Operating Systems

Lecture 5a

Dr Ronald Grau

School of Engineering and Informatics

Spring term 2018

Previously

Evaluation of scheduling algorithms

- Deterministic evaluation
- Probabilistic evaluation
  - Queueing models
  - Little's Law
- Stochastic evaluation
  - Simulation models

Today

Process Synchronisation

- Inter-Process Communication
- Race conditions
- Communication models
- Critical section
- Software vs. Hardware solutions
- Condition synchronisation

Inter-Process Communication

Context

- Processes:
  - share data and resources
  - cooperate to work on common tasks

- There is a need to synchronise activities, i.e. coordinate:
  - access to shared data and resources
  - sequences of operations of different processes

Problem: Race conditions lead to inconsistent results because of how the execution of instructions may interleave

Race Conditions

Example

Process 1:
```
x++;
```
```
MOV A, x
ADD A, 1
MOV x, A
```

Process 2:
```
x--;
```
```
MOV B, x
SUB B, 1
MOV x, B
```

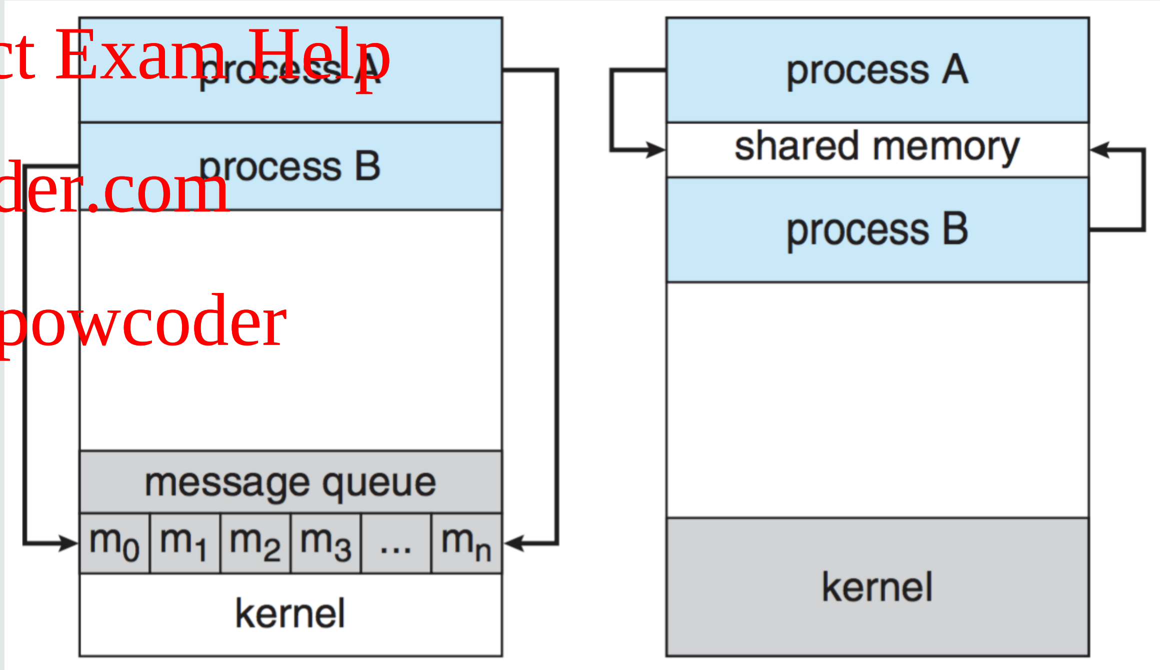
Possible interleaved execution:

MOV A, x	MOV A, x	MOV B, x
ADD A, 1	MOV B, x	SUB B, 1
MOV x, A	SUB B, 1	MOV A, x
MOV B, x	MOV x, B	ADD A, 1
SUB B, 1	ADD A, 1	MOV x, A
MOV x, B	MOV x, A	MOV x, B
x_{old}	$x_{old} + 1$	$x_{old} - 1$

Communication Models

Mechanisms

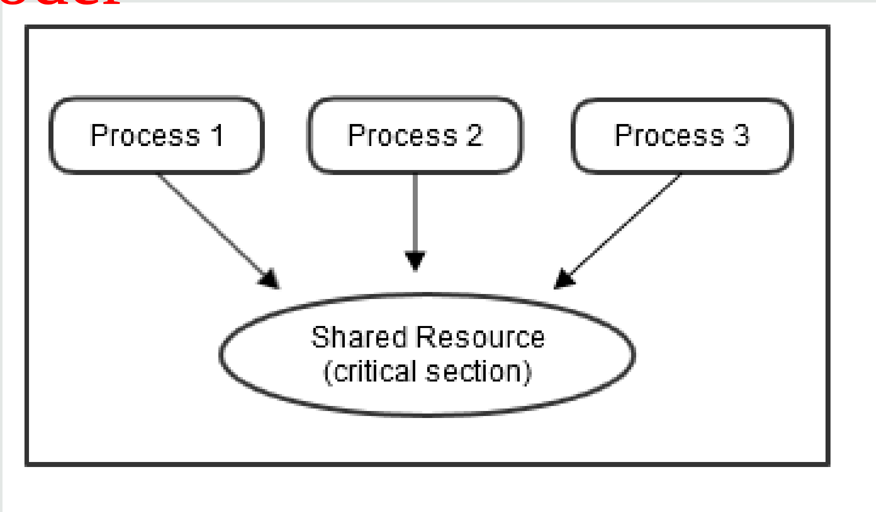
- Shared memory:
  - Commonly accessible memory region
  - Requires synchronisation!
- Message passing (MP):
  - `send` and `receive` messages
  - Synchronisation is implicit!
- Support by operating system kernel, e.g. POSIX SHM, Pipes, RPC, Sockets, MPI, …



Critical Section

Process 1:		**Process 2:**
`entry protocol`		`entry protocol`
`x++;`	Critical section	`x--;`
`exit protocol`		`exit protocol`

- Mutual exclusion:
  Only one of the processes should execute in its critical section at any point in time.
- Entry and exit of critical sections must be coordinated.



Critical Section

Any solution must ensure the following:

- Mutual exclusion: If a process is executing in its critical section, then other processes must not execute in their critical section.

- Progress (Prevent deadlock): If no process is executing in its critical section and some processes wish to enter their critical sections, then the selection of which process may enter cannot be postponed indefinitely.

- Bounded waiting (Prevent starvation): If a process wishes to enter the critical section then only a bounded number of processes may enter the critical section before it.

Solutions

Software

- Assume atomicity of read and write operations

Hardware-supported

- More powerful instructions (e.g. test-and-set)

Higher-level APIs

- Wrap low-level primitives into library:
  Semaphores, monitors, condition variables, etc
- Thread-safe data structures:
  Blocking queues, synchronised maps, etc

Software solutions

Classical algorithms

- Dekker / Peterson

- Assume two processes P_0 and P_1 alternate execution (can be generalised to any number of processes)

- Assume atomic read and write

- Synchronisation over shared variables (e.g. turn, flag)
- Busy waiting:

```
while (condition) {
    /* do nothing */
}
```

Software solutions

Idea 1: Turns

- Assume $i, j \in \{0,1\}$ $i \neq j$
- Shared variable:
  `int turn;`
- Process P_i


```
do {
    // …

    while (turn != i) {
        /* do nothing */
    }

    /* critical section */
    turn = j;
    // …
}
while (!done)
```

Mutual exclusion: Yes

But: If one process finishes, the other one remains blocked forever.

Software solutions

Idea 2: Flags

- Shared variables:
  `boolean flag[2] = {false, false};`
- Process P_i


```
do {
    // …

    while (flag[j]) {
        /* do nothing */
    }
    flag[i] = true;

    /* critical section */
    flag[i] = false;
    // …
}
while (!done)
```

If one process finishes the other one can continue.

But mutual exclusion not guaranteed!

Software solutions

Idea 3: Flag first, then wait

- Shared variables:
  `boolean flag[2] = {false, false};`
- Process P_i

Mutual exclusion: Yes

Possible deadlock

```
do {
   // …
   flag[i] = true;

   while (flag[j]) {
      /* do nothing */
   }

   /* critical section */
   flag[i] = false;
   // …
}
while (!done)
```

Software solutions

Idea 4: Wait and try again

Mutual exclusion: Yes

Possible livelock

```
do {
    // …
    flag[i] = true;

    while (flag[j]) {
        flag[i] = false;
        // wait a little…
        flag[i] = true;
    }

    /* critical section */
    flag[i] = false;
    // …
}
while (!done)
```

Software solutions

Dekker's algorithm

- `flag` request to enter
- `turn` determines which one enters the critical section first

Mutual exclusion: Yes!

Deadlocks prevented: Yes!

Starvation prevented: Yes!

```
do {
   // …
   flag[i] = true;

   while (flag[j]) {
      flag[i] = false;
       while (turn == j) {
           /* do nothing */
      }
       flag[i] = true;
   }
   /* critical section */
   turn = j;
   flag[i] = false;
   // …
}
while (!done)
```

Software solutions

Peterson's algorithm

- More elegant implementation of Dekker's algorithm
- Shared variables: `boolean flag[2] = {false, false};`
- Process P_i


```
do {
    // …
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j) {
        /* do nothing */
    }

    /* critical section */
    flag[i] = false;
    // …
}
while (!done)
```

Hardware-Supported Solutions

Interrupt disabling

- Protect low-level instruction sequences in kernel
- Suitable for user processes?
- Suitable in multi-core processors?

More powerful instructions

- Test-and-set, compare-and-swap, compare-and-exchange

  Rationale: Two or more operations that are executed atomically

Hardware-Supported Solutions

Test-and-Set

- An instruction that performs the following atomically:


```
_Bool testandset(_Bool *b) {
    if(!*b) {
        *b = 1;
        return 1;
    }
    return 0;
}
```

- **Usage:**

  Shared variable `_Bool b;`

```
do {
    // …
    while (!testandset(&b)) {
        /* do nothing */
    }
    /* critical section */
    b=0;
    // …
}
while (!done)
```

Condition Synchronisation

Ensure processes perform actions in desired order

- Example: data transfer from process P_1 to P_2
  - P_1 writes to shared memory, P_2 reads it
  - Goal: avoid duplication or loss of data
- E.g. using `wait()` and `notify()` in Java:


```
do {
   data.wait();
   readdata = data // critical
   data.notify();
   process(readata);
}
while (true)
```

```
do {
   newdata = generate();
   if(init)
     init = false;
   else
     data.wait();
   data = newdata // critical
   data.notify();
}
while (true)
```

Summary

Inter-Process Communication (IPC)

- Communicate information from one process to another
- Avoid conflicts and inconsistencies when accessing shared data and resources (race conditions)
- Coordinate the sequencing of operations

Communication Models

- Shared Memory
- Message Passing

Synchronisation of Critical Sections

- Mutual exclusion, making progress & preventing starvation
- Software and hardware-supp. solutions

Condition synchronisation

- Sequencing of process instructions

Read

- Tanenbaum & Bos., Modern Operating Systems
  - Chapter 2.3 & 2.5
- Silberschatz et al., Operating System Concepts
  - Chapter 3.4 – 3.6, Chapter 6
- https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html

Next Lecture

- Introduction
- Operating System Architectures
- Processes
- Threads - Programming
- Process Scheduling - Evaluation
- **Process Synchronisation**
- Deadlocks
- Memory Management
- File Systems
- Input / Output
- Security and Virtualisation