Week 4 (Jan 25 - 29)

Lecture 1 (Jan 25)

Interprocess Communication and Synchronization

Problem: Race Conditions

- Processes share mem
- both read/write shared mem
- Can't guarantee read/write is atomic
- Can cause eroneous results

Critical Regions

- Area of code where only one process can be at a time
- Provide mutual exclusion to prevent race conditions
- Four conditions must hold
  - No two processes may use critical regions at once
  - No assumptions may be made on CPU speed or #
  - NO processes outside critical region may block another process
  - Processes may not wait forever to enter critical region

Strict alternation * Keep variable to keep track of whose turn it is * Waiting process waits for turn * Avoids race condition, but doesn't satisfy criterion

Bakers Algorithm

- Each process takes a number.
  - Two processes may receive same number
    - If they get number at clsoe to same time
  - Use process ID as tie breaker

Use Hardware

- Prior methods
  - Complex
  - Busy wait wastes CPU time
- Solution: Use hardware
- Several hardware methods
  - Test and Set (Atomic)
  - Atomic Swap
  - Turn off interrupts
    - Process not switched out unless it asks (not good for user programs)
- Semaphores
  - No busy wait

- Implementation
  - Semaphore S accessed by two atomic ops
    - Down (S) (AKA P)
      - At start of critical region
      - Blocks until region open
    - Up (S) (AKA S)
      - At end
      - unblocks those waiting to Down

Rendevous * Want to guarantee some operation occurs after another * Have process that should come second wait for a signal from first process

Types of Semaphores * Counting * Binary

Monitors

- High Level Synchro Primitive
  - Multiple entry points
  - Only one process in monitor at once
  - Mutual Exclusion
- Provided by high level lang
  - Variable belonging to monitor protected
- Prob: How can proc wait inside monitor?
  - Can't sleep, no one can enter
  - Sol: Use condition vars
- Two Ops
  - Wait: Suspend until signaled
  - Signal: Wake up processes waiting for signal

Locks and COndition vars * Provide monitors using special data types * Locks (Acquire, release) * Conditional (wait signal) * Lock usage * Get lock - enter monitor * Lock can be made with semaphores

Message Passing * Sync by sending messages * Two primitives * Send * Recv * ACK - how to know recv got message * Prob: Authentication - how do you know message is from correct process?

Barriers - Hold processes until all reach the barrier

Lecture 2 (Jan 27)

Livelock Example: * Two people in elevator, both saying after you, then attempting to go

IPC - Interproccess Communication

If processes aren't allowed to communicate, there can never be race condition

Communication: Affecting an observable change * If i manipulate data you can read, we're communicating

Bakery: Take a turn but declare your intent to take a turn. Take a number

Implementation of Barriers with Semaphores: Barrier b; /* contains two semaphores */ *b.bsem.value = 0;* / for barrier */ *b.mutex.value = 1;* / for mutex */ *b.waiting = 0; b.maxproc = n;* / n processes needed at barrier */

```
HitBarrier Barrier *b)
{
    SemDown (&b->mutex); // Lock because we're changing stuff
    // if everyone is done
    if(++b->waiing >= b->maxproc) {
        // Let everyonoe be free!
        while(--b->waiting >0) {
            SemUp (&b->bsem);
        }
        SemUp (&b->mutex); // unlock
    }
    else
    {
        SemUp (&b->mutex);
        SemDown (&b->bse); // unlock
    }
}
```

Classical Sync Problems

Bounded Buffers

Readers and Writers

Dining philosophers

Goal : Use semaphores

Bounded Buffers

```
// Shared
const int n;
// Num slots empty, num slots full, mutex
Semaphore empty(n), full(0), mutex(1);
Item buffer[n];

// Producer
int in = 0;
Item pitem;
while (1)
{
    // produce an item into pitem
    empty.down();
    mutex.down();
    buffer[in] = pitem;
    in = (in+1) % n;
    mutex.up();
    full.up();
}

// Consumer
int out = 0;
Item citem;
while (1)
{
```

```
    full.down();
    mutex.down();
    citem = buffer[out];
    out = (out+1) % n;
    mutex.up();
    empty.up();
}
```