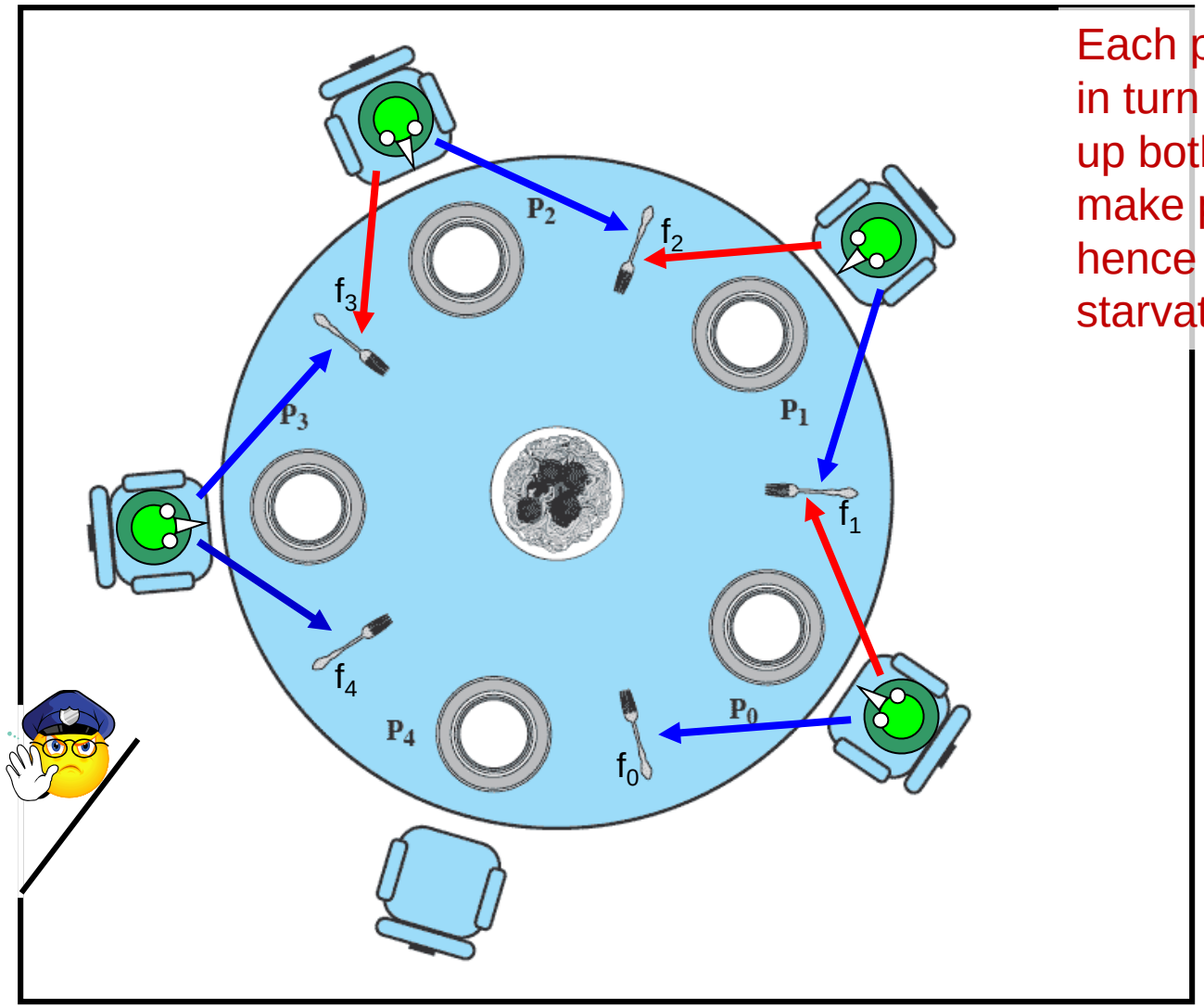
Dining Philosophers Problem



Dining Philosophers Problem

At least one philosopher will pick up both forks and make progress, hence no deadlock

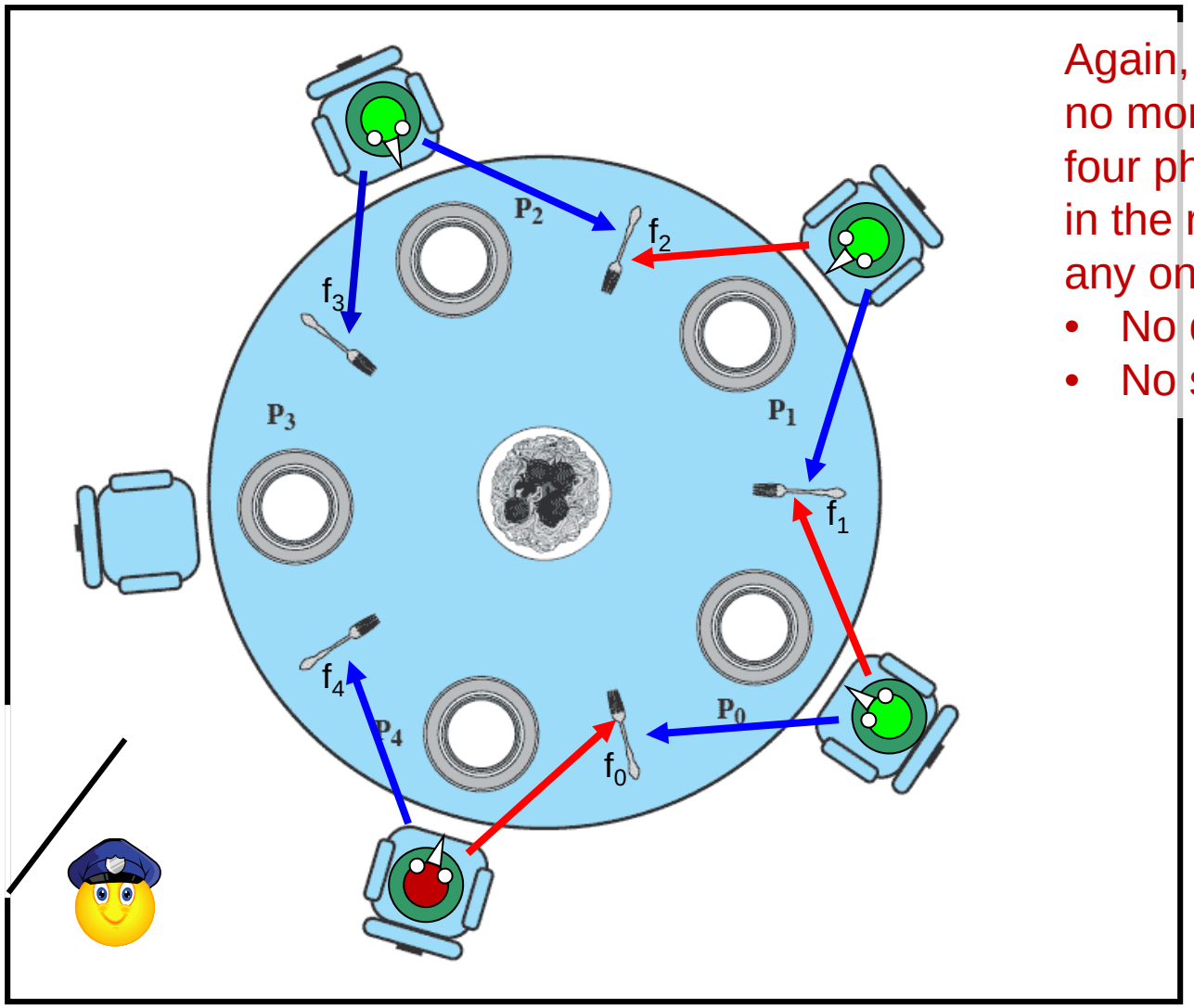
The room semaphore prevents more than four philosophers from entering the room at the same time



Each philosopher in turn will pick up both forks and make progress, hence no starvation

Dining Philosophers Problem

Once first philosopher is done eating, he leaves the room, and the last one can enter



Again, there are no more than four philosophers in the room at any one time:

- No deadlock
- No starvation

Third Solution?

- Can we find a solution that imposes a total ordering on resources and which only uses semaphores for forks?
  - Hint: how can we write the code to deny one of the four conditions for deadlock
- Try it
- You will need to use this solution in your project

Solution Using A Monitor

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)          /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork[left])
      cwait(ForkReady[left]);          /* queue on condition variable */
   fork[left] = false;
   /*grant the right fork*/
   if (!fork[right])
      cwait(ForkReady[right]);         /* queue on condition variable */
   fork[right] = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork[left] = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork[right] = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]           /* the five philosopher clients */
{
   while (true) {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);      /* client releases forks via the monitor */
   }
}
```

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Pipes

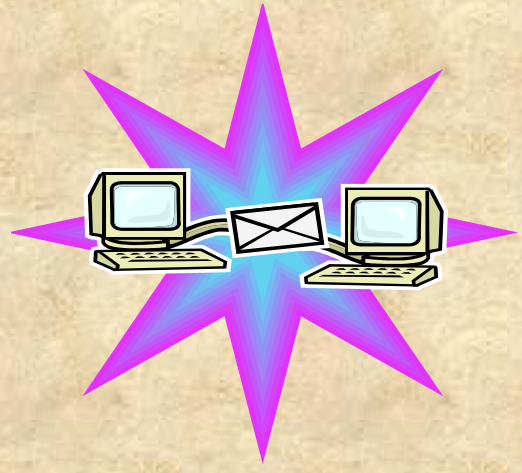
- Circular buffers allowing two processes to communicate on the producer-consumer model
  - first-in-first-out queue, written by one process and read by another

Two types:

- Named
- Unnamed

Messages

- A block of bytes with an accompanying type
- UNIX provides ***msgsnd*** and ***msgrcv*** system calls for processes to engage in message passing
- Associated with each process is a message queue, which functions like a mailbox



Shared Memory

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

Semaphores

- Generalization of the `semWait` and `semSignal` primitives
  - no other process may access the semaphore until all operations have completed

Consists of:
• current value of the semaphore • process ID of the last process to operate on the semaphore • number of processes waiting for the semaphore value to be greater than its current value • number of processes waiting for the semaphore value to be zero

The sem_op() and sem_ctl() system calls for UNIX System V semaphores is very convoluted and obscure

Most programmers write wrapper functions to implement wait / signal using the sem_op and sem_ctl system calls

Semaphores

- Generalization of the `semWait` and `semSignal` primitives
  - no other process may access the semaphore until all operations

Or, most UNIX programmers use alternative implementations of semaphores, such as the POSIX semaphores

consists of:

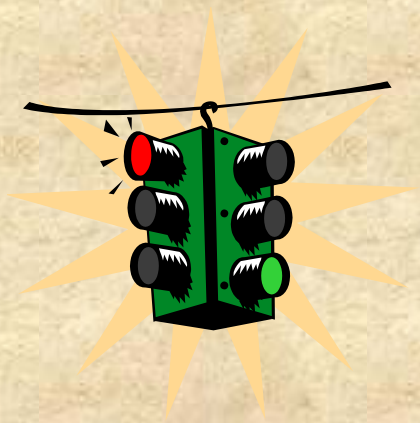
- current value of the semaphore
- process ID of the last process to operate on the semaphore
- number of processes waiting for the semaphore value to be greater than its current value
- number of processes waiting for the semaphore value to be zero

The sem_op() and sem_ctl() system calls for UNIX System V semaphores is very convoluted and obscure

Most programmers write wrapper functions to implement wait / signal using the sem_op and sem_ctl system calls

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
  - similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
  - performing some default action
  - executing a signal-handler function
  - ignoring the signal

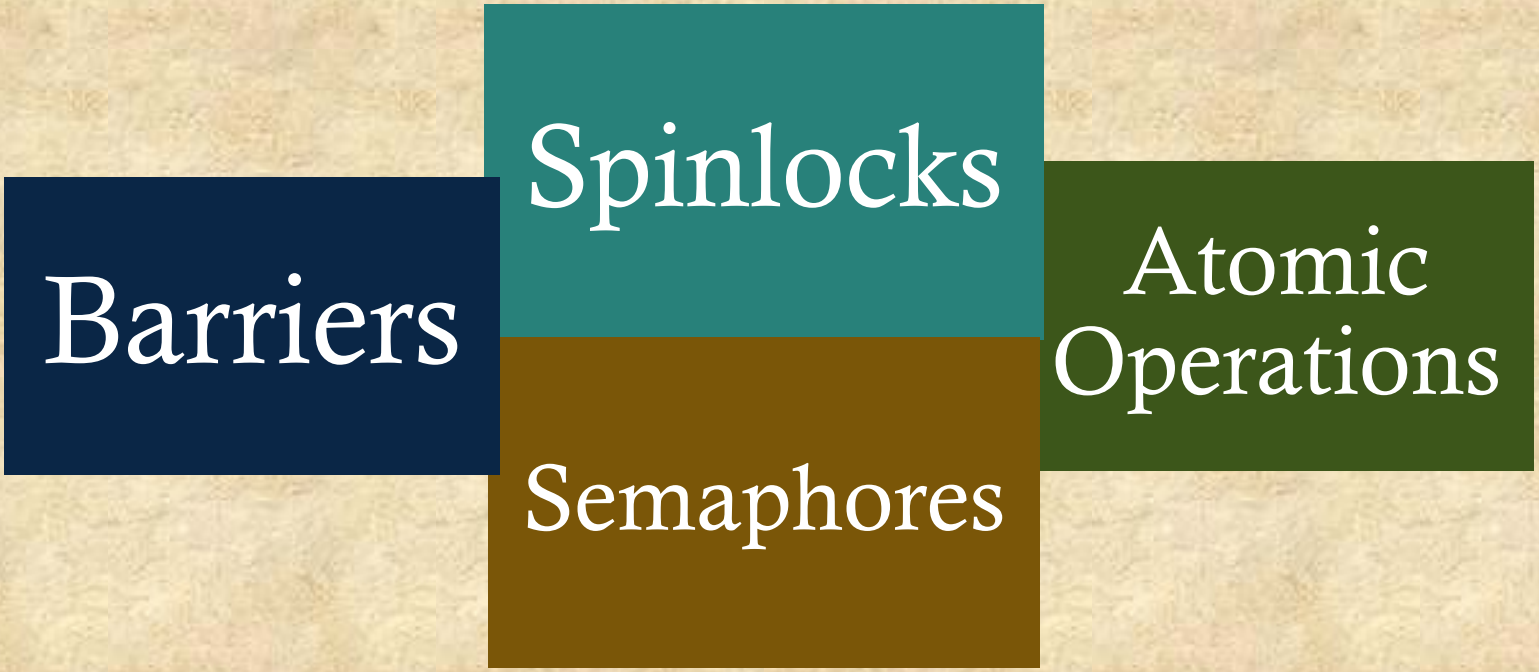


UNIX Signals

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Linux Kernel Concurrency Mechanism

- Includes all the mechanisms found in UNIX plus:



Atomic Operations

- Atomic operations execute without interruption and without interference
- Simplest of the approaches to kernel synchronization
- Two types:

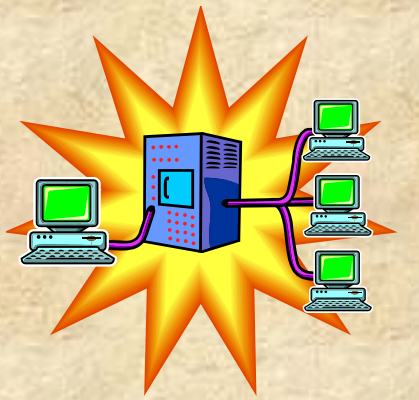
Integer Operations

- operate on an integer variable
- typically used to implement counters

Bitmap Operations

- operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable

Linux Atomic Operations



Atomic Integer Operations	
`ATOMIC_INIT (int i)`	At declaration: initialize an atomic_t to i
`int atomic_read(atomic_t *v)`	Read integer value of v
`void atomic_set(atomic_t *v, int i)`	Set the value of v to integer i
`void atomic_add(int i, atomic_t *v)`	Add i to v
`void atomic_sub(int i, atomic_t *v)`	Subtract i from v
`void atomic_inc(atomic_t *v)`	Add 1 to v
`void atomic_dec(atomic_t *v)`	Subtract 1 from v
`int atomic_sub_and_test(int i, atomic_t *v)`	Subtract i from v; return 1 if the result is zero; return 0 otherwise
`int atomic_add_negative(int i, atomic_t *v)`	Add i to v; return 1 if the result is negative; return 0 otherwise (used for implementing semaphores)
`int atomic_dec_and_test(atomic_t *v)`	Subtract 1 from v; return 1 if the result is zero; return 0 otherwise
`int atomic_inc_and_test(atomic_t *v)`	Add 1 to v; return 1 if the result is zero; return 0 otherwise
Atomic Bitmap Operations	
`void set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr
`void clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr
`void change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr
`int test_and_set_bit(int nr, void *addr)`	Set bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_clear_bit(int nr, void *addr)`	Clear bit nr in the bitmap pointed to by addr; return the old bit value
`int test_and_change_bit(int nr, void *addr)`	Invert bit nr in the bitmap pointed to by addr; return the old bit value
`int test_bit(int nr, void *addr)`	Return the value of bit nr in the bitmap pointed to by addr

Spinlocks

- Most common technique for protecting a critical section in Linux
- Can only be acquired by one thread at a time
  - any other thread will keep trying (spinning) until it can acquire the lock
- Built on an integer location in memory that is checked by each thread before it enters its critical section
- Effective in situations where the wait time for acquiring a lock is expected to be very short
- Disadvantage:
  - locked-out threads continue to execute in a busy-waiting mode

Linux Spinlocks

`void spin_lock(spinlock_t *lock)`	Acquires the specified lock, spinning if needed until it is available
`void spin_lock_irq(spinlock_t *lock)`	Like spin_lock, but also disables interrupts on the local processor
`void spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`	Like spin_lock_irq, but also saves the current interrupt state in flags
`void spin_lock_bh(spinlock_t *lock)`	Like spin_lock, but also disables the execution of all bottom halves
`void spin_unlock(spinlock_t *lock)`	Releases given lock
`void spin_unlock_irq(spinlock_t *lock)`	Releases given lock and enables local interrupts
`void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)`	Releases given lock and restores local interrupts to given previous state
`void spin_unlock_bh(spinlock_t *lock)`	Releases given lock and enables bottom halves
`void spin_lock_init(spinlock_t *lock)`	Initializes given spinlock
`int spin_trylock(spinlock_t *lock)`	Tries to acquire specified lock; returns nonzero if lock is currently held and zero otherwise
`int spin_is_locked(spinlock_t *lock)`	Returns nonzero if lock is currently held and zero otherwise

Semaphores

- User level:
  - Linux provides a semaphore interface corresponding to that in UNIX SVR4
- Internally:
  - implemented as functions within the kernel and are more efficient than user-visable semaphores
- Three types of kernel semaphores:
  - binary semaphores
  - counting semaphores
  - reader-writer semaphores

Linux Semaphores

Traditional Semaphores	
void sema_init(struct semaphore *sem, int count)	Initializes the dynamically created semaphore to the given count
void init_MUTEX(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 1 (initially unlocked)
void init_MUTEX_LOCKED(struct semaphore *sem)	Initializes the dynamically created semaphore with a count of 0 (initially locked)
void down(struct semaphore *sem)	Attempts to acquire the given semaphore, entering uninterruptible sleep if semaphore is unavailable
int down_interruptible(struct semaphore *sem)	Attempts to acquire the given semaphore, entering interruptible sleep if semaphore is unavailable; returns -EINTR value if a signal other than the result of an up operation is received
int down_trylock(struct semaphore *sem)	Attempts to acquire the given semaphore, and returns a nonzero value if semaphore is unavailable
void up(struct semaphore *sem)	Releases the given semaphore
Reader-Writer Semaphores	
void init_rwsem(struct rw_semaphore, *rwsem)	Initializes the dynamically created semaphore with a count of 1
void down_read(struct rw_semaphore, *rwsem)	Down operation for readers
void up_read(struct rw_semaphore, *rwsem)	Up operation for readers
void down_write(struct rw_semaphore, *rwsem)	Down operation for writers
void up_write(struct rw_semaphore, *rwsem)	Up operation for writers

Barriers

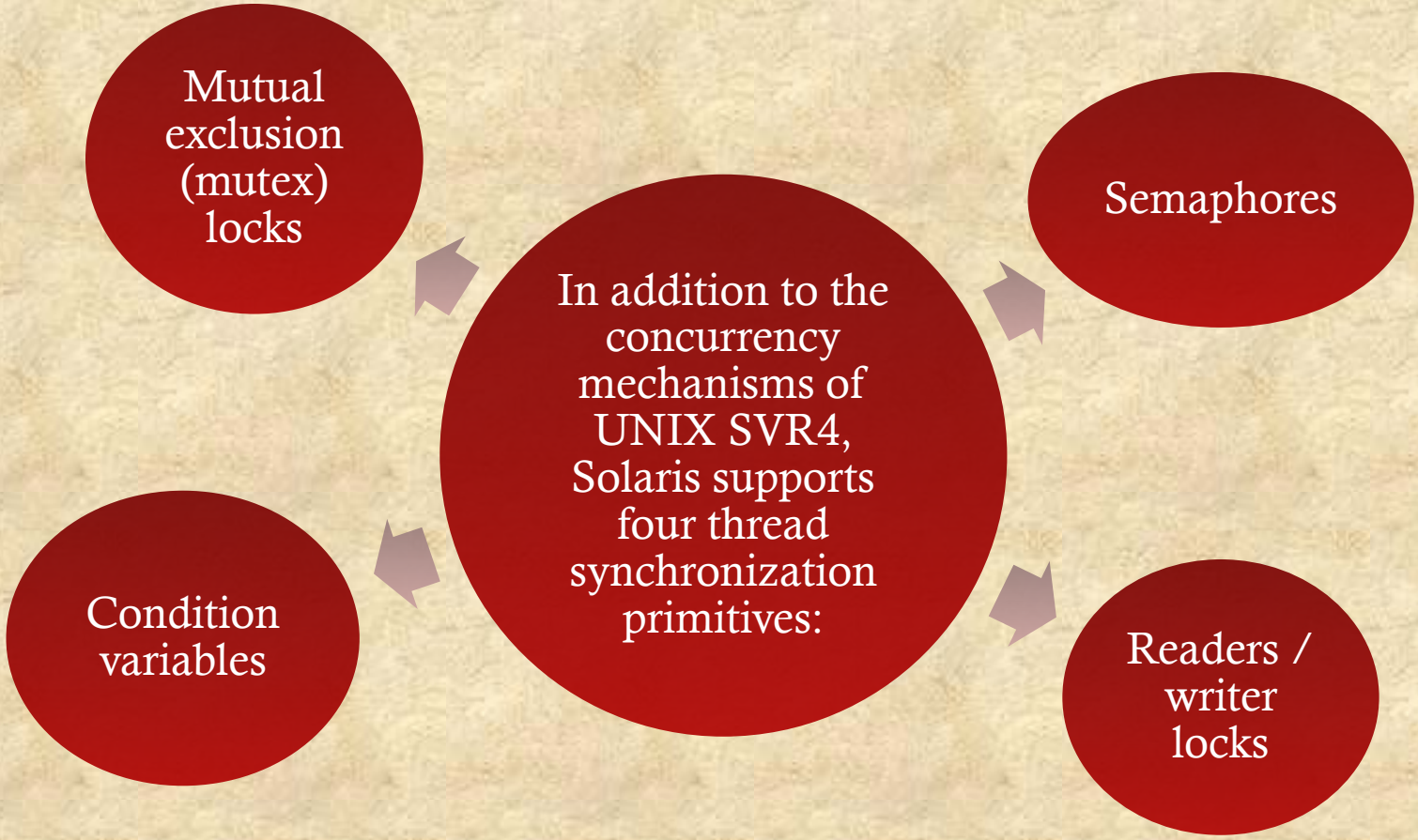
- enforce the order in which instructions are executed

`rmb()`	Prevents loads from being reordered across the barrier
`wmb()`	Prevents stores from being reordered across the barrier
`mb()`	Prevents loads and stores from being reordered across the barrier
`Barrier()`	Prevents the compiler from reordering loads or stores across the barrier
`smp_rmb()`	On SMP, provides a `rmb()` and on UP provides a `barrier()`
`smp_wmb()`	On SMP, provides a `wmb()` and on UP provides a `barrier()`
`smp_mb()`	On SMP, provides a `mb()` and on UP provides a `barrier()`

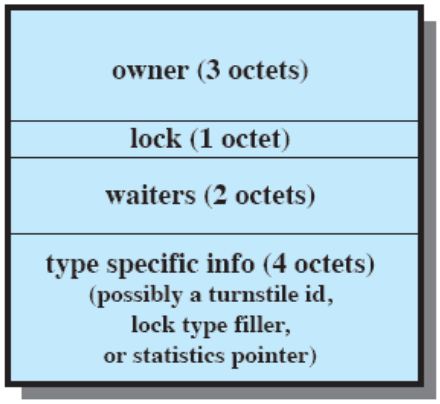
SMP = symmetric multiprocessor
UP = uniprocessor

Table 6.6 Linux Memory Barrier Operations

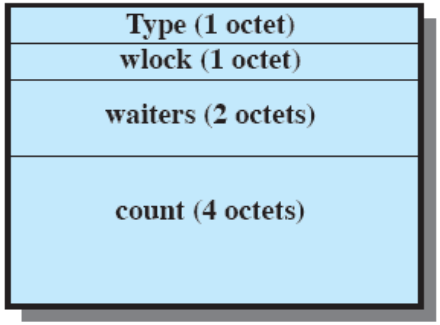
Solaris Kernel Synchronization



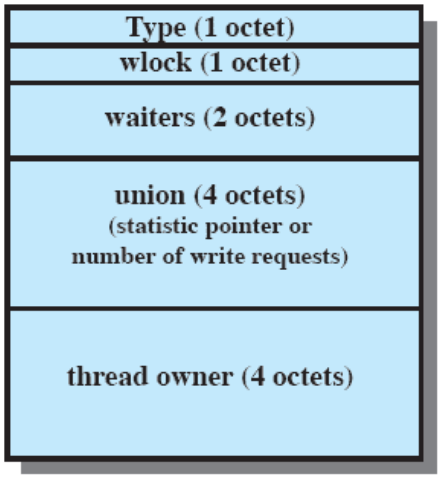
Solaris Data Structures



(a) MUTEX lock



(b) Semaphore



(c) Reader/writer lock

(d) Condition variable

Figure 6.15 Solaris Synchronization Data Structures

Mutual Exclusion (MUTEX) Lock

- Used to ensure only one thread at a time can access the resource protected by the mutex
- The thread that locks the mutex must be the one that unlocks it
- A thread attempts to acquire a mutex lock by executing the `mutex_enter` primitive
- Default blocking policy is a spinlock
- An interrupt-based blocking mechanism is optional

Semaphores

Solaris provides classic counting semaphores with the following primitives:

- sema_p() Decrements the semaphore, potentially blocking the thread
- sema_v() Increments the semaphore, potentially unblocking a waiting thread
- sema_tryp() Decrements the semaphore if blocking is not required

Solaris Semaphores

- Solaris Semaphore operations are:
  - sema_init
  - sema_destroy
  - sema_post — signal semaphore
  - sema_wait — wait on semaphore
  - sema_trywait — atomically decrements the semaphore, if count is greater than zero; otherwise, it returns an error.
- sema_init must be called prior to semaphore use.
- sema_destroy releases semaphore if it is no longer used.

POSIX Thread Semaphores

- POSIX Semaphore operations are:
  - sem_open - creates semaphore
  - sem_init - initializes semaphore
  - sem_wait - wait on semaphore
  - sem_trywait - wait on semaphore w/o blocking
  - sem_post - signals semaphore
  - sem_getvalue - gets value of semaphore w/o affecting its state
  - sem_unlink - removes semaphore
  - sem_close - makes semaphore unavailable to process
  - sem_destroy - deletes/destroys semaphore in the kernel

Readers/Writer Locks

- Allows multiple threads to have simultaneous read-only access to an object protected by the lock
- Allows a single thread to access the object for writing at one time, while excluding all readers
  - when lock is acquired for writing it takes on the status of `write lock`
  - if one or more readers have acquired the lock its status is `read lock`

Condition Variables

A condition variable is used to wait until a particular condition is true

Condition variables must be used in conjunction with a mutex lock

Windows 7 Concurrency Mechanisms

- Windows provides synchronization among threads as part of the object architecture

Most important methods are:

- executive dispatcher objects
- user mode critical sections
- slim reader-writer locks
- condition variables
- lock-free operations

Wait Functions

Allow a thread to block its own execution

Do not return until the specified criteria have been met

The type of wait function determines the set of criteria used

Table 6.7 Windows Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Notification event	An announcement that a system event has occurred	Thread sets the event	All released
Synchronization event	An announcement that a system event has occurred.	Thread sets the event	One thread released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File	An instance of an opened file or I/O device	I/O operation completes	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Note: Shaded rows correspond to objects that exist for the sole purpose of synchronization.

Critical Sections

- Similar mechanism to mutex except that critical sections can be used only by the threads of a single process
- If the system is a multiprocessor, the code will attempt to acquire a spin-lock
  - as a last resort, if the spinlock cannot be acquired, a dispatcher object is used to block the thread so that the kernel can dispatch another thread onto the processor

Slim Read-Writer Locks

- Windows Vista added a user mode reader-writer
- The reader-writer lock enters the kernel to block only after attempting to use a spin-lock
- It is *slim* in the sense that it normally only requires allocation of a single pointer-sized piece of memory

Condition Variables

- Windows also has condition variables
- The process must declare and initialize a CONDITION_VARIABLE
- Used with either critical sections or SRW locks
- Used as follows:
  1. acquire exclusive lock
  2. while (predicate()==FALSE) SleepConditionVariable()
  3. perform the protected operation
  4. release the lock

Releases lock and puts process to sleep
Lock is reaquired when process awakes

Lock-free Synchronization

- Windows also relies heavily on interlocked operations for synchronization
  - interlocked operations use hardware facilities to guarantee that memory locations can be read, modified, and written in a single atomic operation

"Lock-free"

- synchronizing without taking a software lock
- a thread can never be switched away from a processor while still holding a lock

Summary

- Deadlock:
  - the blocking of a set of processes that either compete for system resources or communicate with each other
  - blockage is permanent unless OS takes action
  - may involve reusable or consumable resources
    - Consumable = destroyed when acquired by a process
    - Reusable = not depleted/destroyed by use
- Dealing with deadlock:
  - prevention – guarantees that deadlock will not occur
  - detection – OS checks for deadlock and takes action
  - avoidance – analyzes each new resource request

Summary

- Principles of deadlock
  - Reusable/consumable resources
  - Resource allocation graphs
  - Conditions for deadlock
- Deadlock prevention
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- Deadlock avoidance
  - Process initiation denial
  - Resource allocation denial
- Deadlock detection
  - Deadlock detection algorithm
  - Recovery
- Android interprocess communication
- UNIX concurrency mechanisms
  - Pipes
  - Messages
  - Shared memory
  - Semaphores
  - Signals
- Linux kernel concurrency mechanisms
  - Atomic operations
  - Spinlocks
  - Semaphores
  - Barriers
- Solaris thread synchronization primitives
  - Mutual exclusion lock
  - Semaphores
  - Readers/writer lock
  - Condition variables
- Windows 7 concurrency mechanisms