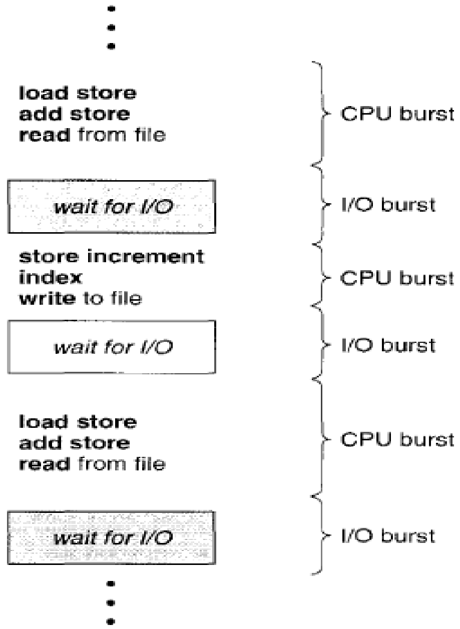
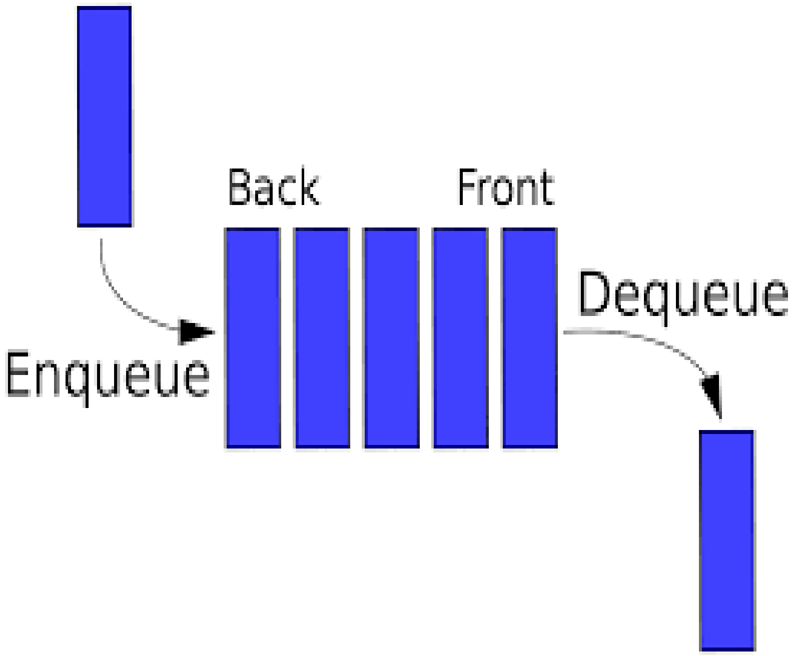
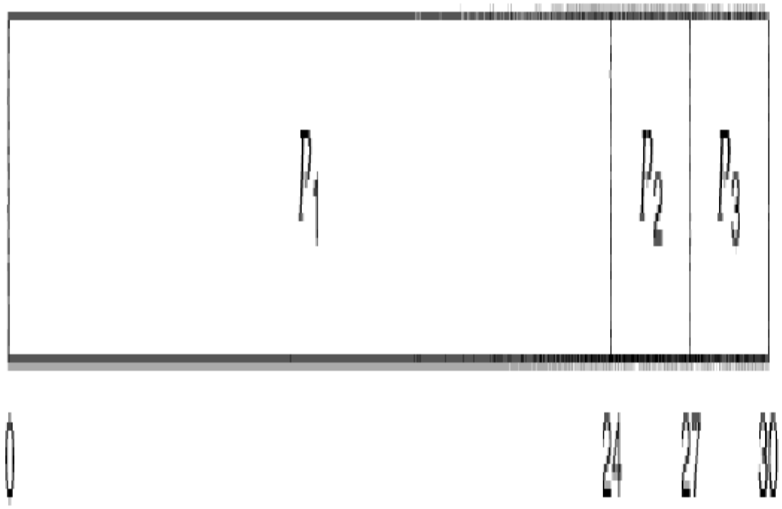


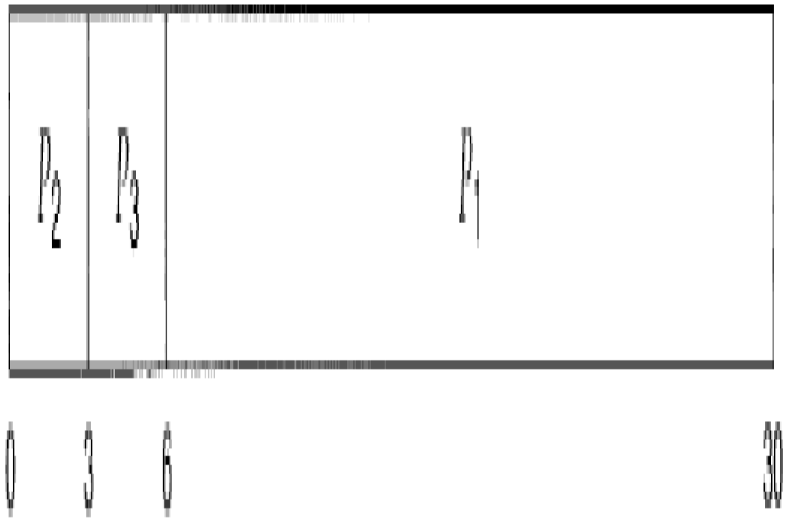
Eventually, the final CPU burst ends with a system request to terminate execution



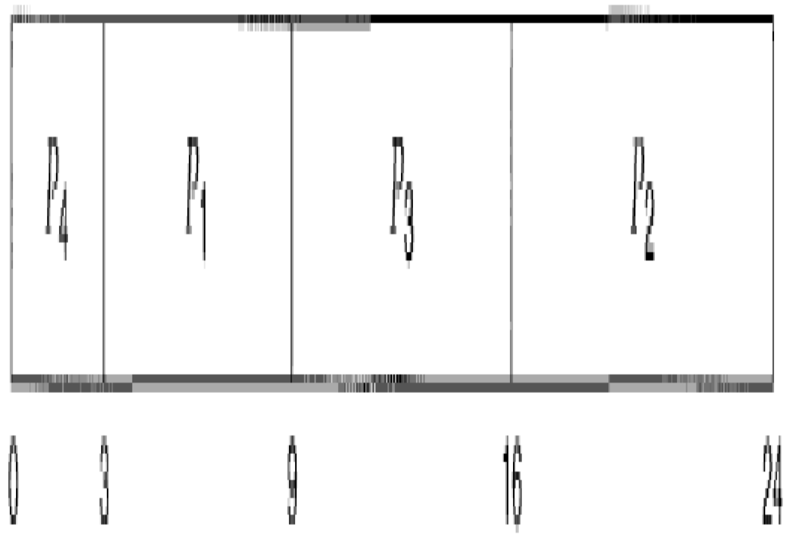
Scheduling Algorithms First-Come, First-Served Scheduling (FCFS) This is the simplest scheduling algorithm. The process that requests the CPU first is allocated the CPU first. The implementation of the FCFS scheduling is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process a

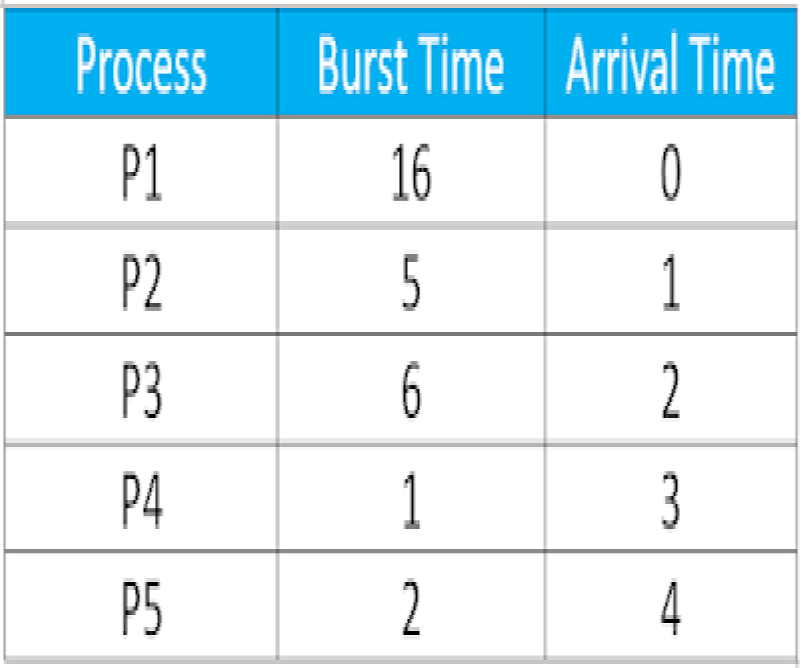


Gantt chart



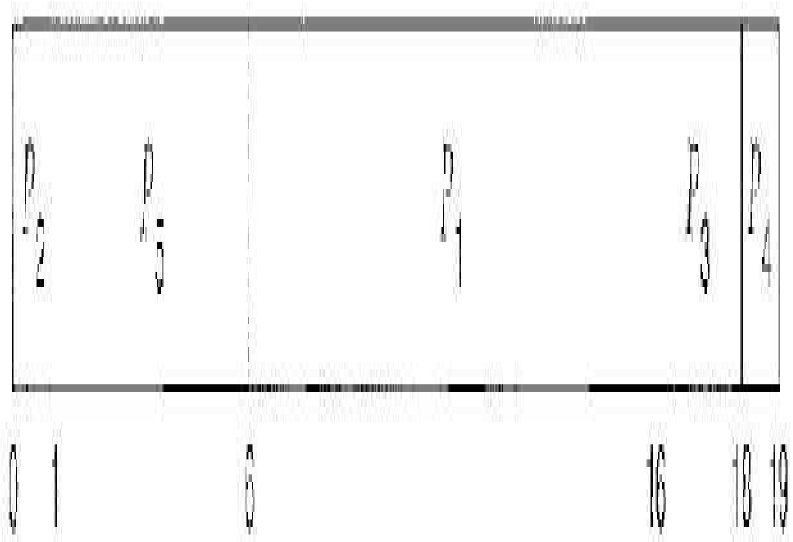
Waiting times


P4
P1
P3
P2
0
3
9
16
24

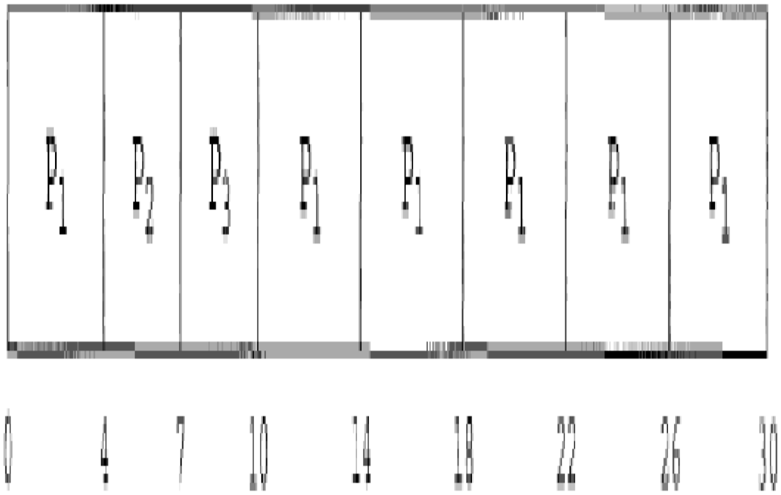


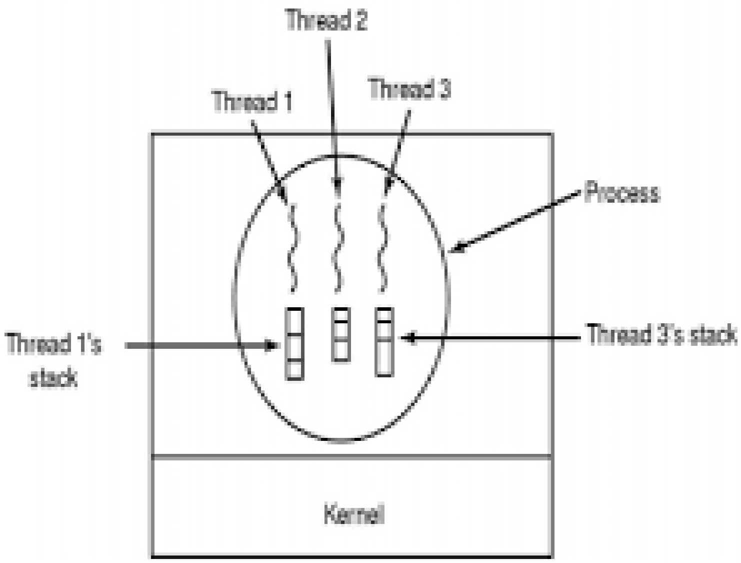
| Process | Burst Time | Arrival Time |
|---|---|---|
| P1 | 16 | 0 |
| P2 | 5 | 1 |
| P3 | 6 | 2 |
| P4 | 1 | 3 |
| P5 | 2 | 4 |

In Preemptive SJF Scheduling, jobs are put into the ready queue as they come. A process with shortest burst time begins execution. If a process with even a shorter burst time arrives, the current process is removed or preempted from execution, and the shorter job is allocated CPU cycle.



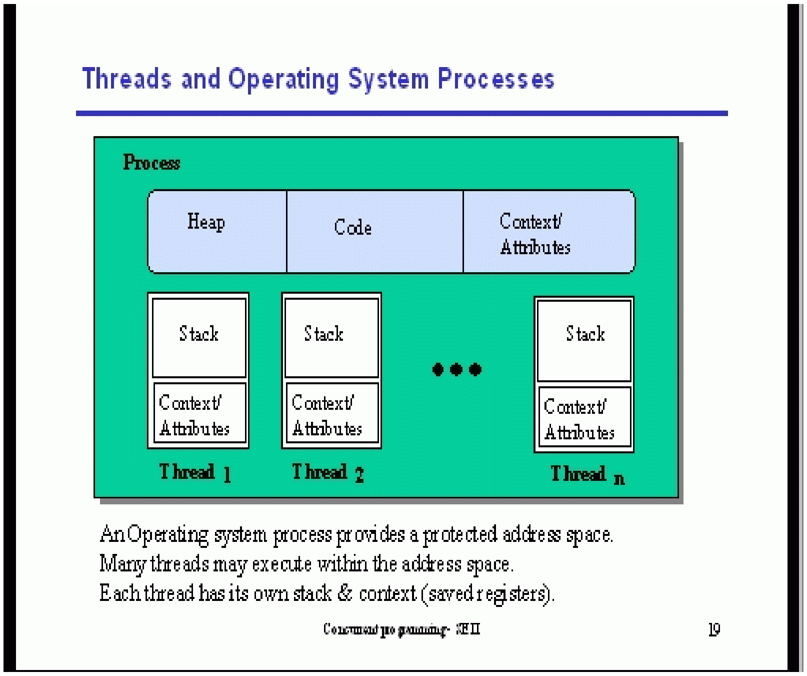
The waiting time is: P1=6; P2=0; P3=16; P4=18; P5=1. The average waiting time = (6+0+16+18+1)/5=8.2 msec.


P1
P2
P3
P1
P1
P1
P1
P1
0
4
7
10
14
18
22
26
30



**Each thread has its own stack.**

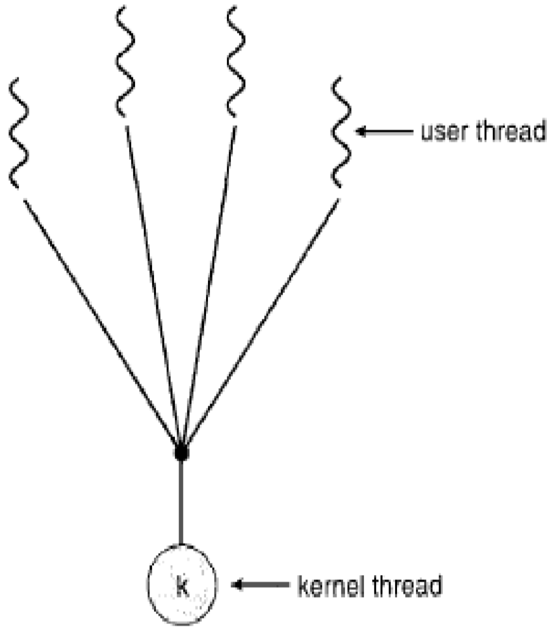
Single threaded and Multi-threaded processes

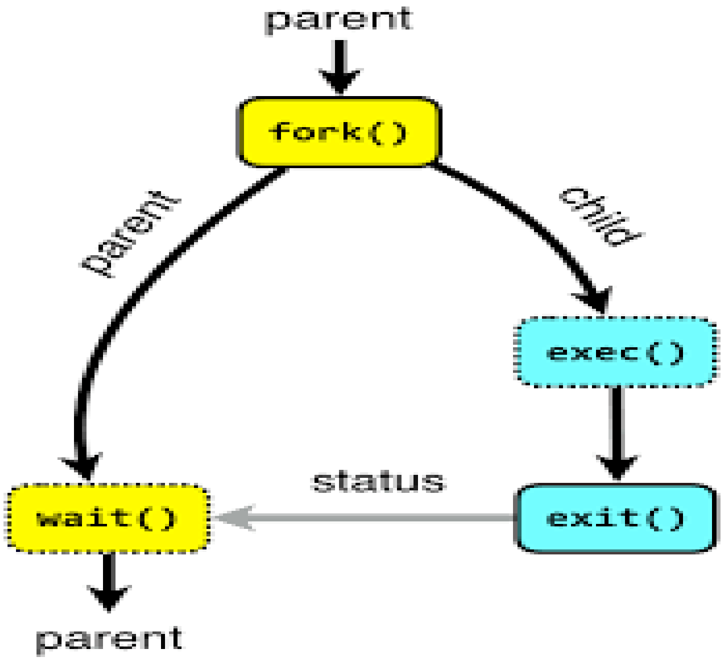


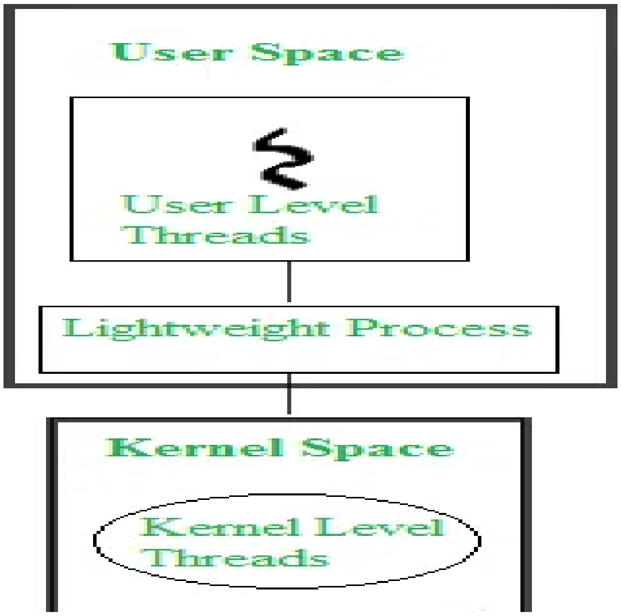
## Threads and Operating System Processes

An Operating system process provides a protected address space.
Many threads may execute within the address space.
Each thread has its own stack & context (saved registers).

Concurrent programming - SE II 19


user thread
k
kernel thread


parent
fork()
parent
child
exec()
status
wait()
exit()
parent



Systems implementing many to-many model place an intermediate data structure between the user and kernel threads. This data structure-typically known as a lightweight process (LWP) is shown in Figure. To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is kernel threads th

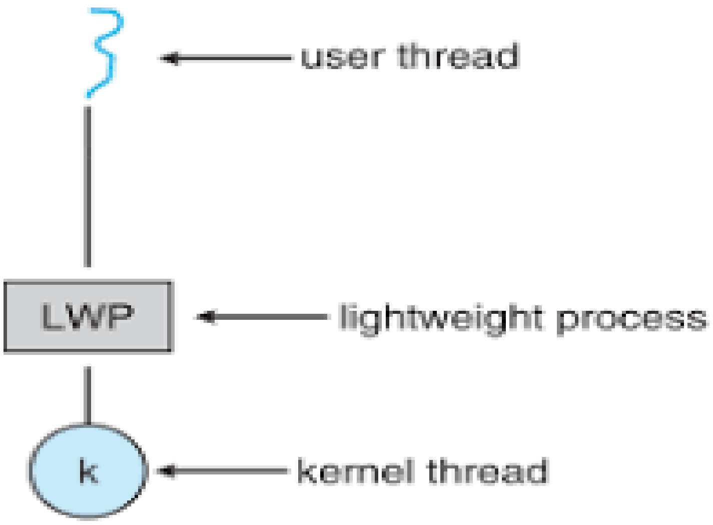
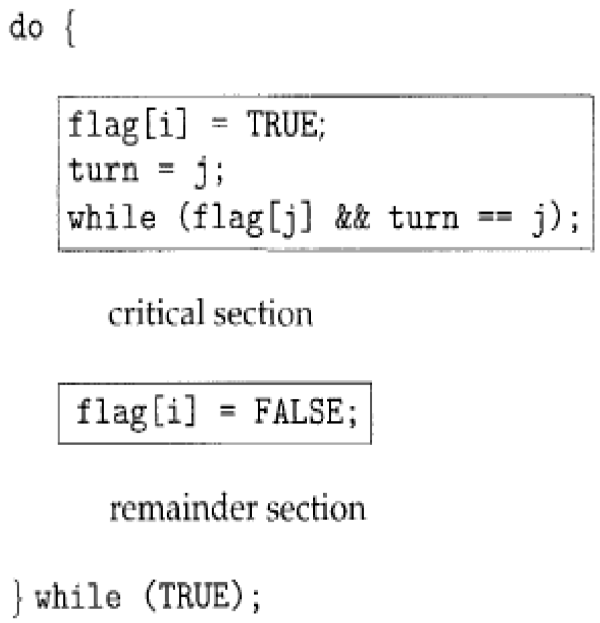


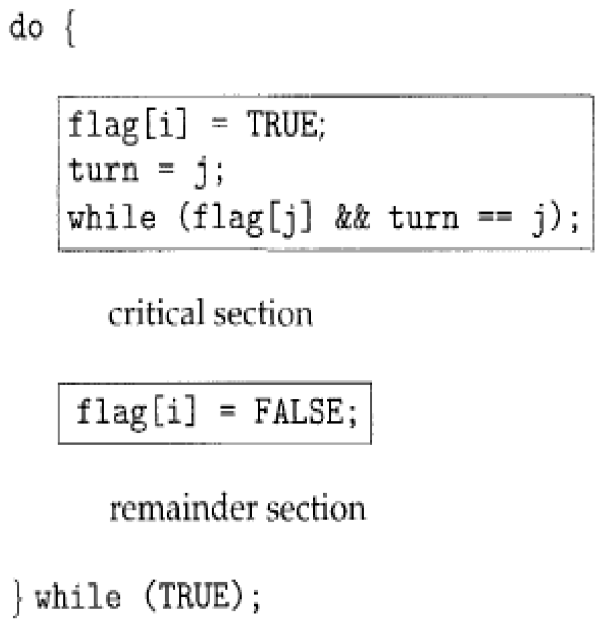
**Figure 4.13** Lightweight process (LWP).

Light Weight Process (LWP)



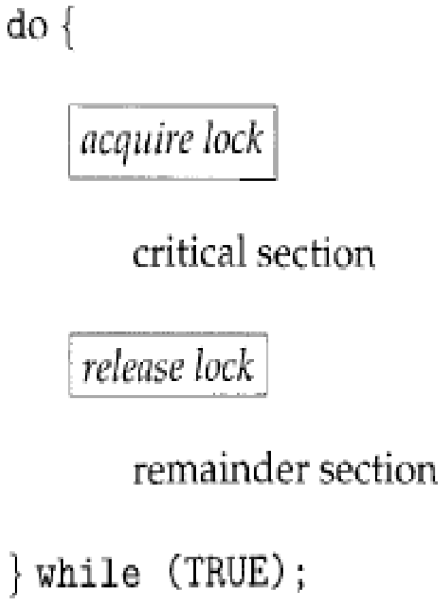
```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.



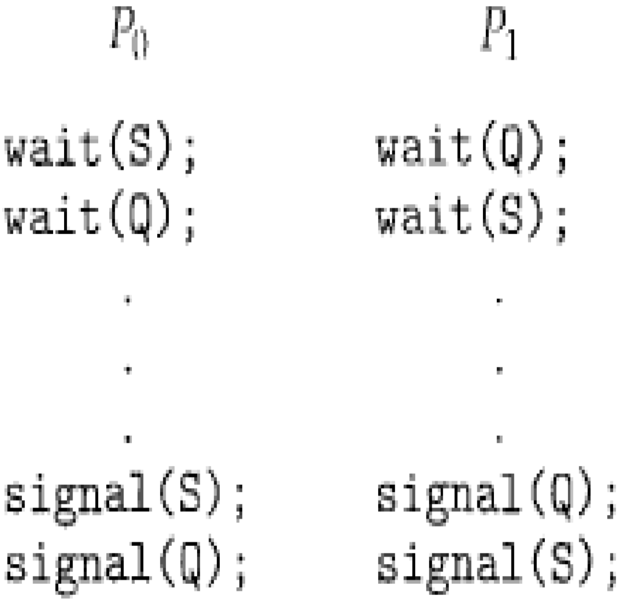
```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

The structure of process Pi in Peterson’s solution



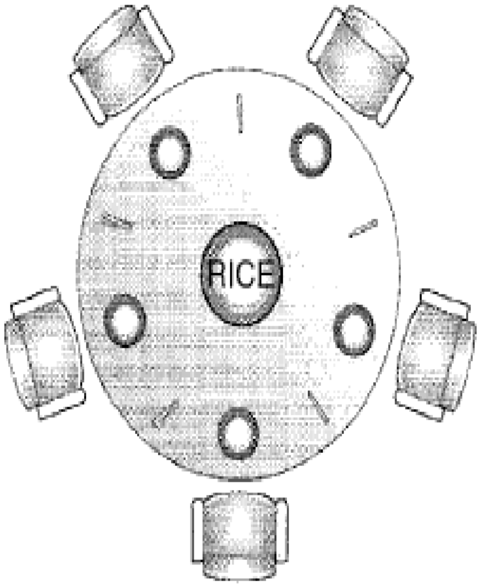
```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Solution to the critical section problem using locks.

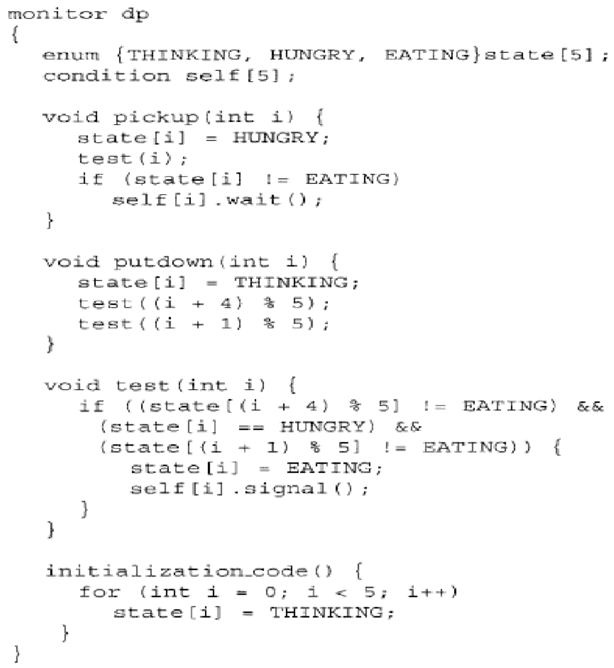


```
      P0               P1

wait(S);          wait(Q);
wait(Q);          wait(S);
    .                 .
    .                 .
    .                 .
signal(S);        signal(Q);
signal(Q);        signal(S);
```

Deadlocks and Starvation The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal 0 operation. When such a state is reached, these processes are said to be deadlocked.



The Dining-Philosophers Problem



```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
           state[i] = EATING;
           self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

A monitor solution to dining philosopher problem