Lecture 41: Solution to Critical Section Problem

The Lecture Contains:

- Solution to Critical Section Problem
- Mutual Exclusion
- Progress
- Bounded Wait
- Solution Issues
- Two Process Critical Section Solution
- Solution to Critical Section Problem
- Synchronization Support in OS/ISA
- Support in ISA
- Implementing Locks Using Swap
- Other Supports From ISA
- Support From The OS
- Multiprocessor Issues
- Semaphores
- Mutual Exclusion Using Semaphore

Previous Next

Module 21: Problem and Solution

Lecture 41: Solution to Critical Section Problem

Solution to Critical Section Problem

- Requirements
  - Mutual Exclusion
  - Progress
  - Bounded Wait
- We can make no assumptions on
  - Processor speed
  - Relative speeds of processes
  - Time to execute any critical/remainder section
  - Time to execute any entry/exit code

Mutual Exclusion

- Statement of obvious
  - If a process P_i is in critical section at some point in time, no other process should be permitted to enter its critical section.
- This is clearly the most basic requirement for the solution.

Previous Next

Progress

- Given:There is no process in the critical section
  - And one or more processes want to enter the critical sections
  - Then one of them must be permitted to enter the critical section.
- One those processes waiting to enter the critical section must take part in the arbitration.
  - This arbitration must be done in a finite amount of time.

Bounded Wait

- A scheme of fairness and ensuring no starvation.
- There is an upper bound on the number of times that other processes are allowed to enter their critical sections between a process making its request and here quest getting granted.

Previous Next

Solution Issues

- Preemptive kernel
    - A process may be preempted when in kernel mode
- Non-preemptive kernel
    - A process may not be preempted when in kernel mode
- Preemptive kernels are difficult
    - Especially for SMP machines.
- Threads on multi-processors face independent OS.
- Preemptive kernels are essential
    - In embedded systems with RT guarantees
- Windows 2000, XP are non-preemptive
- Linux kernel became preemptive since Linux 2.6

Two Process Critical Section Solution

- Processes are P0 and P1.
- Processes share a common variable turn(=0 or 1).
- If turn= i, *Pi* is permitted to enter the critical section.

while (1) {
While (turn != i);
Critical section
turn = 1-i;
Remainder section
}

Mutual Exclusion: ✓
Progress: X
Bounded Wait: X

Previous Next

Solution to Critical Section Problem

- Consider two processes P0 and P1.
- Shared variables (for solution to CSP)
    - int turn; boolean flag[2];
    - flag[*i*] is true when P_i is ready to enter its critical section.

while (1) {
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
Critical section
flag[i] = FALSE;
Remainder section
}

Mutual Exclusion: ✔
Progress: ✔
Bounded Wait: ✔

Synchronization Support in OS/ISA

- Synchronization code can be written using locks
  while (1) {
  Non critical code
  Acquire Lock
  Critical Section Code
  Release Lock
  }
- Implementation of Lock require support from the ISA
    - TestAndSet instruction, Swap instruction.

Previous Next

Support in ISA

- Recall: All instructions in a processor execute uninterrupted.
- Within a single processor, instructions are atomic.
- Pentium ISA provide xchg instruction. (Swap instruction)
  xchg(register r, memory_address a) {
  t = r; r = *a; *a = t;
  }
  - One Read and One write in memory and register each.

Implementing Locks Using Swap

- Acquire Lock:
  Register AX = TRUE;
  While (AX=TRUE) xchg(AX, &lock);
- Release Lock:
  - Lock = false;

Previous Next

Other Supports from ISA

- Some processors support TestAndSet instruction.
  ```
  boolean TestAndSet(boolean *mem) {
  boolean ret = *mem;
  *mem = TRUE;
  return ret;
  }
  ```
- Acquire Lock:
  while TestAndSet(&lock) ;
- Release Lock:
  lock = false;

Support From The OS

- If OS is non-preemptive
  - System calls can be provided for
    - Acquire Lock and Release Lock.
  - Process can not be preempted while acquiring and releasing locks.
- If OS is preemptive.
  - System calls are tricky to support but not impossible.

Previous Next

Module 21: Problem and Solution

Lecture 41: Solution to Critical Section Problem

Synchronization Support in OS/ISA

- Synchronization code can be written using locks
  while (1) {
  Non critical code
  Acquire Lock
  Critical Section Code
  Release Lock
  }
- Implementation of Lock require support from the ISA
  - TestAndSet instruction, Swap instruction
- OS may provide system calls to
  - Acquire lock or release lock.
  - For preemptive OS kernels, hard to implement these calls

Solutions for Multi-processes

```
boolean waiting[n], lock;
waiting[i] = TRUE;
key = TRUE;
while (waiting[i] && key) key = TestAndSet(&lock);
waiting[i] = FALSE;
// Critical Section
j = (i+1)%n;
while ((j != i) && waiting[j]==FALSE) j = (j+1)%n;
if (j==i) lock=FALSE; else waiting[j] = FALSE;
// Remainder Section
```

Multiprocessor Issues

- Multiple processors share a single bus.
  - Arbitration is for bus cycles
  - Atomicity across processors is at the granularity of bus cycle.
- TestAndSet or Swap instructions require
  - At least one read and one write cycle
- Bus arbitration logic has to be instructed to give bus for two cycles.
  - Lock instruction prefix in Pentium
  - For example lock xchg %ax, mem16
- Lock instruction causes an arbitration sequence to be done for the entire instruction.
  - Atomic instruction execution across multiple processors

Previous Next

Semaphores

- A data structure abstraction for lock

```
class semaphore {
private: int s;
public:
void wait(void) {while (s < 0) ;
s--;
}
void signal(void) {
s++;
}
}
```

wait and signal are Atomic Methods.
Also known as P and V.
Also known as down and up.

Mutual Exclusion Using Semaphore

```
semaphore mutex;
:
mutex.wait();
// Critical Section
mutex.signal();
// Remainder Section
:
```

Previous Next