OPERATING SYSTEMS

by
Marwa Yusuf

Lecture 11
Sunday 29-11-2020

Chapter 2 (2.4.4 to 2.5.2)

Processes and Threads

Scheduling in Real Time Systems

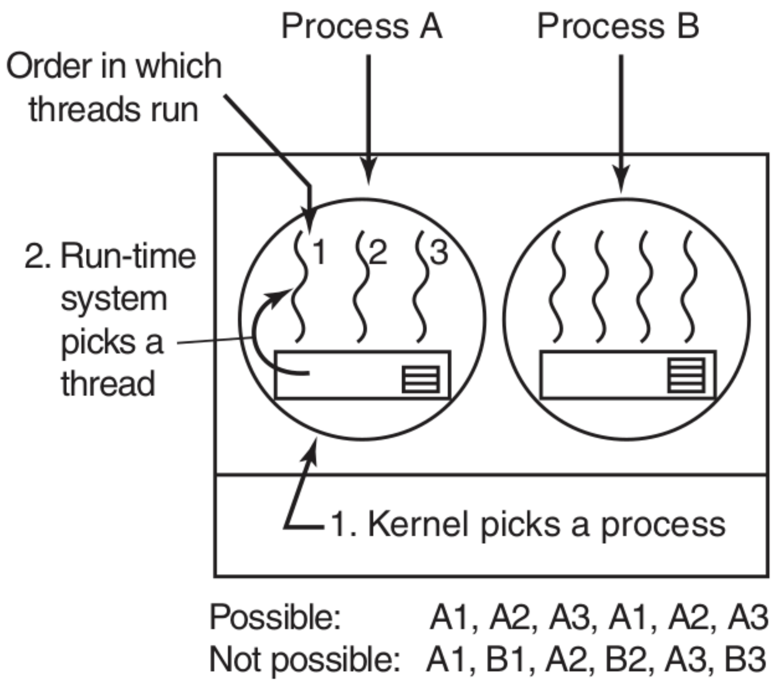
- Time is paramount.
- Usually: receive external input, process it and act within a constrained time.
- Ex: compact disc player, patient monitoring in a hospital intensive-care unit, the autopilot in an aircraft, and robot control in an automated factory.
- Hard vs. Soft real time systems.
- Program divided into processes (known and predictable in advance), usually short lived (within 1 sec). The scheduler is responsible for meeting deadlines.
- **Periodic** vs **aperiodic** events.
- **Schedulable** system: $\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant 1$
  - Ex: Processes periods: 100, 200, 500 msec ---- time per event: 50, 30, 100
  - 0.5 + 0.15 + 0.2 <= 1 → OK
  - Fourth process with period 1 sec: Ok as long as lower than 150 msec time per event.
  - Assumption: switching time is negligible.
- Static scheduling (in advance perfect information about work to be done and deadlines), vs. dynamic scheduling.

Policy vs. Mechanism

- What if a parent process has info about its children and can take decisions about their scheduling? Scheduler does not accept input from processes.
- Sol: separate policy from mechanism. Parameterized algorithm where parameters are filled by processes.
- Ex: Database application
  - Mechanism: Priority scheduling.
  - Policy: Parent process assigns values to children.

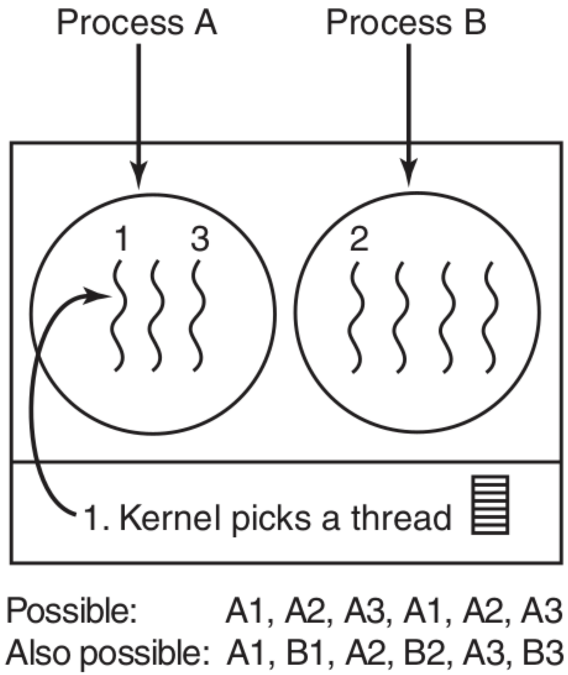
Thread Scheduling

- User-level threads:
  - Threads scheduler (within the threads runtime system) schedules threads.
  - Anti-social threads do not affect other processes.
  - Commonly round robin or priority.
  - No clock to interrupt, but they are supposedly cooperating.



Thread Scheduling (cont.)

- Kernel-level threads:
  - The system scheduler chooses the next thread to run.
  - May (or may not) take into account the process of this thread.



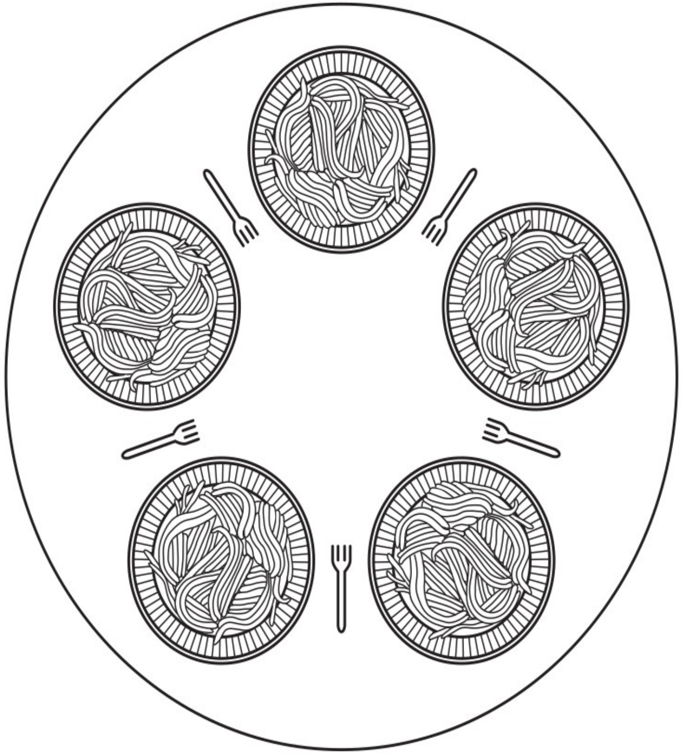
Thread Scheduling (cont.)

- Switching is faster in user-level threads.
- A thread blocking blocks only itself (not the entire process) in kernel-level threads.
- Switching to a thread within the same process is faster (memory map and cache … etc) →
  - The kernel may prefer this choice if the two threads are equally important.
- In user level threads, scheduling may be according to the application needs (like the server example with dispatcher and workers). In kernel-level threads, this is not possible (except for priorities).

Classical IPC Problems

The Dining Philosophers Problem

- Presented and solved by Dijkstra in 1965, then used to test new synchronization solutions.
- A philosopher either thinks or eats (spaghetti with two forks!). When hungry, he tries to acquire forks in order.



The obvious (WRONG) Solution

```
#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 */
      }
}
```

- The problem is …….
- One modification: pick left fork, look for right fork. If not available put down left fork and try again after some time (say after 5 seconds).
- Again the problem is …….
- **Starvation**
- Does **random** waiting time solve the problem? Ex: sending a packet over the network vs safety control in a nuclear power plant.

Another Starvation-Free solution

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

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                /* philosopher is thinking */
down(mutex)
                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 */
up(mutex)
        }
}
```

- The problem is ....

Deadlock-Free with max. Parallelism

```
#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 */
```

Main code for each 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 */
    }
}
```

Deadlock-Free with max. Parallelism

```
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]);
        }
}
```

The Readers & Writers Problem

- Models access to a database (as example).
- Possible multiple readers at the same time, but **only one** writer at a time.

One Possible Solution

```
void reader(void)
{
        while (TRUE) {                          /* repeat forever */
                down(&mutex);                   /* get exclusive access to rc */
                rc = rc + 1;                    /* one reader more now */
                if (rc == 1) down(&db);         /* if this is the first reader ... */
                up(&mutex);                     /* release exclusive access to rc */
                read_data_base( );              /* access the data */
                down(&mutex);                   /* get exclusive access to rc */
                rc = rc – 1;                    /* one reader fewer now */
                if (rc == 0) up(&db);           /* if this is the last reader ... */
                up(&mutex);                     /* release exclusive access to rc */
                use_data_read( );               /* noncritical region */
        }
}


void writer(void)
{
        while (TRUE) {                          /* repeat forever */
                think_up_data( );               /* noncritical region */
                down(&db);                      /* get exclusive access */
                write_data_base( );             /* update the data */
                up(&db);                        /* release exclusive access */
        }
}
```

The Readers & Writers Problem

- What is the problem with the prev. sol.?

The Readers & Writers Problem

- Writers may starve when there is a continuous stream of arriving readers.
- One solution is that: whenever there is a waiting writer, subsequently arriving readers are not admitted, they wait till the writer finishes its work.