Process Synchronization

- Background
- The Critical-Section Problem
- Semaphores
- Classical Problems of Synchronization
- Monitors

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating/concurrent processes.
- That mechanism/logic is called process synchronization.

The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process P_i

repeat

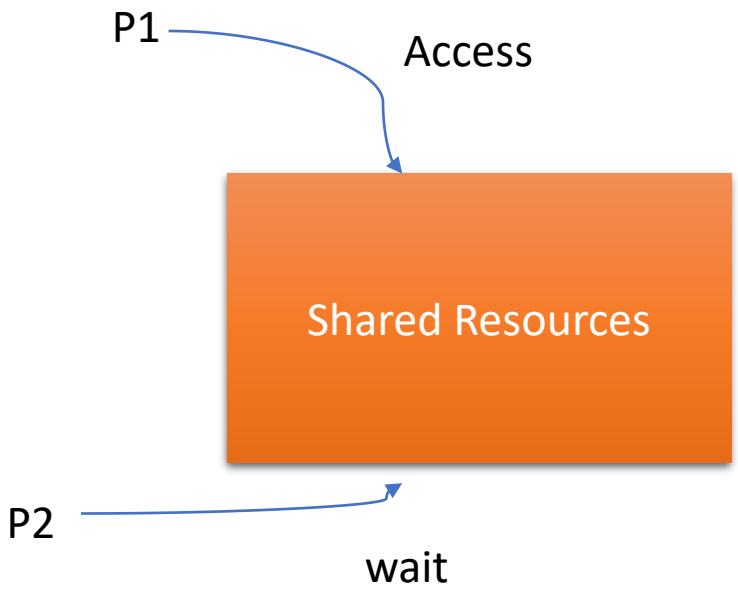
entry section

critical section

exit section

reminder section

until *false*;



Solution to Critical-Section Problem

1. **Mutual Exclusion**. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical sections.

If one processing is accessing the shared data other should wait.

2. **Progress**. All the concurrent process involved in the synchronization (mutual exclusion) must progress and driven to completion.

3. **Bounded Waiting**. There should be definite waiting time for processes following mutual exclusion.

Semaphore

- It is the solution to critical section problem.
- Helps in achieving mutual exclusion.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*): **while** $S \leq 0$ **do** *no-op*;
S := *S* – 1;

signal (*S*): *S* := *S* + 1;

Example: Critical Section of *n* Processes

- Shared variables
  - **var** *mutex* : *semaphore*
  - initially *mutex* = 1
- Process P_i

repeat

wait(*mutex*); 0: locked

critical section

signal(*mutex*); 1: opened

remainder section

until *false*;

Semaphore Implementation

- Define a semaphore as a record

type *semaphore* = **record**

value: *integer*

L: **list of** *process*;

end;

- Assume two simple operations:
  - Block: suspends the process that invokes it.
  - wakeup(*P*): resumes the execution of a blocked process P.

Implementation (Cont.)

- Semaphore operations now defined as

wait(*S*): *S.value* := *S.value* – 1;
if *S.value* < 0
then begin
add this process to *S*.L;
*blo*ck;
end;

signal(*S*): *S.value* := *S.value* + 1;
if *S.value* $\leq$ 0
then begin
remove a process *P* from *S.L*;
wakeup(*P*);
end;

Semaphore as General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

B can not be executed until unless process Pi execute the signal(flag) instruction

wait (*S*): **while** $S \leq 0$ **do** *no-op*;
$S := S - 1$;
signal (*S*): $S := S + 1$;

Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.

Q1

The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

```
do {
   flag[i] = TRUE;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; // do nothing
         flag[i] = TRUE;
      }
   }

      // critical section

   turn = j;
   flag[i] = FALSE;

      // remainder section
} while (TRUE);
```

Figure 6.25 The structure of process P_i in Dekker's algorithm.

The structure of process P_i (i == 0 or 1) is shown in Figure 6.25; the other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

Classic Synchronization Problems

- Dining-Philosophers Problem
- Bounded-Buffer Problem
- Readers and Writers Problem

Dining-Philosophers Problem

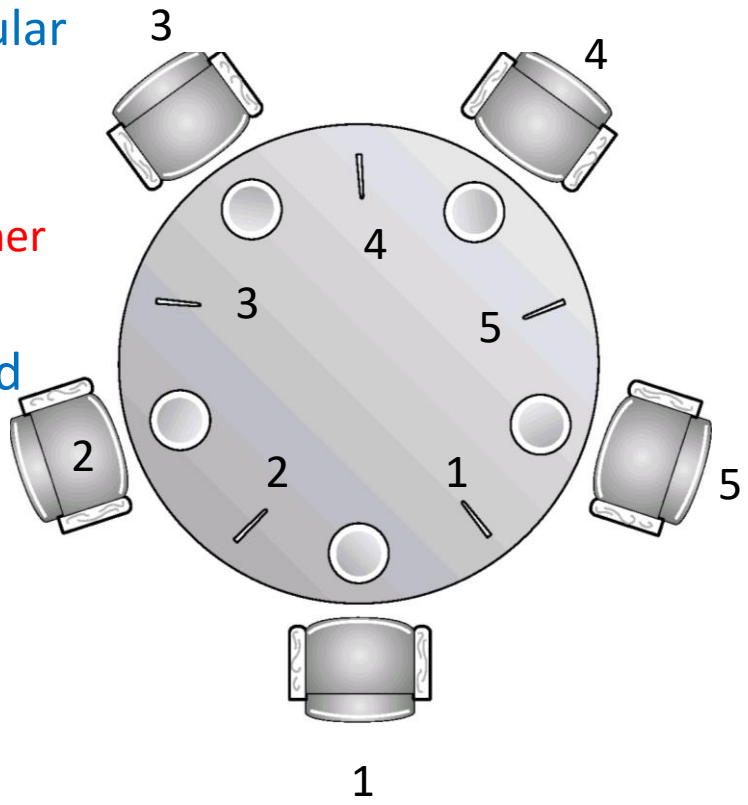
Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.

When a philosopher thinks, she does not interact with her colleagues.

From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).

A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.

When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.



It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

- One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing await () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

```
semaphore chopstick[5];
```

where all the elements of chopstick are initialized to 1.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
} while (TRUE);
```

Figure 6.15 The structure of philosopher *i*.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.

Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible solutions to the deadlock problem are listed next.

1. Allow at most four philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available
3. Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick

Monitors

- Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait (mutex) before entering the critical section and signal (mutex) afterward.

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

To deal with such errors, researchers have developed high-level language constructs. In this section, we describe one fundamental high-level synchronization construct-the monitor type.

Monitors

- The monitor construct ensures that only one process at a time is active within the monitor.

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Figure 6.16 Syntax of a monitor.

Monitors

- Consequently, the programmer does not need to code this synchronization constraint explicitly.