

High-level Synchronization

CS 256/456

Dept. of Computer Science, University of Rochester

2/7/2007

CSC 256/456 - Spring 2007

1

Recap of Last Class

- Concurrent access to shared data may result in data inconsistency - race condition.
- The Critical-Section problem
 - Pure software solution
 - With help from the hardware
- Problems with busy-waiting-based synchronization
 - Waste CPU, particularly when context switch occurs while a process is inside critical section
- Solution
 - Avoid busy wait as much as possible (yield the processor instead).
 - If you can't avoid busy wait, you must prevent context switch during critical section (disable interrupts while in the kernel)

2/7/2007

CSC 256/456 - Spring 2007

2

Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore S - integer variable which can only be accessed via two atomic operations
- Semantics (roughly) of the two operations:
 - wait(S) or P(S):**
wait until $S > 0$;
 $S--$;
 - signal(S) or V(S):**
 $S++$;
- Solving the critical section problem:
 - Shared data:
semaphore mutex=1;
 - Process P_i :
wait(mutex);
critical section
signal(mutex);
remainder section

2/7/2007

CSC 256/456 - Spring 2007

3

Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {
    int value;
    proc_list *L;
} semaphore;
```
- Assume two simple operations:
 - block suspends the process that invokes it.
 - wakeup(P) resumes the execution of a blocked process P.
- Semaphore operations now defined as (both are atomic):

```
wait(S):
    S.value--;
    if (S.value < 0) {
        add this process to S.L;
        block;
    }
signal(S):
    S.value++;
    if (S.value <= 0) {
        remove a process P from S.L;
        wakeup(P);
    }
```

Does this completely solve the critical section problem?
How to make sure wait(S) and signal(S) are atomic?
So have we truly removed busy waiting?

2/7/2007

CSC 256/456 - Spring 2007

4

Mutex Lock (Binary Semaphore)

- Mutex lock - a semaphore with only two state: locked/unlocked
- Semantics of the two (atomic) operations:
lock(mutex):
wait until mutex==unlocked;
mutex=locked;
unlock(mutex):
mutex=unlocked;
- Can you implement mutex lock using semaphore?
- How about the opposite?

Implement Semaphore Using Mutex Lock

- Data structures:
mutex_lock L1, L2;
int C;
- Initialization:
L1 = unlocked;
L2 = locked;
C = initial value of semaphore;
- wait operation:
lock(L1);
C --;
if (C < 0) {
unlock(L1);
lock(L2);
}
unlock(L1);
- signal operation:
lock(L1);
C ++;
if (C <= 0)
unlock(L2);
else
unlock(L1);

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Dining-Philosophers Problem

Bounded Buffer Problem

■ Shared data

```
buffer;
```

■ Producer process

```
while (1) {  
    ...  
    produce an item in nextp;  
    ...  
    add nextp to buffer;  
    ...  
}
```

■ Consumer process

```
while (1) {  
    ...  
    remove an item from buffer to nextc;  
    ...  
    consume nextc;  
    ...  
}
```

- Protecting the critical section for safe concurrent execution.
- Synchronizing producer and consumer when buffer is empty/full.

Bounded Buffer Solution

Shared data

```
buffer;
semaphore full=0;
semaphore empty=n;
semaphore mutex=1;
```

Producer process

```
while (1) {
    ...
    produce an item in nextp;
    ...
    wait(empty);
    wait(mutex);
    add nextp to buffer;
    signal(mutex);
    signal(full);
    ...
}
```

Consumer process

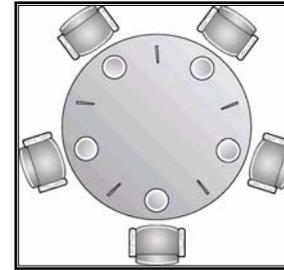
```
while (1) {
    ...
    wait(full);
    wait(mutex);
    remove an item from buffer to nextc;
    signal(mutex);
    signal(empty);
    ...
    consume nextc;
    ...
}
```

2/7/2007

CSC 256/456 - Spring 2007

9

Dining-Philosophers Problem



Philosopher i ($1 \leq i \leq 5$):

```
while (1) {
    ...
    eat;
    ...
    think;
    ...
}
```

- eating needs both chopsticks (the left and the right one).

2/7/2007

CSC 256/456 - Spring 2007

10

Dining-Philosophers Solution

Shared data:

```
semaphore chopstick[5];
Initially all values are 1;
```

Philosopher i :

```
while(1) {
    ...
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    eat;
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ...
    think;
    ...
};
```

Deadlock?

2/7/2007

CSC 256/456 - Spring 2007

11

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- Native support for mutual exclusion.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}
```

2/7/2007

CSC 256/456 - Spring 2007

12

Condition Variables in Monitors

- To allow a process to wait within the monitor, a condition variable must be declared, as
`condition x, y;`
- Condition variable can only be used with the operations `wait` and `signal`.
 - The operation
`x.wait();`
means that the process invoking this operation is suspended until another process invokes
`x.signal();`
 - The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the `signal` operation has no effect.
- Unlike semaphore, there is no counting in condition variables

2/7/2007

CSC 256/456 - Spring 2007

13

Two Semantics of Condition Variables

- Hoare semantics:
 - p_0 executes `signal` while p_1 is waiting \Rightarrow p_0 immediately yields the monitor to p_1
 - The logical condition holds when P_1 gets to run
- ```
if (resourceNotAvailable()) Condition.wait();
/* now available ... continue ... */
. . .
```
- Brinch Hansen ("Mesa") semantics:
  - $p_0$  executes `signal` while  $p_1$  is waiting  $\Rightarrow$   $p_0$  continues to execute, then when  $p_0$  exits the monitor  $p_1$  can receive the signal
  - The logical condition may not hold when  $P_1$  gets to run

2/7/2007

CSC 256/456 - Spring 2007

14

## Dining Philosophers Example

```
monitor dp {
 enum {thinking, eating} state[5];
 condition cond[5];

 void pickup(int i) {
 while (state[(i+4)%5]==eating || state[(i+1)%5]==eating)
 cond[i].wait();
 state[i] = eating;
 }

 void putdown(int i) {
 state[i] = thinking;
 cond[(i+4)%5].signal();
 cond[(i+1)%5].signal();
 }

 void init() {
 for (int i=0; i<5; i++)
 state[i] = thinking;
 }
}
```

2/7/2007

CSC 256/456 - Spring 2007

15

## Synchronization in Practice

- User program synchronization
  - for threads
  - for processes
- OS kernel synchronization

2/7/2007

CSC 256/456 - Spring 2007

16

## User Program Synchronization for Threads

- All threads share the same address space
- When only need to protect a short critical section (busy waiting is OK)
  - software/hardware spin locks
  - still has the risk of context switch in the middle of critical section
- For complex synchronization (busy waiting is not OK)
  - semaphore, mutex lock, condition variable, ...
  - may need kernel help
- In pthreads
  - mutex lock and condition variable
  - condition variable must be used together with a mutex lock

2/7/2007

CSC 256/456 - Spring 2007

17

## Synchronization Primitives in Pthreads

- Mutex lock
  - pthread\_mutex\_init
  - pthread\_mutex\_destroy
  - pthread\_mutex\_lock
  - pthread\_mutex\_unlock
- Condition variable (used in conjunction with a mutex lock)
  - pthread\_cond\_init
  - pthread\_cond\_destroy
  - pthread\_cond\_wait
  - pthread\_cond\_signal
  - pthread\_cond\_broadcast

2/7/2007

CSC 256/456 - Spring 2007

18

## User Program Synchronization for Processes

- Processes naturally do not share the same address space
- Process synchronization:
  - semaphore
  - shared memory
  - pipes

2/7/2007

CSC 256/456 - Spring 2007

19

## Spin or Block in Synchronization

- Multi-processor synchronization.
- A process is waiting for an event, triggered by another process.
- Should it spin wait or yield the processor?

2/7/2007

CSC 256/456 - Spring 2007

20

## Disclaimer

- Parts of the lecture slides contain original work of Abraham Silberschatz, Peter B. Galvin, Greg Gagne, Andrew S. Tanenbaum, and Gary Nutt. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).