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

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

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
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.

Bank Balance Problem Revisited

Process A
Deposit(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?

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

What We Are Trying to Solve

Critical Section Problem
- A number of processes each of which have a critical section of code.

Desires
- Mutual Exclusion: Only one process can be executing their critical section at a time
- Progress: No process not in its entry, critical, or exit sections can block any other process wishing to enter its critical section.
- Bounded waiting: A process can’t wait forever to enter its critical section

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?
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
Mutual Exclusion using Locks

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)

Locks

Used for mutual exclusion

Two operations:
- **Acquire**: obtains the lock. Only one thread may have the lock at a time.
  - If the lock is already acquired by another thread, acquire blocks
- **Release**: Releases the lock. If another thread is waiting for the lock, it is awoken

Implementation

```java
class Lock

  Boolean locked = false;

  Acquire()
    if (locked)
      put thread on wait queue
      put thread to sleep
    else
      locked = true;

  Release()
    if thread waiting on wait queue,
      put the thread on the ready queue
    else
      locked = false
```

Locks

Used for mutual exclusion

Two operations:
- **Acquire**: obtains the lock. Only one thread may have the lock at a time.
  - If the lock is already acquired by another thread, acquire blocks
- **Release**: Releases the lock. If another thread is waiting for the lock, it is awoken

Implementation

```java
class Lock

  Boolean locked = 0;

  Acquire()
    while (tset(&locked) == 1)
      ;

  Release()
    locked = 0;
```
**Locks**

*Used for mutual exclusion*

```java
class Lock {
    Boolean locked = 0;
    Boolean lockGuard = 1;
    
    void Acquire() {
        if (locked)
            put thread on wait queue
        go to sleep
    else
        locked = 1;
    }
    
    void Release() {
        if thread waiting on wait queue,
            put thread on ready queue
        else
            locked = 0;
    }
}
```

**Semaphores**

*Synchronization variable*

- Value is non-negative

*Two atomic operations:*

- **down** (P or wait)
  - If value is zero, put process to sleep
  - Else, decrement value
- **up** (V or signal)
  - 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

---

**Bank Balance Problem Part VII**

**Process A**

```java
Deposit(int amt) {
    balanceSemaphore.P();
    balance = balance + amt;
    balanceSemaphore.V();
}
```

**Process B**

```java
Withdraw(int amt) {
    balanceSemaphore.P();
    balance = balance - amt;
    balanceSemaphore.V();
}
```

**Initialization**

- balanceSemaphore=Semaphore(1)

**Any problems?**

---

**Semaphore**

*Machine independent*

*Simple*

*Solve Mutual Exclusion*

*Solve Synchronization*

*No busy wait*

*Can acquire multiple resources*

*Can have many different critical sections*

*Can work for more than 2 processes*
Semaphore Implementation

P() and V() 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)

Synchronization Examples

From the (free) book *Little Book of Semaphores* by Allen Downey
- [http://www.greenteapress.com/semaphores](http://www.greenteapress.com/semaphores)

Signal Problem

Two threads A and B
- The statement a1 in A should execute before the statement b1 in B

A:

statement a1
statement a2

B:

statement b1
statement b2

- Initialization

Rendezvous Problem

Two threads A and B
- The statement a1 in A should execute before the statement b2 in B
- The statement b1 in B should execute before the statement a2 in B

A:

statement a1
statement a2

B:

statement b1
statement b2

- Initialization
Multiplex Problem

There are a number of different threads
- No more than \( n \) threads can be running their critical section at a time

Code common to all threads:
critical section

- Initialization

Barrier Problem

There are a number of different threads
- No more than \( n \) threads can be running their critical section at a time

Initialization

```c
int count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Code common to all threads:
rendezvous

- Initialization

```c
int count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Barrier Problem

There are a number of different threads
- No more than \( n \) threads can be running their critical section at a time

Initialization

```c
int count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Code common to all threads:
rendezvous

- Initialization

```c
int count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```
Turnstile

A turnstile can be:
- locked so no threads can pass
- If unlocked, allows one thread at a time to pass

```python
turnstile = Semaphore(1)
turnstile.P();
turnstile.V();
```

Barrier Problem

There are a $n$ different threads
- All of them should execute their rendezvous code before any of them executes the critical point code

Initialization
```
count = 0
mutex = Semaphore(1)
barrier = Semaphore(0)
```

Reusable Barrier Problem

There are a $n$ different threads
- All of them should execute their rendezvous code before any of them executes the critical point code. Also, when all have executed critical point, turnstile is locked again

Initialization
```
count = 0
mutex = Semaphore(1)
turnstile = Semaphore(0)
```
Reusable Barrier Problem

Code common to all threads:
rendezvous
critical point

• Initialization
count = 0
mutex = Semaphore(1)
turnstile = Semaphore(0)

t2 = Semaphore(1)

Solution, as is often the case with synchronization problems, is hard to prove correct.

Useful statements to prove:
• Only the nth thread can lock or unlock turnstiles
• Before a thread can unlock the first turnstile, it must lock the second (and vice versa). Therefore, no thread can get ahead of the others by more than one turnstile

Queue Problem

Imagine two types of threads: leaders and followers. Each waits in a queue for a match before calling dance().

• If a follower arrives, it looks for a leader in the queue. If so, both proceed
• If a leader arrives, it looks for a follower in the queue. If so, both proceed

Note:
• We should never have both followers and leaders in the queue
• A leader or follower in the queue should proceed before some newer leader or follower
Queue Problem

Follower:
dance()

Leader:
dance()

Initialization
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)

Exclusive Queue Problem

Imagine two types of threads: leaders and followers. Each waits in a queue for a match before calling dance.
- If a follower arrives, it looks for a leader in the queue. If so, both proceed
- If a leader arrives, it looks for a follower in the queue. If so, both proceed

Note:
- We should never have both followers and leaders in the queue
- A leader or follower in the queue should proceed before some newer leader or follower
- Only a single leader/follower pair can be executing dance concurrently

FIFO Queue Problem

Imagine two types of threads: leaders and followers. Each waits in a queue for a match before calling dance.
- If a follower arrives, it looks for a leader in the queue. If so, both proceed
- If a leader arrives, it looks for a follower in the queue. If so, both proceed

Note:
- We should never have both followers and leaders in the queue
- A leader or follower in the queue should proceed before some newer leader or follower
- Only a single leader/follower pair can be executing dance concurrently
- Followers must execute before later followers
- Leaders must execute before later leaders
Producer/Consumer problem

Description:
- Producer thread is producing data
- Consumer thread 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

Producer/Consumer Solution

Initialization
- emptySlots = Semaphore(N); fullSlots = Semaphore(0)
- mutex = Semaphore(1)

Questions
- Why does producer emptySlots.P() but fullSlots.V()?
- Is order of P important?
- Is order of V important?
- Could we have two mutexes: one for each pool?
- What would we change to have 2 consumers? 2 producers?

Implementing Semaphores

```java
class Semaphore {
    int count  // if neg. says how many are waiting
    int countGuard
    P() {
        while (tset(&countGuard) == 1) {
            count--;
            if (count < 0)
                add this process to wait queue
                sleep this process
            else
                countGaurd=0
        }
    }
    V() {
        while (tset(&countGuard) == 1) {
            count++
            if (count <= 0)
                remove a process from wait queue and wake it
        }
        countGuard = 0
    }
}
```

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 two 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);
      }
    }
```
Dining Philosophers

Initialization:
- `mutex = semaphore(1);`
- `s = Semaphore(0)[5];`
- `state = Array[5](not_hungry)`

```java
class philosopher(i) {  
    loop {  
        Think();  
        mutex.P();  
        state[i] = hungry  
        test(i);  
        mutex.V();  
        s[i].P();  
        Eat();  
        mutex.P();  
        state[i] = not_hungry  
        test(left(i));  
        test(right(i));  
        mutex.V();  
    }  
}
```

test(i) {
    if (state[i] = hungry &&
        state[left(i)] != hungry &&
        state[right(i)] != hungry) {
        state[i] = eating
        s[i].V();
    }
}

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 P()
- Leaving any procedure in monitor automatically does V() (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
```java
monitor {  
    int balance;
    Withdraw(int amt)  
    {  
        balance = balance - amt;
    }  
    Deposit(int amt)  
    {  
        balance = balance + amt;
    }
}
```

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 s.V(), then s.P(), the P() doesn't block.
- In condition variable, if operation is c.signal(), then c.wait(), 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 is was waiting for is true.

Producer/Consumer Solution

Initialization
- `emptySlots = n;`
- `fullSlots = 0;`

```java
monitor ProducerConsumer {  
    notFull = condition();  
    notEmpty = condition();  
    count = 0;  
    Producer() {  
        loop {  
            item = produce();  
            ProducerConsumer.insert(item)  
        }  
    }  
    Consumer {  
        loop {  
            ProducerConsumer.remove();  
            consume(item);  
        }  
    }  
}
```

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

Consumer {
  loop {
    ProducerConsumer.remove();
    consume(item);
  }
}
Using Condition Variables

while (!conditionOfInterest)
    condition.wait()

Do work

update state information and signal appropriate conditions

Readers/writers problem

Description:
- Reader thread reads data from data structure
- Writer thread modifies data in data structure

Constraints:
- Any number of readers can be in critical section at once
- Writer must have exclusive access to the critical section
- What about if readers and writers both want to execute. Who gets to?

Readers/Writers Problem

monitor readerWriter
    canRead = Condition(monitor.lock)
    canWrite = Condition(monitor.lock)
    aw = ar = ww = wr = 0

preRead()
    while (aw + ww > 0)
        wr++;
        canRead.wait()
        wr--
        ar++

postRead
    ar--
    if (ar == 0 && ww > 0)
        canWrite.signal()

write()
    while(ar + aw > 0) canWrite.wait()
    //do writing
    if (ww > 0) canWrite.signal()
    else if (wr > 0) canRead.signal()