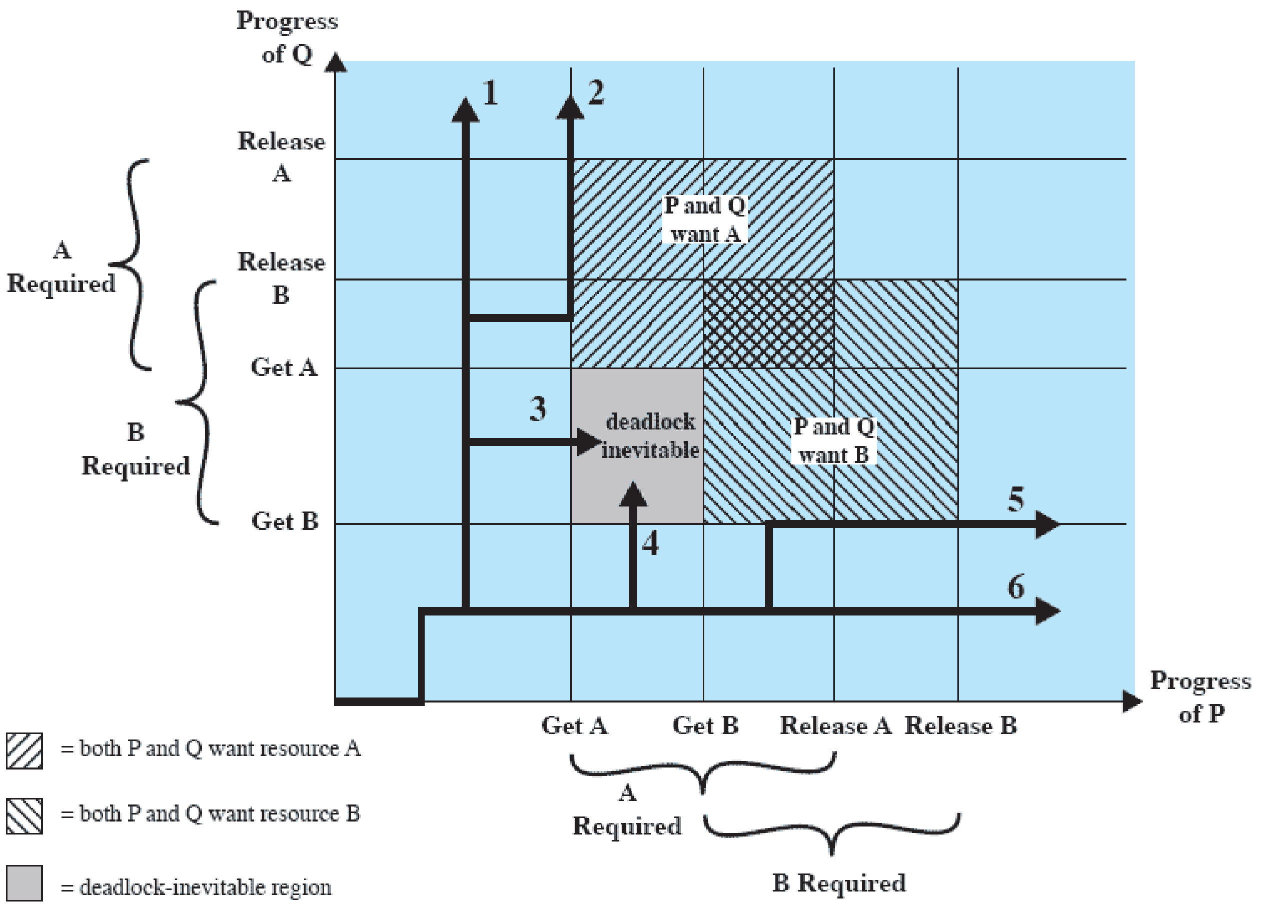
Chapter 6
Concurrency: Deadlock and Starvation

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other
- No efficient solution in the general case *(distinguish between universal properties and application-specific properties)*
- Involve conflicting needs for resources by two or more processes

Joint Progress Diagram



= possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

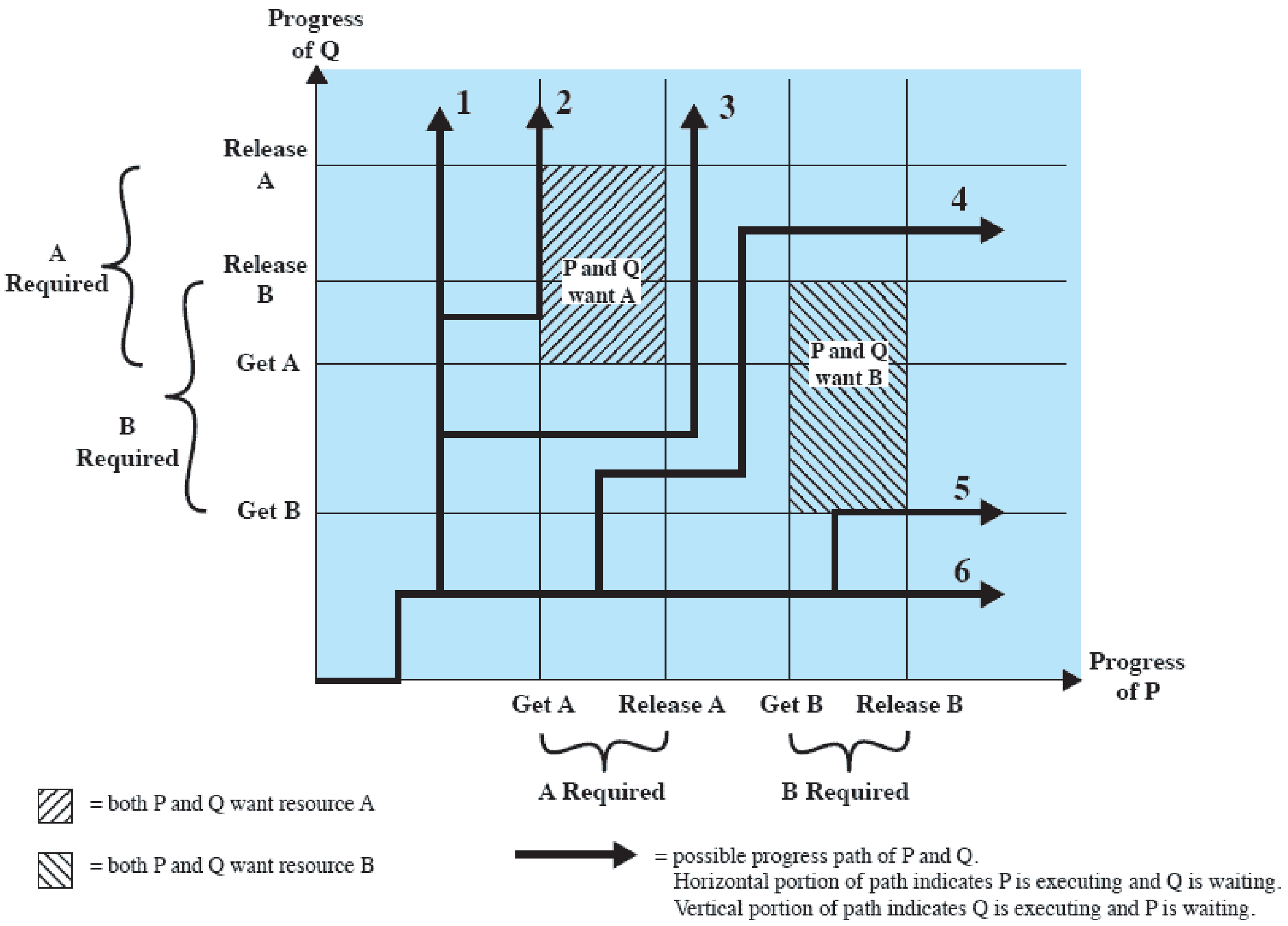
Figure 6.2 Example of Deadlock

Lets say we have processe P and Q and two resources A and B. P calls get(A), get(B), release(A), release(B). Q calls get(B), get(A), release(B), release(A).

In the first path of execution (on a uniprocessor) we execute P so it now owns A. Then we start process Q who now owns B. So now P is stuck waiting on Q to release B who is stuck waiting for P to release A. Ahhhhhh deadlock nooo.

Note: this graph is odd, axis ticks are lines to execute and arrows are paths of execution.

Deadlock Avoidance



Progress of Q
Release A
Release B
Get A
Get B
A Required
B Required
1
2
3
4
5
6
P and Q want A
P and Q want B
Progress of P
Get A
Release A
Get B
Release B
A Required
B Required
= both P and Q want resource A
= both P and Q want resource B
= possible progress path of P and Q.
Horizontal portion of path indicates P is executing and Q is waiting.
Vertical portion of path indicates Q is executing and P is waiting.

Figure 6.3 Example of No Deadlock [BACO03]

A way to get around this is to just attempt to avoid deadlocks in the first place. Think about the order of resource use to design it so that no one has a hold and wait situation. In this case no consecutive get commands.

Better would be for P to get(A), release(A), get(B), release(B) and do the same thing on process Q.

Reusable & Consumable Resources

Reusable:

- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Deadlock occurs if each process holds one resource and requests the other *(in case only one resource element exists)*
- … or a large amount in small chunks

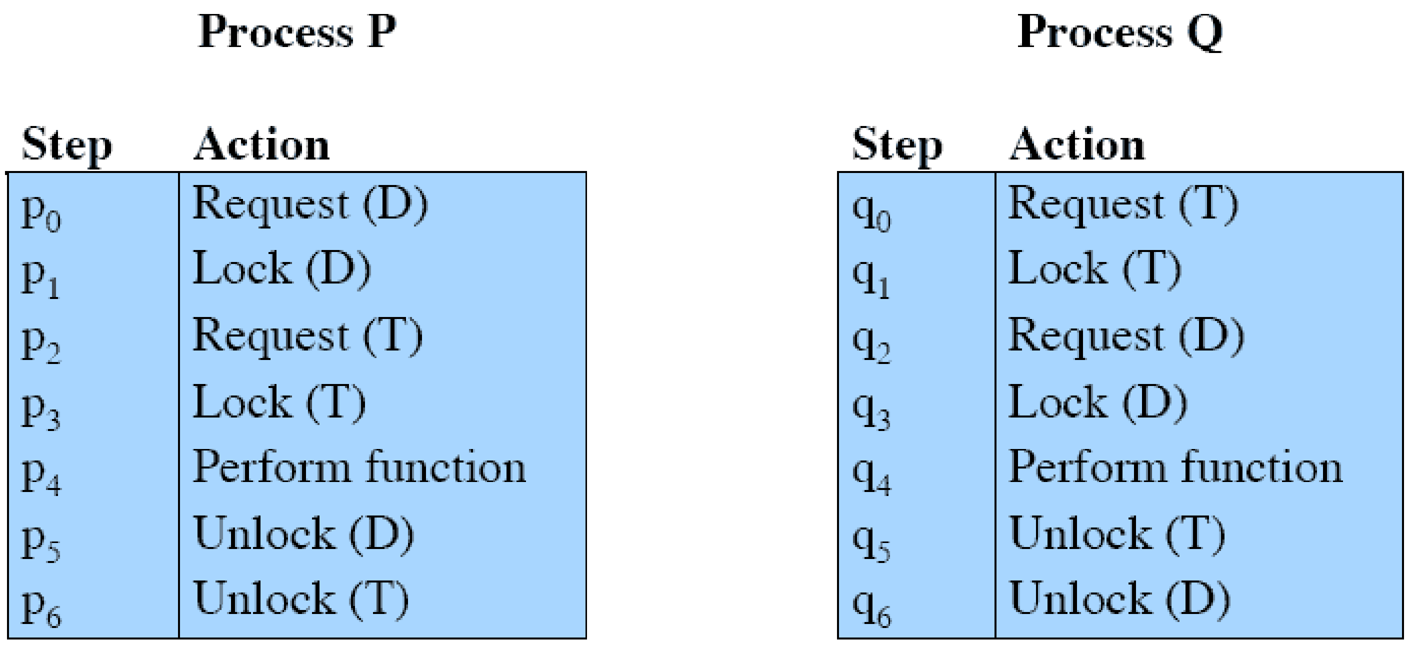
Consumable:

- Created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information in I/O buffers
- Deadlock may occur if a Receive() message is blocking

There are two types of resources to be blocked on, reusable and consumable.

Reusable: IO, main and secondary memory, devices, data structures. Dead lock occurs if each processn holds and then requests, only one element exists Consumable: created and destroyed elements. Deadlock occurs if receive signal is blocked.

Reusable Resources



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

Deadlock occurs if: p0 p1 q0 q1 p2 q2

These are just resources that dont go away as you use them. These you have to release before you request again if you want assurance that a deadlock wont occur.

Reusable Resources

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

```
P1
. . .
Request 80 Kbytes;
. . .
Request 60 Kbytes;
```

```
P2
. . .
Request 70 Kbytes;
. . .
Request 80 Kbytes;
```

- Deadlock occurs if both processes progress to their second request

In this example we run out of memory no matter the order of execution.

Consumable Resources

- Deadlock occurs if receives blocking

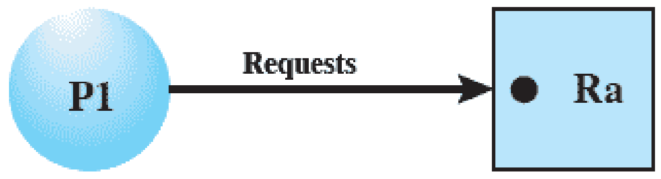
```
P1
. . .
Receive(P2);
. . .
Send(P2, M1);
```

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

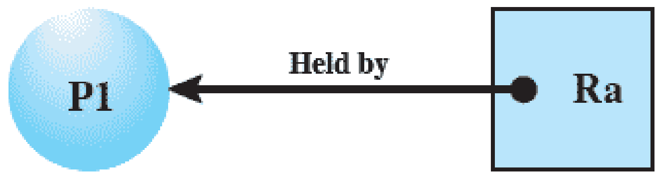
Here is an example of message passing. If each message queue is empty we have a deadlock because neither program can reach the point at which they send the message.

Resource Allocation Graphs

- Directed graph that depicts a state of the system of resources and processes



(a) Resouce is requested



(b) Resource is held

Squares are resources and circles are processes.

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 awaiting assignment of others
- No preemption (wrt. resources)
  - No resource can be forcibly removed form a process holding it
  - Requires rollback mechanism or saving state

Mutual exclusion is the cause of deadlocks since we block on resources, if we dont have mutual exclusion everyone just accesses willy nilly. Deadlocks also need to have a instance where a process holds onto a resource for more than one cycle to give other processes time to attempt to get it. We also need to not have preemption of resource so we cannot force an resource from an entity.

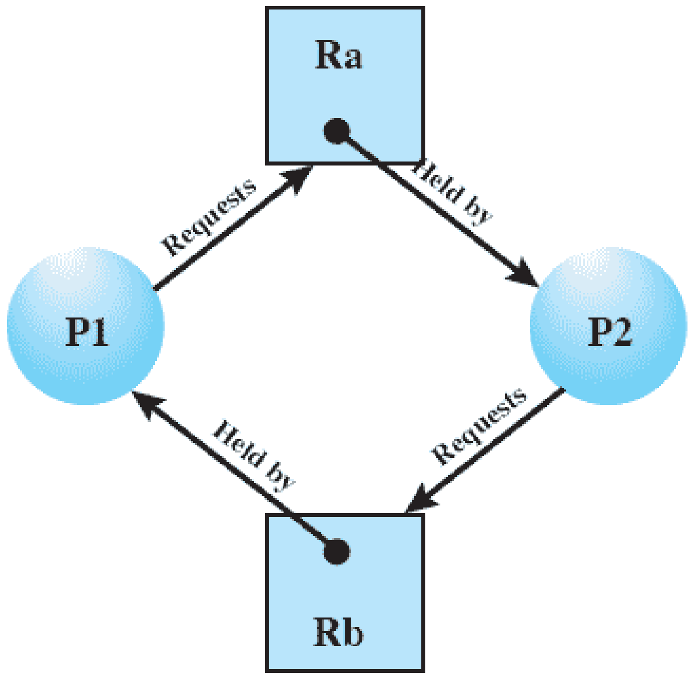
Conditions for Deadlock

- 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

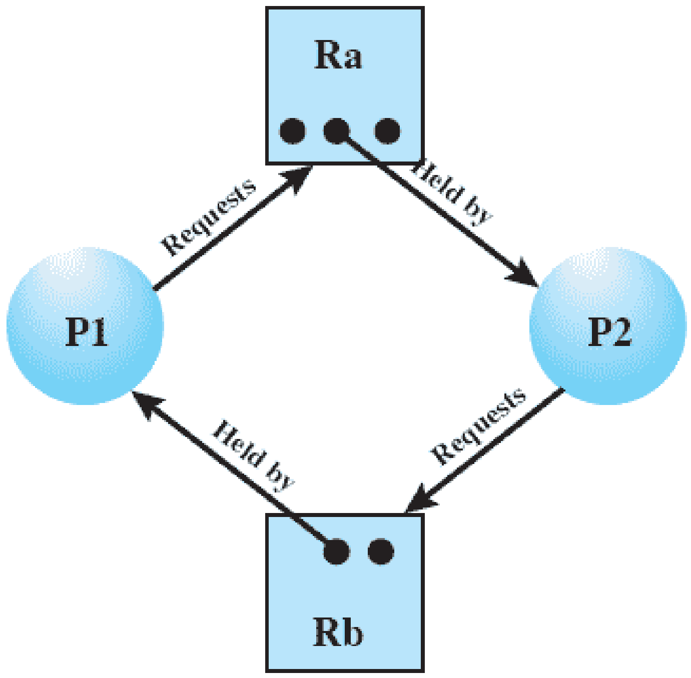
This is the main condition for a deadlock. One process must hold a resource that is needed by a process that holds a resource the initial process needs.

If we are missing any of the above conditions we wont get deadlocks.

Resource Allocation Graphs



(c) Circular wait



(d) No deadlock

Resource Allocation Graphs

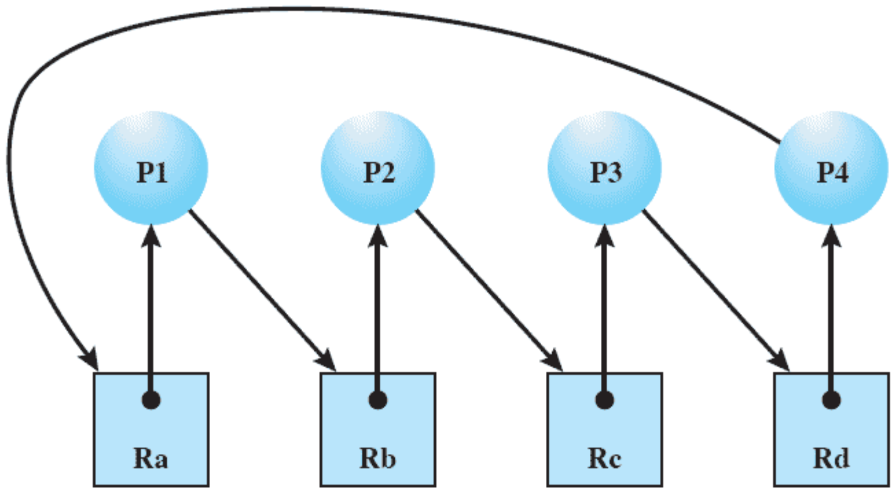


Figure 6.6 Resource Allocation Graph for Figure 6.1b

Possibility of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait

Necessary conditions

Existence of Deadlock

- Mutual Exclusion
- No preemption
- Hold and wait
- Circular wait

Sufficient conditions

Deadlock Prevention

... Design the system in such a way that no deadlock can occur. (remove one of the conditions)

- Mutual Exclusion
  - Must be supported by the OS
  - Can we remove this?
- Hold and Wait
  - Require a process request all of its required resources at one time (e.g, Ravenscar profile)

Hold and Wait: Try to allocate all of the resources needed at beginning. This also means that we dont waste anytime.

Deadlock Prevention

- No Preemption
  - Process must release resource and request again if it’s denied
  - OS may preempt a process to require it releases its resources (=> no two processes have same priority)
  - Requires saving and restoring state
- Circular Wait
  - Define a linear ordering of resource types
  - Lock resources following the ordering

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 process requests

Instead of attacking crude assumptions we try to make descisions at runtime. Banker's algorithm avoids incremental requests for resource allocation is an example of one such algorithm

Two Approaches to Deadlock Avoidance

1. Do not start a process if its demands might lead to deadlock
   – If the sum of all requested resources exceeds the resource budget, then don’t admit the process.
   – Problems of this approach?

2. Do not grant an incremental resource request to a process if this allocation might lead to deadlock

Make a descision based on if a process should be started. Requires upfront knowledge of what resources are required by that resource (R). Need to keep track of the unallocated resources (V). C (claim matrix) keeps table of processes and the total resources they will need (processes down, resources across) with claims. A (allocation matrix) keeps table of processses and resources allocated to them (process down, resource across) with allocations. We have m processes and n resources, so C_{mn} is the max allocation of R_n to P_m and A_{mn} is current allocation of R_n to P_m.

Rules.

1. $R_i = v_j + \sum A_{ij}$
2. $C_{ij} =\leq R_j$
3. $A_{ij} \leq C_{ij}$

Disadvantages

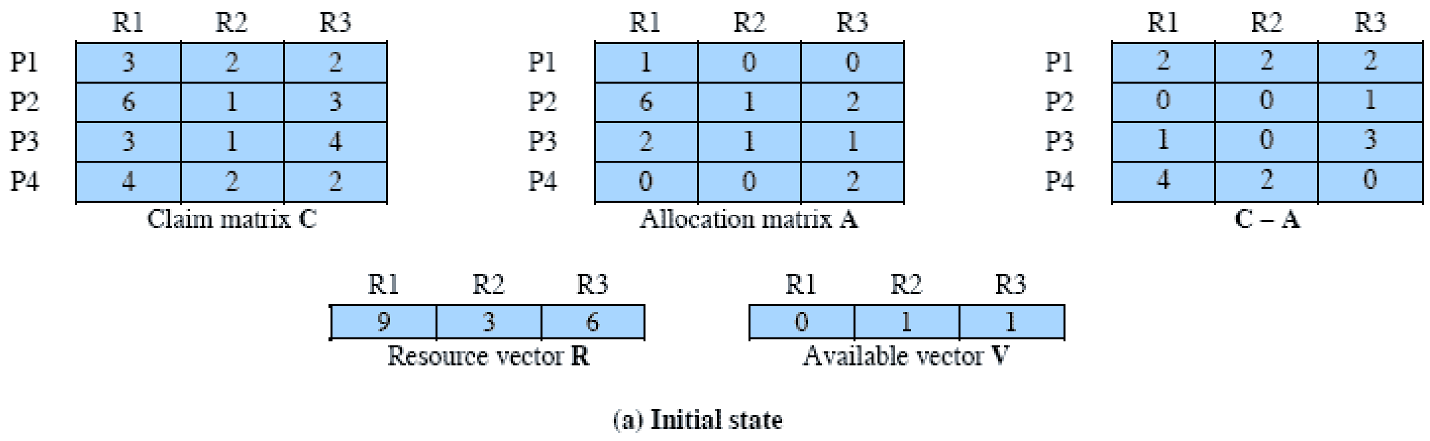
- fixed number of resources (doesnt work for consumable resources)
- must state max resources upfront which many processes dont know at their start
- independent processes (doesnt account for processes that must wait on each other)
- no processes terminating while holding resources

Dont start process if $C_{(n+n)} + \sum C_{ij} > R_j$ basically dont start the process if it requires resources that are already in use.

Resource Allocation Denial

- Referred to as the banker's algorithm
- State of the system is the current allocation of resources to process
- **Safe state** is where there is at least one sequence that does not result in deadlock
- **Unsafe state** is a state that is not safe
- <u>Goal: always have a safe state</u>

Are we in a safe 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	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**

Every time you make a resource request you update matrices and keep going to the end to see if you have a safe state (all allocation are cleared).

ex) The only process we can start with is P_2 because the available vector v shows what is available and the matrix c-a show what each process will request, only process two is cleared. So we run process two to completion.

Determination of a Safe State

	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

Here every resource allocated to R_2 get put on the available vector and update its row in c-a.

Now we can run any program. We repeate this process until we see that every process can execute completely so we know we have a safe state.

Determination of a Safe State

Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

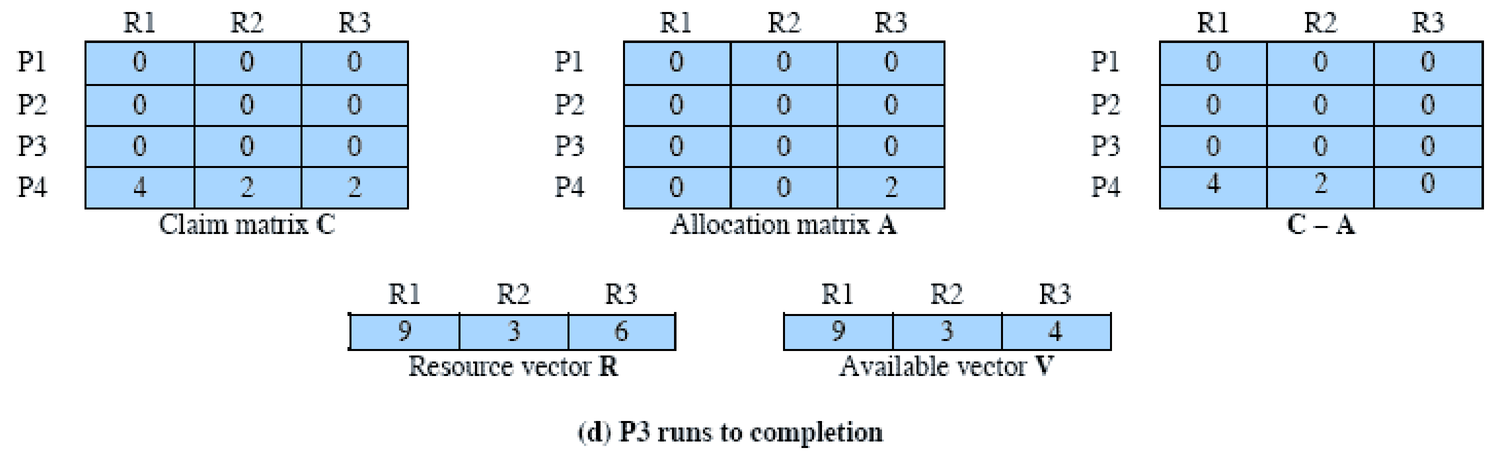
R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
7	2	3

(c) P1 runs to completion

Determination of a Safe State



Claim matrix **C**

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

Allocation matrix **A**

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

C – A

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

Resource vector **R**

R1	R2	R3
9	3	6

Available vector **V**

R1	R2	R3
9	3	4

(d) P3 runs to completion

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**

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

We see here from the initial state the the only process we can run is P_2 making available = [0,1,1] (we took the incremental resources 1 x R_1 and 1x R_3 away to allocate them to P_2). This step is done because of incremental allocation (we dont go to termination and return the resources instead we try to start a new process between incremental allocations). Now we cannot start any process safely, called a unsafe state.

Deadlock Avoidance Logic

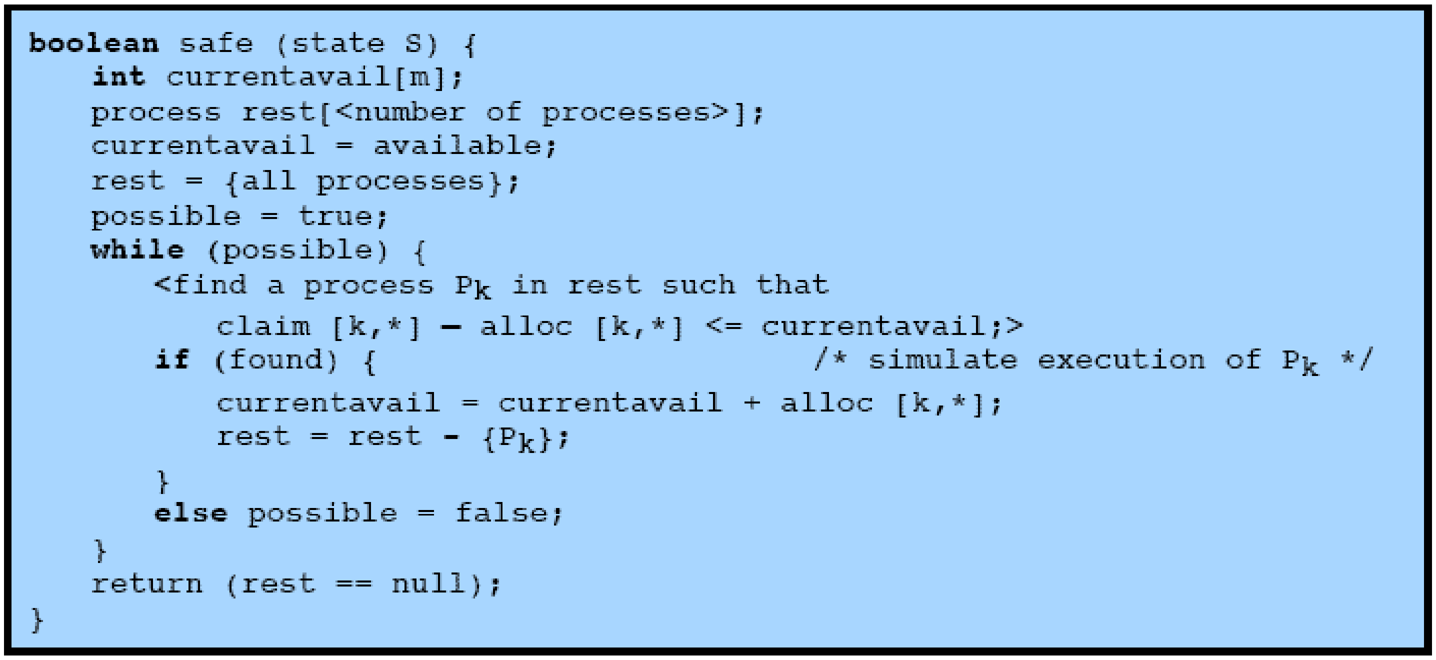
```
struct state {
       int resource[m];
       int available[m];
       int claim[n][m];
       int alloc[n][m];
}
```

(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 >;
}
```

(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

Deadlock Avoidance

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

To use the baker We algorithm we need to fit the above criteria.

Deadlock Detection

	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**

We model outstanding requests using a simplified version ($Q = C - A$ at that instant) seen above. Now we run the same algorithm as last time. Want to find if deadlock and look for processes that can be deadlocked.

- mark all processes that have no resources allocated.
- w := V initialize to available vector
- look for a process that you can satisfy (room in available vector for values in Q), these are not in a deadlock state
- release all resources in available vector (like running to completion)
- repeat for all processes (unless a deadlock is reached)

Applied methodology of bankers algorithm but removed the need to state all required resources at the start. With this we can mock deadlock transactions before letting them run through.

Strategies Once Deadlock Detected

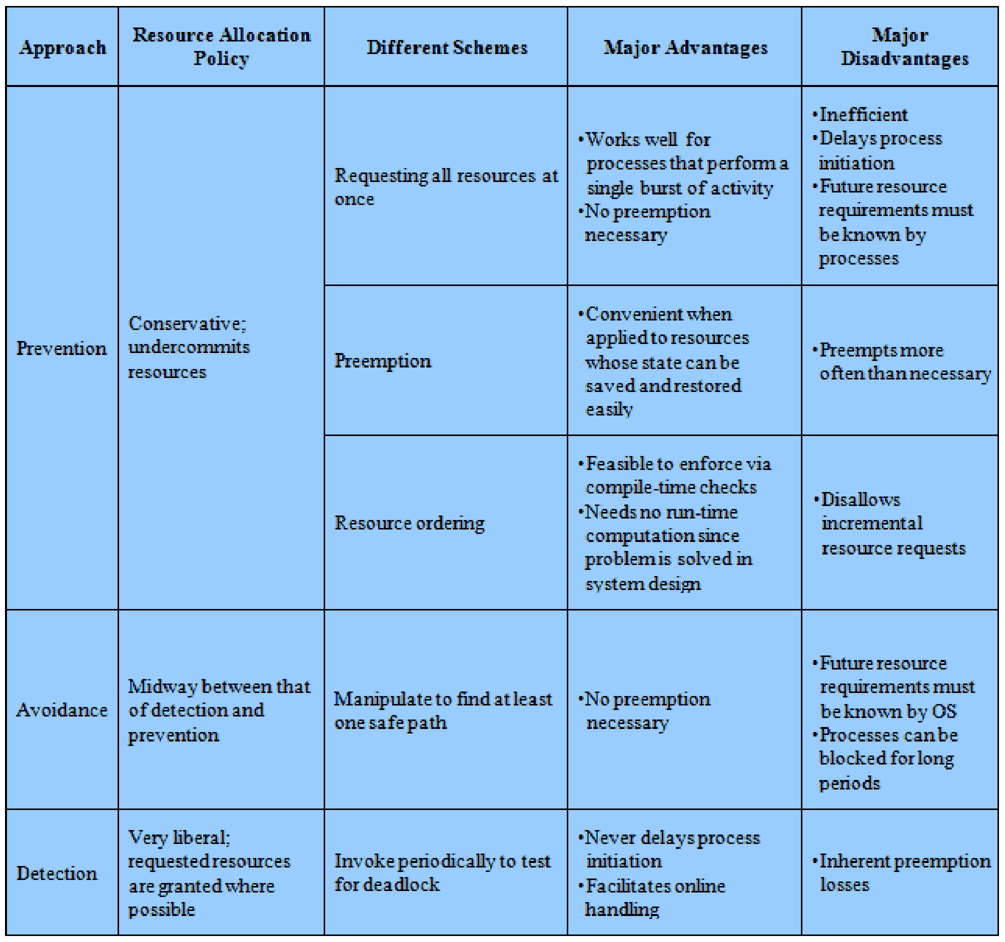
- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur

Strategies Once Deadlock Detected

- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Advantages and Disadvantages

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

Dining Philosophers Problem

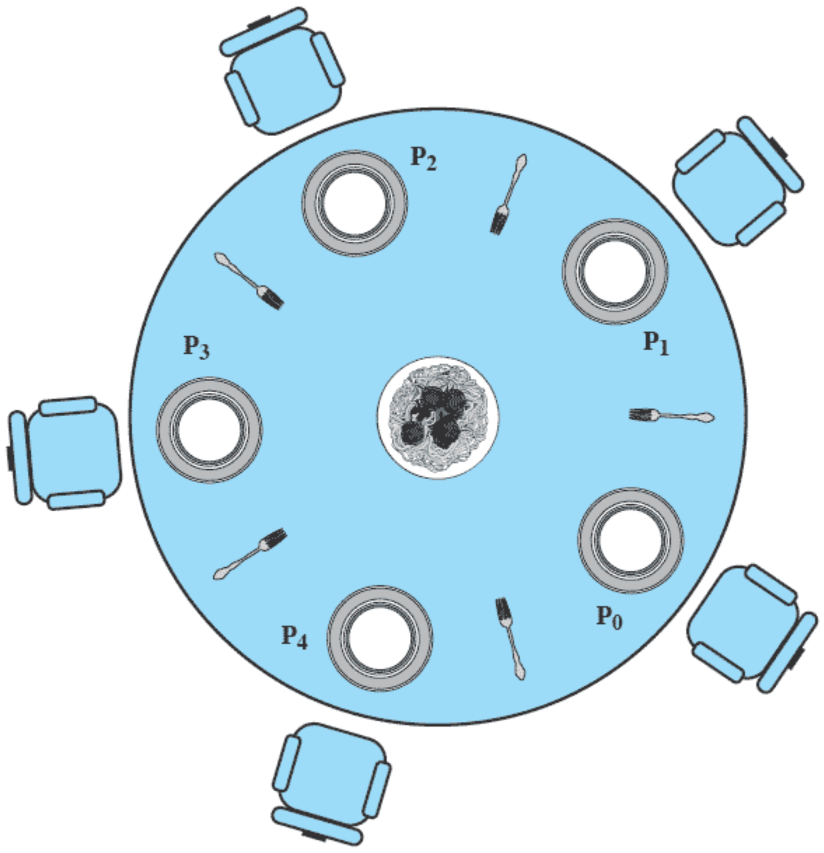
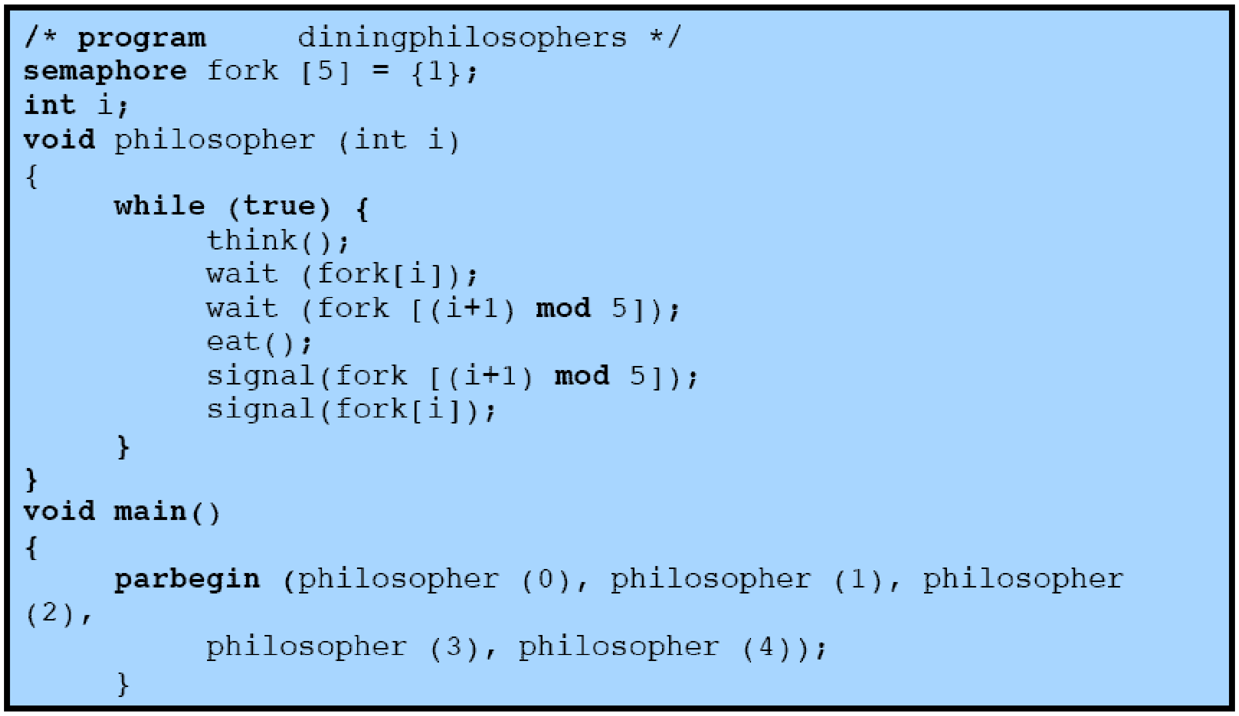


Figure 6.11 Dining Arrangement for Philosophers

A deadlock is reached when every philosopher wants to eat at the same time.

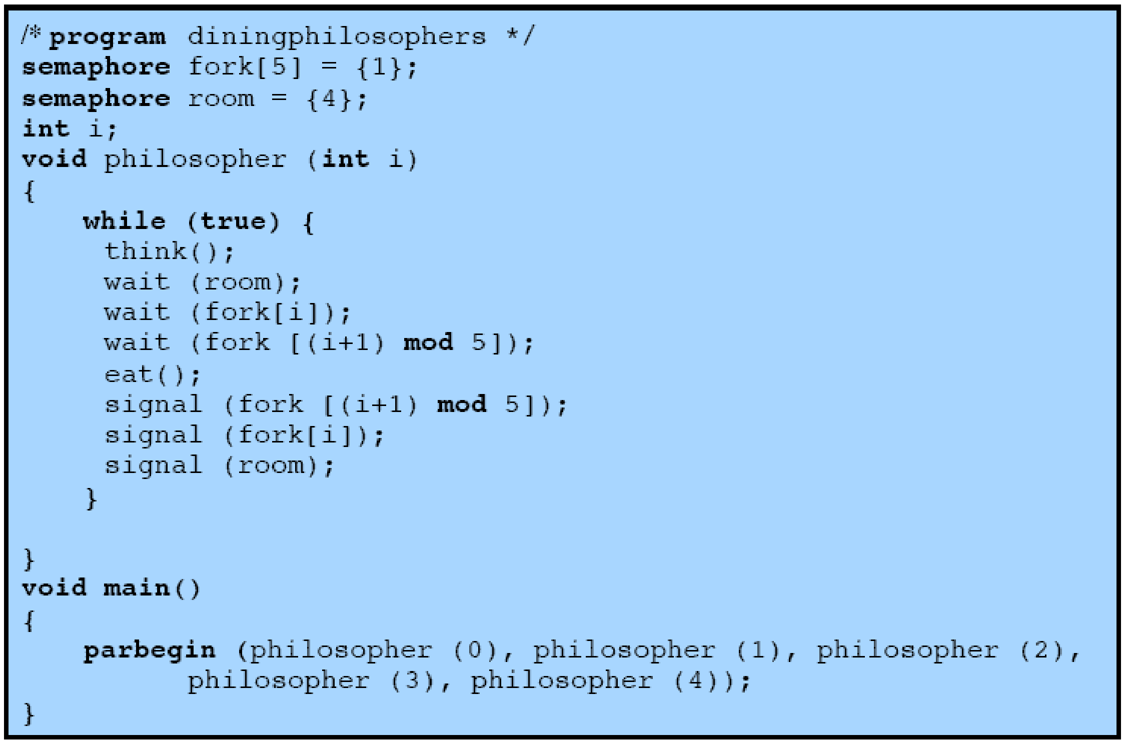
Dining Philosophers Problem



```
/* 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

One solution is to have a bouncer that only allows four people in at a time. We can do this by wrapping the logic or the philosophers in a semaphore that only allows four in at a time.

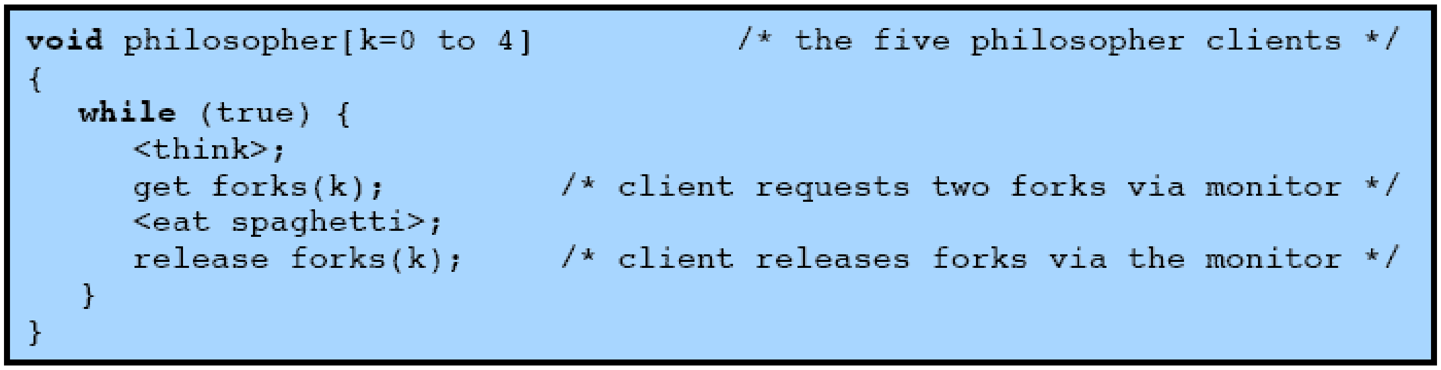
Dining Philosophers Problem

```
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]);
}
```

This is the monitor solution. Wrap get forks and release forks in it. When someone wants to pick up a fork you check for availability for the left fork and right fork and signal for availability.

Dining Philosophers Problem



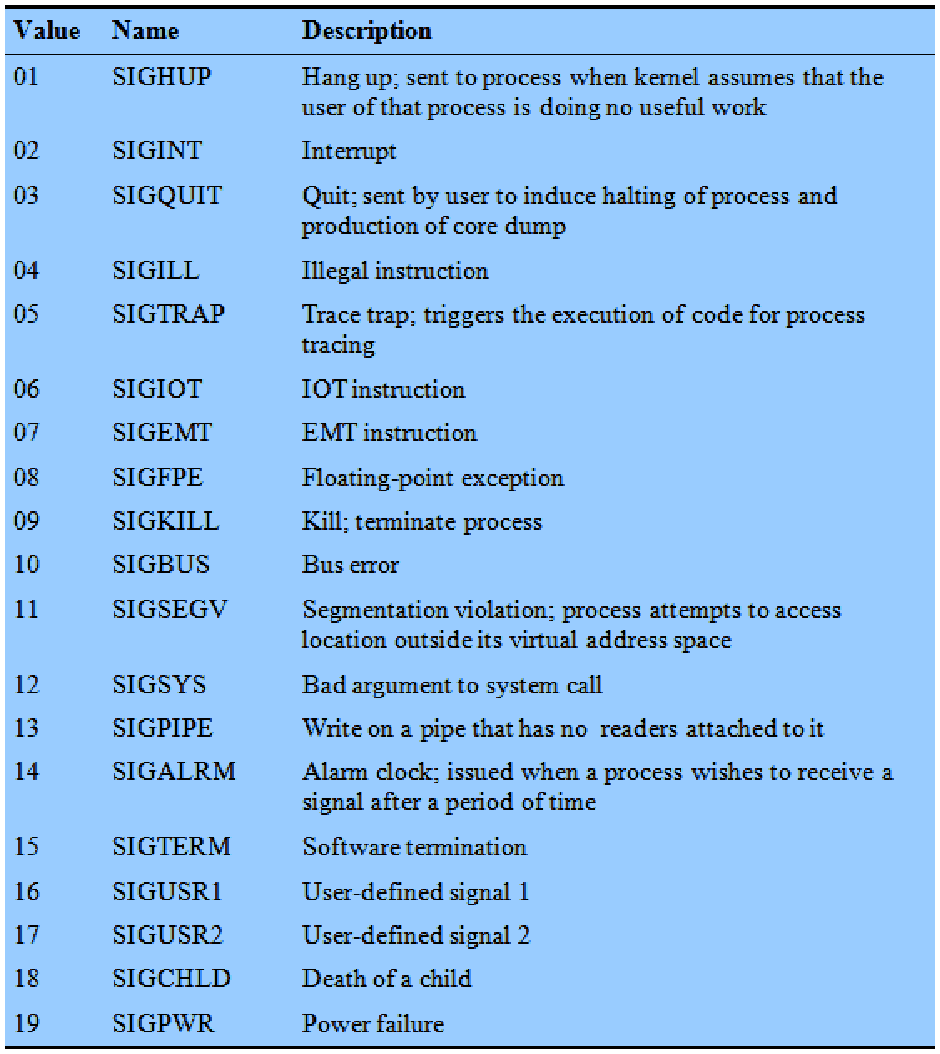
```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release forks(k);      /* client releases forks via the monitor */
   }
}
```

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

UNIX Concurrency Mechanisms

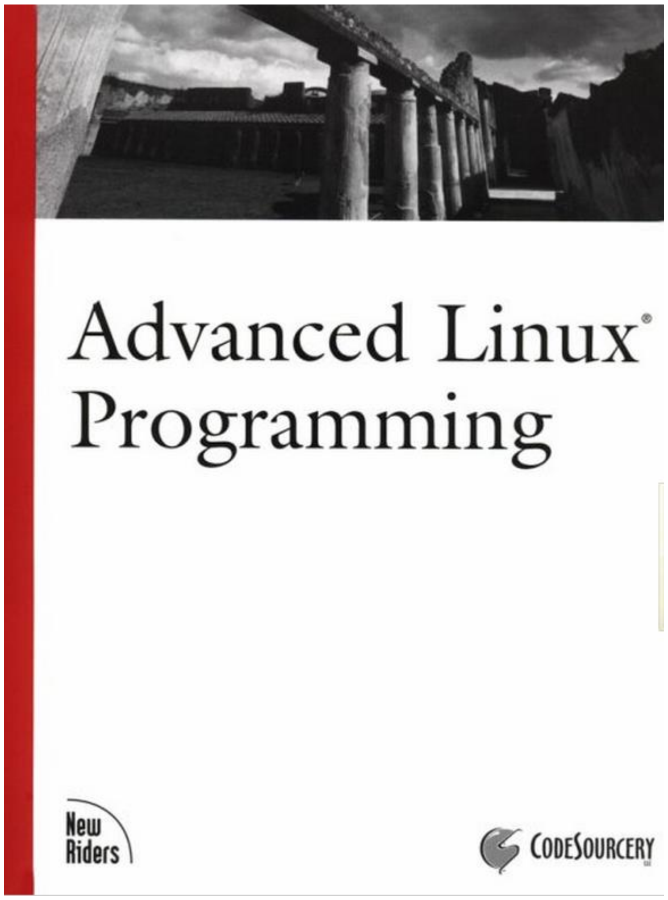
- Pipes: Producer/consumer data passing between two programs
- Messages
- Shared memory
- Semaphores
- Signals: implement asynchronous events

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

Try it out yourself…



http://www.advancedlinuxprogramming.com/