Operating Systems

Lecture 5

Synchronization of processes

Summary

- This lecture shows the problem of synchronization of processes:
  - Example: producent-consument case.
  - Critical section
    - Dekker's algorithm for 2 processes
    - Bakery algorithm
  - Synchronization mechanisms
    - Semaphores
    - Message queues
    - Monitors
    - Thread synchronization in Java
  - Hardware synchronization mechanisms.

Summary

- This lecture shows the problem of synchronization of processes:
  - Example: producent-consument case.
  - Critical section
    - Dekker's algorithm for 2 processes
    - Bakery algorithm
  - Synchronization mechanisms
    - Semaphores
    - Message queues
    - Monitors
    - Thread synchronization in Java
  - Hardware synchronization mechanisms.

Producer-Consument problem

- Assume two kinds of processes:
  - Producer: creates „products” and puts them into the buffer.
  - Consumer: consumes „products” from the buffer.
- Many producers and consumers are allowed and they work simultaneously.
- There is only one buffer.

Producer's algorithm

```
while true do
    Produce one item
    Put it in the buffer
done
```

Consumer's algorithm

```
while true do
    Pick one item from the buffer
    Consume this item
done
```

Where is the problem?

- Access to the buffer usually requires many steps:
  - Putting into the buffer includes:
    - Checking the current position of the free place,
    - Inserting a new item at this place,
    - Changing the free place indicator to the next available position.
  - Removing from the buffer includes:
    - Picking the item from the given place,
    - Reclaiming this place as free.

Where is the problem?

- Without additional mechanisms we cannot be sure, that instructions from one operation are not interleaved with instructions from others.

- Check current position of the free place
- Insert a new item at this place
- Change the free place indicator to the next available position.

- Check current position of the free place
- Insert a new item at this place
- Change the free place indicator to the next available position.

Proper access to some resources

REQUIRES

synchronization of processes

Summary

- This lecture shows the problem of synchronization of processes:
  - Example: producent-consument case.
  - Critical section
    - Dekker's algorithm for 2 processes
    - Bakery algorithm
  - Synchronization mechanisms
    - Semaphores
    - Message queues
    - Monitors
    - Thread synchronization in Java
  - Hardware synchronization mechanisms.

Critical section

- Critical section is a part of code or sequence of instructions, which **MUST NOT** be executed in parallel.
- Critical section code is usually surrounded by additional synchronization code.
- This code may be either “algorithmic” (additional overhead), or OS-supported (OS constitutes a global synchronizer).

Dekker's algorithm

- **Dekker's algorithm** may be used to protect access to a critical section for 2 processes. It uses the following data structures:
  - Boolean array *flag*[] with 2 elements *flag*[0] and *flag*[1], both set to **false** at the beginning.
  - Integer indicator *turn* (which may be equal to 0 or 1), initially 0.
  - Processes are determined by their indexes (0 or 1).
- Whenever a process *p* wants to enter the critical section, it sets „its” *flag*[*p*] to true. Then it checks the opponent's *flag*[1-*p*] value. It can proceed if it is set to false.
- The *turn* indicator breaks symmetry – it denotes, who's turn it is to enter the critical section in case, when both processes want to do it simultaneously.

Dekker's algorithm

```
// flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn  = 0  // or 1
```

```
P0:
   flag[0] = true;
   while (flag[1] == true) {
      if (turn ≠ 0) {
         flag[0] = false;
         while (turn ≠ 0) {
           // busy wait
         }
         flag[0] = true;
      }
   }

   // critical section
   ...
   turn    = 1;
   flag[0] = false;
   // remainder section
```

```
P1:
   flag[1] = true;
   while (flag[0] == true) {
      if (turn ≠ 1) {
         flag[1] = false;
         while (turn ≠ 1) {
           // busy wait
         }
         flag[1] = true;
      }
   }

   // critical section
   ...
   turn    = 0;
   flag[1] = false;
   // remainder section
```

Dekker's algorithm

- Dekker's algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation.
- It includes busy waiting in the loop, so it may have big impact on system's performance.
- It works only for 2 processes.

Lamport's bakery algorithm

- **Lamport's bakery algorithm** may be used to protect access to a critical section for any number of processes (N). It works in a way similar to the queueing system at the Post Office.
- It uses the following data structures:
  - Integer array *number*[1..N] with all values set to 0 at the beginning.
  - Boolean array *entering*[1..N] with all values set to false at the beginning.
  - Processes are determined by their indexes (1 to N).

Lamport's bakery algorithm (cont.)

- Whenever a process *p* wants to enter the critical section, it sets *entering*[*p*] to true. Then it determines *number* by selecting the highest not chosen number. This selection is done by scanning the values stored in the *number* array. The selected number is stored at *number*[*p*].
- **A process with the lowest number selected is allowed to enter the critical section.** If 2 or more processes select the same number, it's their index, what decides (process with a lower index enters first).

Lamport's bakery algorithm

```
// declaration and initial values of global variables
Entering: array [1..NUM_PROCESSES] of bool = {false};
Number: array [1..NUM_PROCESSES] of integer = {0};

lock(integer i) {
    Entering[i] = true;
    Number[i] = 1 + max(Number[1], ..., Number[PROCESSES]);
    Entering[i] = false;
    for (j = 1; j <= PROCESSES; j++) {
        // Wait until thread j receives its number:
        while (Entering[j]) { /* nothing – busy waiting */ }
        // Wait until all threads with smaller numbers or with the same
        // number, but with higher priority, finish their work:
         while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i)))
            { /* nothing */ }
     }
}
```

Lamport's bakery algorithm (cont.)

```
unlock(integer i) {
    Number[i] = 0;
}

Process(integer i) {
    while (true) {
        lock(i);
        // The critical section goes here...
        unlock(i);
        // non-critical section...
    }
}
```

Summary

- This lecture shows the problem of synchronization of processes:
  - Example: producent-consument case.
  - Critical section
    - Dekker's algorithm for 2 processes
    - Bakery algorithm
  - Synchronization mechanisms
    - Semaphores
    - Message queues
    - Monitors
    - Thread synchronization in Java
  - Hardware synchronization mechanisms.

Synchronization mechanisms

- The presented synchronization algorithms are provided by either operating systems, or programming languages.

Semaphores

- Semaphore is used to protect access to a critical section. It is implemented by OS.
- It may be considered as a „special” variable (it has two additional special operations defined).
- In general, a semaphore can have any non-negative value.
  - If this value is 0, it means, that access to critical section is locked and any process trying to get access to it will be suspended.
  - Otherwise, the access to critical section is allowed.
- Binary semaphores allow for only 2 values: 0 = semaphore down (section locked) or 1 = semaphore up (section accessible).

Semaphores

- Access to a semaphore is done by one of the two functions:
  - Lower the semaphore: P(*s*)
    - If the current value of semaphore *s* is greater then 0, it is decreased and process can execute the following code.
    - If the value of semaphore *s* equals to 0, the process is suspended. It may continue its execution when semaphore is raised by another process (the one which executes a critical section now).
  - Raise the semaphore: V(*s*)
    - It increases the current value of semaphore *s*.
    - If the value of semaphore was 0 and there was a process suspended, it is unlocked and the semaphore's value is automatically decreased.

Semaphores

- The P() operation is executed before a process enters the critical section and V() should be executed after it has left the critical section.
- If V() is ommited or operations are performed in the wrong order, it may lead to a deadlock:

- P1:

```
    P(S);
    P(Q);
    critical sections
    V(Q);
    V(S);
```

- P2:

```
    P(Q);
    P(S);
    critical sections
    V(S);
    V(Q);
```

Semaphores

- In the correct implementation, all the semaphores are locked in the same order in all the processes:

- P1:

```
P(Q);
P(S);
critical sections
V(S);
V(Q);
```

- P2:

```
P(Q);
P(S);
critical sections
V(S);
V(Q);
```

Message queues

- Message queue is a communication mechanism implemented in the OS kernel, thus governed by OS. Message queues implement a FIFO model.
- Access to a message queue consists in either:
  - putting a message at the end of the queue. If the size of a queue is limited, operation can cause a process suspension (until there is a space in the queue).
  - picking a message from the beginning of the queue. A process is suspended if the queue is empty.
- Access to a message queue is exclusive, which means, that any operation is „atomic” (cannot be interleaved with another operation on the same queue).

Monitors

- Monitors are implemented by programming languages.
- Monitor can be considered as a class, which methods constitute critical section. Monitor attributes are accessible only from these methods.
- They provide functionality similar to semaphores, but in a more „structured” way.
- Only one process can access monitor's methods at a time.
  - If there are more such processes, they are stored on a queue.
  - Whenever a process leaves a monitor and there are other processes waiting to access it, one of them is freed.

Monitors

- Monitor also provides additional type of an internal queue of *condition* type. These queues have two operations:
  - wait(*cond*) - makes the process suspend and put at the end of the queue
  - signal(*cond*) – resumes execution of the first process in a queue (if not empty).
    - This operation „pushes” theexecuting process out of the monitor, but it is placed at the beginning of the queue of waiting processes, so it will continue it's execution just after the woken process leaves monitor.

Thread synchronization in Java

- Java implements a simplified monitor mechanism – each object can be a monitor.
- Each method, which should be mutually-exclusive, should be declared as *synchronized*. In this way, only one thread can access it at a time (others are suspended).

```
public class Counter {
    private int c = 0;
    public synchronized void inc() {
      c++;
    }
    public synchronized void dec() {
      c--;
    }
    public synchronized int val() {
      return c;
    }
  }
```

Thread synchronization in Java

- One can also use a *synchronized*(*object*) syntax, if creation of a special object is not necessary.

```
public class Sync extends Thread {
    static Object section = new Object();
    …
    public void run() {
        …
        synchronized(section) {
            /* critical section */
            …
        }
        …
    }
    …
}
```

Thread synchronization in Java

- Java provides basic monitor functionality
  - Each object contains one condition queue and each thread accessing it can execute the wait() method, which suspends it and releases the object.
  - Any other process in the same critical section can execute notify() (or notifyAll()) method to wake up one (or all) of suspended threads.

Summary

- This lecture shows the problem of synchronization of processes:
  - Example: producent-consument case.
  - Critical section
    - Dekker's algorithm for 2 processes
    - Bakery algorithm
  - Synchronization mechanisms
    - Semaphores
    - Message queues
    - Monitors
    - Thread synchronization in Java
  - Hardware synchronization mechanisms.

Hardware synchronization (examples)

- TestAndSet() - a basic function, which may be hardware-supported.

```
function TestAndSet (var x: boolean): boolean;
  begin
    TestAndSet := x;
    x := true
  end;
```

- Exchange(x, y) – exchanges two values in an uninterruptable way.

Typical concurrent access problems

- Producer-consumer problem (again)
- Readers and writers.
- Philosophers problem.

Producer-consument with semaphores

- Producer:

```
repeat
    *produce an item*;
    P(empty);
    P(S);
    buffer [(first+count) mod N]
          := produced item;
    count++;
    V(S);
    V(full)
until false
```

- Consumer

```
repeat
    P(full);
    P(S);
    aquired item := buffer [first];
    first := (first + 1) mod N;
    count := count - 1;
    V(S);
    V(empty);
    *consume*
until false
```

Readers and writers

- Assume that there is a library and two classes of users: readers, which can only read, and writers, which can read and write.
  - At a time, in the library can be only ONE writer, or any number of readers.
  - In general, it is assumed, that a reader can enter a library if it is empty or there are only other readers inside, while a writer can enter it only if the library is empty.
- Improper solution may lead to starvation of any group of library users:
  - If readers can enter freely, they may keep writers from entering forever.
  - If writers have higher priority than readers, it may lead to the opposite behavior.

Readers and writers

- Possible solution:
  - 2 queues, in which readers and writers can wait to enter the library.
  - A reader can enter the library if it is empty or if there is no writer waiting in the queue. Otherwise, he must stay in the line.
  - A writer can enter the library if it is empty and when there is no-one waiting in the line. Otherwise, he must stay in the line.
  - Whenever a writer leaves the library, all the waiting readers can get inside.
  - Whenever the last reader leaves the library, the first waiting writer can get inside.

Readers and writers

```
monitor readers_and_writers;
    var
      counter: integer;
      readers, writers: condition;

    procedure reader_enters;
    begin
      if (counter = -1) or not empty(writers) then wait(readers);
      counter := counter + 1;
      signal(readers)
    end;

    procedure reader_leaves;
    begin
      counter := counter - 1;
      if counter = 0 then signal(writers)
    end;
```

Readers and writers

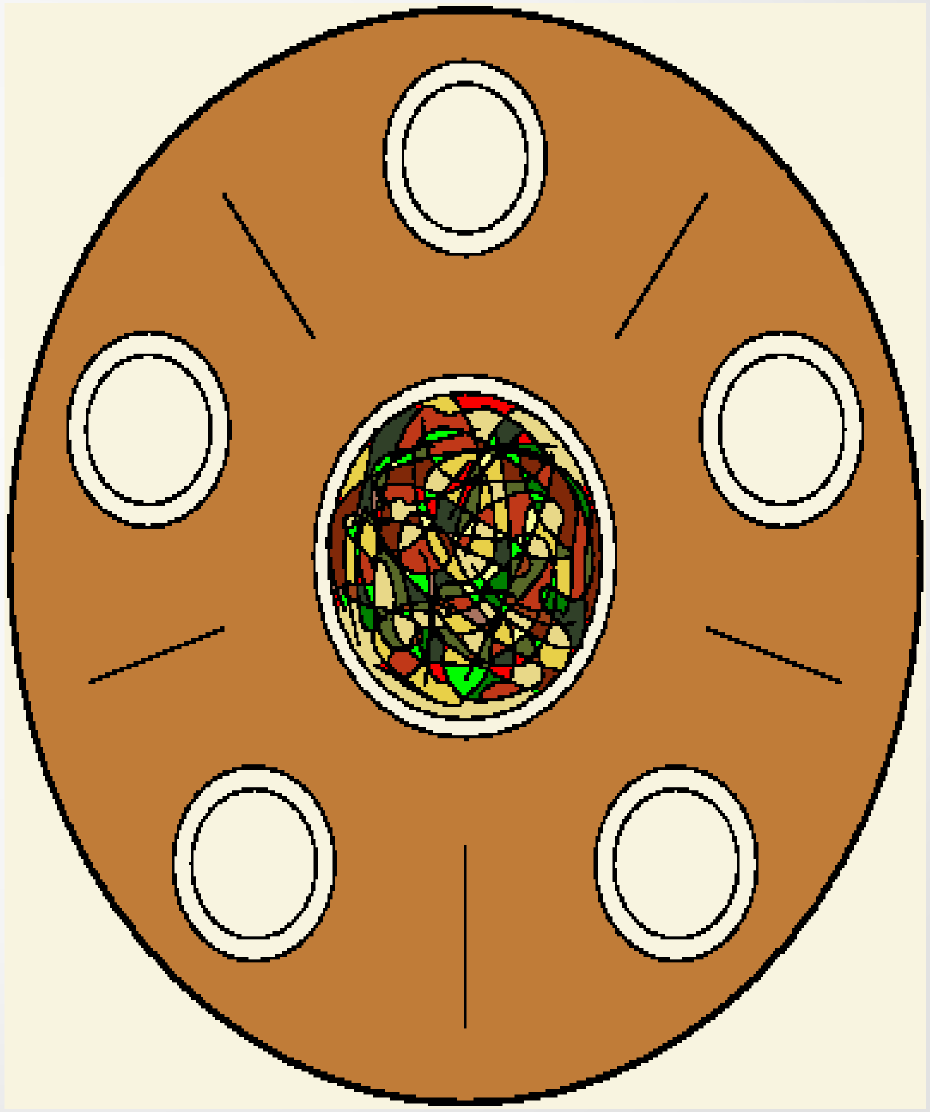
```
  procedure writer_enters;
  begin
    if counter ≠ 0 then wait(writers);
    counter := -1
  end;

  procedure writer_leaves;
  begin
    counter := 0;
    if empty(readers) then signal(writers) else signal(readers)
  end;

begin
  counter := 0;
end;
```

The five philosophers

- There are 5 philosophers. They sit at the round table and think. Whenever a philosopher gets hungry, he may eat something, and then he continues thinking.
- Whenever a philosopher gets hungry, he gets some food from the bowl in the middle of the table, then he must take 2 sticks (one from his left and one from his right) and only then he may eat.



The five philosophers

- The problem: there are only 5 sticks on the table (!).
- Possible solution: it is enough to limit the number of concurently eating (or willing to eat) philosophers to 4.

```
var
    sticks: array [0..4] of semaphore;
    plates: semaphore;

  procedure philosopher(i: 0..4);
  begin
    repeat
      think();
      P(plate);
      P(sticks[i]);
      P(sticks[(i + 1) mod 5]);
      eat();
      V(sticks[i]);
      V(sticks[(i + 1) mod 5]);
      V(plate);
    until false
  end;
```

Thank You