Semaphores

- Three operations:
  - InitSem(S: Semaphore; value: integer)
    - start w/ initial non-negative value as capacity
  - Wait(S: Semaphore) -- grab resource if available
    - if S > 0 then S := S - 1 else suspend in queue assoc. w/ S
  - Signal(S: Semaphore) -- release
    - If processes waiting then wake up, else S := S + 1

Protect Critical Section

Pattern of use:

```
Wait(S);    -- grab token
{            
  Critical region
}           
Signal(S);  -- release token
```

Solve producer-consumer
Suppose have Buffer[0..MaxBuffSize-1]

Main program:

```plaintext
CreateProcess(Producer, WorkSize); -- create producer
CreateProcess(Consumer, WorkSize); -- create consumer
BufferStart := 0; BufferEnd := -1; Count := 0;
InitSem(NonEmpty, 0) -- semaphore w/o non-empty slots
InitSem(NonFull, MaxBuffSize) -- semaphore w/MaxBuffSize open slots
InitSem(MutEx, 1) -- semaphore for mutual exclusion
StartProcesses
end;
```

**Questions**

- Why is Mutex semaphore needed?
- What happens if interchange order of Wait’s in each?
- Semaphores very low level -- easy to make mistakes!
Monitors

• High level concept due to Per Brinch Hansen
  - originally developed for Simula

• Provide ADT w/condition variables, each of which has an associated queue

• Suspend (wait) enqueues process while continue dequeues processes

Producer-Consumer w/ Monitor

```pascal
type buffer = monitor;

var store: array[0..MaxBuffSize-1] of char;
  BufferStart, BufferEnd, BufferSize: integer;
  nonfull, nonempty: queue;

begin (* initialization *)
  BufferEnd := -1;
  BufferStart := 0;
  BufferSize := 0
end;
```

```pascal
procedure entry insert(ch: char);
begin
  if BufferSize = MaxBuffSize then suspend(nonfull);
  BufferEnd := (BufferEnd + 1) % MaxBuffSize;
  store[BufferEnd] := ch;
  BufferSize := BufferSize + 1;
  continue(nonempty)
end;
```

```pascal
procedure entry delete(var ch: char);
begin
  if BufferSize = 0 then suspend(nonempty);
  ch := store[BufferStart];
  BufferStart := (BufferStart + 1) % MaxBuffSize;
  BufferSize := BufferSize -1;
  continue(nonfull);
end;
```

Producer-Consumer w/ Monitor

```pascal
procedure entry insert(ch: char);
begin
  if BufferSize = MaxBuffSize then suspend(nonfull);
  BufferEnd := (BufferEnd + 1) % MaxBuffSize;
  store[BufferEnd] := ch;
  BufferSize := BufferSize + 1;
  continue(nonempty)
end;
```

```pascal
procedure entry delete(var ch: char);
begin
  if BufferSize = 0 then suspend(nonempty);
  ch := store[BufferStart];
  BufferStart := (BufferStart + 1) % MaxBuffSize;
  BufferSize := BufferSize -1;
  continue(nonfull);
end;
```

Producer-Consumer w/ Monitor

```pascal
procedure entry insert(ch: char);
begin
  if BufferSize = MaxBuffSize then suspend(nonfull);
  BufferEnd := (BufferEnd + 1) % MaxBuffSize;
  store[BufferEnd] := ch;
  BufferSize := BufferSize + 1;
  continue(nonempty)
end;
```

```pascal
procedure entry delete(var ch: char);
begin
  if BufferSize = 0 then suspend(nonempty);
  ch := store[BufferStart];
  BufferStart := (BufferStart + 1) % MaxBuffSize;
  BufferSize := BufferSize -1;
  continue(nonfull);
end;
```

Producer-Consumer w/ Monitor

```pascal
procedure entry insert(ch: char);
begin
  if BufferSize = MaxBuffSize then suspend(nonfull);
  BufferEnd := (BufferEnd + 1) % MaxBuffSize;
  store[BufferEnd] := ch;
  BufferSize := BufferSize + 1;
  continue(nonempty)
end;
```

```pascal
procedure entry delete(var ch: char);
begin
  if BufferSize = 0 then suspend(nonempty);
  ch := store[BufferStart];
  BufferStart := (BufferStart + 1) % MaxBuffSize;
  BufferSize := BufferSize -1;
  continue(nonfull);
end;
```

Producer-Consumer w/ Monitor

```pascal
procedure entry insert(ch: char);
begin
  if BufferSize = MaxBuffSize then suspend(nonfull);
  BufferEnd := (BufferEnd + 1) % MaxBuffSize;
  store[BufferEnd] := ch;
  BufferSize := BufferSize + 1;
  continue(nonempty)
end;
```

```pascal
procedure entry delete(var ch: char);
begin
  if BufferSize = 0 then suspend(nonempty);
  ch := store[BufferStart];
  BufferStart := (BufferStart + 1) % MaxBuffSize;
  BufferSize := BufferSize -1;
  continue(nonfull);
end;
```

Producer-Consumer w/ Monitor

```pascal
procedure entry insert(ch: char);
begin
  if BufferSize = MaxBuffSize then suspend(nonfull);
  BufferEnd := (BufferEnd + 1) % MaxBuffSize;
  store[BufferEnd] := ch;
  BufferSize := BufferSize + 1;
  continue(nonempty)
end;
```

```pascal
procedure entry delete(var ch: char);
begin
  if BufferSize = 0 then suspend(nonempty);
  ch := store[BufferStart];
  BufferStart := (BufferStart + 1) % MaxBuffSize;
  BufferSize := BufferSize -1;
  continue(nonfull);
end;
```
Java
Parallelism & Concurrency

Before Parallelism

- Program on single processor:
  - One call stack (w/ each stack frame holding local variables)
  - One program counter (current statement executing)
  - Static fields
  - Objects (created by new) in the heap (nothing to do with heap data structure)

Multiple Threads/Processors

- New story:
  - A set of threads, each with its own call stack & program counter
  - No access to another thread's local variables
  - Threads can (implicitly) share static fields / objects
  - To communicate, write somewhere another thread reads

Shared Memory

*Threads, each with own unshared call stack and current statement (pc for “program counter”) local variables are numbers/null or heap references*

*Heap for all objects and static fields*
Parallel Programming in Java

• Creating a thread:
  1. Define a class C extending Thread
     • Override public void run() method
  2. Create object of class C
  3. Call that thread’s start method
     • Creates new thread and starts executing run method.
     • Direct call of run won’t work, as just be a normal method call
     • Alternatively, define class implementing Runnable, create thread w/it as parameter, and send start message

Parallelism Idea

• Example: Sum elements of an array
  • Use 4 threads, which each sum 1/4 of the array

• Steps:
  • Create 4 thread objects, assigning each their portion of the work
  • Call start() on each thread object to actually run it
  • Wait for threads to finish
  • Add together their 4 answers for the final result

First Attempt

class SumThread extends Thread {
    int lo, int hi, int[] arr; // fields to know what to do
    int ans = 0; // for communicating result
    public void run() { ... }
}

to sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){ // do parallel computations
        ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        ts[i].start(); // use start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}

Correct Version

class SumThread extends Thread {
    int lo, int hi, int[] arr; // fields to know what to do
    int ans = 0; // for communicating result
    public void run() { ... }
}

to sum(int[] arr){
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++){ // do parallel computations
        ts[i] = new SumThread(arr,i*len/4,(i+1)*len/4);
        ts[i].start(); // start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ts[i].join(); // wait for helper to finish!
        ans += ts[i].ans;
    return ans;
}
Thread Class Methods

- void start(), which calls void run()
- void join() -- blocks until receiver thread done
- Style called fork/join parallelism
  - Code on previous slide generates error message as join can throw exception
    - InterruptedException
- Some memory sharing: arr field
- Later learn how to protect using synchronized.

Actually not so great.

- If do timing, it’s slower w/ small arrays than sequential!!
- Want code to be reusable and efficient as core count grows.
  - At minimum, make #threads a parameter.
- Want to effectively use processors available now
  - Not being used by other programs
  - Can change while your threads running

Divide & Conquer

- Divide in half, w/ one thread per half.
  - Each half further subdivided w/ new threads, etc. until down to single elements
  - Depth is O(log n), which is optimal
  - Then total time w/numProc processors
    - O(n/numProc + log n)

In practice

- Creating all threads and communication swamps savings so
  - use sequential cutoff about 1000
  - Don’t create two recursive threads
    - one new and reuse old.
    - Cuts number of threads in half.
Even Better

- Java threads too heavyweight -- space and time overhead.
- ForkJoin Framework solves problems
- Added in Java 7.

To Use Library

- Create a ForkJoinPool
- Instead of subclass Thread, subclass RecursiveTask<V> (or RecursiveAction)
- Override compute, rather than run
- Return answer from compute rather than instance vble
- Call fork instead of start
- Call join that returns answer
- To optimize, call compute instead of fork (rather than run)
- See ForkJoinFrameworkDivideConquerParallelSum

Handling Concurrency in Java

See ATM example