Synchronisation

Daniel Bosk[1]

Department of Information and Communication Systems (ICS),
Mid Sweden University, Sundsvall.

sync.tex 2035 2014-10-14 14:14:46Z danbos

Overview

1. Synchronisation
   - Where does the problem arise?
   - The Critical-Section Problem
   - Tools to solve the problem
2. Deadlocks
   - The requirements
   - Dining Philosophers Problem

Literature

This lecture covers the second half of the topic process management. It gives an overview of Chapter 5 “Process Synchronization” and Chapter 7 “Deadlocks” in [SGG13a; SGG13b], or Chapter 6 “Synchronization” and Chapter 7 “Deadlocks” in [SGG09].

Overview

1. Synchronisation
   - Where does the problem arise?
   - The Critical-Section Problem
   - Tools to solve the problem
2. Deadlocks
   - The requirements
   - Dining Philosophers Problem

Mittuniversitetet
MID SWEDEN UNIVERSITY

Where does the problem arise?

```
i = 0
while (True):
  i = i + 1
```

Where does the problem arise?

- When we have a preemptive scheduler or multiprocessor system, multiple processes can try to use the same resource at the same time.
- When the outcome of the execution depends on the order of execution of the processes we say we have a race condition.

The Critical-Section Problem

- We can sort out the parts of the code of a program which are suceptible to race conditions, these parts are called the critical sections.
- The code just before the critical section is called the entry section.
- The code in the end of the critical section is the exit section.
- The code after the critical section is the remainder section.
- The Critical-Section Problem is about the design of a protocol for several processes to use for cooperation around their critical sections.

Mittuniversitetet
MID SWEDEN UNIVERSITY

The Critical-Section Problem

To solve the Critcal-Section Problem an algorithm (protocol) must fulfull the following requirements:

1. Mutual exclusion. If a process P_i is executing in its critical section, no other process may do so.
2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes *not* in their remainder sections can participate in the decision.
3. Bounded waiting. There exits a bound for how long a process which has requested to enter its critical section may have to wait before allowed to enter.

Tools to solve the problem

- Peterson's solution.
- Locks.
- Semaphores.

Tools to solve the problem

Peterson's Solution

```
while (True):
  flag[i] = True
  turn = j
  while ( flag[j] and turn == j ):
    pass

  # critical section

  flag[i] = False

  # remainder section
```

Tools to solve the problem

- On modern computer architectures we need special hardware instructions to help us, e.g. TestAndSet or AtomicSwap.

Tools to solve the problem

TestAndSet

```
# lock = [False]
while ( True ):
  while ( TestAndSet( lock ) ):
    pass

  # critical section

  lock = False

  # remainder section
```

Tools to solve the problem

AtomicSwap

```
# lock = [False]
while ( True ):
  key = [True]
  while ( key[0] ):
    AtomicSwap( lock, key ) )

  # critical section

  lock = False

  # remainder section
```

Tools to solve the problem I

Bounded waiting

```
# lock = [False]
while ( True ):
  waiting[i] = True
  key = True
  while ( waiting[i] and key[0] ):
    key = TestAndSet( lock )

  waiting[i] = False

  # critical section

  j = ( i + 1 ) \% n
  while ( ( j != i ) and not waiting[j] ):
    j = ( j + 1 ) \% n

  if ( j == i ):
```

Tools to solve the problem II

Bounded waiting

```
        lock = False
    else:
        waiting[j] = False

    # remainder section
```

Tools to solve the problem

Semaphores

- Use two operations `wait()` and `signal()`.
- A binary valued semaphore is called a mutex lock, since it provides mutual exclusion.
- We have also counting semaphores.

Tools to solve the problem

Semaphores

```
def wait( S ):
  while ( S <= 0 ):
    pass
  S -= 1
```

```
def signal( S ):
  S += 1
```

Mittuniversitetet
MID SWEDEN UNIVERSITY

Tools to solve the problem

Semaphores

```
while ( True ):
  wait( mutex )

  # critical section

  signal( mutex )

  # remainder section
```

Overview

1. Synchronisation
   - Where does the problem arise?
   - The Critical-Section Problem
   - Tools to solve the problem
2. Deadlocks
   - The requirements
   - Dining Philosophers Problem

The requirements

For a deadlock to occur, the following requirements must be fulfilled:

1. Mutual exclusion.
2. Hold and wait.
3. No preemption.
4. Circular wait.

The converse, to prevent deadlocking we must guarantee at least one of the above requirements is not fulfilled at any time.

The requirements

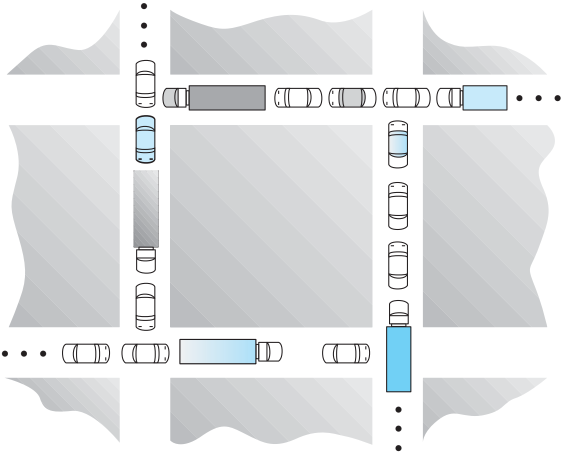

Figure: A deadlock. Image: [SGG13b].

The requirements

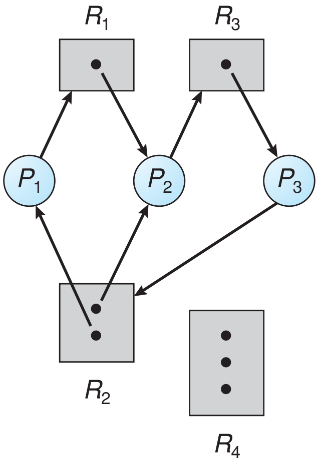

Figure: Another deadlock. Image: [SGG13b].

The requirements

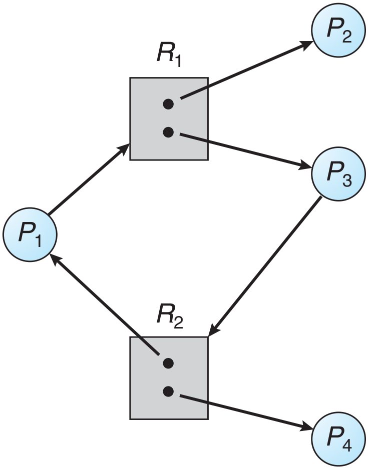

Figure: A similar situation which is not a deadlock. Image: [SGG13b].

Dining Philosophers Problem

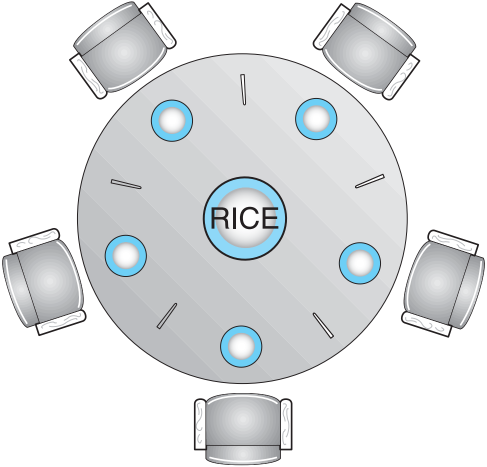

Figure: The setup of the Dining Philosophers Problem. Image: [SGG13b].

Referenser I

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 8th ed. International Student Version. Hoboken, N.J.: John Wiley & Sons Inc, 2009.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 9th ed. International Student Version. Hoboken, N.J.: John Wiley & Sons Inc, 2013.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 9th ed. Hoboken, N.J.: John Wiley & Sons Inc, 2013.