CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2016 Lecture 16

(iClicker quiz)

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Questions from Last Time

- When you call fork ( ) how does a child process start running along with the parent process?
  - The fork n creates a separate address space for the child.
  - The child process has an exact copy of all the segments of the parent process except what fork ( ) returns
  - Some Optimization: In modern UNIX virtual memory pages in both processes may refer to the same pages of physical memory until one of them writes to such a page: then it is copied of its parent's data structures.

Hopefully the font is too tiny for you to see

Questions from Last Time

- How are locks supported by hardware?
  - Currently atomic read-modify-write
- Atomic instruction in x86?
  - LOCK instruction prefix, which applies to san instruction does a read-modify-write on memory (INC, XCHG, CMPXCHG etc)
- In RISK processors? Instruction-pairs
  - LL (Load Linked Word), SC (Store Conditional Word) instructions in MIPS
  - LDREX, STREX in ARM

Questions from Last Time

- Which is used more for synchronization? Software or hardware
  - Hardware, along with code needed
- How does test-and-set implement a lock? review

Critical Section

```
do {

     entry section
          critical section
     exit section
          remainder section
 } while (true);
```

Request permission to enter

Housekeeping to let processes to enter other

Solution to Critical-Section Problem

We want

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Speed assumptions
  - Assume that each process executes at a nonzero speed
  - No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution

- **Software solution** to the critical section problem
  - Restricted to two processes
- No guarantees on modern architectures

  Machine language instructions such as load and store implemented differently
- Good algorithmic description
  - Can shows how to address the 3 requirements

Synchronization: Hardware Support

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - test memory word and set value
  - swap contents of two memory words

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
     while (test_and_set(&lock)) ; /* do nothing */

             /* critical section */
      ….. 
     lock = false;
             /* remainder section */
    …  ..

  } while (true);
```

To break out:
Return value of TestAndSet should be
FALSE

Lock FALSE: not locked.
If two TestAndSet() are executed *simultaneously*, they will be executed *sequentially* in some arbitrary order

Colorado State University

Solution using compare_and_swap

- Shared integer "lock" initialized to 0 0 = unlocked;
- Solution:

Expected=0, new=1

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
   ; /* do nothing */
  /* critical section */
lock = 0;
  /* remainder section */
} while (true);
```

- By itself, does not guarantee bounded waiting. But see next.

Colorado State University

Bounded-waiting Mutual Exclusion with test_and_set

For process i:

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Shared Data structures initialized to FALSE

- `boolean waiting[n];`
- `boolean lock;`

The entry section for process i :

- First process to execute TestAndSet will find key == false ; ENTER critical section,
- EVERYONE else must wait

The exit section for process i:

Part I: Finding a suitable waiting process j and enable it to get through the while loop,

or if thre is no suitable process, make lock FALSE.

Bounded-waiting Mutual Exclusion with test_and_set

The previous algorithm satisfies the three requirements

- **Mutual Exclusion**: The first process to execute TestAndSet(lock) when lock is false, will set lock to true so no other process can enter the CS.
- **Progress**: When a process exits the CS, it either sets lock to false, or waiting[j] to false (allowing j to get in) , allowing the next process to proceed.
- **Bounded Waiting**: When a process exits the CS, it examines all the other processes in the waiting array in a circular order. Any process waiting for CS will have to wait at most n-1 turns

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

`acquire() {`
`      while (!available)`
`       ; /* busy wait */`
	`release() {`
	`      available = true;`
	`    }`

- **Usage**

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```


Colorado State University

Semaphores by Dijkstra

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()`
    - Originally called `P()` and `V()` **based on Dutch words**
- Definition of the `wait()` **operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the `signal()` **operation**

```
signal(S) {
    S++;
}
```

Binary semaphore: When s is 0 or 1, it is a mutex lock

Colorado State University

Semaphores in navy

"I think he just said *I'm on a boat LOL ...*"

Colorado State University

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider ***P_1*** and ***P_2*** that require ***S_1*** to happen before ***S_2***

  Create a semaphore "**`synch`**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Colorado State University

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Quiz time

- Is it time for the iClicker quiz?

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

- ```
  typedef struct{
  int value;
  struct process *list;
  } semaphore;
  ```
  ```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

```
typedef struct{
   int value;
   struct process *list;
   } semaphore;
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Colorado State University

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
      ...
       /* produce an item in next_produced */
      ...
    wait(empty);
    wait(mutex);
       ...
       /* add next produced to the buffer */
       ...
    signal(mutex);
    signal(full);
 } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```