Critical Sections with lots of Threads

Yi Shi

Fall 2018

Xi'an Jiaotong University

Review: Race conditions

- Definition: timing dependent error involving shared state
  - Whether it happens depends on how threads scheduled
- Hard to detect:
  - All possible schedules have to be safe
    - Number of possible schedule permutations is huge
    - Some bad schedules? Some that will work sometimes?
  - they are intermittent
    - Timing dependent = small changes can hide bug

The Fundamental Issue: Atomicity

- Our *atomic* operation is not done atomically by machine
  - Atomic Unit: instruction sequence guaranteed to execute indivisibly
  - Also called “critical section” (CS)

$\Rightarrow$ When 2 processes want to execute their Critical Section,

- One process finishes its CS before other is allowed to enter

Revisiting Race Conditions

Process a:

```
while(i < 10)
    i = i +1;
print "A won!";
```

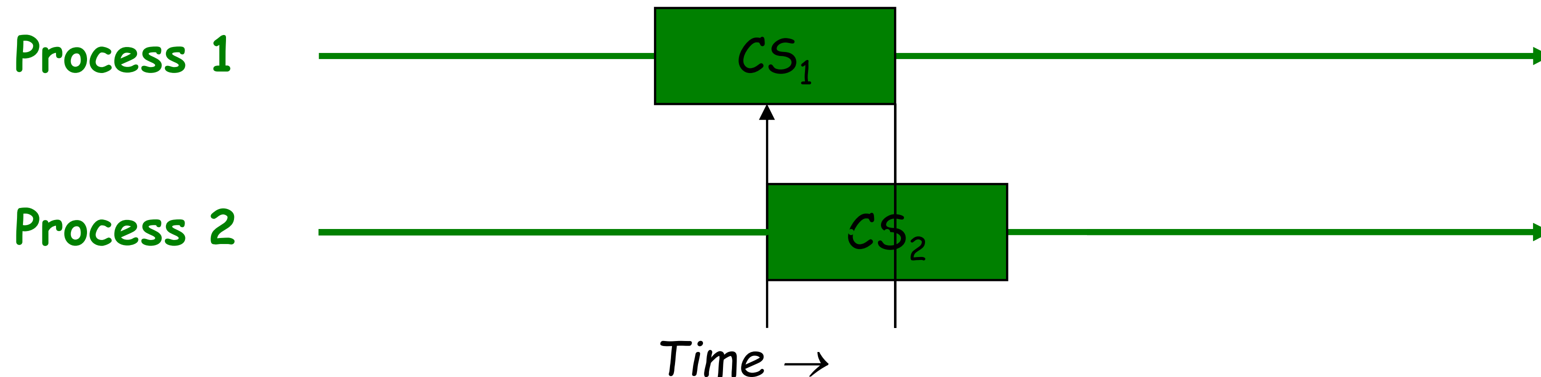
Process b:

```
while(i > -10)
    i = i - 1;
print "B won!";
```

- Who wins?
- Will someone definitely win?

Critical Section Problem

- Problem: Design a protocol for processes to cooperate, such that only one process is in its critical section
  - How to make multiple instructions seem like one?



Processes progress with non-zero speed, no assumption on clock speed

Used extensively in operating systems:
Queues, shared variables, interrupt handlers, etc.

Solution Structure

Shared vars:

Initialization:

Process:

. . .

. . .

Entry Section

Critical Section

Exit Section

Added to solve the CS problem

Solution Requirements

- **Mutual Exclusion**
  – Only one process can be in the critical section at any time
- **Progress**
  – Decision on who enters CS cannot be indefinitely postponed
    - No deadlock
- **Bounded Waiting**
  – Bound on #times others can enter CS, while I am waiting
    - No livelock
- Also efficient (no extra resources), fair, simple, …

Refresher: Dekker's Algorithm

- Assumes two threads, numbered 0 and 1

```
CSEnter(int i)
{
  inside[i] = true;
  while(inside[j])
  {
    if (turn == j)
    {
      inside[i] = false;
      while(turn == j) continue;
      inside[i] = true;
    }
}}
```

```
CSExit(int i)
{
  turn = j;
  inside[i] = false;
}
```

Peterson's Algorithm (1981)

```
CSEnter(int i)
{
      inside[i] = true;
      turn = j;
      while(inside[j] && turn == j)
            continue;
}
```

```
CSExit(int i)
{
  inside[i] = false;
  }
```

- Simple is good!!

Peterson's Algorithm (1981)

1. Mutual exclusion:

Initial: inside[*i*] = false; inside[*j*] = false; turn = *i*

Process i

```
do{
    inside[i] = true;
    turn = j;
    while(inside[j] && turn == j);

    Critical section

    inside[i] = false;

    Remainder section

} while(TRUE)
```

Process j

```
do{
    inside[j] = true;
    turn = i;
    while(inside[i] && turn == i);

    Critical section

    inside[j] = false;

    Remainder section

} while(TRUE)
```

Peterson's Algorithm (1981)

2. Liveness:

Initial: inside[*i*] = false; inside[*j*] = false; turn = *i*

Process i

```
do{
    inside[i] = true;
    turn = j;
    while(inside[j] && turn == j);

    Critical section

    inside[i] = false;

    Remainder section

} while(TRUE)
```

Process j

```
do{
    inside[j] = true;
    turn = i;
    while(inside[i] && turn == i);

    Critical section

    inside[j] = false;

    Remainder section

} while(TRUE)
```

Process i

Process j

Peterson's Algorithm (1981)

3. Bounded waiting:

now: inside[*i*] = true; inside[*j*] = ture; turn = *i*

Process i

```
do{
    inside[i] = true;
    turn = j;
    while(inside[j] && turn == j);

    Critical section

    inside[i] = false;

    Remainder section

} while(TRUE)
```

Process j

```
do{
    inside[j] = true;
    turn = i;
    while(inside[i] && turn == i);

    Critical section

    inside[j] = false;

    Remainder section

} while(TRUE)
```

Can we generalize to many threads?

- Obvious approach won't work:

```
CSEnter(int i)
{
    inside[i] = true;
    for(J = 0; J < N; J++)
        while(inside[J] && turn == J)
            continue;
}
```

```
CSExit(int i)
{
    inside[i] = false;
}
```

- Issue: Who's turn next?

Critical Sections with Atomic Hardware Primitives
test_and_set Instruction

- Definition:

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

Solution using TestAndSet

- Shared boolean variable *lock*., initialized to false.
- Solution:

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

                //    critical section

        lock = FALSE;

                //      remainder section
}
```

Assumes that test_and_set is compiled to a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)

Swap Instruction

- Definition:

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

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );
                    //    critical section
         lock = FALSE;
                   //      remainder section
}
```

Problem:
Hard and complicated for programmers to use

Presenting critical sections to users

- CSEnter and CSExit are possibilities
- But more commonly, operating systems have offered a kind of locking primitive
- We call these *semaphores*

Semaphores

- An integer with atomic increment and decrement
- Integer ‘S’ that (besides init) can only be modified by:
  - P(S) or S.wait(): decrement or block if already 0
  - V(S) or S.signal(): increment and wake up process if any
- These operations are *atomic* (indivisible)

```
semaphore S;
```

Some systems use the operation **wait()** instead of **P()**

These systems use the operation **signal()** instead of **V()**

```
P(S) {
    while(S ≤ 0)
        ;
    S--;
}
```

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

Semaphore Types

- Counting Semaphores:
  - Any integer
  - Used for applying/releasing resource
- Binary Semaphores
  - Value is limited to 0 or 1
  - Used for mutual exclusion (mutex)
- Synchronization
  - Value is limited to 0 or 1
  - Used for synchronization

Shared: semaphore S
Init: S = n;

Shared: semaphore S
Init: S = 1;

Shared: semaphore S
Init: S = 0;

Process i
P(S);
Critical Section
V(S);

Process i
Si;
V(S);

Process j
P(S);
Sj

Semaphore Implementation

- Must guarantee that no two processes can execute P() and V() on the same semaphore at the same time
  - No process may be interrupted in the middle of these operations
- Thus, implementation becomes the critical section problem where the P and V 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.

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.

Implementing Semaphores

- Busy waiting (spinlocks)
  - Consumes CPU resources
  - No context switch overhead
- Alternative: Blocking
- Should spin or block?
  - Less time ⇒ spin
  - More time ⇒ block
  - Single-processor vs. multiprocessor
  - A theory result:
    - Spin for as long as block cost
    - If lock not available, then block

```
typedef struct semaphore {
    int value:
    ProcessList L;
} Semaphore;

void P(Semaphore *S) {
    S->value = S->value - 1;
    if (S->value < 0) {
            add this process to S->L;
            block();
    }
}

void V(Semaphore *S) {
    S->value = S->value + 1;
    if (S->value <= 0) {
            remove process P from S->L;
            wakeup(P);
    }
}
```

Implementing Semaphores

- Per-semaphore list of processes
  - Implemented using PCB link field
  - Queuing Strategy: FIFO works fine
    - Will LIFO work?

Common programming errors

Whoever next calls P() will freeze up. The bug might be confusing because that other process could be perfectly
you'll
he
te!

ss J won't respect
n even if the other
the rules correctly.
e we've done two
operations this way, other
might get into the CS
ppropriately!

A typo. Process I will get stuck (forever) the second time it does the P() operation. Moreover, every *other* process will freeze up too when trying to enter the critical section!

Process

P(S)
CS
P(S)

V(S)
CS
V(S)

CS

More common mistakes

- Conditional code that can break the normal top-to-bottom flow of code in the critical section
- Often a result of someone trying to maintain a program, e.g. to fix a bug or add functionality in code written by someone else

```
P(S)
if(something or other)
   return;
CS
V(S)
```

What’s wrong?

```
Shared: Semaphores mutex, empty, full;

Init: mutex = 1;  /* for mutual exclusion*/
      empty = N;  /* number empty bufs */
      full = 0;   /* number full bufs */
```

Oops! Even if you do the correct operations, the *order* in which you do semaphore operations can have an incredible impact on correctness

What if buffer is full?

Producer

```
do {
   . . .
   // produce an
   . . .
   P(mutex);
   P(empty);
   . . .
   // add nextp to buffer
   . . .
   V(mutex);
   V(full);
} while (true);
```

mer

```
   ull);
   utex);

   // remove item to nextc
   . . .
   V(mutex);
   V(empty);
   . . .
   // consume item in nextc
   . . .
} while (true);
```

In Summary…

- Fundamental Issue
  - Programmers *atomic* operation is not done atomically
  - Atomic Unit: instruction sequence guaranteed to execute indivisibly
  - Also called “critical section” (CS)
- Critical Section Implementation
  - Software: Dekker’s, Peterson’s
  - Hardware: test_and_set, swap
    - Hard for programmers to use
  - Operating System: semaphores
- Implementing Semaphores
  - Could have a thread put itself on a “wait queue”, then context switch to some other thread (an “idle thread” if needed)
  - The OS designer makes these decisions and the end user shouldn’t need to know