Process Management

Scheduling in Real-Time Environment

- Real-time systems are generally categorized as **hard real-time** (absolute deadline that must be met) and **soft real-time** (missing an occasional deadline is undesirable but tolerable).
- Real-time behavior is achieved by dividing a program into a number of processes, each of which behavior is predictable and known in advance.
- These **processes** are generally **short lived** and can **run** to completion in well **under a sec**.
- It is the job of the scheduler to schedule these processes in such a way that all deadlines are met.
- The **events** that a real-time system may have to respond to can be further categorized as **periodic** (occur at regular interval) and **aperiodic** (occur unpredictably).
- Suppose, there are *m* periodic events and event *i* occurs with period P_i and requires C_i sec of CPU time to handle each event, then the load can be handled only if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- A process that fails to meet this condition can not be scheduled because the total amount of CPU time the process wants collectively is more than the CPU can deliver.

Process Management

- Real-time **scheduling algorithms** can be **static** (scheduling decision is made before the system starts running) or **dynamic** (scheduling decision is made at run time after the execution has started).
- **Thread Scheduling:**
  - **Round robin scheduling** and **priority scheduling** are the most commonly used scheduling algorithms.
  - For **user level threads**, as the kernel is not aware of the threads, it operates the processes as it does. The **thread scheduler** inside a process then choose which thread to run.
  - In **kernel level threads**, it can pick a particular thread to run.
  - It does not have to take into account which process the thread belongs to.

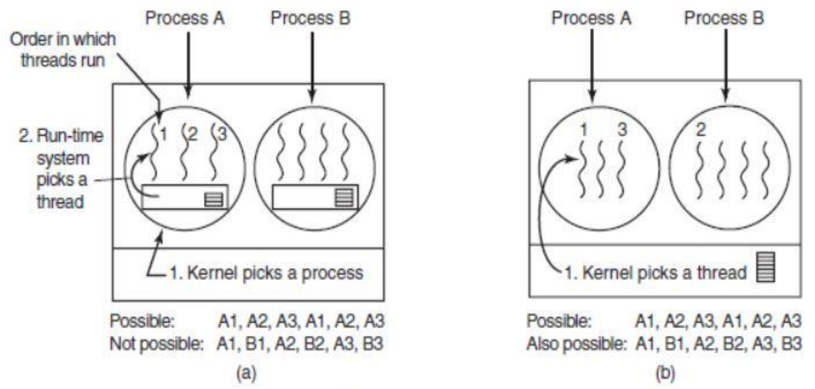

Process A
Process B
Order in which threads run
2. Run-time system picks a thread
1 2 3
1. Kernel picks a process
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3
(a)
Process A
Process B
1 3
2
1. Kernel picks a thread
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3
(b)

Figure 2-44. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Process Management

Classical IPC Problem

Dining Philosophers Problem

- The life of a philosopher consists of alternating periods of eating and thinking
- When a philosopher is sufficiently hungry, he tries to acquire both left and right forks, one at a time, in either order.
- If successful, he eats for a while, then puts down the forks and continues.
- So it is needed to make a system so that each philosopher never gets stuck.
- The obvious solution does not work. What happens if all five philosophers take their left fork simultaneously? None will be able to get their right forks, and the result is **deadlock**.

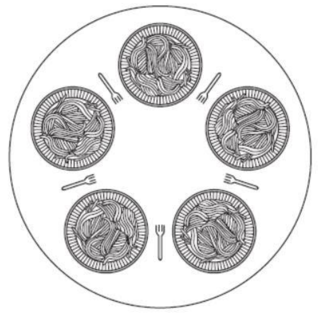
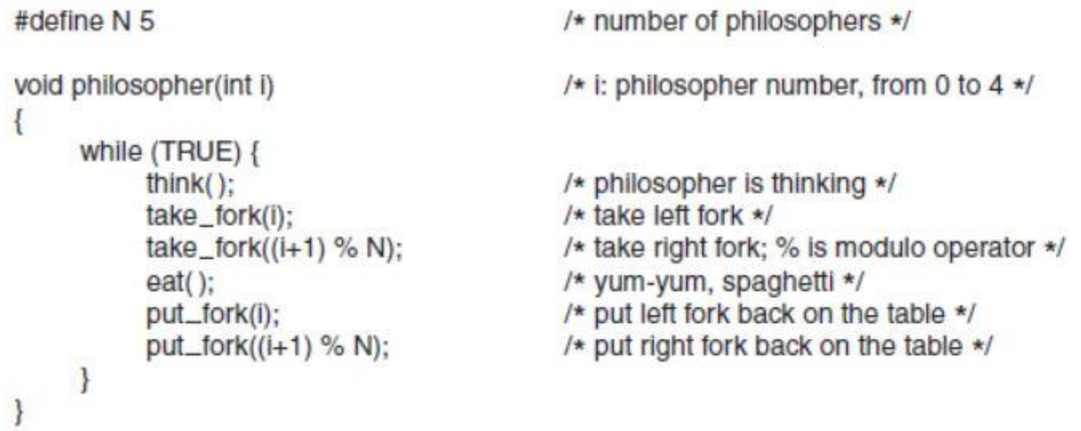


Figure 2-45. Lunch time in the Philosophy Department.

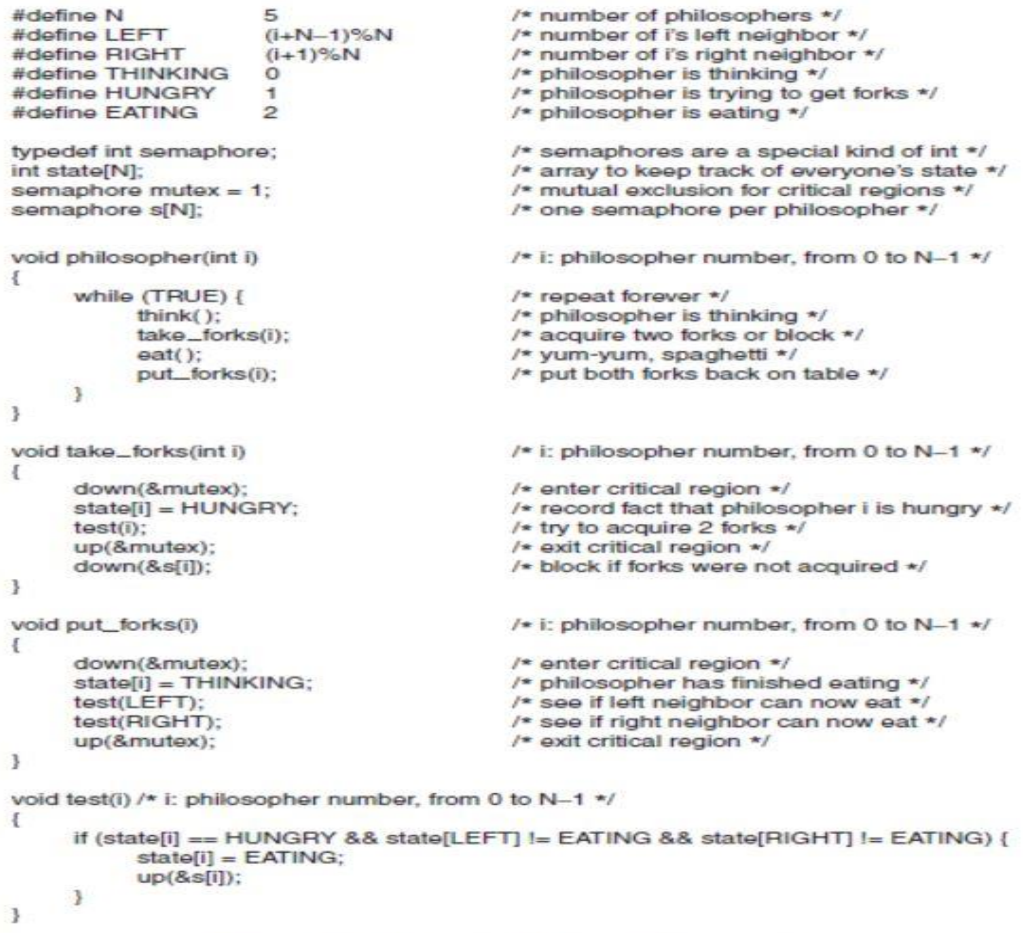

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                  /* philosopher is thinking */
          take_fork(i);              /* take left fork */
          take_fork((i+1) % N);      /* take right fork; % is modulo operator */
          eat( );                    /* yum-yum, spaghetti */
          put_fork(i);               /* put left fork back on the table */
          put_fork((i+1) % N);       /* put right fork back on the table */
     }
}
```
Figure 2-46. A nonsolution to the dining philosophers problem.

Process Management

- The modification of the procedure does not work either. After taking the left fork, the procedure checks to see if the right fork is available. If not available, the philosopher puts down the left fork, waits for some time, and repeats the same process. Problem is, if all the five philosophers do this process simultaneously, this will stay forever.
- A situation where all the programs continue to run indefinitely but fail to make any progress is called **starvation**.
- One solution is to use semaphore. Before acquiring a fork, a philosopher will set semaphore value to down, and when he is done will set up to semaphore. But in this case, only one philosopher will be able to eat at a time.
- The solution that allows maximum parallelism is given in figure 2.47.

Process Management



```
#define N           5                  /* number of philosophers */
#define LEFT        (i+N−1)%N          /* number of i's left neighbor */
#define RIGHT       (i+1)%N            /* number of i's right neighbor */
#define THINKING    0                  /* philosopher is thinking */
#define HUNGRY      1                  /* philosopher is trying to get forks */
#define EATING      2                  /* philosopher is eating */

typedef int semaphore;                 /* semaphores are a special kind of int */
int state[N];                          /* array to keep track of everyone's state */
semaphore mutex = 1;                   /* mutual exclusion for critical regions */
semaphore s[N];                        /* one semaphore per philosopher */

void philosopher(int i)                /* i: philosopher number, from 0 to N−1 */
{
     while (TRUE) {                    /* repeat forever */
          think( );                    /* philosopher is thinking */
          take_forks(i);               /* acquire two forks or block */
          eat( );                      /* yum-yum, spaghetti */
          put_forks(i);                /* put both forks back on table */
     }
}

void take_forks(int i)                 /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                     /* enter critical region */
     state[i] = HUNGRY;                /* record fact that philosopher i is hungry */
     test(i);                          /* try to acquire 2 forks */
     up(&mutex);                       /* exit critical region */
     down(&s[i]);                      /* block if forks were not acquired */
}

void put_forks(i)                      /* i: philosopher number, from 0 to N−1 */
{
     down(&mutex);                     /* enter critical region */
     state[i] = THINKING;              /* philosopher has finished eating */
     test(LEFT);                       /* see if left neighbor can now eat */
     test(RIGHT);                      /* see if right neighbor can now eat */
     up(&mutex);                       /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N−1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

Figure 2-47. A solution to the dining philosophers problem.