Operating System

Chapter 6. Concurrency: Deadlock and Starvation

Lynn Choi
School of Electrical Engineering

Deadlock

❑ Definition

- A set of processes is deadlocked when each process in the set is blocked awaiting an event (or a resource) that can only be triggered (released) by another blocked process in the set

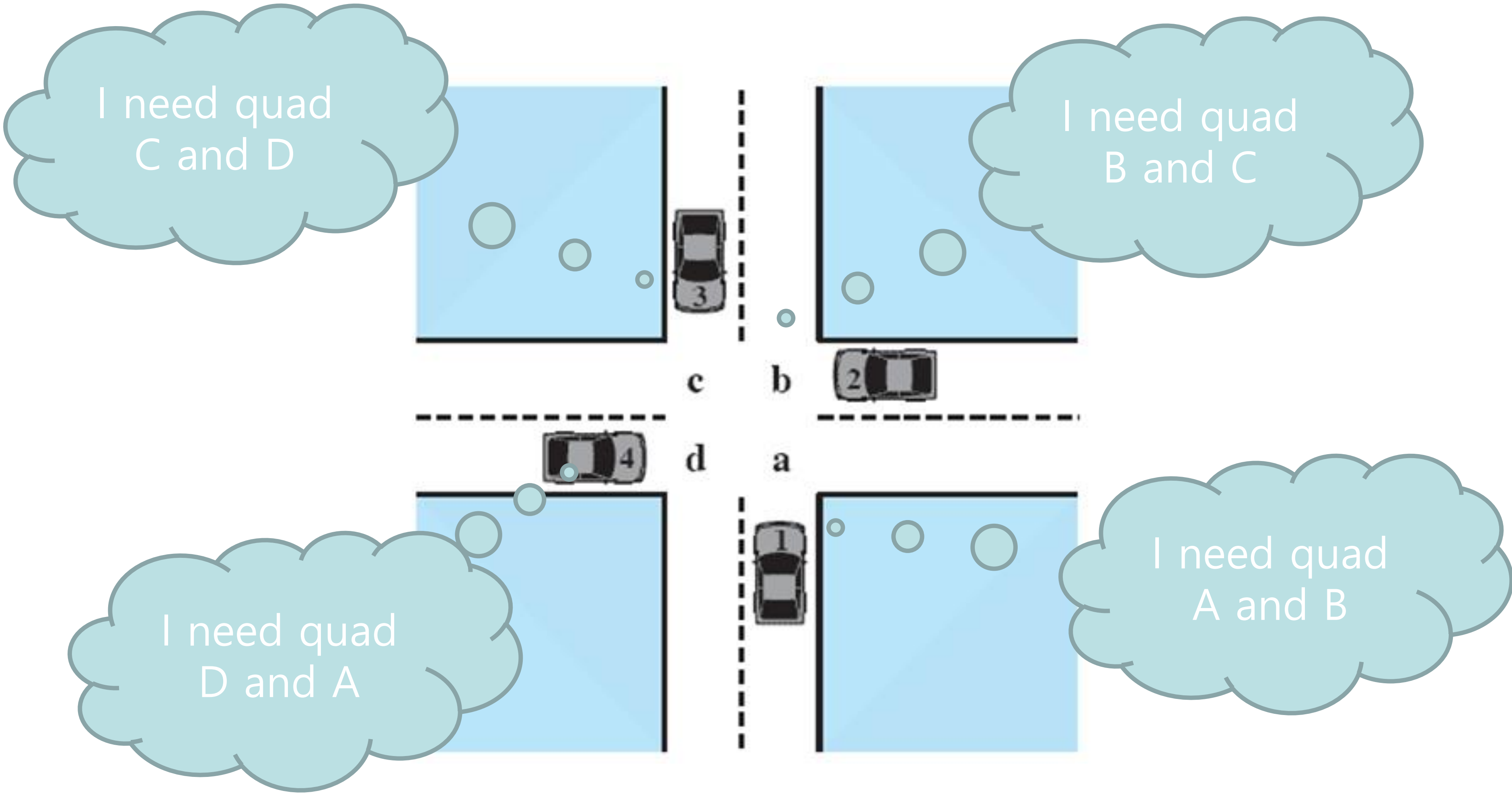
❑ Examples

- 4 cars arrive at a four-way stop

❑ Two general categories of resources

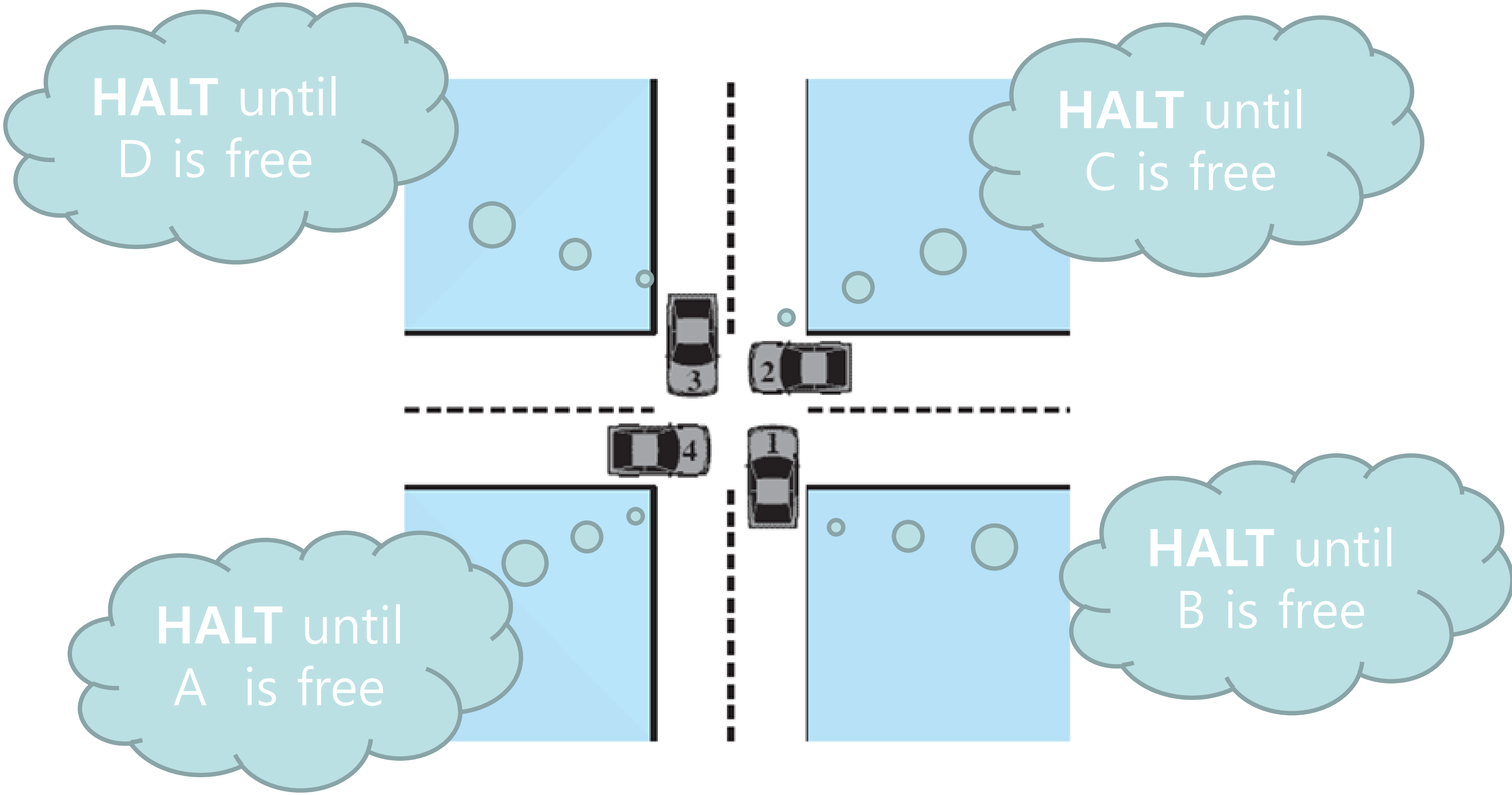
- Reusable resource
  - Can be safely used by only one process at a time and is not depleted by that use
  - Examples: processors, memory, I/O devices, files, databases and semaphores
- Consumable resource
  - Can be created (produced) and destroyed (consumed)
  - Typically, there is no limit on the number of consumable resources
  - Examples: interrupts, signals, messages, data in I/O buffers

Potential Deadlock



Source: Pearson

Actual Deadlock



Source: Pearson

Reusable Resource Example

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

$p_0p_1q_0q_1p_2q_2$ leads to a deadlock!

Source: Pearson

Consumable Resource Example

- 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

- Deadlock occurs if the Receive is blocking

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

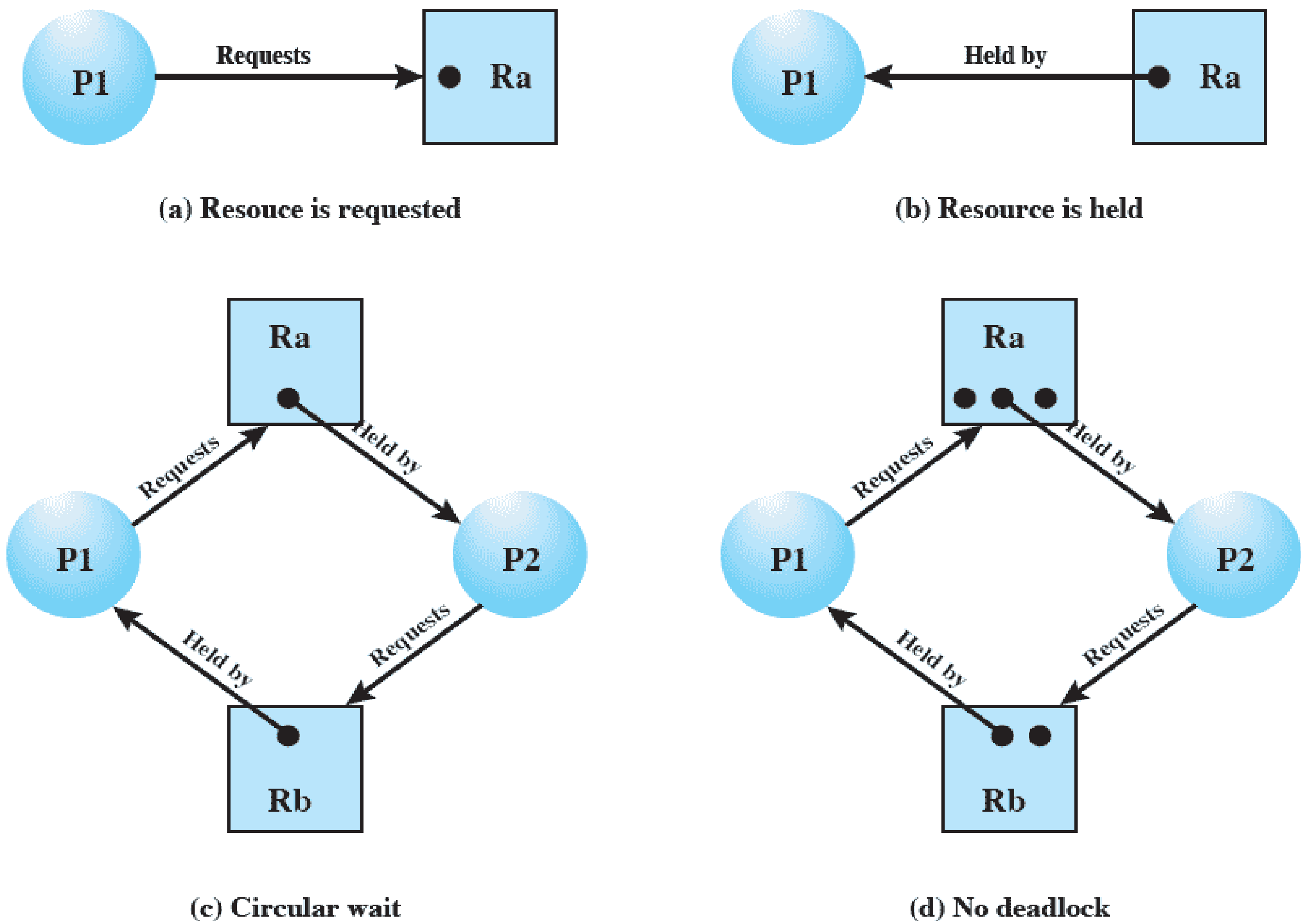
Source: Pearson

Conditions for Deadlock

- **Mutual exclusion**
  - Only one process may use a resource at a time
- **Hold and wait**
  - A process may hold allocated resources while waiting for the other resources
- **No preemption**
  - No resource can be forcibly removed from a process holding it
- **Circular wait**
  - A closed chain of processes exists such that each process holds at least one resource needed by the next process in the chain

- **Note that**
  - The first three conditions are necessary but not sufficient conditions for a deadlock.
    - The fourth condition is actually a consequence of the first three.
  - Given that the first three conditions exist, a sequence of events may occur that lead to an unresolvable circular wait.

Resource Allocation Graphs



Source: Pearson

Circular Wait Example

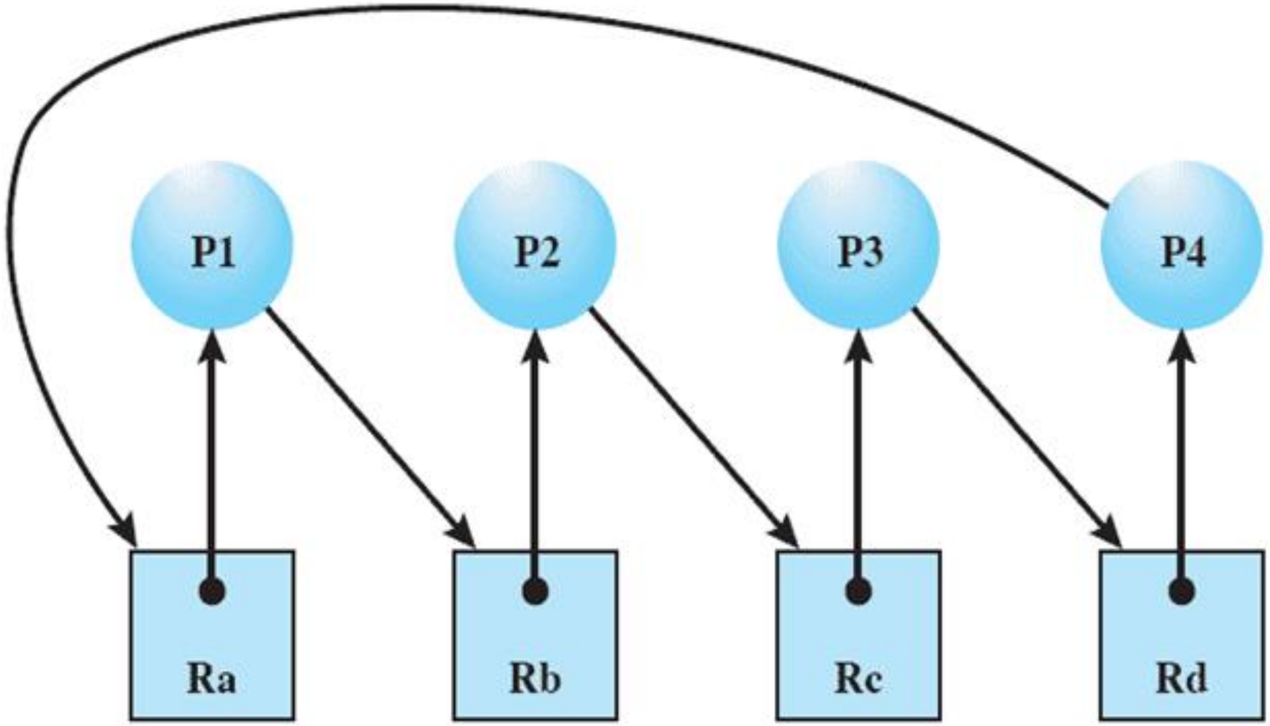


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Source: Pearson

Three Approaches for Deadlocks

- **Deadlock prevention**
  - Adopt a policy that eliminates one of the conditions 1 through 4
- **Deadlock avoidance**
  - Make the appropriate choices dynamically based on the current state of resource allocation
- **Deadlock detection**
  - Allow the deadlock to occur, attempt to detect the presence of deadlock, and recover if a deadlock is detected

Deadlock Detection, Prevention, 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

Source: Pearson

Deadlock Prevention

- **Mutual exclusion**
  - We cannot prevent this first condition
    - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the system
- **Hold and wait**
  - Require that a process request all of its required resources at once and block the process until all the requests can be granted simultaneously
- **No preemption**
  - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
  - Alternatively, if a process requests a resource that is currently held by another process, OS may preempt the second process
- **Circular wait**
  - Define a linear ordering of resource types
  - If a process has been allocated resources of type R, than it may subsequently request only those resources of types following R in the ordering

Deadlock Avoidance

- **Deadlock avoidance**
  - A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
    - Requires knowledge of future resource requests
- **Two approaches**
  - Process initiation denial
    - Do not start a process if its demands may lead to a deadlock
  - Resource allocation denial
    - Do not grant a resource request to a process if this allocation might lead to a deadlock
- **Advantages**
  - It is less restrictive than deadlock prevention
  - It is not necessary to preempt and rollback processes, as in deadlock detection
- **Require**
  - The maximum resource requirement must be known in advance

Process Initiation Denial

- Consider a system of ***n*** processes and ***m*** different types of resources
- Let's define the following vectors and matrices:

- Resource = $\mathbf{R} = (R_1, R_2, \ldots, R_m)$
- Available = $\mathbf{V} = (V_1, V_2, \ldots, V_m)$
- Claim = $\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \cdot & \cdot & & \cdot \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{bmatrix}$
- Allocation = $\mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \cdot & \cdot & & \cdot \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{bmatrix}$

Process Initiation Denial

- **The following relationship holds**
  - $R_j = V_j + \sum_{i=1}^{n} A_{ij}$ for all j
    - All resources are either available or allocated.
  - $C_{ij} \leq R_j$ for all i, j
    - No process can claim more than total amount of resources
  - $A_{ij} \leq C_{ij}$ for all i, j
    - No process is allocated more resources than it originally claimed

- **Policy: start a new process P_{n+1} only if**
  - $R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$ for all j
    - A process is only started if the maximum claim of all current processes plus those of the new process can be met

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

Determination of a Safe State

- System state consists of 4 processes and 3 resources

	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	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
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**

(a) **Initial state**

Source: Pearson

- **Is this a safe state?**
  - Can any of 4 processes run to completion?
    - P2 can run to completion!

P2 Runs to Completion

- After P2 completes, P2 releases its resources

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

Claim matrix C

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

Allocation matrix A

	R1	R2	R3
P1	2	2	2
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
6	2	3

Available vector V

(b) P2 runs to completion

Source: Pearson

- Then, we can run any of P1, P3, or P4
- Assume we select P1

P1 Runs to Completion

	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**

(c) **P1 runs to completion**

Source: Pearson

P3 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) P3 runs to completion

Source: Pearson

Determination of an Unsafe State

	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**

(a) **Initial state**

	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**

Source: Pearson

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

- Is this a safe 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 >;
}
```

(b) resource alloc algorithm

Source: Pearson

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

Source: Pearson

Deadlock Detection

- **Deadlock prevention is very conservative**
  - Limit access to resources by imposing restrictions on processes
- **Deadlock detection does the opposite**
  - Resource requests are granted whenever possible
- **A check for deadlock can be made**
  - As frequently as each resource request
    - (+) Lead to early detection
    - (+) Can employ a relatively simple detection algorithm because it is based on incremental changes to the system state
    - (–) Consume considerable processor time
  - Or less frequently depending on how likely it is for a deadlock to occur

Deadlock Detection Algorithm

- **Instead of Claim (C), a Request (Q) matrix is defined**
  - Q_{ij} represents the amount of resources of type *j* requested by process *i*
- **Initially, all processes are unmarked** (deadlocked)
- **The algorithm proceeds by marking processes that are not deadlocked.**
- **Then, the following steps are performed**
  1. *Mark each process that has a row of all zeros in the Allocation matrix*
  2. *Initialize a temporary vector* **W** *to equal the Available vector*
  3. *Find an index* ***i*** *such that process is currently unmarked and the* ***i****th row of Q is less than or equal to* **W**. *If no such row is found, terminate the algorithm*
  4. *If such a row is found, mark process* ***i*** *and add the corresponding row of the allocation matrix to* **W**. *Return to step 3*
- **A deadlock exists if and only if there are unmarked processes at the end of the algorithm**

Deadlock Detection Algorithm

Request matrix Q

	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

Allocation matrix A

	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

Resource vector

R1	R2	R3	R4	R5
2	1	1	2	1

Available vector

R1	R2	R3	R4	R5
0	0	0	0	1

Figure 6.10 Example for Deadlock Detection

Source: Pearson

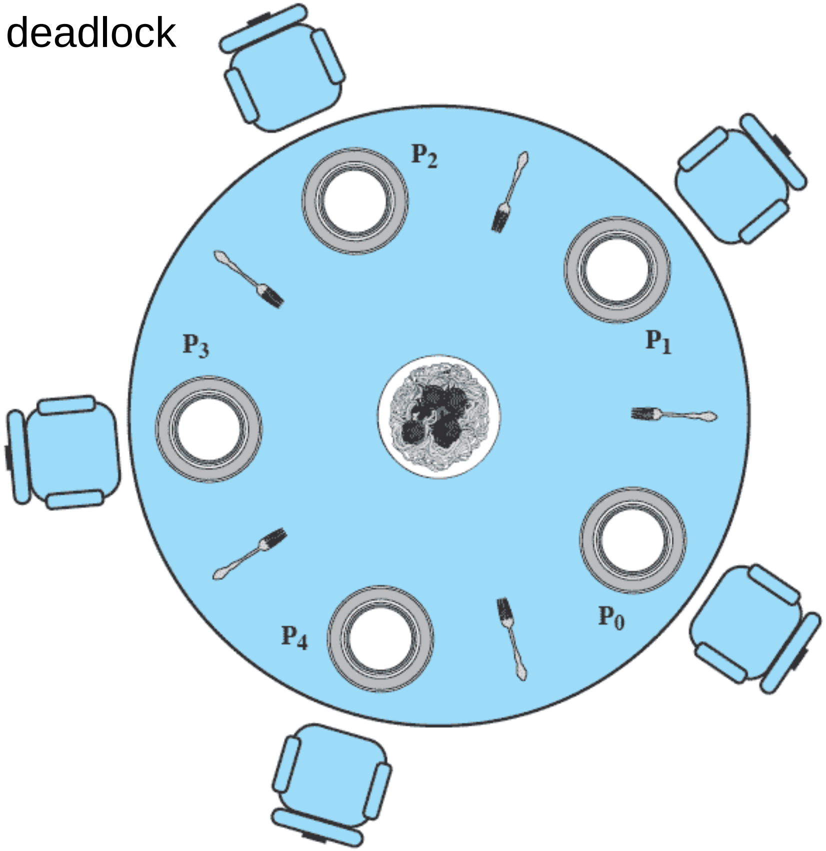
- Mark P4 because P4 has no allocated resources
- Set **W** = (0 0 0 0 1)
- The request of P3 is less than or equal to W, so mark P3 and set
  - **W** = **W** + (0 0 0 1 0) = (0 0 0 1 1)
- No other unmarked process has a row in Q that is less than or equal to W. Therefore, terminate the algorithm. P1 and P2 are deadlocked!

Deadlock Recovery

- **Recovery options in order of increasing sophistication**
  - Abort all deadlocked processes.
    - The most common solution adopted by OS
  - Backup each deadlocked process to a previous checkpoint and restart
    - Require rollback and restart mechanism
    - The original deadlock may recur
    - However, the nondeterminism of concurrent processing may ensure that this does not happen
  - Successively abort deadlocked process until deadlock no longer exists
    - After each abortion, the detection algorithm must be re-invoked
  - Successively preempt resources until deadlock no longer exists
    - A process that has a resource preempted must be rolled back to a prior point before its acquisition of the resource

Dining Philosophers Problem

- No two philosophers can use the same fork at the same time
  - Mutual exclusion
- No philosopher must starve to death
  - Avoid starvation and deadlock



Source: Pearson

Figure 6.11 Dining Arrangement for Philosophers

Solution I using Semaphore

```
/* 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

Source: Pearson

Solution II using Semaphore

```
/* 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 *Source: Pearson*

Solution using 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 */
   }
}
```

Source: Pearson

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocess communication and synchronization including:
  - **Pipes**
    - First-in-first-out queue, written by one process and read by another
    - Implemented by a circular buffer, allowing two processes to communicate on the producer-consumer model
    - Example: ls | more, ps | sort, etc.
  - **Messages**
    - UNIX provides *msgsnd* and *msgrcv* system calls for processes to engage in message passing
    - A message is a block of bytes
    - Each process is associated a *message queue*, which functions like a mailbox

UNIX Concurrency Mechanisms

- **Shared memory**
  - Common block of virtual memory shared by multiple processes
  - Fastest form of interprocess communication
  - Mutual exclusion is provided for each location in shared-memory
  - A process may have read-only or read-write permission for a memory location

- **Semaphores**
  - Generalization of the semWait and semSignal primitives defined in Chapter 5
  - Increment and decrement operations can be greater than 1
    - Thus, a single semaphore operation may involve incrementing/decrementing a semaphore and waking up/suspending processes.
    - Provide considerable flexibility in process synchronization

UNIX Concurrency Mechanisms

- **Signals**
  - A software mechanism that informs a process of the occurrence of asynchronous events (similar to a hardware interrupt)
  - Sending a signal
    - Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process.
    - Kernel sends a signal for one of the following reasons:
      - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
      - Another process has invoked the kill system call to explicitly request the kernel to send a signal to the destination process.
  - Receiving a signal
    - A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.
    - Two possible ways to react:
      - Default action (ignore, terminate the process, terminate & dump)
      - *Catch* the signal by executing a user-level function called a signal handler.
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt.

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

Source: Pearson

Signal Concepts (cont)

- **A signal is *pending* if it has been sent but not yet received.**
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- **A process can *block* the receipt of certain signals.**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
- **Kernel maintains `pending` and `blocked` bit vectors in the context of each process.**
  - `pending` – represents the set of pending signals
    - Kernel sets bit k in `pending` whenever a signal of type k is delivered.
    - Kernel clears bit k in `pending` whenever a signal of type k is received
  - `blocked` – represents the set of blocked signals
    - Can be set and cleared by the application using the `sigprocmask` function.

Receiving Signals

- **Suppose kernel is returning from exception handler and is ready to pass control to process *p*.**
- **Kernel computes `pnb = pending & ~blocked`**
  - The set of pending nonblocked signals for process *p*
- **If `(pnb == 0)`**
  - Pass control to next instruction in the logical flow for *p*.
- **Else**
  - Choose least nonzero bit *k* in `pnb` and force process *p* to receive signal *k.*
  - The receipt of the signal triggers some *action* by *p*
  - Repeat for all nonzero *k* in `pnb`.
  - Pass control to next instruction in logical flow for *p*.

Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`
- Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum`
  - SIG_DFL: revert to the default action on receipt of signals of type `signum`.
  - Otherwise, handler is the address of a user-defined function called *signal handler*
    - Called when process receives signal of type `signum`
    - Changing the default action by passing the address of a handler to the signal function is known as "*installing*" the handler.
    - The invocation of the handler is called "*catching*" the signal
    - The execution of the handler is referred as "*handling*" the signal.
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

Signal Handling Example

```
#include "csapp.h"

void handler(int sig) {
    static int beeps = 0;
    printf("BEEP\n");
    if (++beeps < 5) Alarm(1); /* next SIGALRM
will be delivered in 1s */
    else {
        printf("BOOM!\n");
        exit(0);
    }
}

int main() {

    Signal(SIGALRM, handler); /* install
SIGALRM handler */
    Alarm(1); /* next SIGALRM will be delivered
in 1s */
    while (1) { ; /* signal handler returns
control here each time */ }
    exit(0);
}
```

```
linux> ./alarm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
linux>
```

Source: Pearson

Homework 5

- Exercise 6.5
- Exercise 6.8
- Exercise 6.9
- Exercise 6.11
- Exercise 6.14