Process P

Step	Action
p_0	Request (D)
\mathbf{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)
\mathbf{q}_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
\mathbf{q}_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

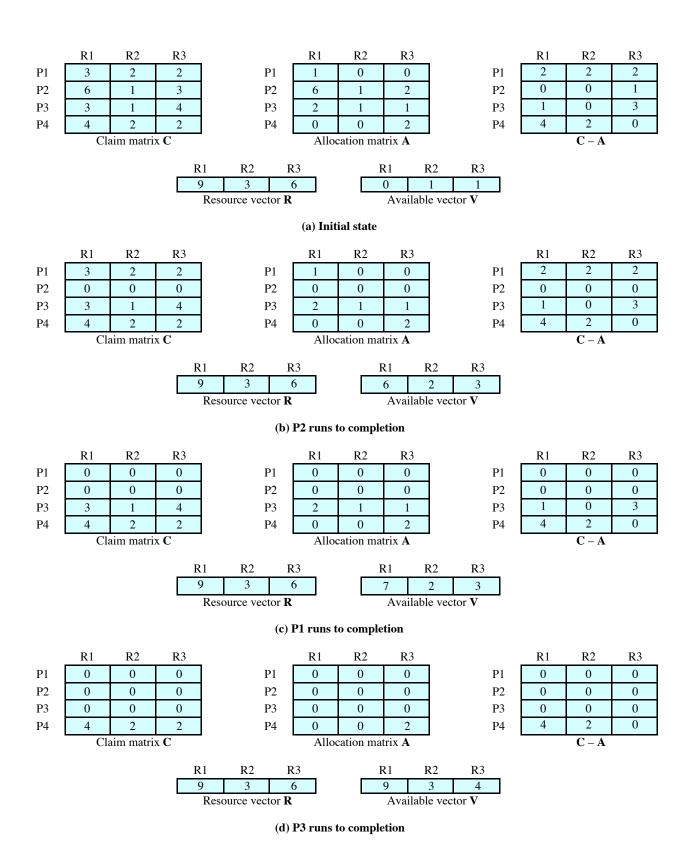
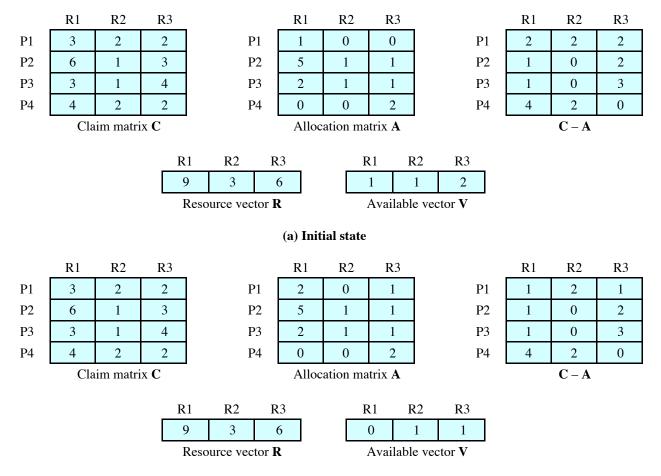


Figure 6.7 Determination of a Safe State



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

Figure 6.8 Determination of an Unsafe State

```
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

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

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

Figure 6.9 Deadlock Avoidance Logic

```
diningphilosophers */
/* program
semaphore fork [5] = {1};
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

```
/* 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

```
monitor dining controller;
                          /* condition variable for synchronization */
cond ForkReady[5];
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)
                               /* queue on condition variable */
     cwait(ForkReady[left]);
  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;
                            /* 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]);
```

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

```
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
  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]);
```

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