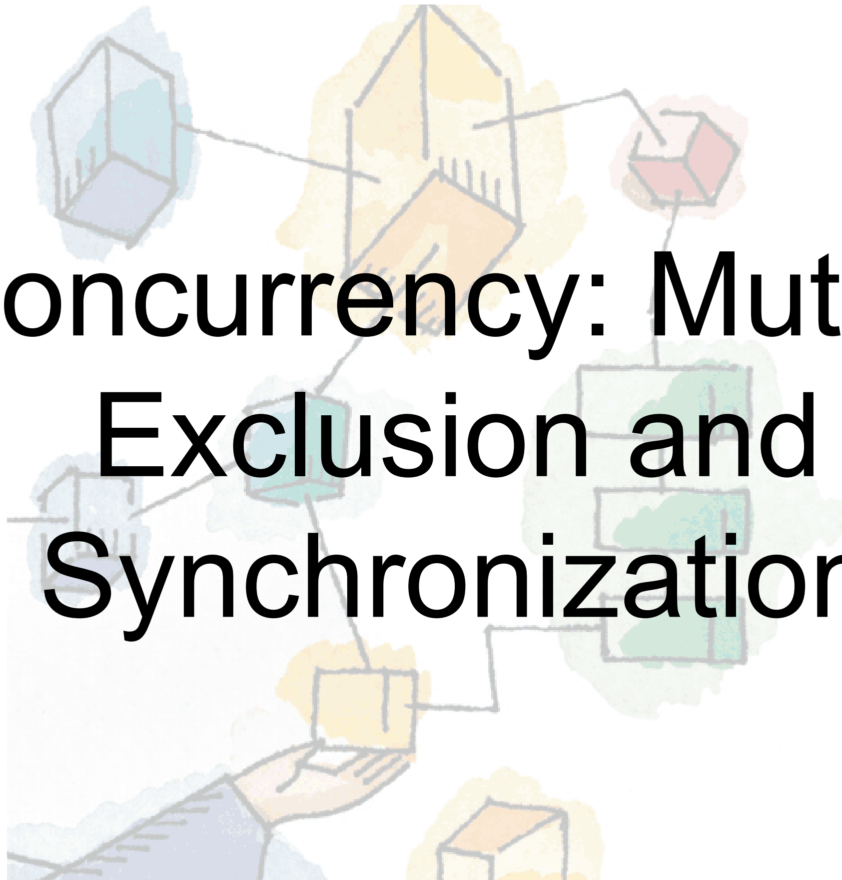
Concurrency: Mutual Exclusion and Synchronization



Outline

- Race condition
- Critical section
- Mutual exclusion
- Hardware support
  - Atomic operations
  - Special machine instructions
    - Compare&Swap
    - Exchange

Multiple Processes/Threads

- The design of modern Operating Systems is concerned with the management of multiple processes and threads
  – Multiprogramming
  – Multiprocessing
- Big Issue is Concurrency
  – Managing the interaction of processes/threads

Race Condition

- A race condition (or data race) occurs when
  - Multiple processes or threads read and write shared data items
  - They do so in a way where the final result depends on the order of execution of the processes/threads
  - The output depends on who finishes the race last

A Simple Example on a Single Processor System

- count++ could be implemented as

  register1 = count
  register1 = register1 + 1
  count = register1

- count-- could be implemented as

  register2 = count
  register2 = register2 - 1
  count = register2

A Simple Example on a Single Processor System

- Consider:
  - process A increments count and process B decrements count simultaneously
  - the execution interleaving with “count = 5” initially:

S0: process A execute register1 = count {register1 = 5}
S1: process A execute register1 = register1 + 1 {register1 = 6}
S2: process B execute register2 = count {register2 = 5}
S3: process B execute register2 = register2 - 1 {register2 = 4}
S4: process A execute count = register1 {count = 6 }
S5: process B execute count = register2 {count = 4}

Critical Section

- In the count example, the correct result will be guaranteed if one process starts increment/decrement only after the other one has finished decrement/increment
  - Need to have mechanism, i.e. mutual exclusion - lock/unlock to serialize the processes/threads entering their critical sections
- When a process executes code that manipulates shared data (or resource), we say that the process is in its Critical Section
  - Want to make sure only one process in its critical section at any time
- A general structure:

…

entry section

critical section

exit section

noncritical section

…

Atomic Operation

- A sequence of instructions appears to be indivisible
- The idea behind making a series of actions atomic is simply expressed with the phrase “all or nothing”
- It should either appear as if
  - all of the actions you wish to group together occurred
  - or that none of them occurred
  - with no in-between state visible
- Sometimes, the grouping of many actions into a single atomic action is called a transaction
  - an idea developed in great detail in the world of databases and transaction processing

Mutual Exclusion

- Only one process at a time is allowed in the critical section for a resource
- No assumptions are made about relative process speeds or number of processes
- For mutual exclusion to work we need
  - hardware support – to provide basic atomic operation primitives
  - OS – to ensure efficiency

Mutual Exclusion: Hardware Support

- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion
  - Work in uniprocessor systems
- Disadvantages:
  - The efficiency of execution could be noticeably degraded
  - This approach will not work in a multiprocessor architecture
  - Serious security issues – thus user process/threads must never disable interrupts

Mutual Exclusion: Hardware Support

- Special Machine Instructions:
  - Compare&Swap Instruction
    - also called a “compare and exchange instruction”
  - Exchange Instruction
- These are atomic instructions
  - Operations are indivisible

Compare&Swap Instruction

```
int compare_and_swap (int *word,
   int testval, int newval)
{               =0              =1
    int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
}
```

- If `word = 1`, unchange, and return `1`
- If `word = 0`, `word = 1`, and return `0`

Compare&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));

}
```

Busy waiting

(a) Compare and swap instruction

Exchange instruction

```
void exchange (int register, int memory)
{
   int temp;
   temp = memory;
   memory = register;
   register = temp;
}
```

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));
}
```

Busy waiting

(b) Exchange instruction

Special Machine Instructions: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- It is simple and therefore easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable

Special Machine Instructions: Disadvantages

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time
- Starvation is possible when a process leaves a critical section and more than one process is waiting
  - Some process could indefinitely be denied access.
- Deadlock is possible