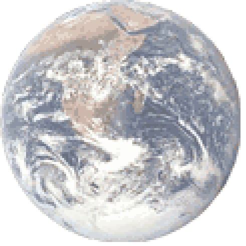
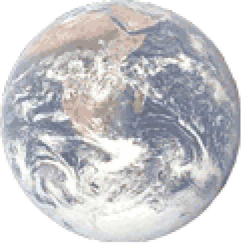


Concurrency:
Deadlock and Starvation



Agenda

- ✓ Principles of Deadlock
- ✓ Deadlock Prevention
- ✓ Deadlock Avoidance
- Deadlock Detection
- An Integrated Deadlock Strategy
- Dining Philosophers Problem
- Linux Kernel Concurrency Mechanisms

Deadlock Detection

•Deadlock **prevention** strategies are very conservative; they solve the problem of deadlock by **limiting** access to resources and by imposing restrictions on processes.

•**At the opposite extreme**, deadlock detection strategies do not limit resource access or restrict process actions.

Deadlock Detection

•With deadlock **detection**, requested resources are granted to processes whenever possible.

Periodically, the **OS** performs an **algorithm** that allows it to detect the circular wait condition described earlier in condition (4).

A common algorithm for deadlock detection

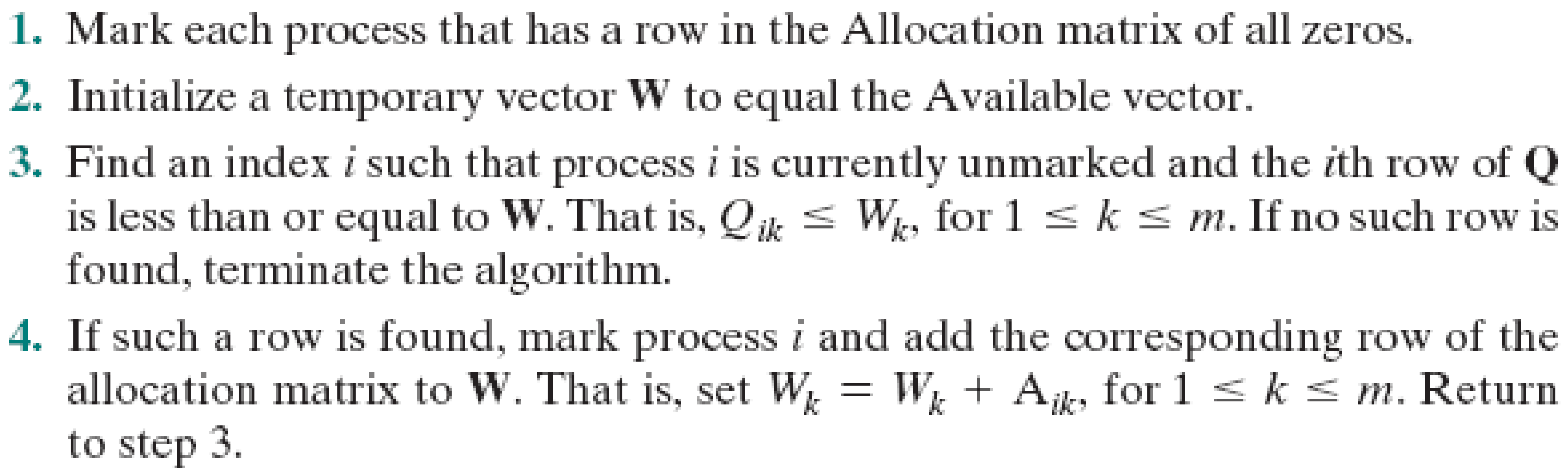
The **Allocation matrix** and **Available vector** described in the previous part are used.

In addition, a **request matrix Q is** defined such that ***Qij*** represents the amount of resources of type *j requested by process i.*

The algorithm *proceeds by marking* processes that are not deadlocked.

Initially, all processes are unmarked.

Then the following steps are performed:



1. Mark each process that has a row in the Allocation matrix of all zeros.
2. Initialize a temporary vector **W** to equal the Available vector.
3. Find an index *i* such that process *i* is currently unmarked and the *i*th row of **Q** is less than or equal to **W**. That is, $Q_{ik} \leq W_k$, for $1 \leq k \leq m$. 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**. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

•A deadlock **exists if and only if** there are unmarked processes at the end of the algorithm.

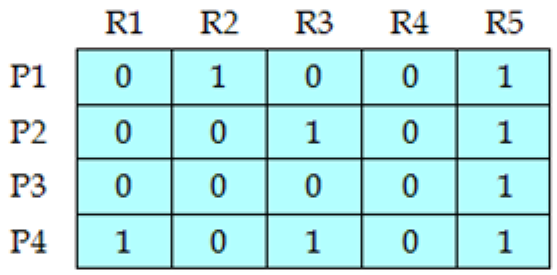
•Each **unmarked** process is deadlocked.

The strategy in this algorithm is to find a process whose resource requests can be satisfied with the available resources, and then assume that those resources are granted and that the process runs to completion and releases all of its resources.

The algorithm then looks for another process to satisfy.
Note that this algorithm does **not guarantee** to prevent deadlock; that will depend on the order in which future requests are granted.

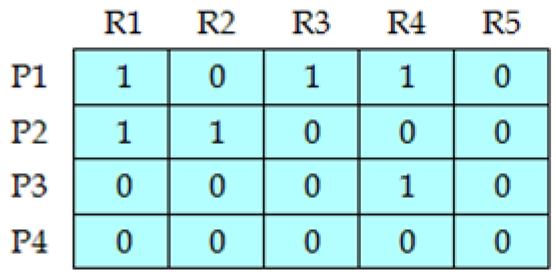
All that it does is determine if deadlock currently exists.

Deadlock Detection Example



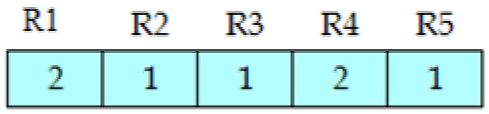
	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix **Q**



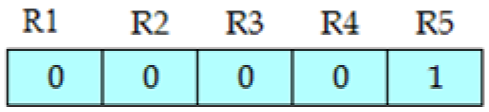
	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

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

We can use this Figure to illustrate the deadlock detection algorithm.

<u>The algorithm proceeds as follows:</u>

1. Mark P4, because P4 has no allocated resources.

2. Set W = (0 0 0 0 1).

3. The <u>request of process P3</u> is less than or equal to W , so mark P3 and set

W = W + (0 0 0 1 0) = (0 0 0 1 1).

4. No other unmarked process has a row in Q that is <u>less than or equal to</u> W.

Therefore, terminate the algorithm.

The algorithm concludes with P1 and P2 unmarked, indicating that these processes are deadlocked.

Strategies once Deadlock Detected

Once deadlock has been detected, some strategy is needed for recovery.

1. Abort all deadlocked processes
2. Back up each deadlocked process to some previously defined checkpoint, and restart all process
3. Successively **abort** deadlocked processes until deadlock no longer exists
4. Successively **preempt** resources until deadlock no longer exists

Selection Criteria Deadlocked Processes for strategies 3 and 4

- **Least amount of processor time consumed so far**
- **Least number of lines of output produced so far**
- **Most estimated time remaining**
- **Least total resources allocated so far**
- **Lowest priority**

Strengths and Weaknesses of the Strategies

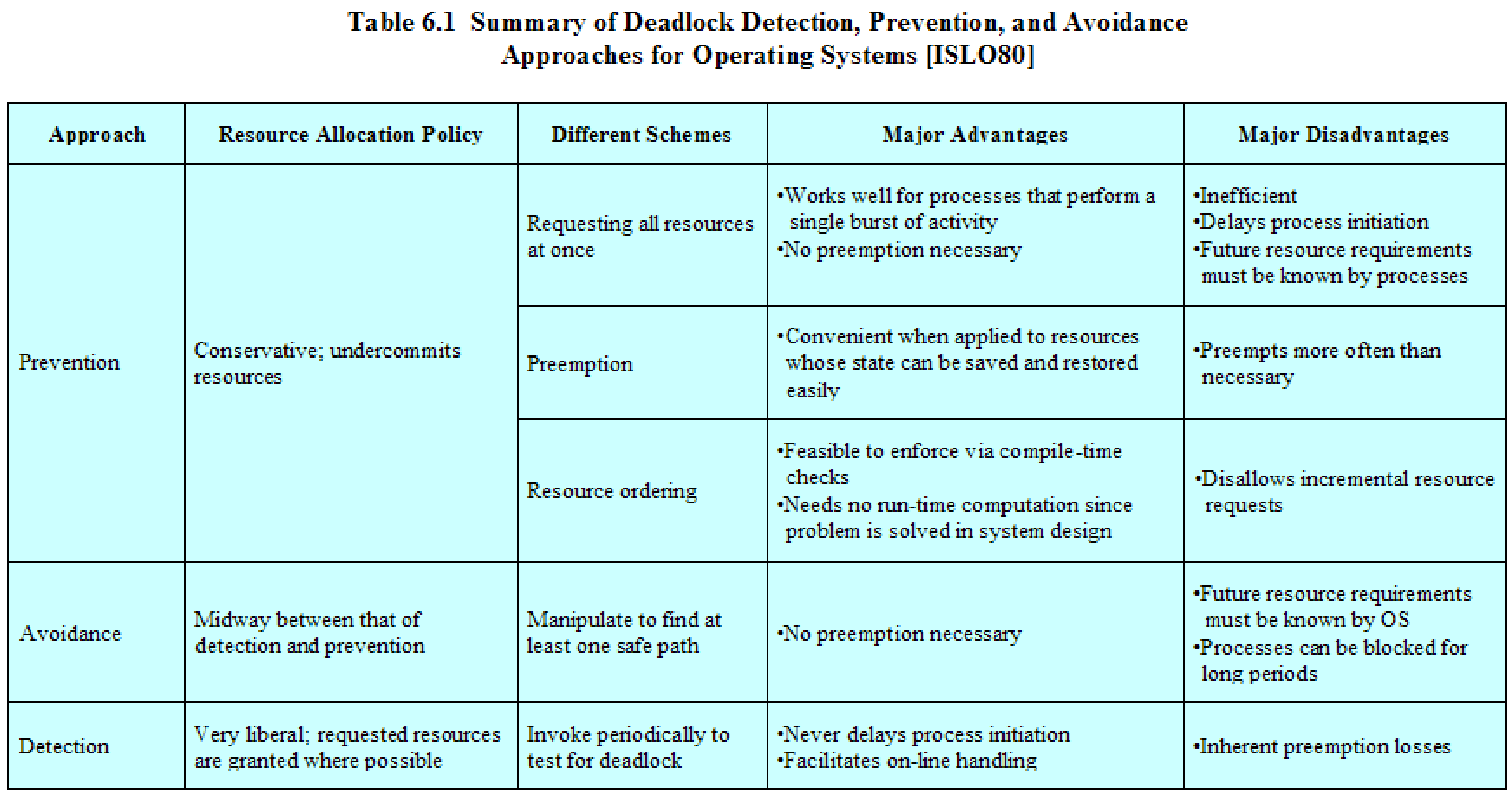


Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

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 on-line handling	•Inherent preemption losses

An Integrated Deadlock Strategy

Rather than attempting to design an OS facility that employs only one of these strategies, it might be more efficient to use different strategies in different situations.

[HOWA73] suggests one approach: page 278
7th ed.

Example: Dining Philosophers

- Problem statement:
  1. Five philosophers sit around a circular table.
  2. Each leads a simple life alternating between <u>thinking</u> and <u>eating</u> rice (**<u>spaghetti</u>**).
  3. In front of each philosopher is a **<u>bowl of rice</u>** that is constantly replenished by a dedicated wait staff.
  4. There are exactly **<u>five chopsticks</u>** on the table, one between each adjacent pair of philosophers.
  5. Eating rice (in the most proper manner) requires that a philosopher use **<u>both adjacent chopsticks</u>** (**<u>forks</u>**) (simultaneously).

Develop a concurrent program <u>free of deadlock</u> and imprecise delay that models the activities of the philosophers.

Dining Philosophers Problem

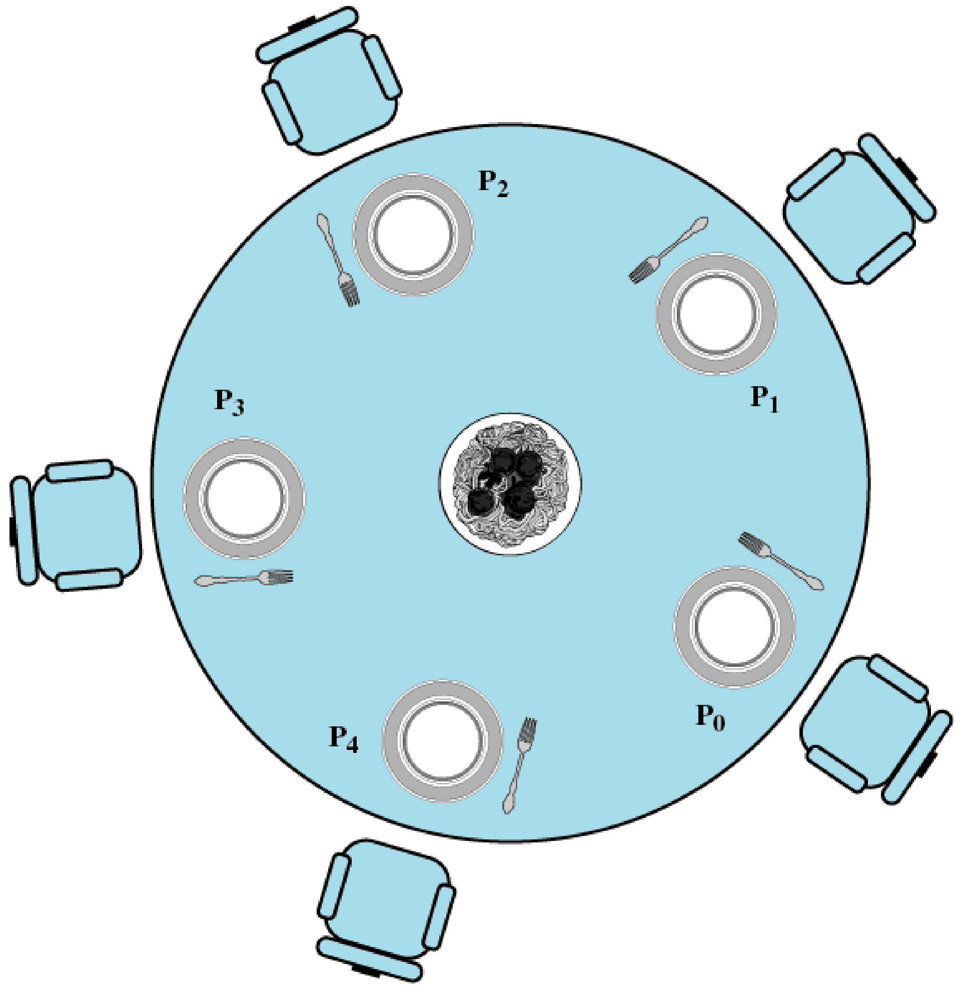
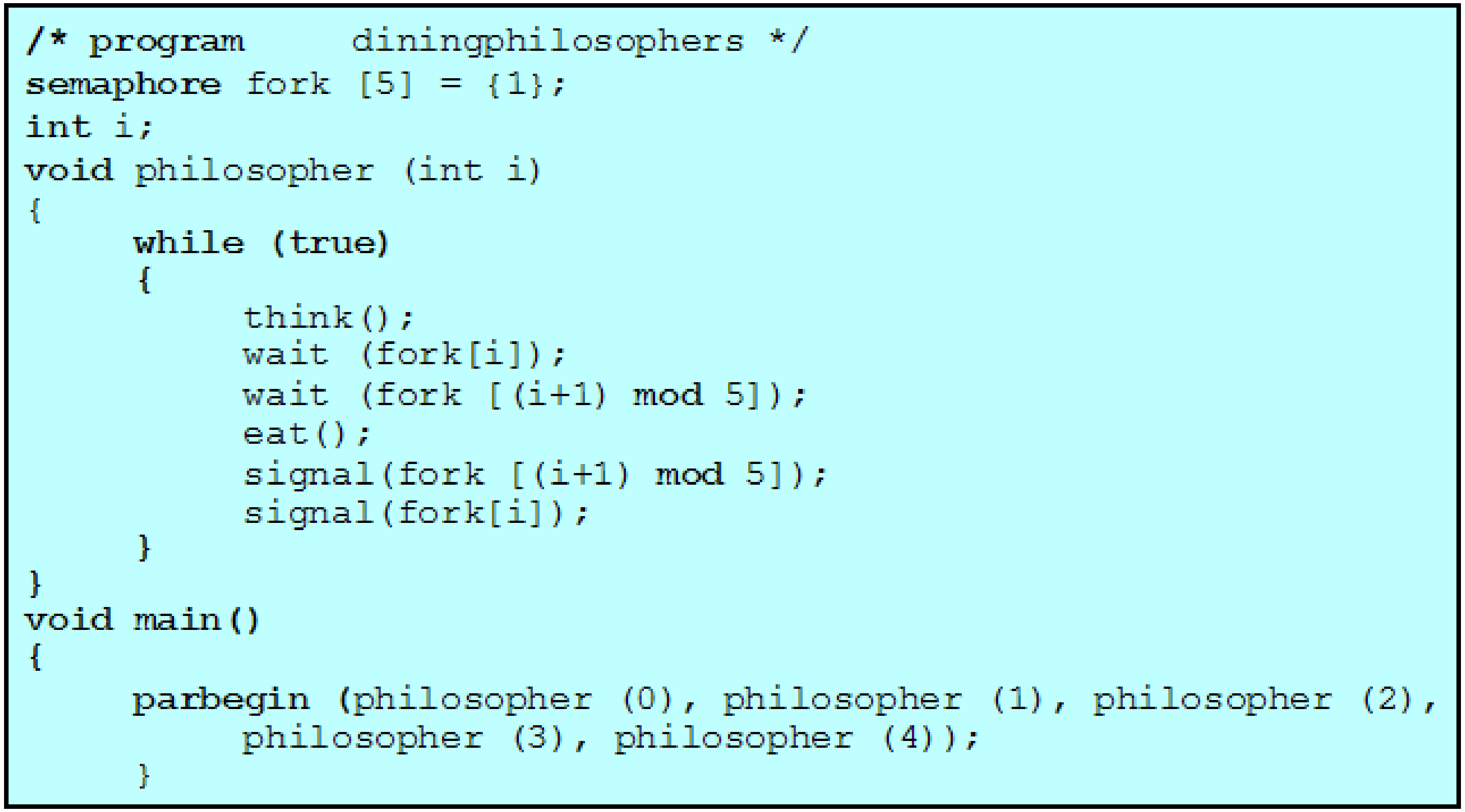


Figure 6.11 Dining Arrangement for Philosophers

Dining Philosophers Problem



```
/* 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

Dining Philosophers Problem

```
/* 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

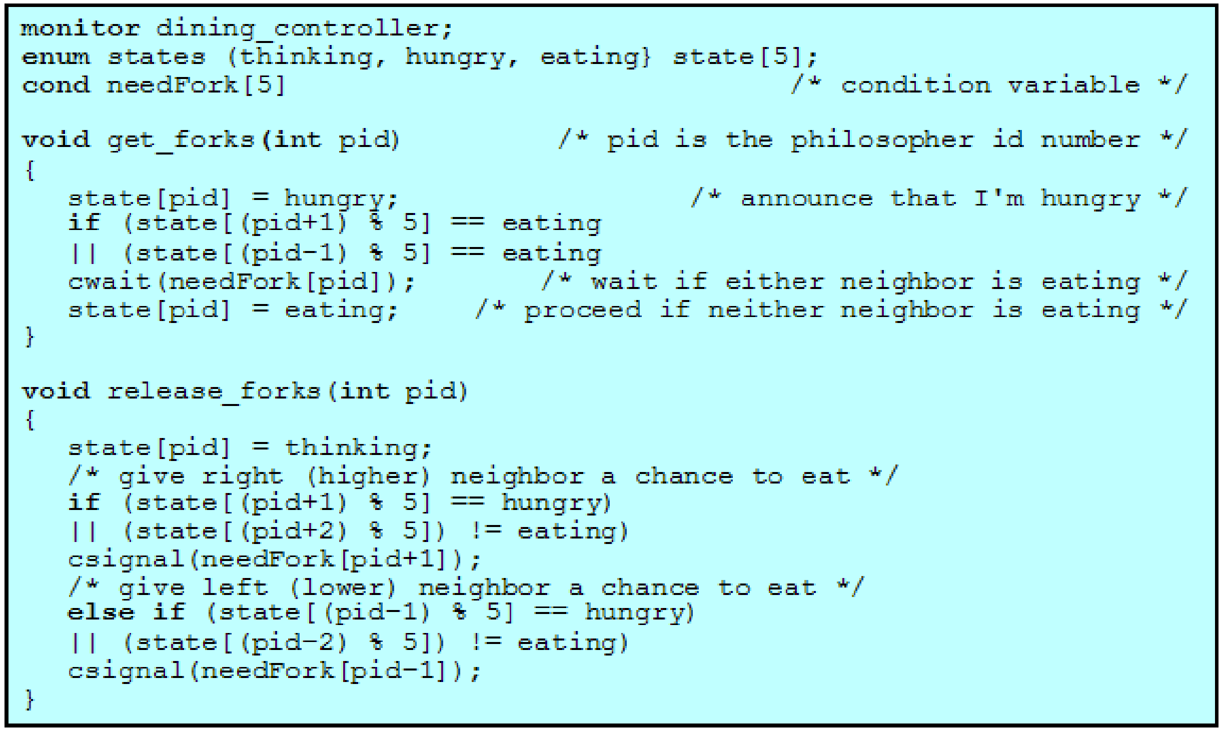
```
monitor dining_controller;
cond ForkReady[5];         /* condition variable for synchronization */
boolean fork[5] = {true};        /* availability status of each fork */

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 */
   }
}
```

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

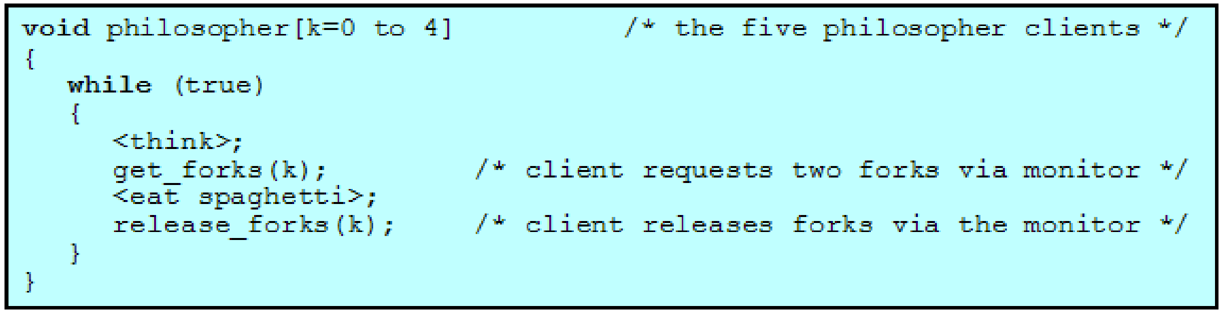
Dining Philosophers Problem



```
monitor dining_controller;
enum states (thinking, hungry, eating} state[5];
cond needFork[5]                                 /* condition variable */

void get_forks(int pid)           /* pid is the philosopher id number */
{
   state[pid] = hungry;                       /* announce that I'm hungry */
   if (state[(pid+1) % 5] == eating
   || (state[(pid-1) % 5] == eating
   cwait(needFork[pid]);           /* wait if either neighbor is eating */
   state[pid] = eating;       /* proceed if neither neighbor is eating */
}

void release_forks(int pid)
{
   state[pid] = thinking;
   /* give right (higher) neighbor a chance to eat */
   if (state[(pid+1) % 5] == hungry)
   || (state[(pid+2) % 5]) != eating)
   csignal(needFork[pid+1]);
   /* give left (lower) neighbor a chance to eat */
   else if (state[(pid-1) % 5] == hungry)
   || (state[(pid-2) % 5]) != eating)
   csignal(needFork[pid-1]);
}
```



```
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 */
   }
}
```

Figure 6.17 Another Solution to the Dining Philosophers Problem Using a Monitor

Linux Kernel Concurrency Mechanisms

Linux includes all of the concurrency mechanisms found in other UNIX systems, such as SVR4, including **pipes**, **messages**, **shared memory**, and **signals**.

In addition, Linux 2.6 includes a rich set of concurrency mechanisms specifically intended for use when a thread is executing in kernel mode.

That is, these are mechanisms used within the kernel to provide concurrency in the execution of kernel code.

Task-2

- Examine the Linux kernel concurrency mechanisms.

- Sheet -1

Sheet - 1

Review Questions

6.1 Give examples of reusable and consumable resources.

6.2 What are the three conditions that must be present for deadlock to be possible?

6.3 What are the four conditions that create deadlock?

6.4 How can the hold-and-wait condition be prevented?

6.5 List two ways in which the no-preemption condition can be prevented.

6.6 How can the circular wait condition be prevented?

6.7 What is the difference among deadlock avoidance, detection, and prevention?

Problems

6.1 Show that the four conditions of deadlock apply to Figure 6.1a .

6.2 Show how each of the techniques of prevention, avoidance, and detection can be applied to Figure 6.1 .

6.3 For Figure 6.3 , provide a narrative description of each of the six depicted paths, similar to the description of the paths of Figure 6.2 provided in Section 6.1 .

6.5 Given the following state for the Banker's Algorithm.
6 processes P0 through P5
4 resource types: A (15 instances); B (6 instances)
C (9 instances); D (10 instances)
Snapshot at time T0:

Available

A	B	C	D
6	3	5	4

	Current allocation				**Maximum demand**			
Process	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P0	2	0	2	1	9	5	5	5
P1	0	1	1	1	2	2	3	3
P2	4	1	0	2	7	5	4	4
P3	1	0	0	1	3	3	3	2
P4	1	1	0	0	5	2	2	1
P5	1	0	1	1	4	4	4	4

a. Verify that the Available array has been calculated correctly.
b. Calculate the Need matrix.
c. Show that the current state is safe, that is, show a safe sequence of processes. In addition, to the sequence show how the Available (working array) changes as each process terminates.
d. Given the request (3,2,3,3) from Process P5. Should this request be granted? Why or why not?