# Process Synchronization

# Multiple Processes

- Operating System design is concerned with the management of processes and threads:
    - Multiprogramming
    - Multiprocessing
    - Distributed Processing

# Concurrency & Shared Data

- Concurrent processes may share data to support communication, info exchange,...
- Threads in the same process can share global address space
- Concurrent sharing may cause problems
- For example: lost updates

# Concurrency: Key terms

| | |
|---|---|
| **atomic operation** | A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes. |
| **critical section** | A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code. |
| **deadlock** | A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something. |
| **livelock** | A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work. |
| **mutual exclusion** | The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources. |
| **race condition** | A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution. |
| **starvation** | A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen. |

# Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

# Race Condition

- Occurs when multiple processes or threads read and write shared data items
- The final result depends on the order of execution
    - the “loser” of the race is the process that updates last and will determine the final value of the variable

# Operating System Concerns

- Design and management issues raised by the existence of concurrency:
  - The OS must:
    - be able to keep track of various processes
    - allocate and de-allocate resources for each active process
    - protect the data and physical resources of each process against interference by other processes
    - ensure that the processes and outputs are independent of the processing speed

# Process Interactions

| **Degree of Awareness** | **Relationship** | **Influence that One Process Has on the Other** | **Potential Control Problems** |
|---|---|---|---|
| Processes unaware of each other | Competition | •Results of one process independent of the action of others<br><br>•Timing of process may be affected | •Mutual exclusion<br><br>•Deadlock (renewable resource)<br><br>•Starvation |
| Processes indirectly aware of each other (e.g., shared object) | Cooperation by sharing | •Results of one process may depend on information obtained from others<br><br>•Timing of process may be affected | •Mutual exclusion<br><br>•Deadlock (renewable resource)<br><br>•Starvation<br><br>•Data coherence |
| Processes directly aware of each other (have communication primitives available to them) | Cooperation by communication | •Results of one process may depend on information obtained from others<br><br>•Timing of process may be affected | •Deadlock (consumable resource)<br><br>•Starvation |

# Resource Competition

- Concurrent processes come into conflict when they use the same resource (competitively or shared)
  - for example: I/O devices, memory, processor time, clock
- Three control problems must be faced
  - Need for mutual exclusion
  - Deadlock
  - Starvation
- Sharing processes also need to address coherence

# Need for Mutual Exclusion

- If there is no controlled access to shared data, processes or threads may get an inconsistent view of this data
- The result of concurrent execution will depend on the order in which instructions are interleaved.
- Errors are timing dependent and usually not reproducible.

# An Example

- Assume P1 and P2 are executing  this code and share the variable **`a`**
- Processes can be preempted at any time.
- Assume P1 is preempted after the input statement, and P2 then executes entirely
- The character echoed by P1 will be the one read by P2 !!

a

P1 P2

```
static char a;

void echo()
{
  cin >> a;
  cout << a;
}
```

cin
cout

Preempted

5
6
6
6

# What’s the Problem?

- This is an example of a *race condition*
- Individual processes (threads) execute sequentially in isolation, but concurrency causes them to interact.
- We need to prevent concurrent execution by processes when they are changing the same data. We need to enforce *mutual exclusion*.

# The Critical Section Problem

- When a process executes code that manipulates shared data (or resources), we say that the process is in its critical section (CS) for that shared data
- We must enforce mutual exclusion on the execution of critical sections.
- Only one process at a time can be in its CS (for that shared data or resource).

# The Critical Section Problem

- Enforcing mutual exclusion guarantees that related CS’s will be executed *serially* instead of *concurrently*.
- The critical section problem is how to provide mechanisms to enforce mutual exclusion so the actions of concurrent processes won’t depend on the order in which their instructions are interleaved

# The Critical Section Problem

- Processes/threads must request permission to enter a CS, & signal when they leave CS.
- Program structure:
  - entry section: requests entry to CS
  - exit section: notifies that CS is completed
  - remainder section (RS): code that does not involve shared data and resources.

RS

entry-CS

CS

exit-RS

RS

- The CS problem exists on multiprocessors as well as on uniprocessors.

# Mutual Exclusion and Data Coherence

- Mutual Exclusion ensures data coherence if properly used.
- Critical Resource (CR) - a shared resource such as a variable, file, or device
- Data Coherence:
  - The final value or state of a CR shared by concurrently executing processes is the same as the final value or state would be if each process executed serially, in some order.

# Deadlock and Starvation

- Deadlock: two or more processes are blocked permanently because each is waiting for a resource held in a mutually exclusive manner by one of the others.
- Starvation: a process is repeatedly denied access to some resource which is protected by mutual exclusion, even though the resource periodically becomes available.

# Mutual Exclusion

```
        PROCESS 1 */

void P1
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

```
        /* PROCESS 2 */

void P2
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

• • •

```
        /* PROCESS n */

void Pn
{
   while (true) {
      /* preceding code */;
      entercritical (Ra);
      /* critical section */;
      exitcritical (Ra);
      /* following code */;
   }
}
```

# Requirements for Mutual Exclusion

token

P1 P2

- Mutual Exclusion: must be enforced
- Non interference: A process that halts must not interfere with other processes
- No deadlock or starvation || Liveness
- Progress:A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

turn := 1

CS

turn := 2

# token := 2

CS

turn := 1

# Mutual Exclusion: Hardware Support

- **Interrupt Disabling**
  - uniprocessor system
  - disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will not work in a multiprocessor architecture

# Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Compare&Swap Instruction
    - also called a “compare and exchange instruction”
    - a **compare** is made between a memory value and a test value
    - if the old memory value = test value, swap in a new value to the memory location
    - always return the old memory value
    - carried out atomically in the hardware.

# Mutual Exclusion: Hardware Support

- Compare&Swap Instruction
  - Pseudo-code definition of the hardware instruction:

```
compare_and_swap (word, test_val, new_val)
if (word ==test_val)
    word = new_val;
return new_val
```

# Compare and Swap Instruction

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true) {
     while (compare_and_swap(bolt, 0, 1) == 1)
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

(a) Compare and swap instruction

word = bolt
test_val = 0
new_val = 1

If bolt is 0 when the C&S is executed, the condition is false and P enters its critical section. (leaves bolt = 1)
If bolt = 1 when C&S executes, P continues to execute the while loop. It's busy waiting ( or spinning)

# Exchange Instruction

```
    /* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
    int keyi = 1;
    while (true) {
        do exchange (keyi, bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction