CS 4410
Operating Systems

Synchronization
Spinlocks - Semaphores

Summer 2013

Cornell University

Today

- How can I synchronize the execution of multiple threads of the same process?
- Example
- Race condition
- Critical-Section Problem
- Spinlocks
- Semaphors
- Usage

Problem Context

- Multiple threads of the same process have:
  - Private registers and stack memory
  - Shared access to the remainder of the process “state”
- Preemptive CPU Scheduling:
  - The execution of a thread is interrupted unexpectedly.
- Multiple cores executing multiple threads of the same process.

Share Counting

- Mr Skroutz wants to count his $1-bills.
- Initially, he uses one thread that increases a variable *bills_counter* for every $1-bill.
- Then he thought to accelerate the counting by using two threads and keeping the variable *bills_counter* shared.

Share Counting

bills_counter = 0

- Thread A

  while (machine_A_has_bills)

  bills_counter++

- Thread B

  while (machine_B_has_bills)

  bills_counter++

***print** bills_counter*

- What it might go wrong?

Share Counting

- Thread A

  r1 = *bills_counter*

  r1 = r1 +1

  bills_counter = r1

- Thread B

  r2 = *bills_counter*

  r2 = r2 +1

  bills_counter = r2

- If *bills_counter = 42, what are its possible values after the execution of one A/B loop ?*

Shared counters

- One possible result: everything works!
- Another possible result: lost update!
- Called a “**race condition**”.

Race conditions

- Def: *a timing dependent error involving shared state*
  - It depends on how threads are scheduled.
- Hard to detect

Critical-Section Problem

bills_counter = 0

- Thread A

  while (my_machine_has_bills)

  - **enter critical section**

    bills_counter++

  - ***exit critical section***

- Thread B

  while (my_machine_has_bills)

  - **enter critical section**

    bills_counter++

  - ***exit critical section***

***print** bills_counter*

Critical-Section Problem

- The solution should satisfy:
  - Mutual exclusion
  - Progress
  - Bounded waiting

- *enter section*
  - *critical section*
- *exit section*
  - *remainder section*

General Solution

- LOCK
- A process must acquire a lock to enter a critical section.
- Hardware or Software-based implementation

TestAndSet

- **Hardware** instruction.
- **Test** and **modify** the content of **one word atomically**.

```
boolean TestAndSet(boolean *target){
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

Share Counting

bills_counter = 0
lock = FALSE

- Thread A

```
while (machine_A_has_bills){
    while (TestAndSet(&lock))
        ;
    bills_counter++
    lock = FALSE;
}
```

- Thread B

```
while (machine_B_has_bills){
    while (TestAndSet(&lock))
        ;
    bills_counter++
    lock = FALSE;
}
```

***print** bills_counter*

```
boolean TestAndSet(boolean *target){
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

Swap

- **Hardware** instruction.
- Swap the contents of two **words atomically**.

```
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Share Counting

bills_counter = 0
lock = FALSE

- Thread A

```
while (machine_A_has_bills){
    keyA = TRUE;
    while (keyA == TRUE)
        Swap(&lock, &keyA);
    bills_counter++
    lock = FALSE;
}
```

- Thread B

```
while (machine_B_has_bills){
    keyB = TRUE;
    while (keyB == TRUE)
        Swap(&lock, &keyB);
    bills_counter++
    lock = FALSE;
}
```

***print** bills_counter*

```
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

But...

- TestAndSet and Swap are complicated for application programmers to use.
- What is the alternative?

Semaphores

- Integer value
- Atomic operations
  - wait
  - signal

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

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

Share Counting

bills_counter = 0
S = 1

- Thread A

```
while (machine_A_has_bills){
    wait(S);
    bills_counter++
    signal(S);
}
```

- Thread B

```
while (machine_B_has_bills){
    wait(S);
    bills_counter++
    signal(S);
}
```

***print** bills_counter*

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

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

Spinlock

- This implementation of Semaphors, TestAndSet and Swap are spinlocks.
  - They require **busy waiting**.
- The waiting processes should **loop continuously** in the entry code.
- Valuable **CPU** cycles are **wasted**.
- Solution:
  - **Block** the waiting process.
  - **Signal** blocked process when the semaphore is “available”.

Semaphores

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

```
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);
        }
}
```

Usage

- Binary semaphore (mutex)
  - Ranges between 0 and 1.
  - Ex. Only one process can access a resource.
- Counting semaphore
  - Ranges, usually, between 0 and N.
  - Ex. N resources and M processes that share the resources.
- Synchronization
  - Ranges between 0 and 1.
  - Ex. Process A should do task At after process B having done task Bt.

Today

- How can I synchronize the execution of multiple threads of the same process?
- Example
- Race condition
- Critical-Section Problem
- Spinlocks
- Semaphors
- Usage