

CSE 120

Summer, 2005

Day 2
Synchronization
Inter-Process Communication (IPC)

Instructor: Neil Rhodes

Independent Processes

Definition: one that can't affect or be affected

- State not shared in any way by any other process
- Deterministic and reproducible
 - Input state alone determines results
- Can stop and restart with no bad effects
 - No side-effects, other than timing
- Example: program that calculates the 10th prime

Sharing state examples

- Sharing a file
- Uses input from another process
- Generates output for another process
- Want to use the same resources as another process
 - (disk/printer/peripheral)

From theoretical point of view, independent processes are interesting

- In real life, most processes aren't independent

2

Cooperating Processes

Computation based on collection of cooperating processes sharing some state

- Want reproducible results
- Don't care about runtime/interleaving
- Can we rerun a set of cooperating processes and have it execute *exactly* the same way?
 - Not at a micro level—runtime/interleaving may be different
 - System clock must be set to same starting value
 - Disk heads must be at same locations
 - Data structures in kernel must be identical
 - Disk layout the same
- Can we get the same results?
 - Yes, possible

Why have cooperating processes?

- Sharing: one database of parts, many sales agents
- Speed: One process reads while another processes
- Modularity

3

Bank Balance Problem

Process A

```
Deposit(int amt)
{
    balance = balance + amt;
}
```

Process B

```
Withdraw(int amt)
{
    balance = balance - amt;
}
```

Assumptions

balance is a shared variable
Read and assignment are each atomic

Question

- If balance starts at 100, and we do a Deposit(50) and Withdraw(30) simultaneously, what is the ending balance?

Race Condition

- Result of computation depends on exactly which process runs when

4

Atomic operation

Indivisible

- Either completely finishes, or doesn't do anything
- Can't be interrupted

Loads and stores of single value atomic (in hardware)

- `int a;`
`int b = 3;`

`a = b`

Loads and stores of aggregate values not atomic

- `struct MyStruct a, b;`

...
`a = b;`

Normally, hardware guarantees that a single instructions is atomic

We can build up higher-level operations out of a low-level atomic operation

- Semaphores, Monitors, Signals, Mutexes.

5

Bank Balance Problem Revisited

Process A

```
Deposit(int amt)
{
    while (guard == 1)
        ;
    guard = 1;
    balance = balance + amt;
    guard = 0;
}
```

Process B

```
Withdraw(int amt)
{
    while (guard == 1)
        ;
    guard = 1;
    balance = balance - amt;
    guard = 0;
}
```

Guard is there to protect against both processes trying to manipulate balance

- Starts out at 0

Any problems?

6

Bank Balance Problem Revisited

Process A

```
Deposit(int amt)
{
    while (guard == 1)
        ;
    guard = 1;
    balance = balance + amt;
    guard = 0;
}
```

Process B

```
Withdraw(int amt)
{
    while (guard == 1)
        ;
    guard = 1;
    balance = balance - amt;
    guard = 0;
}
```

Guard is there to protect against both processes trying to manipulate balance

- Starts out at 0

Any problems?

7

Bank Balance Problem Part III

Process A

```
Deposit(int amt)
{
    while (turn == B)
        ;
    balance = balance + amt;
    turn = B;
}
```

Process B

```
Withdraw(int amt)
{
    while (turn == A)
        ;
    balance = balance - amt;
    turn = A;
}
```

turn tells whose turn it is

- 0 means A, 1 means B

Any problems?

8

Bank Balance Problem Part IV

Process A

```
Deposit(int amt)
{
    Turn off interrupts
    balance = balance + amt;
    Turn on interrupts
}
```

Process B

```
Withdraw(int amt)
{
    Turn off interrupts
    balance = balance - amt;
    Turn on interrupts
}
```

turn tells whose turn it is

- 0 means A, 1 means B

Any problems?

9

Bank Balance Problem Part V (Peterson's solution)

Process A

```
Deposit(int amt)
{
    flagA = true;
    turn = A;
    while (flagB && turn == A)
        ;
    balance = balance + amt;
    flagA = false;
}
```

Process B

```
Withdraw(int amt)
{
    flagB = true;
    turn = B;
    while (flagA && turn == B)
        ;
    balance = balance - amt;
    flagB = false;
}
```

Initialization

- turn = A
- flagA = flagB = false

Any problems?

10

Bank Balance Problem Part VI (Peterson's solution)

Process A

```
Deposit(int amt)
{
    while (tset(lock) == 1)
        ;
    balance = balance + amt;
    lock = 0;
}
```

Process B

```
Withdraw(int amt)
{
    while (tset(lock) == 1)
        ;
    balance = balance - amt;
    lock = 0;
}
```

Test and Set (tset)

- Hardware operation that does the following (atomically)
 - register = memory value
 - memory value = 1
 - set condition code

Any problems?

11

Desires

Don't want more than one process manipulating shared data simultaneously

Definitions

- **Critical region:** region of code where only one process should be executing
- **Mutual Exclusion:** if one process is using a resource, another process is excluded from that resource

Requirements for a solution

- No two processes may be inside their critical regions (mutual exclusion)
- No assumption may be made about the speed or numbers of CPUs
- No process running outside its critical region may block another process
 - Requiring alternating Deposit/Withdrawal not OK
- No process should have to wait forever to enter its critical region (starvation)

Desires for a solution

- Efficient: don't use lots of resources when waiting (no busy waiting)
- Simple: should be easy to use

12

Mutual Exclusion

How to use:

- Lock before manipulating shared data
- Unlock afterwards
- Do not lock again if you've already locked it
- Don't unlock unless you should (usually, you locked it)
- Don't spend lots of time in critical region
- Don't fail in critical region (make sure to unlock on exception)

13

Semaphores

Synchronization variable

- Value is non-negative

Two atomic operations:

- down (P)
 - If value is zero, put process to sleep
 - Else, decrement value
- up (V)
 - If process is waiting, wakeup up one waking process
 - Else, increment value

Binary semaphores

- Value is only 0 or 1

Counting semaphores

- Value represents some count

14

Bank Balance Problem Part VII

Process A

```
Deposit(int amt)
{
    down(balanceSemaphore);
    balance = balance + amt;
    up(balanceSemaphore);
}
```

Process B

```
Withdraw(int amt)
{
    down(balanceSemaphore);
    balance = balance - amt;
    up(balanceSemaphore);
}
```

Initialization

- Semaphore balanceSemaphore=1;

Any problems?

15

Semaphore

Machine independent

Simple

Solve Mutual Exclusion

Solve Synchronization

No busy wait

Can acquire multiple resources

- Each with its own semaphore

Can have many different critical sections

- Use different semaphore for each different critical section

Can work for more than 2 processes

16

Semaphore Implementation

Down and Up are atomic operations

- The entire procedures are critical sections

Must achieve mutual exclusion for those procedures using a lower-level mechanism

- Test-and-set lock
- Turning off interrupts (unless multiprocessor)
- Peterson's Algorithm

Still may have busy-wait

- But critical section is very short (not many instructions)

17

Producer/Consumer problem

Description:

- Producer process is producing data
- Consumer process is consuming data
- Buffer holds excess produced data

Constraints:

- If buffer is full, producer must wait (scheduling)
- If buffer is empty, consumer must wait (scheduling)
- No two processes manipulating buffer at the same time (mutual exclusion)

Solution

- Use separate semaphore for each constraint
 - emptySlots
 - fullSlots
 - bufferMutex

18

Producer/Consumer Solution

Initialization

- emptySlots = N; fullSlots = 0; mutex = 1;

```
producer:
down(emptySlots);
down(mutex);
get empty buffer from
empty buffer pool;
up(mutex);
Produce data in buffer
down(mutex);
put buffer in full
buffer pool
up(mutex);
up(fullSlots);

Consumer:
down(fullSlots);
down(mutex);
get full buffer from full buffer pool;
up(mutex);
Consume data in buffer
down(mutex);
Add empty buffer to empty buffer pool
up(mutex);
up(emptySlots);
```

Questions

- Why does producer down(emptySlots) but up(fullSlots)?
- Is order of downs important?
- Is order of ups important?
- Could we have two mutexes: one for each pool?
- What would we change to have 2 consumers? 2 producers?

19

Dining Philosopher's problem

Five philosopher's eating at a Chinese restaurant

- Round table
- There's a chopstick to the left and a chopstick to the right
- Each philosopher needs to chopsticks to eat
- Each philosopher loops:
 - Think
 - Eat

Come up with a simulation with following characteristics:

- No central control
- Efficient (two philosophers can eat at a time)
- Symmetric: All philosophers use the same algorithm

- If we don't need efficient, it's easy:

```
philosopher() {
    loop {
        Think();
        down(mutex);
        Eat();
        up(mutex);
    }
}
```

20

Dining Philosophers

Initialization:

- semaphore mutex=1;
- semaphore s[5] = 0;
- int state[5] = not_hungry

```
test(i) {
    if (state[i] == hungry &&
        state[left(i)] != hungry &&
        state[right(i)] != hungry){

        state[i] = eating
        up(s[i]));
    }
}

philosopher(i) {
    loop {
        Think();
        down(mutex);
        state[i] = hungry
        test(i);
        up(mutex);
        down(s[i]);

        Eat();

        down(mutex);
        state[i] = not_hungry
        test(left(i));
        test(right(i));
        up(mutex);
    }
}
```

21

Monitors

Higher-level construct than semaphores

- Must be built-in to language
- Can't make mistakes as easily as with semaphores
- Solves mutual exclusion, but not synchronization

What happens

- Entering any procedure in monitor automatically does down
- Leaving any procedure in monitor automatically does up
 - Exactly like "synchronized" functions in Java
- Only access to data in monitor is via mutually-exclusive procedures
 - In Java, non-synchronized functions still have access

Example

```
monitor {
    int balance;

    Withdraw(int amt)
    {
        balance = balance - amt;
    }
    Deposit(int amt)
    {
        balance = balance + amt;
    }
}
```

22

Synchronization with Monitors: Condition Variables

Condition variable supports 3 operations

- Wait: blocks current process and lets another process enter monitor. Will not continue until woken up
- Signal: wakes up a single blocked process on that condition variable.
- Broadcast: wakes up all blocked processes on that condition variable.

No history

- In semaphore, if operations are up(s), then down(s), the down doesn't block.
- In condition variable, if operation is signal(c), then wait(c), the wait still blocks.

Which process wakes up and enters monitor?

- Mesa semantics
 - On signal, signaller keeps monitor mutex.
 - Awakened process waits for monitor mutex just like any other process (condition it was waiting for may no longer be true since some third process may run between signaller and waked). Must loop to check condition
- Hoare semantics
 - On signal, if process waiting on condition, signaller immediately suspended
 - One of processes waiting is awakened, and is guaranteed the condition it was waiting for is true.

23

Producer/Consumer Solution

Initialization

- emptySlots = n;
- fullSlots = 0;

```
Producer() {
    loop {
        item = produce();
        ProducerConsumer.insert(item)
    }
}

Consumer {
    loop {
        ProducerConsumer.remove;
        consume(item);
    }
}
```

```
monitor ProducerConsumer {
    condition notFull, notEmpty;
    int count = 0;

    void insert(item) {
        while (count == n)
            wait(notFull);
        insert item into buffer
        count = count + 1;
        if (count == 1)
            signal(notEmpty);
    }

    item remove() {
        while (count == 0)
            wait(notEmpty);
        remove item from buffer
        count := count - 1;
        if (count == n - 1)
            signal(notFull);
    }
}
```

24