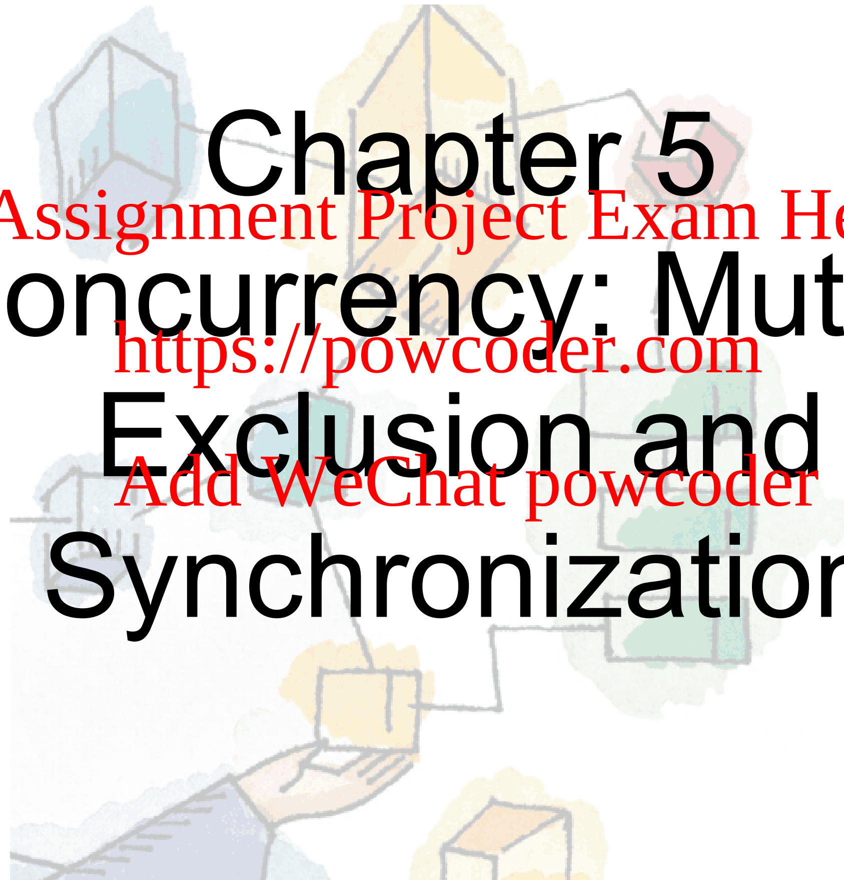
Operating Systems:
Internals and Design Principles
William Stallings



Chapter 5

Concurrency: Mutual

Exclusion and

Synchronization

Outline

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

Multiple Processes

- 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

Race Condition

- A race condition 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.
- The output depends on who finishes the race last.

A Simple Example

Assume chin is a shared variable.

```
void echo()
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

A Simple Example: On a Multiprocessor

Process P1	Process P2
.	.
chin = getchar();	.
	.
	chin = getchar();
.	
chout = chin;	chout = chin;
.	putchar(chout);
putchar(chout);	.
.	.

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 increment count and process B decrement 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

- When a process executes code that manipulates shared data (or resource), we say that the process is in its Critical Section.

- Need to design a protocol that the processes can use to cooperate.

- A general structure:


```
…
entry section
    critical section
exit section
    noncritical section
…
```

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
- A process must not be delayed access to a critical section when there is no other process using it
- A process that halts in its noncritical section must do so without interfering with other processes

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 */;
   }
}
```

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

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

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.