Previous class...

- IPC for passing data
  - Pipe
  - FIFO
  - Message Queue
  - Shared Memory

Compare these IPCs for data passing
## Compare different IPCs for data passing

<table>
<thead>
<tr>
<th>IPC method</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes</td>
<td>Can only be used among parent and child</td>
</tr>
<tr>
<td>FIFO (named pipes)</td>
<td>Pipe is named using a string, so doesn’t have the limitation above</td>
</tr>
<tr>
<td>Message Queues</td>
<td>Supports message boundary and message types / priorities</td>
</tr>
<tr>
<td>Unix-domain Socket</td>
<td>Bi-directional; support both stream and packets; can pass file descriptors</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Data passing doesn’t go through kernel, so it is usually the most efficient one</td>
</tr>
</tbody>
</table>
Outline

• Very important concepts
  – Race condition
  – Critical section
  – Mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Synchronization without busy waiting
Race condition bug

- A race condition exists if the final program result depends on the execution sequence
- Let’s consider a counter, $c$, that is shared by two processes

```c
++c;
```

1. `movl c, %eax // copy to eax`
2. `addl $1, %eax // increment`
3. `movl %eax, c // copy back to c`

- Assume $c = 0$ initially, and the two processes both execute “++c”
- Consider the execution sequence: after process 0 executes (1) (2), it is scheduled out; and then process 1 executes (1) – (3)?
- You get $c=1$ here, while in other execution sequences you may get 2
- A concurrent program as simple as this has a race condition bug
Intuitive attempts to fix the bug

• Use atomic_add instructions
  – atomic_add supports you to do atomic increment
  – It works in that extremely simple example
  – But it is not a general solution; consider another example

```c
// critical section: withdraw $100 from account
(1) if(account >= 100)
(2)  account -= 100;
// Is it possible to withdraw $200, given account = 100?
```

• Disabling interrupts to prevent the process from being schedule out
  – Does not work well on multi-core machines
Intuitive attempts don’t work; we need a formal treatment by introducing new ideas and concepts
Critical section and mutual exclusion

- A **Critical section** is a program region that has to access shared data in a synchronized way (say, a mutual exclusion way); otherwise, race condition may occur.
- **Mutual exclusion** is to make sure no two processes are simultaneously inside critical sections that access the same shared data.
  - Each synchronization primitive that enforces mutex provides the two APIs: `enter_cs()` / `leave_cs()`
  - A critical section is surrounded by the two API calls.
Why is Mutual Exclusion important?

// Why is it impossible to withdraw 200 given account = 100?
enter_cs();
// critical section
if(account >= 100)
    account -= 100;
exit_cs();
Mutual Exclusion

- A good solution to mutual exclusion should satisfy
  - Mutual exclusion
  - No assumptions made about processor speed or number
  - No outside blocker
  - No infinite wait
A failed attempt: strict alternation

- What if process 0 quickly finishes one iteration and wants to execute another, while process 1 has not entered the critical section?
- It violates Condition 3 “no outside blocker”
Outline

• Concepts
  – Race condition, critical section, and mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Synchronization without busy waiting
Dekker’s algorithm – First solution

The two flags indicate the intention of the two processes to enter critical sections, respectively. If both intend to enter, “turn” decides who wins.

```c
//flag[] is boolean array;
flag[0] = false
flag[1] = false
turn = 0    // or 1

// P0:
  flag[0] = true;
  while (flag[1] == true) {
    if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
    }
  }

  // critical section
  ...
  turn = 1;
  flag[0] = false;
  // remainder section

// P1:
  flag[1] = true;
  while (flag[0] == true) {
    if (turn != 1) {
      flag[1] = false;
      while (turn != 1) {
        // busy wait
      }
      flag[1] = true;
    }
  }

  // critical section
  ...
  turn = 0;
  flag[1] = false;
  // remainder section
```

// indicate intension
// If it is not my turn
// back off
// Finally, it is my turn
// re-indicate intension
Compiler optimization and memory ordering

• Issues due to compiler optimizations
  – In the view of Process 0, write to flag[0] is wasted, flag[1] should be cached in registers, and turn has no hope to be changed, so they are optimized by compiler
  – “volatile” ensures that access to volatile variables are not removed or cached

• Issues due to memory ordering, e.g., between loads and stores
  – memory fences are needed
  – GCC: __sync_synchronize() // both a compiler and h/w barrier
  – asm volatile("": :: "memory") is insufficient, as it is just a compiler barrier

```c
x = 3;
y = 4;
x = 5;
while(x == 5) i++;
gcc -O2

x = 5;
y = 4;
while(true) ;
```

```c
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn = 0 // or 1

p0:
  flag[0] = true;
  while (flag[1] == true) {
    if (turn != 0) {
      flag[0] = false;
      while (turn != 0) {
        // busy wait
      }
      flag[0] = true;
    }
  }

  // critical section
  ...
  turn = 1;
  flag[0] = false;
  // remainder section

p1:
  flag[1] = true;
  while (flag[0] == true) {
    if (turn != 1) {
      flag[1] = false;
      while (turn != 1) {
        // busy wait
      }
    }
    flag[1] = true;
  }

  // critical section
  ...
  turn = 0;
  flag[1] = false;
  // remainder section
```
Peterson’s algorithm – simplify Dekker’s

```c
#define FALSE 0
#define TRUE 1
#define N 2

int turn; /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other; /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process; /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

What if two processes execute “turn = process” almost the same time?

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}

Process 0: “turn = 0” // then Process 1: “turn = 1”
Process 0: while( turn == process && ... ) // loop ends as “turn == 1”
It means whoever executes “turn = process” first wins
Lamport’s bakery algorithm – N processes

“A New Solution of Dijkstra's Concurrent Programming Problem”, Lamport 1974

```c
lock (int pid) {
    // enter doorway
    choosing[pid] = 1;
    number[pid] = 1 + max(number[0],...,number[N-1]);
    choosing[pid] = 0;

    // enter bakery
    while (j<N) {
        if (j != pid) {
            while (choosing[j]);
            while (number[j] &&
                ((number[j] < number[pid]) ||
                 ((number[j]==number[pid]) && j<pid)));
            // spin
        }
        j++;
    }
}

unlock (int pid) { number[pid] = 0; }
```

- Flag that I am drawing my number
- Flag that I have drawn my number
- Process j’s still drawing its number
- The process that has the smallest number wins
- Number 0 is reserved to indicate the exit of critical section
Outline

• Concepts
  – Race condition, critical section, and mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Synchronization without busy waiting
Hardware-assisted solutions

test_and_set(int* p){
    int t = *p;
    *p = 1;
    return t;
}

//compare-and-swap
CAS(p, old, nvalue){
    if(*p != old)
        return false;
    *p = nvalue;
    return false;
}

Load-link/store-
conditional, or LL/SC // CAS - equivalent in RISC architectures

XCHG(int *x, int *p){
    int t = *x;
    *x = *p;
    *p = t;
}

//compare-and-swap
CAS(p, old, nvalue){
    if(*p != old)
        return false;
    *p = nvalue;
    return false;
}

Load-link/store-
conditional, or LL/SC // CAS - equivalent in RISC architectures

XCHG(int *x, int *p){
    int t = *x;
    *x = *p;
    *p = t;
}

enter_region() { // lock()
    while(test_and_set(&lock) == 1) ;
}

leave_region() { // unlock()
    lock = 0;
}

enter_region() { // lock();
    a = 1;
    do { XCHG(&a, &lock);
    } while ( a == 1);
}

leave_region() { // unlock()
    lock = 0;
}
A faulty solution to the Producer-consumer problem

```java
int itemCount = 0;

procedure producer() {
    while (true) {
        item = produceItem();
        if (itemCount == BUFFER_SIZE) {
            sleep();
        }
        putItemIntoBuffer(item);
        itemCount = itemCount + 1;
        if (itemCount == 1) {
            wakeup(consumer);
        }
    }
}

procedure consumer() {
    while (true) {
        if (itemCount == 0) {
            sleep();
        }
        item = removeItemFromBuffer();
        itemCount = itemCount - 1;
        if (itemCount == BUFFER_SIZE - 1) {
            wakeup(producer);
        }
        consumeItem(item);
    }
}
```

Wakeup messages may be lost, when itemCount changes from 0 to 1, or from BUFFER_SIZE to BUFFER_SIZE-1
Outline

• Concepts
  – Race condition, critical section, and mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Next lecture: synchronization w/o busy waiting
  – Kernel-assisted solutions: semaphore
  – Language-assisted solutions: monitor
Summary

• Concepts
  – Race condition, critical section, mutual exclusion

• Pure software based locks
  – Dekker’s, Peterson’s, Bakery algorithms

• Atomic read-modify-write (RMW) instructions
  – Test-and-set, xchg, compare-and-swap
  – Mutex based on (RMW)

• How to implement lock() and unlock() using atomic RMW instructions?