COMP 409

Concurrent Programming

CLASS NOTES

Based on professor Clark Verbrugge's notes

Format and figures by Gabriel Lemonde-Labrecque

Contents

About these notes

These notes contain several typos and missing words. Such known errors are colored red. In order to improve the readibility and accuracy of these notes, you can send me (gabriel.lemonde-labrecque@mail.mcgill.ca) your very own class notes and I will gladly merge them with the ones I have here.

1 Lecture: January 4th, 2008

1.1 Final Exam

April 23rd at 2PM

1.2 Syllabus

In this class, we'll be addressing the problem of how dealing with "multiple things happening at the same time". We can divide the problem into the following subclasses:

- Concurrent Programming
  - Think of it as a sports team
  - Team but no coach (no control)
  - Rules, referee (Know what is allowed or not)
- Parallel Programming
  - Think of it as a sports team
  - In this one, there is a coach (direction, specifics, the play)
- Multi-Processing
  - Many processes & if we have multiple CPU's (i.e. can execute at the same time)
  - processes: though are large, heavy weight structures
  - expansive to switch processes
- Multi-Threading
  - Very much like multi-processing but light-weight mechanism for doing things at the same time.
  - keep threads in the same process
  - threads share memory space, etgc. so much less data/info when switching threads

This class **will not** cover distributed programming (or very slightly). This class **will** focus on multi-threading. In a multi-threaded application, each thread executes on its own asynchronously. We don't know from the start what share each of them will get over the CPU. Most problems related to multi-threading come from thread interaction, that is reading/writing shared data and synchronizing in different ways.
We will cover 2 different models for implementing multi-threading.

1. Java (with or without new concurrency API)
   this solution works almost everywhere

2. PThreads (POSIX threads (1003.1c))
   this solution works in the C/C++/Unix world

Other models of threading/communication exist:

- Windows
- Sun
- HP
- IBM
- ...

Check out the handout on "Why threads are a bad idea?".

2 Lecture: January 7th, 2008

2.1 What is a thread (vs a process)?

2.1.1 Properties of a process

- unique identifiers.
- priorities.
- their own address space shared among all threads.
- open file handle table shared among all threads
- program code shared among all threads.
- global variables and data.
- child processes
- signal handlers

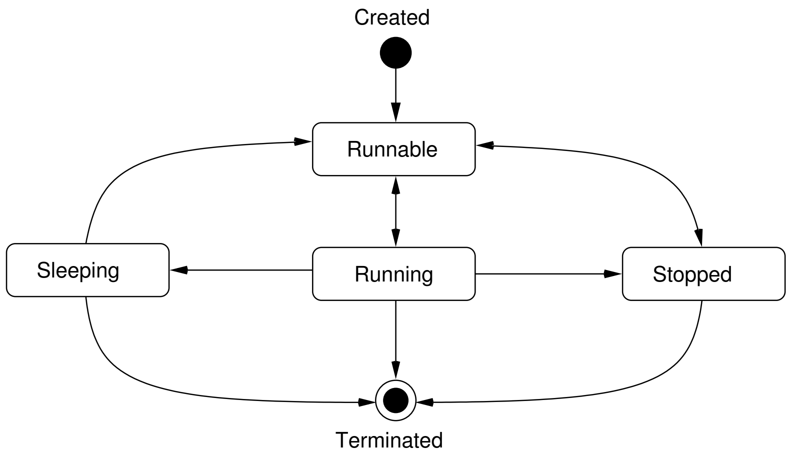
Since all threads share the address space, they can communicate by reading from and writing to this shared memory.

2.1.2 Properties of a thread

(Note that some of the following properties are common to both threads and processes)

- unique identifiers. (just like processes)
- priorities.
- also refered to as lightweight processes
- set of registers including the program counter (IP register)
- their own stack
- their own execution context
- separate call and return
- unique local variables
- separate scheduling policies (obtained from the operating system). In most cases, thread scheduling is performed by the OS itself, but it has not always been the case.

2.2 Lifecycle of a process



2.3 Achieving good performances

Amdahl's Law The program time is splitted into the sequential part and the parallel part. The law states the the speed when we execute under n processors is given as follows:

concurrent program time = sequential part + $\frac{\text{parallel part}}{n}$ (see: linear scaling)

2.3.1 What is speedup?

The speedup is a factor:

$$speedup = \frac{old-time}{new-time}$$

By Amdahl's Law, we have that...

$$s + p = 1 \Rightarrow \frac{1}{(1-p)+\frac{p}{n}}$$

Suppose 75% of our program is "parallelizible", then 25% of our program is intrinsically sequential. Hence, we have that $speedup = \frac{1}{.25+\frac{.75}{n}}$.

- $n = 1, \quad speedup = 1$
- $n = 3, \quad speedup = \frac{1}{.25+\frac{.75}{3}} = 2$

As we add up new processors, we get a much increased speedup factor.

- $n = 75, \quad speedup = \frac{1}{.25+\frac{.75}{75}} \approx 3.8$
- $n = \infty, \quad speedup = \frac{1}{.25+\frac{.75}{\infty}} \to 4$

From these calculations, we see that the benefit from multiple CPUs rapidly reach a limit.

Sometimes, we get significant speedups by parallelization.

2.3.2 What are threads good for then?

Main benefits of multithreading:

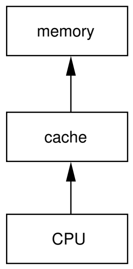
1. **increased responsiveness**, by dedicating a thread to listening to the user
2. **hide latency** (e.g. a process that is waiting for I/O from HD is hidden by executing some other thread while the former is waiting)
   If one thread must wait
   - cache miss
   - I/O

   We can switch to another thread and thus the CPU does not sit idle.

Some programs are “naturally multithreaded” (e.g. Webserver: serving each page is independent).

2.4 Concurrent Hardware

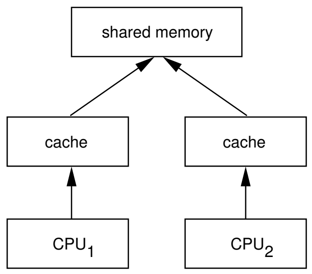
2.4.1 Basic Uniprocessor



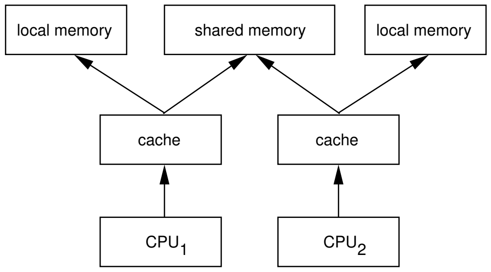
- Can use multithreading
- Switch between threads
- Cooperative
- Pre-emptive scheduling

2.4.2 Multiprocessors

- UMA (Uniform Memory Access) Scheme



- NUMA (Non-UMA) Scheme
  This scheme is more complicated than the UMA model but generally is a more suitable solution.



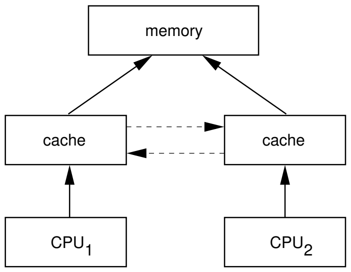
3 Lecture: January 9th, 2008

3.1 Last Time

- What are threads good for?
- Usually an upperbound limit on the amount of parallelism in an application/algorithm
  - Amdahl's Law
  - Hiding latency of I/O and cache misses
- Simultaneous applications are "naturally" multithreaded
- Basic Hardware
  - NUMA
  - UMA

3.2 Basic Hardware (continued)

3.2.1 Cache Coherence Issue



Memory must stay the same for all processes. It ends up being a little expensive. Which brings us to the next section.

3.2.2 On-Chip Multiprocessing (multiprocessors)

$cache$	$cache$
CPU_0	CPU_1

- keep both CPU's on the same actual chip
- this design has been implemented

  e.g. Core Duo

3.3 Granularity

Note: In parallel computing, granularity means the amount of computation in relation to communication, i.e., the ratio of computation to the amount of communication.

Fine-grained parallelism means individual tasks are relatively small in terms of code size and execution time. The data are transferred among processors frequently in amounts of one or a few memory words. Coarse-grained is the opposite: data are communicated infrequently, after larger amounts of computation.

The smaller the granularity, the greater the potential for parallelism and hence speed-up, but the greater the overheads of synchronization and communication.[1]

How can we reduce the "waste" due to single-threaded applications?

3.3.1 Coarse-grained multi-threading (CMT)

The idea is to reduce waste by making basically one big CPU that has support for switching between threads. It includes structures and instructions for managing threads and multiple sets of registers.

switches between threads every n cycles

n = 1 (one processor)

3.3.2 Fine-grained multithreading (FMT)

- Barrel Processing

  Definition: A barrel processor is a CPU that switches between threads of execution on every cycle. This CPU design technique is also known as "interleaved" or "fine-grained" temporal multithreading. As opposed to simultaneous multithreading in modern superscalar architectures, it generally does not allow execution of multiple instructions in one cycle.[2]

  - Cray (Tera)-Architecture
    - Some instructions can take up to ~ 70 cycles to complete.
    - No data cache.
- we need plenty of threads to hide this latency
  - even years ago, we had hardware support for 128 threads

3.4 Simultaneous Multithreading (SMT)

This was proposed by Eggers et al. (U. of Washington) 94/95.

Note: Don't confuse SMT with SMP. Symmetric multiprocessing, or SMP, is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory. Most common multiprocessor systems today use an SMP architecture. In case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.[3]

Definition: SMT is a cross between CMP and FMT.

[1]Source: Wikipedia
[2]Source: Wikipedia
[3]Source: Wikipedia

Definition: Symmetric multiprocessing, or SMP, is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory.

cache : *cache*
CPU_0:CPU_1

The separation between the two CPU is virtual.

- They can act separately
- Can be treated as one big CPU

How does this really improve? Consider a basic superscallar architecture.

Note: A superscalar CPU architecture implements a form of parallelism called Instruction-level parallelism within a single processor.

- has multiple functional units
- it can issue more than one instruction at the same time
- can have multiple Arithmetic Logic Units (ALU) that execute two things at the same time
- can have multiple Floating Point Units (FPU)
- can have multiple Branch Prediction Units (BPU)
- can have multiple Load/Store Units (LSU)

Thread 1	Thread 2
add	*fadd* (floating point add)
add	*fadd*
add	*fadd*
load	*fload*
load	*fload*

e.g. Suppose we have 2 ALU's, 2 FPU's and that we can issue 3 instructions at a time. Moreover, assume adds, fadds take 1 cycle and load, fload take 2 cycles.

In a uniprocessor environment,

cycle	$inst_0$	$inst_1$	$inst_2$	
1	*add*	*add*	–	
2	*add*	*load*	–	
3	–	–	–	this is called **vertical waste**
4	*add*	–	–	

In an SMP or CMP environment,

cycle	$inst_0$	$inst_1$	$inst_2$	$inst_0$	$inst_1$	$inst_2$
1	*add*	*add*	–	*fadd*	*fadd*	–
2	*add*	*load*	–	*fadd*	*fload*	–
3	–	–	–	–	–	–
4	*add*	–	–	*fadd*	–	–

In an FMT environment,

cycle	$inst_0$	$inst_1$	$inst_2$
Thread 1	*add*	*add*	–
Thread 2	*fadd*	*fadd*	–
Thread 1	*add*	*load*	–
Thread 2	*fadd*	*fload*	–
Thread 1	*add*	–	–
Thread 2	*fadd*	–	–

Horizontal waste remains! In practice, we find that we can issue a maximum of 2-3 instuctions/cycle. The difference is that with an SMT machine, the CPU grabs instructions to issue from any available thread.

cycle	$inst_0$	$inst_1$	$inst_2$
1	*add*	*add*	*fadd*
2	*add*	*load*	*fadd*
3	*fadd*	*fload*	–
4	*add*	–	–
5	*fadd*	–	–

In simulation, we get up to 6 instructions/cycle. But when applied in practice, we don't necessarilly get this. On the Intel processor with Hyperthreading, we only get 5 to 20% speed boost.

4 Lecture: January 11th, 2008

4.1 Last Time

- Parallel hardware designs
  - UMA: SMP
  - CMP (On-Chip MP)
  - CMT,FMT: switch threads very rapidly
  - SMT: multiple "virtual CPU's"
- SMT is very nice (in simulation)
  - In practice (Intel, hyperthreading), we only get 5% to 30% speed boost
  - Need the "right" mix of interaction from threads

:
CPU_0:CPU_1

May end up with threads "fishing" over the cache

4.2 "replay architecture"

1	2	3	4	5	6	7	$\rightarrow$

- Replay interacts poorly with SMT
- Use functional units unecessarily
- Hyperthreading/SMT (they're the same thing)
  - Modern Power 5/6
  - Sun UltraSPARC

4.3 Atomicity

In concurrency we care about relations between processes (or threads). If we have 2 completely "independent" threads, the situation is not very interesting.

When are threads independent?

Definition: The **read set** of a thread/process is the set of all variables that the thread reads. The **write set** of a thread/process is the set of all variables that the thread writes. Two threads are **independent** if and only if the write set of each is disjoint from the read and write sets of the other.

What happens when the threads read/write the same variables?

```
int x = y = z = 0;

Thread 1          Thread 2
---------         ---------
x = y + z         y = 1
                  z = 2
```

In the execution of the two latter threads, we might end up with different outputs.

$$\begin{array}{ccc} T1 : x = y + z & T2 : y = 1 & T2 : y = 1 \\ T2 : y = 1 & T2 : z = 2 & T1 : x = y + z \\ T2 : z = 2 & T1 : x = y + z & T2 : z = 2 \\ \hline \Rightarrow x == 0 & \Rightarrow x == 3 & \Rightarrow x == 1 \end{array}$$

We have to consider all possible interleavings of thread states.

e.g. for a single expression like $x = y + z$, we shall get several machine-code instructions
load $r1, y$
load $r2, z$
add $r3, r1, r2$
store $r3, x$
getting the following result

$T1:$	load $r1, y$
$T2:$	$y = 1$ $z = 2$
$T1:$	load $r2, z$ add $r3, r1, r2$ store $r3, x$

$$\Rightarrow x == 2$$

We have to think in terms of atomicity. That is we have to think how are these instructions compiled. Are they atomic or not? Notice that this example shows a resulting x with value 2 even though the expression $y + 3$ never equals 2! We need to figure out what intruction/structs are atomic (i.e. indivisible). Then we can touch at all possible interleavings and then figure out what the program does.

Hence, we need to go down to machine description, and we need to know how statements are compiled. Fortunately, in most cases, we do get some reasonable guarantees:

- A **word-size** assignment is usually atomic.
- Read is usually atomic.
  - on a 32-bit machine, a 32-bit r/w is atomic
  - on a 64-bit machine, 64-bit r/w atomic

Note: In Java, the JVM defines what is atomic or not. Java is really a 32-bit architecture. So this means that in Java, r/w is atomic. That is to say that a 64-bit r/w is not necessarily atomic

So it is possible in Java to have a long x. Suppose we have thread 1 and thread 2 running.

	Thread 1	Thread 2
Time 1:	$x == 0$	$x == -1$

The variable x could be assigned the value 0, -1 OR... it could be something close:

$$\left.\begin{array}{|c|}\hline 1111\ 1111\ 1111\ 1111\ 0000\ 0000\ 0000\ 0000 \\ \hline 0000\ 0000\ 0000\ 0000\ 1111\ 1111\ 1111\ 1111 \\ \hline\end{array}\right\}\text{with only half the word-size filled up}$$

In Java, to make these long assignments atomic, we can use the following code snippet:

```
volatile long x;
```

Notice that what we really need is not so much atomicity but (THIS IS QUITE STRONG AS A PROPERTY)... in fact, what we do need to know is that the partial state of one operation will never be visible to other threads.

5 Lecture: January 14th, 2008

Class is cancelled.

6 Lecture: January 16th, 2008

6.1 Last Time

- Atomicity
  - Some operations are atomic and some are not (e.g. x = y + z)
    - most machines r/w are atomic
    - 64-bit operations on a 32-bit machine are not atomic
  - In Java, we can convert larger operations to "appear" atomic
  - If we can make a series of operations such that no other thread can see intermediate calculations, then it appears atomic

6.2 At-Most-Once (AMO)

$x = \underbrace{y * z + g/w}_{expr}$

If y, z, g and w are not shared variables (i.e. are local) there is no intermediate that can be seen by another thread.

Consider the following code statement:

```
x = expr
```

The right-hand side, `expr`, have one or more critical references. (a reference to a variable that some other thread may write to). "`x = expr`" satisfies, the AMO property if either

1. `expr` has at most one critical reference, and `x` is not read by any other thread
2. `expr` contains no critical reference in which case `x` may be read by another thread.

A statement with the AMO property can be treated as an atomic operation.

e.g.

```
int x = 0, y = 0;

Thread 0        Thread 1
---------       ---------
x = y + 1;       y = y + 1;
```

In Thread 0, we have a critical reference in variable y. But x is not read by thread 1 so we have AMO. In Thread 1, we don't have a critical reference.

Either it will act like:

```
x = y + 1 (Thread 0)
x = y + 1 (Thread 1)
```

or

```
y = y + 1 (Thread 1)
x = y + 1 (Thread 0)
```

We only need to consider these possibilities.

Homework: Break down the following statement into machine instructions.

Hint:

```
x = y + 1
```

translates into

```
load r1,[y]
inc r1
store r1,[x]
```

e.g.

```
x = y + 1       y = x + 1
y = x + 1       x = y + 1
----------      ----------
(x = 1, y = 2)  (x = 2, y = 1)

load r1, y      load r1, x
inc r1          inc r1
store r1,x      store r1,y

        (x = 1, y = 1)
```

What does this show? Why is this important? If we look at the many interleavings to figure out what programs do, how many possible interleavings are there to be considered? If we have n theads doing m atomic actions, how many possible interleavings?

$$\text{number of interleavings} = \frac{(nm)!}{m!^n}$$

6.3 Race Conditions

One thread changes a value that some other thread's using, with no ordering. 2 threads are using the same variable but are not properly ordered/synchronized.

Definition: A **data race** occurs in multithreaded programs when 2 threads across the same memory location, with no ordering constaints, such that at least one of the accesses is a **write**.

Obviously, read-read is not a problem. Mostly, race conditions are considered bad.

We need to enforce some ordering through synchroniztion (not necessarily a fixed ordering). All of your programming should not have race conditions (Almost!). In fact, techniques for synchronizing often use race conditions intentionally). In order to face ordering, we need some way to prevent threads from entering code at different times.

The solution is to use mutual exclusion.

```
Thread 0   Thread 1
   |          |
   |          |
   v          v
   ------------
      enter();    // only 1 thread at a time
         |
         |
 critical section
         |
         |
       exit();
         |
   ------------
   |          |
   |          |
   v          v
```

Setup some entry and exit code to guarantee no more than 1 thread at the same time in the code.
How to implement mutual exclusion. Here's a trivial solution: Assume threads have ID's $\{0, 1\}$.
We are going to have an `enter()` routine:

```
int turn = 0;

enter(int threadid) {
    while(turn != threadid) {} // This is called spinning
}
exit() {
    turn = 1 - threadid;
}
```

7 Lecture: January 18th, 2008

7.1 Last Time

- Larger Atomic Pieces
  - internal/intermediate state is not visible to other thread (At-Most-Once)
  - critical section
- Race Conditions
  - shared variable between threads at least one access is write
  - if we have synchronization, ordering constraints
  - race conditions are usually errors; except sometimes we need them

7.2 Mutual Exclusion

Use...

```
enter();
------------------------
--> critical section <--
------------------------
exit();
```

This is mutually exclusive code (only one thread at a time).

(1) very trivial algorithm threadid's of 0,1

```
int turn = 0;
enter(int id) {
   while(turn != id); // Spin
}
exit(int id) {
    turn = 1 - turn;
}
```

This algorithm, unfortunately, has strong constraints.

T0, T1, T0, T1, ...

toggling, starting from Thread 0 (T0)

T0,T0 T1,T1

not possible

T1,T0,T1

(2)
```
bool flag[0] = false;
bool flag[1] = false;
enter(int id) {
    while(flag[1-id]); // Spin
    flag[id] = true;
}
exit(int id) {
    flag[id] = false;
}
```

We might have

```
 Thread 0                  Thread 1
--------------------      -------------------
 while(flag[1])            while(flag[0])
     // doesn't spin           // doesn't spin
     flag[0] = 1;              flag[1] = true;
```

So both enter the critical section: this is bad.

So, let's make the flags indicate that a thread waits to enter the critical section

(3)
```
 enter(int id) {
     flag[id] = true;
     while(flag[1-id]);  // Spin
 }
 exit(int id) {
     flag[id] = false;
 }
```

Unfortunately, this also doesn't quite do what we want

```
 Thread 0                  Thread 1
--------------------     -------------------
 flag[0] = true;           flag[1] = true;
 while(flag[1]);           while(flag[0]);
 => must spin              => must spin
```

This shows that the above algorithm may cause the threads to end up "deadlock". This, however, does satisfy the Mutual Exclusion (ME) property!

Can we have this Mutual Exclusion property without having a "deadlock"? What we can do is add some "randomness" and let our threads back out and try again.

(4)
```
 enter(int id) {
     flag[id] = true;
     while(flag[1-id]) {
         flag[id] = false;
         -> random delay <-
         flag[id] = true;
     }
 }
```

This algorithm is much the same as the third algorithm but with back out and retry.

This solution usually works. What random delay should we choose?

There's a good chance (especially if the random delay is small) that lock-step behavior will occur (with decreased chance over time)

But what we really want is a solution that works ALL THE TIME. Thus we would not want algorithm (4), because of the remaining (rather small) chance of deadlock..

If we choose a large delay

Thread 0	Thread 1
(just gets into the critical section)	(delayed)
(exit the critical section)	(delayed)
	(delayed)
	(delayed)
	(delayed)
	...

Another counter-argument to this solution is that it does not help perfomance very much.

(5) Dekker's Algorithm (Dijkstra's '68)

(6) Peterson's 2-process tie-breaker algorithm

- will enforce mutual exclusion
- won't have extra delays
- won't deadlock
- flexible

```
int turn = 0;
bool flag[0] = false;
bool flag[1] = false;
enter(int id) {
    flag[id] = true;
    turn = id;
    while(turn == id && flag[1 - id] == true);   // Spinning
}
exit(int id) {
    flag[id] = false;
}
```

```
Thread 0                                        Thread 1
---------------                                 ----------------
flag[0] = true;                                 flag[1] = true;
turn = 0                                        turn = 1
leaves the spin loop since flag[1] == false     while(turn == id \&\& flag[0] == true); // Spin
```

```
Thread 0           Thread 1
---------------    ---------------
flag[0] = true;    flag[1] = true;
turn = 0;          turn = 1;
```

Test all possibilities!

This algorithm is also **starvation-free**, meaning that if one thread is trying to go in its critical section, then it is guaranteed to go in.

8 Lecture: January 21st, 2008

8.1 Last Time

- Mutual-Exclusion (`entry()` function, critical section and `exit()` function)
  - We saw several algorithms but most of them had problems.

- A solution was the Peterson's 2-process tie-breaker (had an extra variable that "breaks ties")
  * This solution was also starvation-free (a process that access its entry function is guaranteed to enter its critical section)
  * Neither Thread 1 nor Thread 2 can dominate the critical section.
  * The tie-breaking variable is shared in read/write and both threads are looping on this variable (and the flag).
  * When one thread changes the value of a shared variable, the other CPU's (threads) need to be made aware.
  * Spinning on shared read/write variables is rather expensive.
  * Spinning on local variables is much faster. (the variable can be stored on the CPU)

8.2 Kessel's Algorithm

- Tries to ensure local spinnings
- Works very similar to Peterson's algorithm except "turn" becomes 2 variables.

```
Peterson's          Kessel's
turn == 0   <==>    turn[0] == turn[1]
turn == 1   <==>    turn[0] == turn[1]
```

  - 2 flag variables (`flag[0]` and `flag[1]`) initally set to false
  - 2 local variables (`local[0]` and `local[1]`)
  - 2 turn variables (`turn[0]` and `turn[1]`)

```
// Thread 0
enter(int id == 0) {
    flag[0] = true;
    local[0] = turn[1];
    turn[0] = local[0];
    await(flag[1] == false || local[0] != turn[1]) {}   // Spin loop
}

// Thread 1
enter(int id == 1) {
    flag[1] = true;
    local[1] = !turn[0];
    await(flag[0] == false || local[1] == turn[0]) {}   // Spin loop
}
```

Note that Peterson's and Kessel's algorithm are in the textbook.

Spinning can be done on a variable stored in that CPU's memory.

- processor 0: local[0],flag[1],turn[1]
- processor 1: local[1],flag[0],turn[0]
- allows for local spinning

8.3 Brief Interruption to Introduce Java and PThreads

8.3.1 Java Thread API

`java.lang.thread` is able to start a thread.
Two ways to start a thread in Java:

(1) Extend the thread class

```
public MyThread extends Thread {
  public void run() { // This is an "Runnable" interface
    // This is the code executed when the thread runs
  }
}
new MyThread.start();
```

(2) Implement Runnable

```
public class_something implements Runnable {
  public void run() {
    // This is the code executed when the thread runs
  }
}
// To start a new thread:
new Thread(new class_something()).start();
```

Notice the first method extends Thread, while the second uses an interface.

When changing the basic ideas of a thread (what the thread provides: functions...), inheritance (first method) is appropriate.

8.4 2 Major Kinds of Threads

8.4.1 Non-Daemon Thread (default)

8.4.2 Daemon Thread

- Exists as services
- Application exits when all threads are done but some threads are just services and should not stop the application from stopping
- Daemon threads are services threads
- Application exits when all non-daemon threads are done

9 Lecture: January 23rd, 2008

9.1 Last Time

- Kessel's Algorithm: local spin, most useful when local memory is cheaper to access
- Java Threads
  - Thread class
    - Extend
    - Implements Runnable

    - Actually start at `thread.start()`

- Daemon vs Non-Daemon Threads
    - daemon don't keep the app alive
    - non-daemon app stays alive

9.2 Java Thread API (continued)

The Thread class often has useful API's methods

- `currentThread()` returns the current thread (the one that makes the call to the `currentThread()` function)
- `isAlive()` (should be called *wasAlive*(). It says if a thread has been alive) the thread on which you call this function may not be alive once the call returns
- `enumerateThreads()` returns the list of the threads sorted in groups (you call group your threads) again, stale information (the list you set back might be changing before you use it)
- `sleep(miliseconds t)` causes the currently executing thread to pause for *t* milliseconds. `sleep()` **does not** guarantee that the thread sleeps for **exactly** *t* milliseconds. However, it guarantees that the thread will sleep for **at least** *t* milliseconds. In practice, you might get a minimum sleep time of 5-10 ms.
- `yield()` causes the currently executing thread to pause and allow other threads to execute. We don't get many guarantees with this function. Sleeping is actually only a suggestion. That is, the thread might not even give up the CPU if OS wants to schedule it next.

9.3 Java Threads Scheduling Model

This is actually how the threads are scheduled by the OS.

9.3.1 Nominally priority-pre-emptive scheduling

Threads of highest priority are always executed first.

e.g. As long as threads with priority 1 and 2 are available to execute, then no thread with priority 3 will be given a CPU share. That is, highest priority threads are executed in a round-robin fashion as long as of them has something to do, no lower priority thread is executed.

In fact, this model is NOT guaranteed. We may not even get pre-emption because pre-emption usually needs OS support. run-to-completion is also possible. We need to insert `yield()` of `sleep()`. We may not even get priorities.

Note: **Priorities in Java**: If the OS does not support priorities, then you might not get the expected result. (i.e. The priorities in Java may not match the OS priorities.)

9.4 Atomicity (continued)

- Including references (ByRef arguments?)
- 32-bits read/write
- 64-bit only with volatile

9.4.1 Basic Synchronisation

```
Object o = new Object();
```

We can use the special keyword synchronized:

```
synchronized(o) { // this is the enter() function
    -----------
    -----------      // this is a
    -----------      // critical section
    -----------
} // this is the exit() function
```

This piece of code is used to mean "Every object has a lock". The code "`synchronized(o) {`" acquires the lock. One thread at a time can own the lock. And the closing curly bracket `}` releases the lock. Note that there is no guarantee of ordering, fairness or starvation-freedom.

The following piece of code is syntactic sugar!

```
synchronized void foo() {
    -----------
    -----------
}
```

It actually stands for the following code:

```
void foo() {
    synchronized(this) {
        -----------
        -----------
    }
}
```

Depending on the implementation of the Java runtimes, one way or the other may be more effective than the other.

9.4.2 Recursive Synchronization in Java

Synchronization in Java can be recursive. We can re-enter locks we own. Must exit an equal number of times.

```
synchronized void foo() {
  foo();
}
```

9.4.3 Joining Threads

What if we need some threads to be done at a certain point? We can use the `join()` function like this:

```
Main:
...
...
t_1.start();
t_2.start();
...
...
...
```

```
t_1.join();
t_2.join();
// Every line that follows the two join() function will
// be executed only when t_1 and t_2 have terminated
```

At a join, the running thread will wait for the target thread to finish execution.

10 Lecture: January 25th, 2008

10.1 Last Time

- Java's Threading model
- Thread + API's
- Priority-pre-emptive scheduling model (but no guarantees)
- Basic synchronized

```
Object o = new Object();
synchronized(o) {
    --------
    --------
}
```

10.2 Compiler Optimization and Mutlithreading

Consider the following code.

```
int x,y;

x = someclass.var;
------------
------------
------------
y = someclass.var;
```

Can we optimize this?

```
x = someclass.var;
------------
------------ // Code where neither x nor someclass.var changes
------------
y = x;
```

In a single-threaded environment, yes. But in a multi-threaded environment, what if another thread changes `someclass.var` between these statements? Then we won't see it. This can be a problem.

```
static int x;

 Thread 1                    Thread 2
 -------------------         -----------------------------------------------
 while(x == 0); // Spin      x = 1; // this should allow Thread 1 to proceed
```

The code for `while(x == 0);` should translate to:

```
load r1, [x]  <---
test r1,0        |
jz ---------------
```

But we may get the following optimization:

```
load r1, [x]
test r1,0
jz  <----
|_______|
```

Notice we now do not re-read `x` inside loop. In practice, this may not work: thread 1's code can be optimized to a self-loop so it never sees the update to `x` by thread 2. We need to tell the compiler not to do this optimization. To do so, we use the `volatile` keyword as follows:

```
volatile int x;
```

Every time `x` is accessed, it must be re-loaded from actual memory. So it cannot store locally. This means the variable (memory location) may change asynchronously. C/C++ also has a `volatile` keyword. Java's `volatile` also counts as synchronization. In that sense, it means that if we declare a variable `volatile`, then it is not considered a data race. All shared data in your programs should either be inside a `synchronized` block (all accesses protected by the same lock). Otherwise, declare it volatile. If neither in synchronized block or declared as volatile, then we will necessarilly have a data race.

```
volatile Object q = new Object()
q.var1
q.var2b
```

The `volatile` keyword offers some guarantees over the variable `q` but in NO WAY on `var1` and `var2`.

10.3 PThreads

Definition: PThreads is a standard for multithreading on Unix-like systems. PThreads stand for POSIX Threads. The standard is defined in the POSIX Standard 1003.1c. It defines an API with specific behavior.

In many places, PThreads will have variable and system-specific behavior. The API is formed of several pieces and there's ways to test how much is implemented on a given system.

PThreads is actually provided in separate library. The argument passed to the compiler depends on which compiler you use:

- `-l pthread`: this links in the library.
- `-pthread`: this links in the library + make the rest of the run-time "thread aware".

The basic run-time relies on global variables (strtoh, errno), this hasto be overriden.
Therefore, we need a link in also the re-entrant run-time library cc_r.
Header file `pthread.h` must be included. This gives the API headers. It redefines some standards (C/C++ macros) and other things. `pthread.h` must be the FIRST include (may work if not first but do this anyway).

10.3.1 How to create a thread

You need to use `pthread_create(handle, attr, start, args);` where the parameters are defined as follows:

- **handle**: your thread reference
- **attr**: thread attributes (can be null)

- **strat**: function pointer to the main function for this thread
- **args**: input argument for that main function (this might also be null)

`attr` is a thread attribute data structure. The attribute specifies the parameters of the thread you're creating. This is actually specifying what "type" of thread it is. Note that types can be reused from thread to thread.

Most PThreads objects are built the same way. Attribute as a "type". The handle you get back is the thread identifier.

PThreads is a data structure. Therefore, YOU CANNOT COMPARE THEM WITH "`==`". USE `pthread_equal( )`.

Thread executes its start routine until it finishes that code or it can exit prematurely. Don't use the `exit()` function because that function would shut the whole application. Use `pthread_exit(retcode)`. This means that return value has to be stored until someone consumes it. After the thread ended, the thread doesn't completely disappear until that return value is consumed.

11 Lecture: January 28th, 2008

Note: Today's class notes are a courtesy of Anton Dubrau

11.1 PThreads (continued)

- All threads have a return value
- All threads must be joined by default
- They leave data behind until joined by `pthread_join(...)`
- All threads by default are **joinable**. WE MUST JOIN THEM!
- Sometimes, you don't really need to wait for the threads to finish
- Make your threads **detached** (cannot join) If you do try, you get implementation dependent behaviour since it is undefined

```
Thread:  joinable
            |
            v
         detached
```

Optional pieces

- Check at compile time or run-time
- Compile time (use `_POSIX_THREAD_ATTR_STACKSIZE`) If defined, we can set the stacksize attr.
- Run time (use `sysconf()` function and `_SC_THREAD_ATTR_STACKSIZE`)

11.2 Execution Scheduling Policies

roughly same as Java (priority pre-emptive)
You are not guaranteed to get the requested because not all systems provide such scheduling models.

- `SCHED_FIFO`
  - Thread with highest priority is executed first.
  - Threads are not time-sliced, i.e. they run to completion.
- `SCHED_RR`
  - round-robin between threads
  - Threads are time-sliced, i.e. pre-emptive.
- `SCHED_OTHER`
  - This scheduling model is not really part of the PThreads standard because implementations vary from one system to another.
  - `sched_yield()` is not part of the PThreads standard, but it is part of the 1003.1b standard.

Definition: Resource **contention**, a general concept in communications and computing, is competition by users of a system for the facility at the same time.[4]

11.3 Contention Scope

- Everyone's thread pooled, operating system does global scheduling (`PTHREAD_SCOPE_SYSTEM`)
- or, your PThreads can be scheduled in your process (`PTHREAD_SCOPE_PROCESS`)

Note: mixing scope, scheduling model is non-trivial. `PTHREAD_SCOPE_SYSTEM` + trying to set scheduling model may need root priviledge.

11.4 Basic Mutual Exclusion (mutex)

Mutexes are part of PThreads.
You create a mutex using attributes

- `pthread_mutex_int(attr)`
- should destroy mutexes with `pthread_mutex_destroy()`
- lock: `pthread_mutex_lock()` (enter)
- unlock: `pthread_mutex_unlock()` (exit)
- also: `pthread_mutex_trylock()`

In Java, the language enforces locks.

In C, PThreads behaviour is not enforced. One thread can lock, another unlock. Unlike Java, PThreads locks don't count. Locking twice by the same thread will deadlock.

Different kind of mutex/locks exist. Lots of vendor specific extensions.

- **DEFAULT** (normal): cannot recursively enter, undefined behavior on errors
- **RECURSIVE**: does allow recursive locking

[4]Source: Wikipedia

- **ERRORCHECK**: reports errors

What we can achieve with mutexes:

- Toggling
- Peterson's, Kessel's

Properties of mutexes:

(1) Mutual exclusion: 1 thread at a time

(2) Absence of deadlock

(3) No unnecessary delay

12 Lecture: January 30th, 2008

12.1 Last Time

- Java
- PThreads
  - Scheduling control
  - Mutexes

12.2 Properties of Mutual Exclusion algorithms

(1) Mutual-Exclusion (a single thread can enter its critical section at a time)

(2) Absence of deadlock

(3) No unnecessary delay

(4) Eventual entry (if the thread is trying to get in the critical section, it will eventually succeed)

Properties 1,2 and 3 are considered **safety properties** (i.e. trying to avoid bad things!!). Property 4 is considered a **liveness property** (i.e. something good must happen at some point). To prove this property's correctness often relies on the scheduling properties whose structure is not always known.

Our previous Mutual-Exclusion algorithms all realied on having two threads. If one is to implement mutual exclusion for more threads, one could use a tree. Threads enter at the leaves (bottom), and at every node one can use a 2-process algorithm to decide which thread goes first ('up'). This is analagous to running a tournament, with n threads. There are n-process versions of Peterson's; however, these are quite complicated.

12.3 Ticket Algorithm

Idea: We have a ticket dispenser and everyone takes a ticket. Next, we process the numbers in order. People wait until it is their turn.

- next (who gets in next)
- number (what the dispenser provides)

What if we want to take the next number while another thread is trying to take a number? ... So must be an atomic operation.

```
enter(int id) {
    <turn[id] = number++;> // the brackets mean the line must be executed atomically
    await(turn[id] == next); // Spin
}
exit(int id) {
    next++ // does this need to be atomic?
}
```

Variation on the ticket algorithm: called the Bakery Algorithm

12.4 Bakery Algorithm

Same thing but with a broken dispenser. The threads decide among themselves who's next. What we do is we inspect every other thread when we enter.

```
 enter(int id) {
     <turn[i] = max(turn(i = 0..n)) + 1> /* We look at the thread id of every other thread
                                            we choose a number one more than the largest
                                            number already awaiting */
     for(int j; j < n; j++) {
         if(j == id) continue;
         await(turn[j] == 0 || turn[id] < turn[j]); // Spin
     }
     exit(int id) {
         turn[id] = 0;
     }
 }
```

How can we implement such brackets < and > efficiently? The algorithms for enforcing Mutual-Exclusion are complex. Can we find something faster and easier?

12.5 Hardware Support for Concurrency

Modern processors actually help us doing so by providing special instructions.

12.5.1 Test-and-Set

```
//sets x to y & returns the old value of x atomically
TS(x,y) {
 <temp = x
  x = y
  return temp>
}
```

We can use this to build a simple lock.

```
int lock = 0 // 0 means CS is free, 1 means CS is busy
enter(int id) {
    while(TS(lock,1) == 1); // Spin
}
exit(int id) {
    lock = 0;
}
```

12.5.2 Fetch-and-Add

```
 // FA(v,c), the idea is to set v = v + c and return the old/new value of v atomically
<temp = v;
v = v + c;
return temp>
// }
```

FA works nicely in the ticket algorithm in order to implement the "`number++`" statement.

12.5.3 Compare-and-Swap

```
 // CAS(x,a,b) Only does the assigment if the comparison succeeds.
 <x = b IF x == a>
```

Returns a boolean to indicate succeed/failure. Easily build locks (algorithm TS). But there is a problem with this algorithm, discovered in '83 by IBM system 370: The A-B-A problem!

12.6 The A-B-A Problem

```
x = v;
.
. // no one changes v between
.
CAS(v,x,-)
```

We have to be careful if the value is changed and then changed back. If we kept version number for each variable, increamented at every write, then CAS could perhaps check the version numbers as well.

The solution to the A-B-A problem is to:

(1) **double-word CAS**: [8bit=value, 24-bit=version number]

(2) **Load-Linked/Store-Conditional LL/SC (/VL)** (VL stands for validate): Uses 2 instructions special read. Puts a reservation number on the value read. Special write checks the reservation. Succeeds or fails in the write. Then check the success and branch on that.

13 Lecture: February 1st, 2008

13.1 Last Time

There is built-in hardware support for special instructions executed atomically. These operations are used to build lock algorithms.

1. **test-and-set**: atomic R/W
2. **fetch-and-add**: atomic addition and read
3. **compare-and-swap**: atomic if and write, careful A-B-A problem; solution: variable version number.

13.2 Load-Linked/Store-Conditional (/Validate) or LL/SC (/VL)

Implemented on Power (IBM) and Sun. Two pieces/instructions (possible to render any two instructions atomic). You can use to make larger atomic code.
Basic Idea:

1. **load**: puts a reservation on that memory location. If anyone writes between, then reservation is erased.

2. **store**: checks the reservation (and sees if it's still valid). If it is, then return true. Otherwise, return false.

PowerPC Syntax:

- **lwarx** (**L**oad **W**ord **a**nd **R**eserve Inde**x**ed) implements LL.
- **stwcx** (**St**ore **W**ord **C**onditional Inde**x**ed) implements SC.

The statement `x++;` is not usually compiled as atomic. It is usually translated to:

```
load x
inc x
store x
```

Using the PowerPC syntax, we can make `x++;` atomic by writing:

```
redoit: //label
lwarx r1,0,x
addi r1,r1,1
stwcx r1,0,x
bne redoit
```

This piece of code load variable x to register $r1$ and puts a reservation on this memory location. Then we add 1 to $r1$ and attempt to write the new value to memory location x. If x was modified by some other thread, then we cycles until the operation succeeds. When it succeeds, the operation on x appears atomic. In practice, we use LL/SC for building simple locks (although all will work).

Wait-free synchronization (difference in expressiveness)

```
LL/SC; CAS
FA, TS
atomic R/W
```

Note: Works for unbounded number of threads

Can we speed up locking?

13.3 Lamport '87 (Fast Mutual Exclusion)

Definition: A new solution to the mutual exclusion problem is presented that, in the absence of contention, requires only seven memory accesses. It assumes atomic reads and atomic writes to shared registers.[5]

- shared variable access is more expensive than local variable
- minimize number of shared variables
- contention is rare!
- when 2 or more threads actually compete for the critical section. In practice, most threads will enter and leave no competition.
- contention, in practice, is usually no more (rule-of-thumbs) $\leq 2\%$ execution time. So we will focus optimization effort on the uncontended case even if contended is slow
- minimize shared variables in the absence of contention
- rough argument:

[5]Source: acm.org

  - every thread executes the same operations
  - we only care about shared variables

- what could be the first operation?
  - Could be a read or a write
  - Read x. If they all read first, they all get the same value (it won't help!)
  - First statement, must be a "write x"

- could the next statement be a write?
  - If it was, we could've encoded both writes as one write.
  - Second operation must be a read. But read x or read y? If it was a "read x", then every thread could write x, read x (everyone just reads what they wrote)

- therefore, must be a "read y"
  - we know that we have
    * write x
    * read y
  - but no point of reading without writing. therefore, there are 2 more statements
    * read x
    * write x

- What is the last operation before any thread enters the critical section? If it was "write y", then it wouldn't help decide who gets in to the critical section. So we know that the last operation is "read x".

Best possible sequence for Mutual-Exclusion:

```
1. write x
2. read y
3. write y
4. read x
```

We can produce an algorithm with timing-related requirements.

```
enter(int i) { // assume i != 0
    start:
    x = i;
    if(y != 0) goto start;
    y = i;
    if(x != i) {
        delay(); /* this function needs to know how long a thread needs to execute
                    the entry code and the critical section */
        if(y != i) goto start;
    }
} // end of enter() (we're done with the entry code)
```

This works, but does require timing info. Delay must be long enough that a thread trying to get inside that finds `y == 0` gets to the end part of the if, or exits the critical section.

Drawbacks:

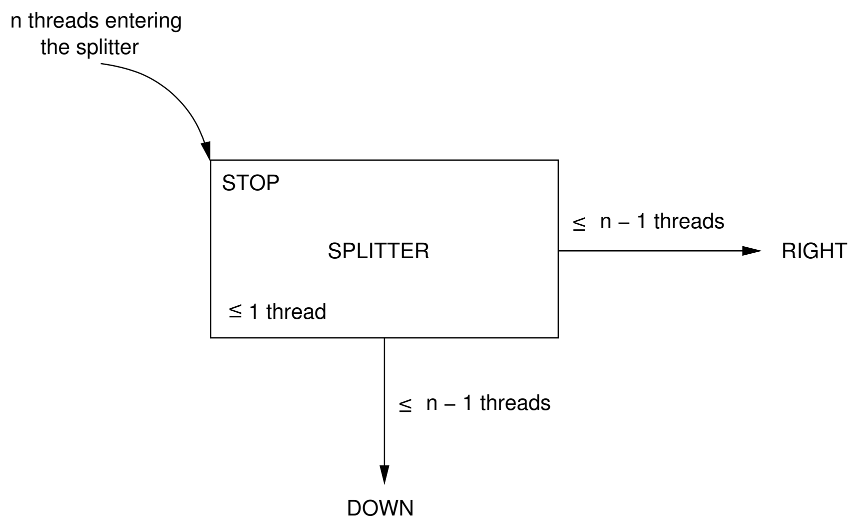
- Requires timing info
- Unecessary delay is possible

13.4 Splitter

We can design a better algorithm with no delay, as something called a splitter.

Can allow n threads into the splitter. The threads then have 3 possibilities:

1. stop: guarantees enter CS (no more than one thread in)
2. $n - 1$ threads go down
3. $n - 1$ threads go right



```
x, y;
x = id; (not 0)
if(y != 0)
    dir = right;
else {
    y = id;
    if(x != id)
        dir = down;
    else
        dir = stop;
}
```

14 Lecture: February 4th, 2008

14.1 Last Time

Fast Mutual Exclusion (Lamport)

- Minimize locking cost
- Contention is rare. Optimize the uncontended case at the expense of the contended.
- splitter: 2 shared varables

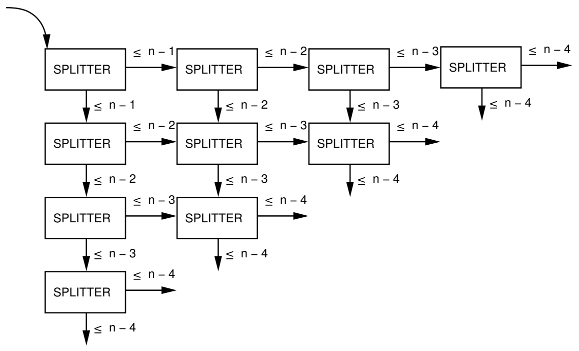
14.2 Splitter (again)

put splitter together to make network of spitters:

If put in chain, then we can make a mutual exclusion algorithm!

We can use this to build a lock. We link them together to form a tree. We have an unbounded number of threads if we have an unbounded number of splitters. But threads may never stop in the splitter.

process renaming grid: Assume thread id's to be $0..n-1$. Then index an array by thread id. In practice, we may find that these IDs can be unbounded. e.g. by creating/destroying a using pointers for ids, we can use splitters to rename our threads.



With n threads entering. If at most one thread enters, it is guaranteed to stop. The idea is to make a network where no matter paths threads take, they must all eventually stop. Each thread then hash unique number. Notice that there is an upper bound on thread id's of $O(n^2)$.

14.3 Practical Lock Design

e.g. How is `synchronized` implemented in Java?

How long should a thread wait to enter a lock?

Spinning is most effective when it's for a short period of time.

Threads can spin

- short critical section (OK)
- long critical section (BAD)

They can tell the OS that they should not be scheduled.

- are not runnable
- e.g. `sleep()`
- relatively slow
- good for long critical section

2 appraches to Mutual exclusion

(1) spinning

(2) blocking

When blocking, if you cannot get into the critical section, just block (sleep) until you can get in.
The way it's modeled in Java is called Thin Locks.

14.3.1 Thin Locks, Mauricio J. Serrano, 1978

There is a third approach to mutual exclusion. This approach is an hybrid design between spinning and blocking.

Let us suppose there is a thread (A) already owning the lock. A new thread (B) tries to acquire the lock and fail because A is already owning it. If we use a Thin Lock, Thread B will react as follows:

1. Spin for a while.
2. If it is spinning for too long, then go to sleep (block).

We say that this approach is adaptive because it starts fast and then switch to a slow mode whenever it is necessary.

We need to keep in mind that we want to optimize the most common case, that is what happens frequently, must execute fast. Notice that it is usually the uncontended case and the shallow recursive locking that happens most often.

Let's define these levels of contention. We will say that there is a...

- **shallow contention** when...
  - One thread owns the critical section.
  - Another thread shows up.
- **deep contention** when...
  - One thread owns the critical section
  - At least one other thread already waiting to get in.
- **shallow recursive locking** when...
  - Only relock something a few times
- **deep recursive locking** when...
  - Lots of recursive locking

Using a Thin Lock, a spinning thread with switch to slow mode when we get to a deep recursive contention.

Lock Word (24 bits):

Lock word in every object. Try to keep it small.

1 bit	15 bits	8 bits	8 bits
"shape" bit	thread id	recursive count	reserved

- **shape bit**:
  - bit 0 indicates whether the lock is a thin lock (fast, spinning)
  - bit 1 indicates a fat lock (slow, waiting)
- **thread id**: The owner of the lock/object, or 0 if the lock is free. We may need this to index a table if you need more than 15 bits for an id.
- **recursive count**: how many times the owner has locked it, minus one.

We can only represent recursive locking up to 256 times. Use C.A.S (compare and swap, or LL/SC). Uncontended, not locked. C.A.S in the lock word looking for 0 in the shape, then, count fields. C.A.S(lock, 0, id): atomically check that it's 0 and change it so our id is in the id field. To unlock, write 0 in the lock word. Lock in a recursive case. our CAS will fail because the owner is not 0. Check that the actual id is ours. If so, then this is recursive locking. increment our recursive count (+256) if this would overflow. However, if the count is already 255, then we need to transition to a fat lock. Similarly, if we find another thread id, then the lock is already owned. Therefore, this is the contention case.

Note: PThreads mutexes are considered fairly slow

Fat locks are implemented by spinning over a mutex lock.

B tries to get a lock, A owns it. B spins until lock is available or changes into a fat lock. If change to fat lock, then it waits on the fat lock `mutex_lock()`. If it becomes available, B enters and then B hanges it into a fat lock. Changing to a fat lock consists of allocating a new mutex to associate with our lock.

Fat Lock (24 bits):

1 bit	23 bits	8 bits
1	index	reserved

23-bit index into a table of mutexes.

15 Lecture: February 6th, 2008

15.1 Last Time

- Splitters
  - process renaming grid
- In Java, use PThreads mutexes to implement `synchronized` (relatively slow)
- Practical locking
- Thin Locks
  - CAS operation
    - Fast, but assume no contention and shallow locking
    - If there is contention, or recursive locking
- Fat Lock (i.e. a PThreads mutex): this algorithm tries to be adaptive (i.e. start fast, and only go to the slow mode if necessary) (one way change!!)

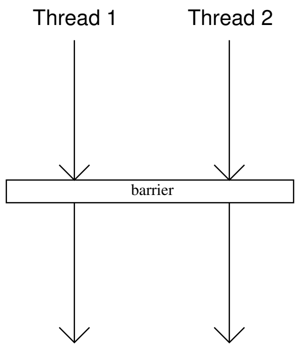
15.2 Tasuki Locks

Fat lock was impoved by Onodera et al. The idea is to use an extra bit (contention bit). Must be in a different location than the lock word. Allows for lock defletion (Fat Lock - Thin Lock).

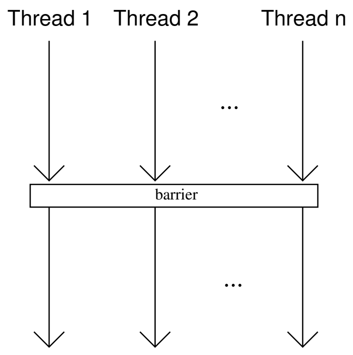
This is a read benefit. Do we have locality of contention? Mostly true, but not always. Dynamically, this is not necessarily true.

15.3 Barrier

The old model the prof showed in an previous class:



A more general barrier:



The purpose of barriers is to prevent threads from executing until they've all reached the barrier.

A single/naive solution:

```
volatile boolean flag = false;
volatile int count = 0;
//n-thread barrier:
<count++;>   // We can implement this atomically with lcount = FA(count,1)+1
             // Note lcount stands for local count
if(count==n) {
    flag = true;
} while (!flag);
```

This does work but only works once: it is not reusable! So we have to think of this scenario when the threads loop around this barrier.

As a solution, we could use (sort of) '2 non-reusable barrier'

- first, ensures all n threads show up
- second, ensures all n threads have left

Here's a reusable solution:

```
boolean localgo;
localgo = flag;
```

```
 lcount = FA(count, 1) + 1
 if(lcount == n) {
     count = 0;
     flag = !flag;
 }
 while(localgo == flag); // Spin
```

Blocking solution:

If spinning would be too expensive, better if we just sleep (block) instead. Wait for the owner thread to exit and let us know the CS is non-free.

```
 Thread 1          Thread 2
 .                 .
 .                 .
 .                 .
 .                 .
 lock             if(locked)
 .                   sleep();
 .                 .
 .                 .
 .                 .
 .                 .
unlock
wakeup
```

Problem of lost wakeups:

Thread gets into the critical section. During that time, another thread has found it locked and it decided to sleep (block). It could enter and leave before the thread that wants to sleep does.

How can we fix this? We need better primitives. We can use semaphores (Dijkstra, 60's).

15.4 Semaphores

A semaphore is an **ADT** implementing an integer value **counter**.

2 operations:

(1) `P()/Down()` operation: tries to decrement the semaphore value. But it cannot decrement it below zero. If it's already set to 0, then it will block (i.e. sleep).

(2) `V()/Up()` operation: tries to increment the semaphore value. This one never blocks! But if we increment it to 1, then you should wakeup a waiting/sleeping thread. Once woken, try to complete the `P()` operation.

`P()` and `V()` are supposed to be atomic.

We will wake up a thread in the up operation if any is waiting. But which thread are we going to wake up? Answer is: an arbitrary one! (We rarely get guarantees)

Blocking:

```
 T1          T2                T3
 |           |                 |
 P()         |                 |
 |           P() -> blocks     |
 |          |                |
 V() -> wake up someone      |
            |                P() -> succeeds
            |
```

So Thread 2 stays asleep (blocks) .

Few kinds of semaphores:

15.4.1 Binary Semaphore

- Binary semaphores (simplest)
- counter is only either 0 or 1
- much like a mutex or synchronized
  - semaphore initial value of 1
  - `enter{ P() }`
  - `exit{ V() }`

Mutex is slightly different. Has a concept of ownership. A mutex is owned by a thread. If you lock a mutex, you own it. Only the owner can unlock.

```
synchronized() {

}
```

The brackets are the `enter()` and `exit()` function!

Binary semaphore doesn't require ownership. 1 thread ends `P()`. 1 enter condition `V()`. Mutex is changed open/unlocked to begin with. Binary semaphore could start at 0.

15.4.2 General n-counting Semaphore

The counter still has a mininimum of 0 but has no specific maximum. We can use this to indicate resource limitations.

e.g. 10 of some resource. all threads do `P()`. 10 of them succeeds; the 11^{th} blocks.

16 Lecture: February 8^{th}, 2008

16.1 Last Time

- Tasuki Locks vs Thin Locks
- Barriers
  - One-Shot Barriers
  - Reusable Barriers
- Semaphores: lost wakeup solution (blocking)
  - Operations `P()`, `V()`
  - Flavours of semaphores
    - Basic semaphores (basic semaphore value is always 0 or 1)
    - Counting semaphores
  - Semaphores do 2 things
    1. Enforce mutual exclusion
    2. Signalling

16.2 Semaphores (continued)

16.2.1 Signalling Semaphores

Binary semaphore, but we start at the value 0

e.g. Build a signal 2-process barrier.

```
Semaphore S1 = 0, S2 = 0;

T1              T2
------          ------
V(S2)           V(S1)
P(S1)           P(S2)
```

16.2.2 Split-Binary Semaphores

Set of binary semaphores such that at most one of them has the 1.
(0,0,0,0,1) (0,0,0,0,0) (1,0,1,0,1,0)

```
Producer:                    Buffer:        Consumer:
--------------------        -----------     ----------------------
Generate data and                           removes data from
puts into the buffer.       [.......]       buffer and consumes it
```

Producer shouldn't overfill the buffer, and no data should be lost.

```
Semaphore lock = 1;
volatile int buffer;
volatile boolean filled;
```

Producer:

```
while(true) {
  P(lock);
  if(!filled) {
    buffer = produce();
    filled = true;
  }
  V(lock);
}
```

Consumer:

```
while(true) {
  P(lock)
    if(filled) {
      consume(buffer);
      filled = false;
    }
}
```

This still spins, not making good use of ability to block. A better solution is to use two semaphores:

```
Semaphore filled, empty;
filled = 0;
empty = 1;
```

Producer (2-semaphore solution):

```
while(true) {
  P(empty);
  buffer-produce();
  V(filled);
}
```

Consumer (2-semaphore solution):

```
while(true) {
  P(filled);
  consume(buffer);
  V(empty);
}
```

Notice that this solution uses split-binary semaphore.

We use this solution if we want more than 1 data from capacity in the buffer.

If the buffer holds n data, then we can just initialize *empty* to n (i.e. change to counting semaphore)

Note that the property that `empty` + `filled` $\leq n$.

Java has semaphores (now) (see `java.util.concurrency`)

But you don't need it because you can build it yourself.

PThreads does not have semaphores. (at least, in the main spec 1003.1c) They are there but not part of the same spec (i.e. 1003.1b).

Semaphores have some disadvantages:

- They do combine signalling and mutual exclusion. This is not ideal.
- `P()`, `V()` are separated (that makes it easy to make mistakes) debugging is difficult.

16.3 Monitors: Another Blocking Model

All operations are collected together.

A monitor is an ADT:

```
|---------------------|
|   private data      |
|---------------------|
|  monitor method     |
|                     |
|  monitor method     |
|                     |
|  monitor method     |
|                     |
|  ...                |
|---------------------|
```

Some properties of monitors:

- Only one thread in the monitor at one time.

- All monitor methods are mutually exclusive.
- Permits data in the monitor is only accessed when inside a monitor methods.
- Monitor methods only access internal, private data and parameter data.

In Java, this is easy:

```
public class monitor {
    private int x;
    public synchronized foo() {

    }
}
```

You can enforce this yourself. Java gives you the tools to build the monitors, but DOES NOT provide any monitors *per se*.

In PThreads, we can build monitors as well by using one mutex. Acquire it at the beginning of any monitor method, and release it at the end. Private data is only accessed in monitor methods. The signalling is done separately, using a special construct called a "condition variable". It Lets you signal other threads.

17 Lecture: February 11th, 2008

17.1 Last Time

- Semaphores
  - Counting Semaphores
  - Binary Semaphores
    - Mutex
    - Signalling
  - Split-Binary Semaphores
- Producer/Consumer Problem
- Monitors
  - Mutual-Exclusion (Mutex Lock)
    - procedures
    - private data
  - Need something to be true before. Proceed further. Check the condition. If not true, then leave the monitor and go to sleep and be wakeup.

17.2 Condition Variables

Syntax: `<await(b) S>`

- *b* is a boolean condition
- *S* is some statement

```
enter()
while(!b) {
    exit();
    // hopefully, some other thread enters and makes b true
    enter();
}
S;
exit();
```

Nicer if we could go to sleep (careful of a lost wakeup problem).

```
while(!b)
    exit() //check if anyone is sleeping -> no, doesn't wake anyone
    sleep()
    enter() // nobody to wake it up
```

Make what's inside the loop atomic so that there is no lost wakeup.

Condition Variables are always associated with monitors. Use "inside" monitors (mostly).

To enter a monitor method, we must:

(1) Acquire a mutex lock

(2) Check the condition variable

(3) `wait()` on it (atomic exit and sleep.)

(4) Signal on it (notification). This signal wakes up a thread which is sleeping if there is one. There is no guarantees of ordering for threads. An arbitrary one is chosen out of the pool of sleeping threads.

In Java,

```
 synchronized(o) { // lock o

     o.wait(); // atomically unlock o and sleep, then wait to be woken
               // re-enter the monitor, must acquire the lock again

 } // unlock o
```

Wait: Release the lock. Go to sleep. Wake up.

Compete for the lock in order to continue. Along with all other threads trying to acquire the lock.

`notify()` in Java is a signalling operation.

```
synchronized(o) {
    o.notify(); // wakes up one arbitrary thread
}
```

Using wait/notify for the Producer/Consumer problem.

```
class M {
    private boolean full = false;
    private int buffer;
    synchronized void produce(int d) {
        if(full) { // If buffer is full
            wait();
        }
        buffer = d;
        full = true;
        this.notify(); // The "this" keyword is not necessary here
    }

    syncrhonized int consume() {
        if(!full) { // If buffer is not full
            // If someone is sleeping, we want to wake them up
            wait();
        }
        rc = buffer;
        full = false;
        notify();
        return rc;
    }
}
```

Java keeps your condition variable and lock together for you. We cannot use the `wait()` function outside the `synchronized` block.

In PThreads, you make the assumption yourself. Monitor by allocating and using a mutex. In all monitor procedures, you lock on enter, and unlock on exit. Allocate the condition variable(s) separately, and you must be sure to always use them correctly.

```
produce()
    mutex_lock(&m) // m is the mutex
    if(full) // ERROR
        cv_wait(&m,&c);  // c is the CV
    buffered; full = true;
    cv_signal(&c)
    mutex_unlock(&m);
```

In Java, you can only notify inside a monitor method.

In PThreads, `cv_signal(&c)` is not always inside the monitor. If you want, you can notify (signal) outside the monitor. Sometimes, our boolean condition may not be guaranteed to not change between notification and the notified thread continuing in the monitor.

Solution: Put the `wait()` in a loop.
Instead of:

```
if(full)
    wait();
```

Do:

```
while(full)
    wait();
```

Double-checks the condition on wakeup and goes to sleep again if necessary. Note that the “if” method is a WRONG way of doing it!
In almost every system (Java and PThreads included), `wait()` may have spurious wakeups. Threads may wake up without notification. That is why you must always `wait()` in a loop.
Spurious Wakeups:

- Allows efficient implementation
- Having a one-to-one mapping is difficult when system calls may or may not complete
- Easier to re-issue wakeups than guarantee 1 goes through.

Simple implementation:

```
cond_wait();
    release_mutex; sleep(random);
cond_signal();
    nop;
```

18 Lecture: February 13th, 2008

18.1 Last Time

- Monitors
  - Mutual-Exclusion (mutex)
  - Condition Variables
    - Java: `wait()`, `notify()`
    - PThreads: `cond_wait()`, `cond_signal()`
    - WRONG WAY:

      ```
      if(b)
          wait();
      ```

      Thread can come up from these wait statements simulteneously.

      RIGHT WAY:

      ```
      while(b)
          wait();
      ```

18.2 Broadcasting

We may have many threads waiting (on the same condition). But so far, we have only seen the `notify()` (or `signal()`) method whose function is to wake up only one (arbitrary) thread.

We also need a broadcast version of `notify()` in order to wake up all of them. To do that, we set up conditions so that all but the initial thread are back to sleep. For example, in the Producer/Consumer problem, we may have many producer and consumer threads (instead of just 1 of each). To guarantee that a producer notify will reach a consumer (and vice versa), use `notifyAll()` (`cond_broadcast()`). In fact, we can do this reasonably efficiently even though it seems to be horribly inefficient.

In Java, how can we implement it?

```
 ----------------------------
 |             |   lock   -|---> T1,T2,T3 are the threads contending for the lock
 |             -------------
 |             |    CV    -|------> T4,T5,T6 are the threads that did a wait() call
 |             -------------
 | object                  |
 ----------------------------
```

`notify()` moves a thread from the cv-queue to the lock queue (so it's not as inefficient as we might think). `notifyAll()` moves the entire `cv_queue` to the `lock_queue`.

```
  T1          T2
   |           |
 lock          |
   |           |
```

```
wait          |
             lock
              |
wakeup <--- notify
  |           |
  |           |
  v           v
 lock       unlock
```

Broadcasting Semantics:

18.2.1 Signal-and-Continue

when we `notify()/signal()`, then we still

(1) **may** own the lock (in PThreads)

(2) **must** own the lock (in Java)

18.2.2 Signal-and-Wait

The notified thread is "passed the lock" from the notifier. The notifier leaves the monitor (i.e. releases the lock). It must re-acquire the lock before continuing. It contends with all other threads trying to lock.

18.2.3 Signal-and-Urgent-Wait

Similarly, to signal-and-wait to start. But when the notified thread is done with the lock (exits or waits (recursive signal-and-urgent-wait)), it passes it back to the notifier.

All of three semantics are equivalent (in terms of expressiveness).

18.2.4 Signal-and-Exit!

This one `notify()` and then terminates.

18.3 Building Semaphores using Monitors

We can use monitors to build semaphores.

```
public class Semaphore {
    private int count;
    private Semphore(int count) {
        this.count = count;
    }
    // has to wait until a resource is free
    public synchronized void down() {
        while(count == 0) {
            try {
                wait();
            } catch(InterruptedException ie) {}
        }
        --count;
    }

    public synchronized void up() {
        ++count;
        notify();
    }
}
```

18.4 Using Multiple Condition Variables

This is easily done in PThreads. In Java, (basic libraries), we have only one condition variable associated with an object. However, in the new concurrency API, there is a PThreads-like model.

We can build our own condition variables in Java though. Use a binary semaphore as a mutex.

```
public class CV {
    public void cv_wait(Mutex m) {
        try {
            synchronized(this) { // this is the "condition variable"
                m.unlock();
                wait();
            }
        } catch(InterruptedException ie) {}
        finally { m.lock(); }
    }
    public void notify() {
        synchronized(this) {
            notify();
        }
    }
}
```

18.4.1 Multiple Producers/Consumers Using Multiple CVs

We use multiple condition variables so that producers only `notify()` consumers and consumers only `notify()` producers.

```
mutex m;
buf[n];
front; // beginning of queue
rear;  // end of queue
count; // number of entries in buffer
CV notfull, notempty;

produce(data) {
    mutex_lock(&m);
    while(count == n)
        cond_wait(&m,&notfull);
    buf[rear] = data
    rear = (rear + 1) % n;
    count++;
    cond_signal(&notempty);
    mutel_unlock(&m);
}
consume() {
    mutex_lock(&m);
    while(count == 0)
        cond_wait(&m,notempty);
    rc = buf[front];
    front = (front + 1) % n;
```

```
     count--;
     cond_signal(&notfull)
     mutex_unlock();
 }
```

19 Lecture: February 15th, 2008

missed the beginning of class

19.1 Readers and Writers Problem

In a mutithreaded database, only one thread is allowed writing.

Desired properties:

- Prevent Writer/Writer conflicts
- Prevent Reader/Writer conflicts
- Allow for Reader/Reader at the same time

Problem of mutual exclusion as we've seen so far only allows 1 thread at a time. Rather than thinking of all readers as distinct threads, think of them as a group/class. Either let in writer of the class reader.

19.1.1 A Solution Using Semaphores

```
int readers = 0;
BinSem r = 1, rw = 1;
```

We have mutual exclusion within our class of readers.

We want mutual exclusion between class of Reader/Writer or Writer/Writer.

Reader:

```
while(true) {
    r.down();
    readers++;
    if(readers==1)
        rw.down();
    r.up();
    --> read <--
    r.down();
    readers--;
    if(readers == 0)
        rw.up();
    r.up();
}
```

Writer:

```
while(true) {
    rw.down();
    --> write <--
    rw.up();
}
```

Once a reader gets in, others can get in. As long as at least one reader is in the DB, writers will not get in. Readers can starve out writers! This is called the reader's preference (weak).

19.1.2 A writer's preference Solution (Monitors)

```
class RW {
    private int nw; // number of writers in the DB {0,1}
    private int ww; // number of waiting writers
    private int nr; // number of readers in the DB

    void reader() {
        synchronized(this) {
            while(nw != 0 || ww > 0) {
                wait(); // prevents to go in the DB
            }
            nr++;
        }
        --> read <--
        synchronized(this) {
            nr--;
            notifyAll();
        }
    }

    void writer() {
        synchronized(this) {
            ww++;
            // while there is a reader or a writer in the database, we want to wait
            while(nr != 0 || nw > 0) {
                wait();
            }
            ww--;
            nw++;
        }
        --> write <--
        synchronized(this) {
            nw--; // removes myself from the DB
            notifyAll();
        }
    }
}
```

Notice, as long as there are writers in the DB or waiting, our readers will not get in. Writers can starve our readers!

19.1.3 A Fair Solution

i.e. Neither reader nor writer's preference (avoid starvation of either group): Toggle back and forth between the 2 groups. Here's a solution using monitor but 2 condition variables.

Note: The following code uses if's rather than while-loops for waiting. Therefore, **the following code must be fixed!**

```
Lock e;
CV okread, okwrite; //Condition variables
```

```
Reader() {
    e.lock();
    if(nw > 0 || ww > 0) { // THIS IS THE EVIL IF!!!!
        wait(okread);
    } else {
        nr++;
    }
    e.unlock();
    --> read <--
    e.lock();
    nr--;
    if(ww > 0) {
        ww--;
        nw++;
        signal(okwrite);
    }
    e.unlock();
}

Writer() {
    e.lock();
    if(nr > 0 || nw > 0 || wr > 0) {
        ww++;
        wait(okwrite);
    } else {
        nw++;
    }
    e.unlock();
    --> write <--
    e.lock();
    nw--;
    if(wr > 0) {
        nr = wr;
        wr = 0;
        signal_broadcast(okread);
    } else if(ww > 0) {
        ww--;
        nw++;
        signal(okwrite);
    }
    e.unlock();
}
```

Note: If x is only accessed inside a synchronized block. That is, if...

```
synchronized(o) {
    x = ...;
    ... = x;
}
```

20 Lecture: February 18th, 2008

Note: Today's class notes are a courtesy of Adam Blahuta

20.1 Last Time

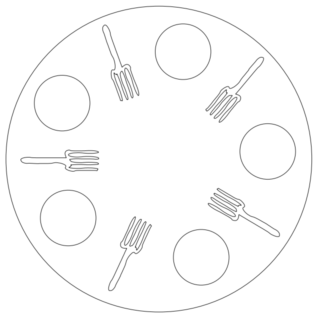
- Reader and Writers
  - Reader can be treated as a group/class because all can read concurrently
  - Writer must have exclusive accessive
    - Mutual Exclusion
      - Group of readers and writers
      - Writers and writers
  - The Reader's Preference Solution: As long as readers are available (reading) no writer can get in (Writers can starve)
  - The Writer's Preference Solution: Writer can lock out readers (Readers can starve)
  - The Fair Solution: Neither group can starve
    - NOTE: In fair solution, last time (reader code), check that n == 0 (i.e. last reader out lets writers in)
  - Building it yourself is possible.
  - Both Java and PThreads do provide R/W locks. Use the `java.util.concurrency` libraries
  - In PThreads, not standard, but usually available as a vendor extension

20.2 Concurrency Problems with Deadlock

Lots of relatively simple problems that exercise concurrency features/problems...

20.3 Dining Philosophers

5 philosophers think and eat. They each have a plate. Between each plate, there is a fork/chopstick. They need an ustensil on either side in order to eat.



The algorithm:

(1) Think for a while.

(2) Get Hungry

(3) Grab both ustensils

   (a) On success, eat and put down chopsticks

   (b) On failure, goto (3) or (1)

20.3.1 Solution 1: Sequential Eating

```
Process() {
    think();
    P(global)
        pick up forks
        eat();
        put down forks
    V(global)
}
```

This avoids fighting over a fork, but doesn't allow concurrency.

20.3.2 Solution 2: One lock per fork

```
Process() {
    think();
    P($f_i$);
    P($f_{i+1}$ > 5);
    eat();
    V($f_{i+1}$ > 5);
    V($f_i$);
}
```

Can end up deadlocked.

20.3.3 Solution 3: Global lock + fine-grained locks

```
Process() {
    P(global);
    P($f_i$);
    P($f_{i+1}$ > 5);
    V(global);
    eat();
    P(global);
    V($f_{i+1}$ > 5);
    V($f_i$);
    V(global);
}
```

Still can end in deadlock.

20.3.4 Solution 4

P1: P(global), P(f1), P(f2), V(global)

P2: P(global), P(f2) $\Leftarrow$ BLOCKS

Now, P1 cannot release the forks and P2 holds global.

We can improve on this solution by not holding the global lock on fork release.

```
Process() {
    think();
    P(global);
    P($f_i$);
    P($f_{i+1}$ > 5);
    eat();
    V(global);
    V($f_{i+1}$);
    V(f_i);
}
```
This avoids the deadlock we had before, but prevents other independent processes from eating.

e.g. P0: P(global), P(fo), P(f1), V(global)

P1: P(global), P(f1) $\Leftarrow$ STUCK UNTIL P0 RELEASES f1

Notice that because P1 is stuck while holding the global semaphore. No other philosopher will be able to eat until P0 is done.

20.3.5 Solution 5: Randomized version

Each process randomly chooses to pick fork f_i, $f_{(i+1)\%5}$ or reverse.

A deadlock here is less likely.

If we are unlucky though, we can still produce deadlock.

20.3.6 Non-uniform solutions

Choose one philosopher to choose right then left, unlike others which grab left then right.

Counting semaphore to ensure no more than 4 philosophers try to eat at once.

20.3.7 Acquire and Release

Try and set both forks. If not succesful at any point, give up and release forks and try again.

Introducing small delay may help.

Still some possibility of lock-step behaviour.

20.4 When does deadlock occur? (Coffman's conditions ['71])

4 properties necessary for deadlock. (resource deadlock)

(1) Serially reusable resources
  a. processes share ressources
  b. Needed under mutual-exclusion
  c. e.g. forks

(2) Incremental Acquisition
  a. processes acquire ressources sequentially
  b. e.g. pick up 1 fork then pick up the other.

(3) No pre-emption
  a. Cannot take someone elses ressource.
  b. e.g. once a philosopher picks up a fork, no one else can take it.

(4) Dependency Cycle
  a. We can build a circular chain of processes, all waiting for each other.
  b. e.g. all philosophers holding left fork and waiting for right fork.

20.5 Deadlock Solution

Ensure one of the 4 properties does not hold. Changing some conditions can mean changing the problem in a fundamental way. Conditions 1,2 (3) are usually fundamental to the definition of the problem. Condition 4 is the main condition to try to break.

21 Lecture: February 20^{th}, 2008

21.1 Last Time

- Readers and Writers
- Concurency problems
  - classic formulations of problems in concurrency
  - dining philosophers (investigated in concurrency as a deadlock problem)
- Deadlock + Necessary properties (resource deadlock)
- The 4 Necessary Properties (Coffman's Conditions)
  (1) Serially reusable resources
  (2) Increment acquisition
  (3) No pre-emption
  (4) Dependency cycle
- The first 3 properties are specific to the problem. The last one, we can prevent a dependency cycle from even occuring and then avoid deadlock.

21.2 Concurrency Problems (continued)

21.2.1 Solution to the Dining Philosophers Problem

One way of achieving this is to impose a (partial) total order on resources. Acquire resources in order.

e.g. With dining philosophers, we order the forks and philosopher acquire in order. we will not have deadlock.

Philosopher must pick up forks in order.

1) $P_0 \rightarrow F_0, F_1$
2) $P_1 \rightarrow F_1, F_2$
3) $P_2 \rightarrow F_2, F_3$
4) $P_3 \rightarrow F_3, F_4$
5) $P_4 \rightarrow F_0, F_4$ (and not $P_4 \rightarrow F_4, F_0$)

$P_0(0) \rightarrow (1)$	$F_0 < F_1$
$P_1(1) \rightarrow (2)$	$F_1 < F_2$
$P_2(2) \rightarrow (3)$	$F_2 < F_3$
$P_3(3) \rightarrow (4)$	$F_3 < F_4$
$P_4(0) \rightarrow (4)$	$F_4 < F_0$

We cannot generate a cycle this way!

So the main solution to avoid deadlock is basically to order resources and acquire in-order.

21.2.2 Other Kinds of Deadlocks

Other kinds of deadlocks that can happen to our program: Thread is spinning forever.
We could have all threads stuck (i.e. not doing anything). Alternatively, threads could be doing work, just not useful work (**livelock**). Or there could be a deadlock, if we keep track of actions that do not progress inside the program.
2 trains approach a crossing, but neither crosses, both wait for each other.

Definition: Divergence: a thread enters actual infinite computation.

Divergence can make some threoretical models more complex. (e.g. in memory models)

21.2.3 The Producer/Consumer Problem

- 1-1
- 1-n
- n-1
- n-m

Issues of Producer/Consumers:

- size of buffer
- ordering of data

21.2.4 The Oriental Gardens Problem

An instanteneously accurate total sum of all currently in the garden, we need to stop entering/exiting (this is 1 process).

1 other process is guarding the entrance. 1 other process is guarding the exit.

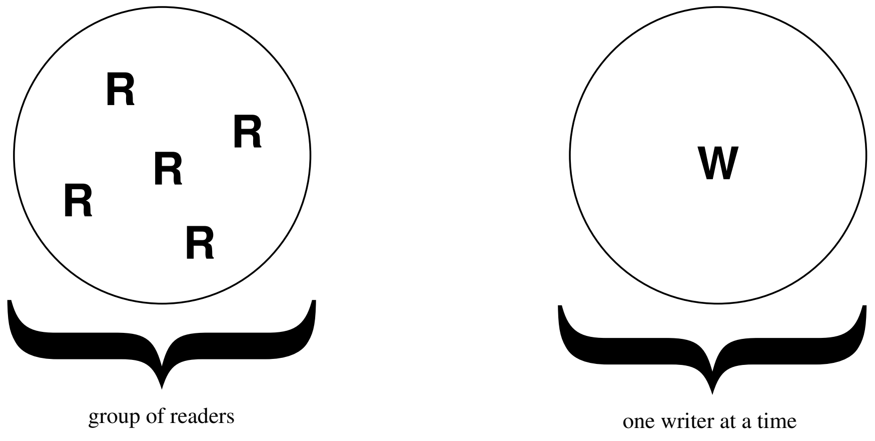
If each door-threads modifiers count under a lock (L_1, L_2)

locks L_1

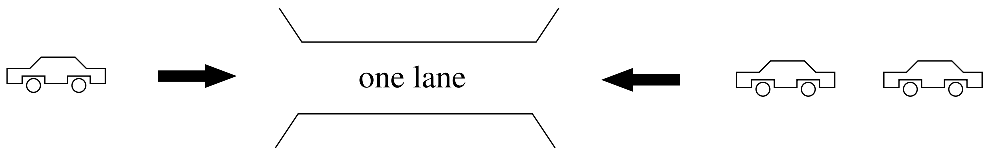
The lock L_2

Now no one can get in/out (can compute the total from individual counts)

21.2.5 The Readers/Writers Problem



21.2.6 The One-Lane Bridge Problem

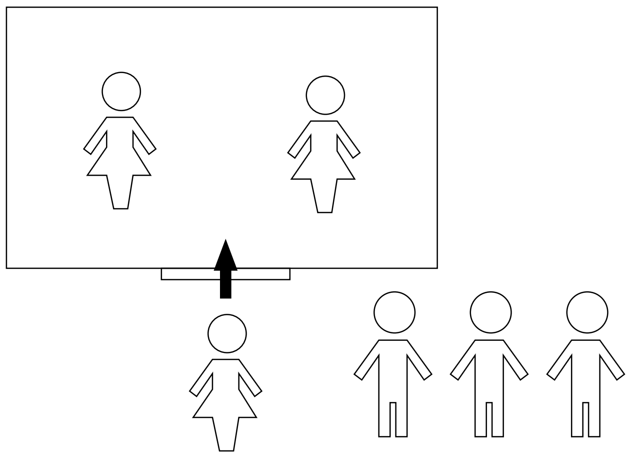


2 groups:

- The one that moves right
- The one that moves left

cannot mix groups in the bridge

21.2.7 The Unisex Bathroom Problem



Men use the bathroom OR women use the bathroom, but not both at once.

21.2.8 The Rollercoaster Problem

This is a condition sync problem.

There are passenger threads ($n = 4$) and a car thread. Some number of seats. No more than passengers, usually less. Passengers compete for seats. Once all seats are full, the rollercoaster car start. Everyone stays on the car while it moves. Once it stops, we let everyone out. Everyone must leave.

21.2.9 The Cigarette Smoker's Problem

The are 3 smokers.

Smoking requires 3 things:

- Paper
- Tobacco
- Matches

Each smoker have an infinite supply of one of the ingredients. An agent drops 2 (random) ingredients in the table. Smokers try and acquire sufficient ingredients to smoke. Agent does not continue until the ingredients have been used. With a lock for each ingredient, easy to get deadlock. Originally proposed as something not solvable by semaphores. In fact, this is not true. You can do this with semaphores (cond. sync).

22 Lecture: February 22^{th}, 2008

Note: Today's class notes are a courtesy of Adam Blahuta

22.1 Last Time

22.2 Concurrency Problems (continued)

22.2.1 The Sleeping Barber Problem

Problem Description:

- One barber chair.
- Multiple waiting chairs.
- Only one person moves at a time.
- Customers enter through in/ exit via out door.
- When no-one is in the shop, barber sleeps in the barber chair.
- When a customer arrives:
  - If no-one else is inside, they wake up the barber and wait to sit in the chair and get their hair cut.
  - If the barber is busy, the customer waits in a waiting chair.
- Once a haircut is finished:
  - Barber opens the exit and waits for customer to leave (close it behind them)
  - Checks for any waiting customers. If there is none, they go to sleep in the barber chair.
  - At least one customer waiting
    - Wake up customer
    - Wait for them to sit down in the barber chair.

Fairly complex model. Requires lots of `wait()/notify()` operations.

At different points, threads must sync up.

- Barriers
- a.k.a. Rendez-vous

22.3 Scheduling and Priorities

Priority-pre-emptive model.

States that threads move through:

- created
- ready
- active
- terminated

OS schedules:

- processes, which can contain threads

    - green threads
    - run-to-completion

- Threads
    - Lightweight process (LWP)
    - Using LWPs, we can produce several thread scheduling models.
        * 1-to-1: Every thread is mapped onto a unique LWP.
        * If there are multiple CPUs, we will get real parallel execution.
        * many-to-1: Several threads mapped to one LWP.
            · Can produce process scheduling model
            · Simpler and will have less scheduling load.
            · Probably will not get much parallelism.
        * many-to-many: OS has maximal flexibility to map threads to LWPs.
            · Can be optimized
            · Very difficult to have scheduling guarantees.
        * Bound threads
            · Associated with a specific CPU.

22.4 Scheduling Guarantees

Suppose you write code:

```
volatile bool go = false;

Thread0 {
    while(!go){
        kill_time();
    }
    do_something_useful();
}

Thread1 {
    go = true;
    long_calc();
}
```

In order to make sure Thread0 can make real progress, we need fairness in our scheduling. Can we guarantee that Thread1 executes? Not necessarily.

Definition: Unconditional fairness: A scheduling policy is unconditionnaly fair if every unconditional instruction is eventually executed.

Conditional Atomic Action: `<await(b); S;>` must be atomic. We cannot change the value of b after await has ended and S will be executed.

Weak Fairness: A scheduling policy is weakly fair if:

- It is Unconditionnally Fair.
- Every conditional atomic action that is ready to execute is eventually executed, provided the condition becomes true and stays true thereafter.

If our condition flag is toggled at different points, cannot guarantee execution of threads as expected.

Strong Fairness: A scheduling policy is strongly fair if:

- It is Unconditionally Fair.
- Every conditional atomic action that is eligible is eventually executed, provided the condition is true infinitely often.

In practice, you usually get weak fairness.

23 Midterm: March 3rd, 2008

24 Lecture: March 5th, 2008

24.1 Last Time

- Scheduling
  - 1-to-1
  - n-to-1
  - m-to-n
  - Java gives no specific guarantees
  - PThreads more defined (optional)
- Fairness
  - Unconditionally Fair
  - Weakly Fair
  - Strongly Fair
  - Need strongly fair to prove
  - Most algorithms weak

24.2 Priorities

Different level of priorities:

- High Priority (very important threads)
- The highest priority is running real-time. (very sparingly)
- Lowest Priority (e.g. screen-savers, idle activity)

As usual, we have few guarantees. Java defines 10 priority levels (1 = min, 10 = max). These levels of priority are mapped to the OS priorities. PThreads, on the other hand, may have more than 32 levels. Some systems even defines up to 128 levels!

The PThreads range may or may not be fully-exploited by the Java range.
Remember: PThreads Scheduling Models

- FIFO
- RR
- other

In PThreads, you get priorities that are separated between the scheduling models.

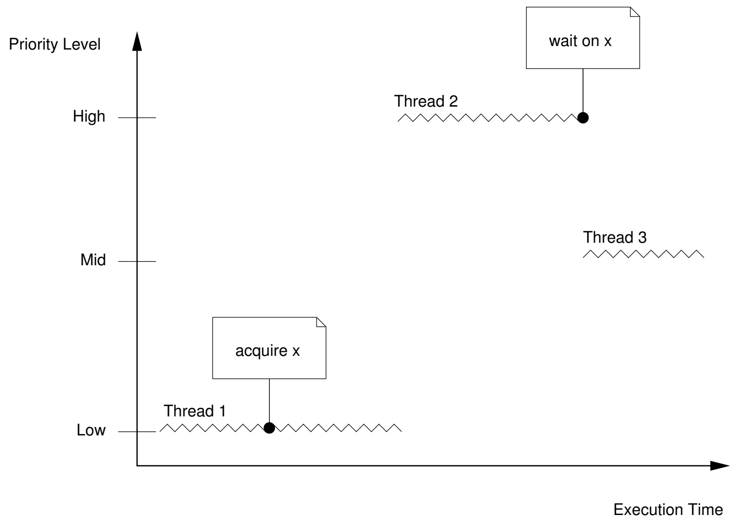
- 32 levels for RR
- 32 levels for FIFO

Priority in RR (respectively to FIFO) is not necessarily connected to primitives in FIFO (respectively to RR). If you min threads in different scheduling models, your priorities will be undefined.

24.2.1 Priority Inversion

Problem where high priority threads are not respected.

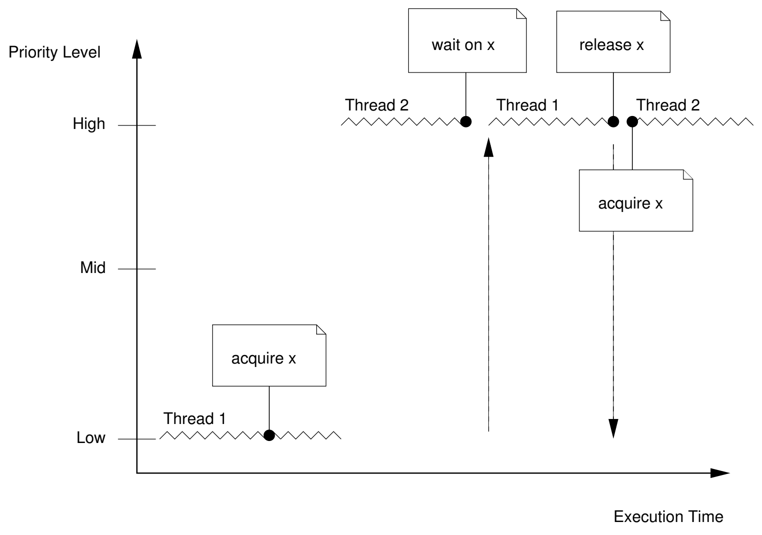
Usually occurs in uniprocessor environment (or few CPU's, this threads) (can happen on multiprocessor too)



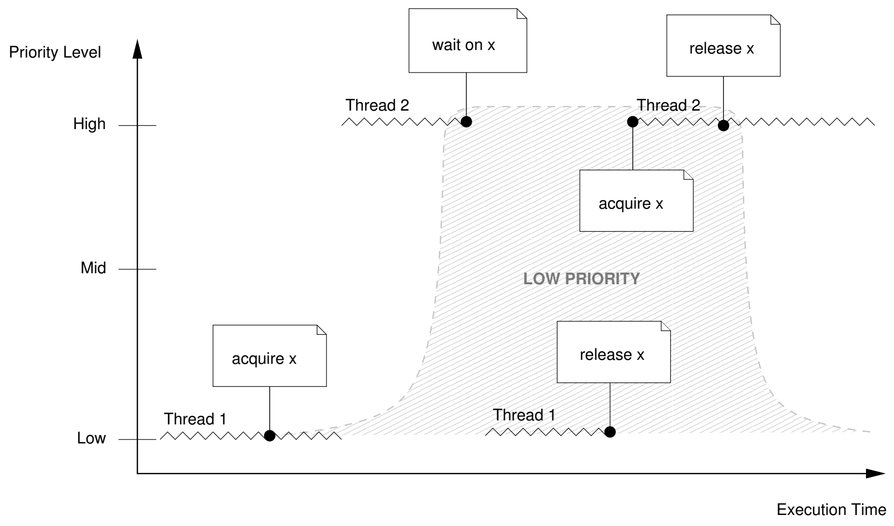
A thread with low priority (Thread 1) acquires a lock. Laton on, a thread with high priority (Thread 2) waits on the same lock. A third thread with mid priority (Thread 3) comes by and prevent the low priority thread to release the lock. The thread with high priority is busy waiting forever.

2 techniques to fix this problem:

(1) **Priority Inheritance**
The owner of a lock is raised to the priority of the highest priority thread waiting on the same lock.



(2) **Priority Ceilings**
We associate a priority with a lock.



In the figure above, the lock is associated with a low priority. When Thread 2 (with high priority) tries to acquire the lock, it becomes a low priority thread, until it has successfully acquired and released the lock.

e.g. Mars Pathfinder

Mars Pathfinder had this problem of priority inversion.

3 threads:

- bus-manager (high)
- meteorological data (low)
- communication (sending data) (med)

Priority inheritance problem:

- low - locked the bus
- med - wants to send (VERY slow)
- high - move info across bus

There also was a monitor thread to make sure everything is ok. It detects bus is stalled and causes resets.

24.3 Termination and Suspension

often wants to 'pause' a thread and 'resume' it later

may want to stop a thread.

Java had these primitives but they've been deprecated (will disappar).

`stop/resume/pause/destroy` no longer works. In fact, destroy was never actually fully implemented!

Why did they remove them?

If you asynchronously stop/pause threads, they might be in arbitrary state (e.g. aquiring a lock, or in a lock in the critical section). Easy to corrupt state this way! Even if you know about your threads/locks, there are still VM-internal, runtime-libraries, locks. If you don't know for sure, you may get corrupt data/deadlock. In Java, if you want to pause or stop threads, req. is to use **pulling**. This effectively tell the threads to stop/pause themselves using a boolean flag. This is where we're actually using `Thread.interrupt()` call.

You can also create your own interrupts.

```
Thread A              Thread B
--------------        --------------------------
                      try {
B.interrupt();            sleep();
                      } catch (InterruptExeption ie) {

                      }
```

Inside some I/O activities, you may not be able to wakeup/interrupt your thread.

25 Lecture: March 7th, 2008

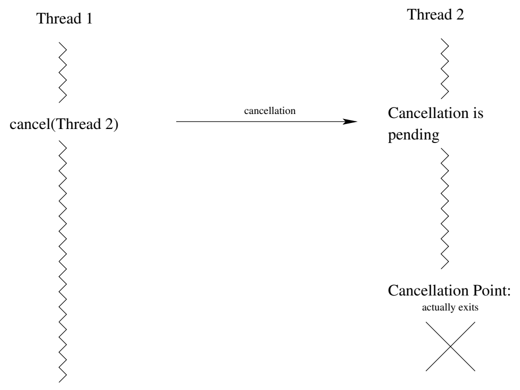
25.1 Last Time

- Priorities
  - Priority-Inversion Problem
    - Solution 1: Priority Inheritance (PThreads may give these)
    - Solution 2: Priority Ceilings (PThreads may give these)
- Safety issues with Termination (stop, pause/resume)
- `Thread.interrupt()`
  - help-pulling solution (ie: pop out of blocked state)

25.2 Termination and Suspension (continued)

PThreads gives you a termination system (cancellation).

Definition: **Cancellation points** are places where it is safe to pause or stop a thread.



You can also register handlers that will be executed when you are cancelled.

Lock

- Register a handler that knows we own the lock
- Cancelled: it knows to unlock
- which: deregister our handler
- cancellation handlers are stacked very muck like `atexit` routines.

Default in PThreads: `PTHREAD_CANCEL_DEFERRED`

- Asynchronous version is a `thread.stop()`
- Still get cancellation handlers

We can add our own cancellation points using the `test_cancel()` function.

In PThreads, most blocking calls (including `sleep(), sem_wait(), open(), read(), fcntl()`) are also cancellation points. If a signal is received during such calls, `errno` is set to `EINTR`. An exception is mutexes. To keep locking cheap, cancellation is not checked.

We can globally (i.e. for all threads) turn these cancellation points on/off using

- `PTHREAD_CANCEL_ENABLE` (default)
- `PTHREAD_CANCEL_DISABLE`

Note: A pending cancellation remains pending if cancellation is disabled.

25.3 Thread-Specific Data/Thread Local Storage

Registers, Stack variables / Thread local.

Globals are shared.

Sometimes, it is convenient to have a variable with global scope but containing thread local values.

In C/C++, there is a global variable called `errno`. After each system call, `errno` is filled in. `EINTR` is one of these potential filled in value. `EBADVAL` also.

Problem is `errno` is a global variable.

```
Thread 1                                    Thread 2
   |                                           |
  x = read()  // sets errno = EBADF          y = read()  // sets errno = EOK
   |                                           |
 if(errno == EBADF) ...
```

Here, Thread 1 may not detect the error message returned by its system call because of Thread 1 and 2 racing over `errno`. To get rid of the race condition, we need to make `errno` capable of storing multiple values (e.g. 1 per thread). Both PThreads and Java give us this capability. Java uses **Thread Local Storage** (TLS). PThreads uses **Thread-Specific Data** (TSD).

Note: java.lang.ThreadLocal
This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID). (Java Sun Documentation)

```
class foo {
    static ThreadLocal errno;
Thread 1:
errno.set(...);
Thread 2:
errno.set(...);
}
```

Implementing this is straightfoward using a hash-table.

Hash Table	
Thread Object	**Thread-specific values**
key1 (for thread1)	value1 (for thread1)
key2 (for thread2)	value2 (for thread2)
key3 (for thread3)	value3 (for thread3)
key4 (for thread4)	value4 (for thread4)
...	...

Notice that accessing a thread-specific variable requires more steps than a regular variable. This can be relatively slow. The cost has has been improved a lot so it is no longer quite so expensive but it certainly is not cheap.

In PThreads, it works a lot the differently key/value pairs.

Keys

	k1	k2	k3	k4
0	o	o	o	o
1	o			
2	o			
3	o			
4	o			

Globally accessible key that all threads may use, but set/get their own value.

`PTHREAD_KEY_CREATE(&key,fn)` where `fn` is a function pointer to the key destructor method. When a key is destroyed, we need to potentially deallocate any memory pointed to by our key values. That's what the destructor is for. Notice, your destructor is arbitrary code. So it may create keys! Destructors may need to be called repeatedly.

But if we call this function over and over and only look for non-null values, we get an unbounded number of iterations. That is why there is a `PTHREAD_DESTRUCTOR_ITERATIONS` as a max.

Motivated by `errno`. In fact, `errno` is treated specially. You don't need TSD syntax to access it. However, it will contain thread-specific values.

25.4 Miscellanous Issues

Consider the following.

```
class Foo {
   private Data d = null;
   public Data getData() {
       if(d==null) {
           d = new Data();
       }
       return d;
   }
}
```

In a multi-threaded context, we may allocate our Data more than once. Can we fix this?

26 Lecture: March 10^{th}, 2008

Class is cancelled.

27 Lecture: March 12^{th}, 2008

27.1 Last Time

- Thread-Specific Data
  - TLS (Java)
  - TSD (PThreads)
- "errno"

27.2 Miscellanous Issues (continued)

```
class Foo {
    private Data d;
    synchronized Data getData() {
        if(d == null)
            d  = newData()
        return d;
    }
}
```

With multiple threads, we may get multiple allocations of `d`.

- Only one will succeed, the rest are garbage-collected.
- Fix for multi-threading, by adding `synchronized`. No duplicate allocations.

27.2.1 Double-Check Locking (DCL)

We can try and avoid synchronization by using code like:

```
Data getData() {
    if(d == null) {
        synchronized(this) {
            if(d == null) {
                d = new Data();
            }
        }
    }
}
```

A compiler might optimized this code.

In Java, use `new Data`

(1) Allocate space for Data object

(2) Call the constructor

We might actually have code that looks like:

```
  d = new Data      (allocation)
  d<init>()         (constructor)
```

If this happens in our synch blocks, then another thread may see the allocated but not initialized object. DCL doesn't usually work. Avoid this, very much to fix. Note that will work in Java, for primitive data types that are declared `volatile`.

27.3 Memory Consistency

Note: "A correctly synchronized program is one with no data races; correctly synchronized programs exhibit sequential consistency, meaning that all actions within the program appear to happen in a fixed, global order."[6]

```
 x = 1
 y = 2
 b = y
 a = x
-------------
(a,b) = (1,2) is the only possibility
```

With concurrent execution, we get more possibilities:

```
Processor 1                         Processor 2
----------------                    --------------------
x = 1                               y = 2
b = y                               a = x
```

[6]Source: Java Concurrency in Practice, page 341

```
P1: x = 1           P1: x = 1            P2: y = 2
P2: y = 2               b = y                a = x
P1: b = y           P2: y = 2            P1: x = 1
P2: a = x               a = x                b = y
----------------    -----------------    --------------------
(1,2)                  (1,0)                 (0,2)
```

However, not all combos for (a,b) are valid!

```
(0,0)
   --          --
  |  \       /   |
  v   \     /    v
x = 1  \   /   y = 2
b = y   \ /    a = x
  |      \       |
  |_____/ \______|
```

(0,0) would involve a cycle of dependency.

Most existing options are intended for a single thread/process context.

A useful hardware optimization is write-buffering (WB):

- some overhead for each write
- batch writes together
  - into a write buffer
  - at some point flush into memory

With concurrent execution, we get more possibilities:

WB-P1	WB-P2	P1	P2	Main Memory
~~x = 1~~	y = 2	x = 1	y = 2	~~x = 0~~ x = 1
~~b = 0~~	a = 0	b = y	a = x	~~y = 0~~ y = 2
				a = 0
				b = 0

(1,2), (0,2), (1,0), + (0,0)

The simplest model for memory

27.3.1 Strict Consistency

Every operation of every processors/threads is executed in order, and operations are interleaved.

WB are not allowed here.

27.3.2 Sequential Consistency

A MP is sequentially consistent (SC) if the result of each execution is the same as if the operation of each processor were executed in sequential order and the operations of all processors appear interleaved.

27.3.3 Sequential vs Strict Consistency

SC requires the appearance of strict consistency.

```
r1 = 3                   r2 = 4
            becomes
r2 = 4                   r1 = 3
```

i.e. you can rearrange instructions if they don't affect the final result.

Most programmers will assume S.C.

Very few machines provide S.C.

- Intel (very close)
- PowerPC (weaken model)
- DEC Alpha (very weak)

It's been estimated 20-80% of current execution performance is due to the use of weaken models than S.C.

15-35% (is that the actual perf. we get?)

27.4 Relaxed Models

27.4.1 Coherence

Much weaker than SC

SC: are canonical time-line for the actions of all processors

with coherence, we get SC on a variable by variable basis.

Time-line for each variable in isolation

28 Lecture: March 14th, 2008

28.1 Last Time

- Double-Checked Locking (DCL)
  - primetime data only
  - composite → hadto get right
- Memory Consitency/Memory Coherence/ Memory Models
  - strict consistency
  - sequential consistency (appears strict)
  - other models (optimization (Write Buffers))
- Need a more released model

28.1.1 Coherence (continued)

SC on a variable-by-variable basis

Writes to a variable as all seem on the same order by all parameters

```
P1          P2
x = 0       .
x = 1       .
y = 2       .
```

Under sequential consistency, the assignment $y = 2, x = 0, x = 0$ is smart but is it coherent?

28.2 Processor Consistency with PRAM (Pipelined RAM)

Reach processor's writes as seen on the same order by every processor.

```
P1                P2
------            -------
x = 1             y = 3
y = 2             x = 100
z = 3             z = 10
x = 4
```

Here, another processor P3 could see:

```
P3                  P4
----------          ----------
x = 1 (P1)          y = 3 (P2)
y = 3 (P2)          x = 1 (P1)
```

They are difficult in general.

- Location Consistency
- Causal Consistency
- Release Consistency

If you see a value for `x`, then you must see all the writes that generated that value.

In C/C++, you get whatever the machine provides. (No guarantee of any sort). You can enforce the visibility (commitment) of values. You probably need assembly/machine language primitives to do this.

28.3 Java Memory Model (JMM)

In Java, because you have a virtual machine (and since this virtual machine is a multiprocessor model) we also get a memory model. The old memory model (Java 1.4 and below) was very broken! (The problem was discovered by W. Pugh). It had a very complex model with a Master Memory, where every launched thread had its own memory. But the problem is that there is a 3 stages process to fetch the variable from master memory into the thread local memory. This fetch had many interleavings due to these 3 stages, and that would take away any sort of guarantees over when the local value is actually published.

In other words, the old memory model also involved a big complicated set of rules and Java designers didn't fully understand them. There are inherent flaws with this old model. Consider the following execution sequence.

(1) `int i = p.x;`

(2) `int j = q.x;` $\leftarrow$ We don't know for sure that $p \neq q$

(3) `int k = p.x;`

We can convert line 3 into the optimized line `int k = i`.

The old model promises coherence, that is something actually stronger than coherence. At that time, coherence was viewed as a minimum guarantee for programmers.

Let us now suppose that p and q are the same object.

```
Thread 1                            Thread 2
----------                          --------------
i = p.x                             p/q.x = 1
j = q.x
k = i  (optimization of k = p.x)
```

Given this optimization, we would get incoherent result as follows:

At time 1: `p/q.x = 0`

At time 2: `p/q.x = 1`

At time 3: `p/q.x = 0`

This optimization of `k = 1` was not allowed because it used to break coherence. But since we don't want to give up such optimization, we are given no choice but to change the memory model. We needed a model reasonably easy to understand and yet offers some properties.

28.4 The New Java Memory Model: Happens-Before Consistency

Happens-Before is a partial ordering that we enforce over runtime actions. We will use Happens-before graphs to verify consistency. Edges of these graphs will indicate precedence.

28.4.1 Happens-Before Graph (HB graph)

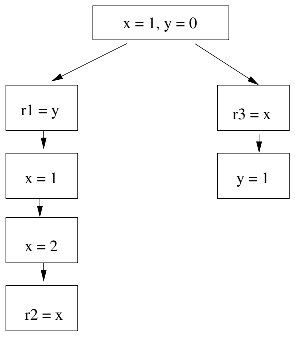
We use a happens-before graph to represent happens-before relationship. Contiguous actions such as locking and unlocking, starting and joining threads, etc. represents the nodes of the HB graph. Edges expresses the happens-before relationship.
Example of HB relationship:

- A `Thread.start()` happens-before every action executed by this thread.
- Every action in a thread happens-before some other thread `Thread.join()` the former.
- `unlock(m)` happens-before `lock(m)`.
- In a `synchronized` block, there is a total order over the actions.

e.g.

```
Thread 1            Thread 2
----------          -----------
r1 = y              r3 = x
x = 1               y = 1
x = 2
r2 = x
```

HB Graph indicating the guaranteed partial order:



What are local values for each read? A read of variable `v` is allowed to see a write to `v` if in the partial ordering.

(1) Read is not ordered before write

(2) There is no interleaving write such that `write` $\leq$ `write'` $\leq$ `read`.

Based on this graph, we can tell which writes a given read can actually see.

29 Lecture: March 17th, 2008

29.1 Last Time

- Memory Consistency
  - Strict Consistency
  - Sequential Consistency
  - Coherence
  - Processor Consistency (PRAM)
- Java Memory Model (JMM)[7]
  - The old model was very broken. Some optimizations violated the consistency property of the old JMM
  - The new model was implemented in JLS3 (Java Language Specification, 3rd Edition), i.e. in Java 1.5
  - Happens-Before Consistency and Graphs
    - Nodes for contiguous action
    - Edges between them represents Happens-Before relationship

29.2 Java Memory Model (continued)

29.2.1 Happens-Before Graph (continued)

Now, we try to figure out which writes a given read can actually see.

`read` is a read of `v`
`write` is a write of `v`

`read` can see `write` if

[7]See Java Concurrency in Practice, Chapter 16 for more details

(1) `read` $\not\rightarrow$ `write`

(2) `write` $\rightarrow$ `write'` $\rightarrow$ `read` (interleaving writes)

Using the Happens-Before Graph, we can add **visibility lines** (arrows with dotted lines) to indicate which writes each read is able to see. Pick, for each read, which writes it actually sees (an independent choice for each read). We can use this to justify whether certain values; code-changes is possible.

Notice, the HB Graph is ordered from runtime actions. But at runtime, you may have different synchronization orders. You must consider all of them.

e.g.

```
Thread 1    Thread 2
--------    --------
x = 1       lock m
lock m      r1 = y
y = 1       unlock m
unlock m    r2 = x
```

One of them will acquire the lock first. Suppose Thread 1 gets the lock first. In this case, there are no data races inside this code because all accesses to shared variables are ordered by Happens-Before relationship.

```
T1           T2
---          ---
x = 1       lock m
lock m      r1 = y
y = 1       unlock m
unlock m    r2 = x
```

Suppose now that Thread 2 gets the lock first. In this case, we do have a data race inside this program.

Construct a template Happens-Before Graph. Think about syncrhonization orders. For each one, you set a destruct-HB-graph. This is supposed to represent all possible behaviour.

Happens-Before Consistency is not too hard to use but it contains some things that are not intended.

```
x = y = 0;

Thread 1        Thread 2
-----------     -----------
(1) r1 = x      (3) r2 = y
(2) y = r1      (4) x = r2

x = y = 42;
```

We have here a **causal loop** (circular argument) because (2) comes from (1) which may come from (4) which in turn comes from (3) which come from (2)!

With a causal loop, we have a circular argument for where a variable's value came from. Thus, any value we inject in here can be validated.

Thread 2 is an out-of-thin-air result and out-of-thin-air result are very unintuitive.

The solution: To disallow all causal cycles.

But some causal cycles are good.

```
a = 1, b = 0;

Thread 1      Thread 2
--------      --------
i = a;        k = b;
j = a;        a = k;
if(i == j)
    b = 2;
```

Is `i == j == k == 2` possible?
That would involve a cycle. Should we actually allow such kind of behavior? In terms of compiler optimization,

```
i = a;
j = a; // we may optimize this by changing 'a' to 'i'
if(i == j) // but then, this will always be executed
    b = 2; // and since this is a write and that b is not read earlier,
           // we can actually put this on top of the code
```

So the compiler may create such a cycle! In other words, this `i == j == k == 2` IS possible!

Optimizations can in fact create causal cycles! Some cycles are good.

As well as Happens-Before Consistency, there is also a system for justifying causality.

Causality:

- Well-formed executions
- Initially an execution that respects Happens-Before Consistency
  - All reads must see writes that happened before them.
  - Then, start committing actions (R/W)
  - Build an entire trace of commited actions
  - Each commit means restart

Use this causality to esure our 42-example cannot occur and the good cycle can occur. Notice that disallowing the 42-example restrict the out-of-thin-air values are not allowed.

30 Lecture: March 19th, 2008

30.1 Last Time

- Java Memory Model
  - Relatively New
  - HB consistency
    - HB-edges (order)
    - Synchronization Order
  - "can see" edges
- Still allows "causal cycles" with lots of unintuitive properties.
- System for justification that tries to establish real causality. (complex)

30.2 Building Correctly Synchronized Programs

All of this is to handle inefficiently synchronized program (i.e. a program with one or more data races).

Write rightly synchronized programs (i.e, no race-conditions, all shared variable accesses are ordered by HB-consistency).

With correctly synchronized programs, you get every strong guarantees such as **sequential consistency**. This is very important to avoid race conditions.

JMM (Java Memory Model) also addressed some side-issues.

Finalization:

```
finalize() {
    ------------
    ------------
}
```

`finalize()` method is called before the memory of this object can be reclaimed.

In fact, actually involving `finalize()` is not guaranteed.

An example of a problem with `finalize()`.

```
class Foo {
    private File f;
    Foo() {
        f = new File();
    }
    public void write(int data) {
        f.write(data);
    }

    // Use this function as a destructor
    void finalize() {
        f.close();
    }
}

f = new Foo();
f.write(42)
f = null; //at ths point, there is no longer any referene to f (f can be garbage-collected)
```

Suppose we optimize it by doing some **inlining** code.

```
f = allocate a new Foo;
//inline the constructor
f.f = new File();
temp = f.f // in order to optimize the reading of f.f (let's say f.f is accessed
           // quite often)
------------------ // at this point, f is no longer referenced and can be
                   // garbage-collected and potentially finalized
temp.write(42);
```

Finalization is in fact usually done with another thread. Our write to f (i.e. 42) may be concurrent with finalization closing the file.

Solution, consider finalization (need synchronization).

Alternatively, avoid finalization.

Some other interesting impacts. Suppose you wirte the following program:

```
x = y = 0

Thread 1              Thread 2
---------             ---------
do {                  do {
    r1 = x;               r2 = y;
} while(r1==0);       } while(r2 == 0);
y = 42;               x = 42;
```

Does this program have a race-condition, or is it correctly synchronized?

Both threads here **diverge** (i.e. they never terminate).

This program actually is correctly synchronized (because it contains no race condition)!!

Some instructions can be moved around by the compiler.

If the statement `y = 42` is moved above the loop, then we get a different behaviour. But this is not allowed since it would violate sequential consistency. Assumptions of forward progress is no longer always allowed.

There are a lot of other subtle cases. Mostly if you use fairly standart synchronization and avoid race-condition, you will not encounter these issues.

30.3 New API inside Java (java.concurrency.utilities)

`java.util.concurrent` is the main package.

This package provides semaphores, condition variables, etc. as well.

It also provide a new model of execution.

Executions:

- Execute chunk of code
- Thread.pooling built-in
- Futures (way of generating concurrent new threads, with a promise of a result)
- Very advanced data structures
  - Queues (blocking and non-blocking)
- Timing
  - Extend timouts to many situations

  - High resolution timers (nano-seconds)

    Note: When requesting a thread to sleep, we usually get it to sleep for 5-10 ms
  - Locks, Barriers
  - Atomic classes and objects

31 Lecture: March 20th, 2008

31.1 Last Time

- Java Memory Model
  - HB Consistency
  - Race-free program has S.C. semantics. 'race-free' is a runtime property (HB Consistency ordering exists between all reads and writes
  - If you execute in a S.C. context could races appear? If never (under any synch order and any interleavings) then it's race-free (and in Java S.C.)

31.2 Java Concurrency API

This API contains lots of useful features.

31.2.1 Basic Locks

```
synchronized() {
}

try {
    m.lock();
} finally {
    m.unlock();
}
```

Locks cannot be interpreted.

If you want interrupts, lock `Interruptibly()`.

interrupted exception

also provides a `TryLock`: attempt to lock (non-blocking)

run also accept **timeouts**.

```
TimeUnit SECONDS
         MILLISECONDS
         MICRO
         NANO
```

31.2.2 Other kinds of locks

- Re-entrant Locks (Normal lock)
- Read-Write locks
- Re-entrant Read-Write locks
  - `readLock()`
  - `writeLock()`

31.2.3 Atomic class

`java.util.concurrent.atomic`

- compare-and-swap
- add-and-get
- test-and-set
- ...

31.2.4 Volatile Arrays

This lets us update a `volatile` field using reflection.

Note that the following code does not provide volatile arrays:

```
volatile int[] a; // COMMON ERROR: THIS DOES NOT DECLARE A VOLATILE ARRAY!!
```

Solution is to use `Atomic.IntegerArray`.

31.2.5 Stamped/Marked References

```
reference + boolean          Marked Reference
(    atomically    )

reference + integer          Stamped Reference
(    atomically    )
```

31.2.6 Counting Semaphore

- FIFO
- non-blocking

31.3 Condition Variable

We can create multiple condition variables with `c = lock.newCondition()`.

But spurious wakeups are still possible.

`waitUninterruptibly()`

31.3.1 Barriers

CountDownLatch

- a **one-shot barrier**
- threads show up
  - countDown()
  - await();

31.3.2 Cyclic Barrier

Barrier that can be used more than once. They are also called **exchanger**.

- Rendez-vous
- Trading values
- `callVExchange(V x)`

31.3.3 Future

Note: A **Future** is a promise.

`new Future(fib(100))`: this piece of code tries to compute this concurrently. It could actually compute it sequentially though.

When calling `f.get();`, we get the result of this computation.

If the task is not finished, then the `get()` call blocks until it is finished.

Executers: This is a standardized invocation for tasks.

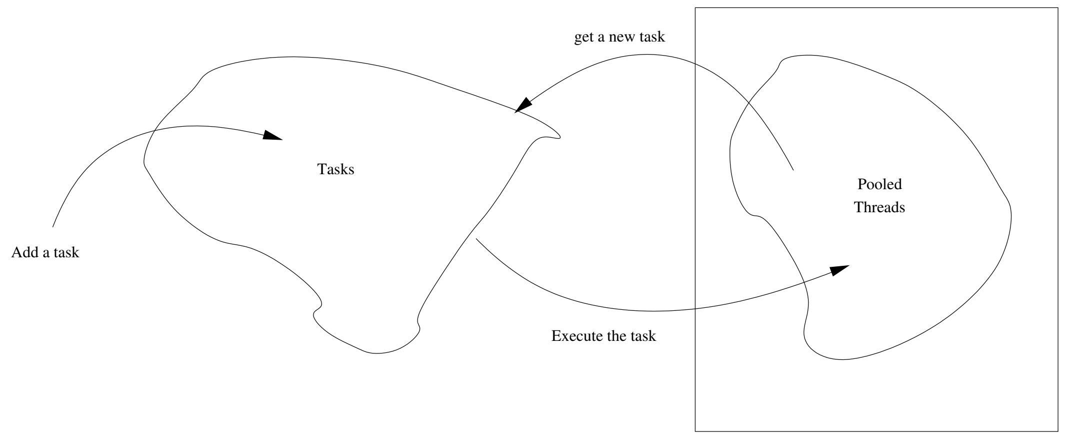
Callable are just like a Runnable:

- They can return a value
- They can throw exeption

Pass these objects to an executer and it will execute them.

We could have an executer that immediatlely executes the task. Normal control flow.

Thread pooling.



CachedThreadPool

- tries to use a thread for each task
- reuses threads when possible

FixedThreadPool

- Fixed, constant number of threads
- Tasks are executed if/when a thread is available

State Thread Executer

- Scheduled Executer
  - Execute a task in the future

31.4 Concurrent Data Structures

```
       -----------
 P --> |         | --> C
       -----------
```

e.g. A Queue

- Bounded Buffer
- Synchronous queue
  - send/receive block

31.4.1 Copy on write structures

- 2 threads that use very similar data structures. Usually the same, logically distinct.
- e.g. using shared libraries between processes

If one thread writes to the shared data structure, the data structure is copied. That is, our thread will have its own version. Can write to without affecting others.

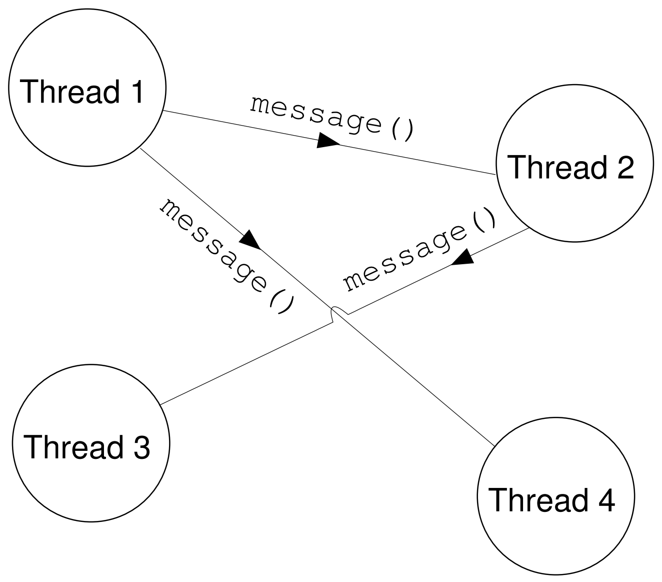
Such data structures are available in Java using the `java.util.concurrent`. Examples of these data structures are:

- ArrayLists
- Sets

But there's a lot more.

31.5 Message Passing

So far, we've been using mostly shared memory. We will now address a new scheme of communication called **message passing** (**channels** to communicate).



There are 2 main flavours of message passing:

- Asynchronous Message Passing
  - sending the message does not block
  - receiving the message blocks until a message shows up
- Synchronous Message Passing
  - receive is the same (blocks)
  - send also blocks until the receiving is done

32 Lecture: March 26th, 2008

32.1 Last Time

- Message Passing (vs Shared Mermory)
- 2 Flavours:
  - Synchronous (both send and receive block)
  - Asynchronous (receive blocks)

32.2 Message Passing (continued)

There is an expressiveness difference (synchronous gives more 'information').

```
     P                  Q
------------    ----------------
 send(Q,23)      x = receive(P)
```

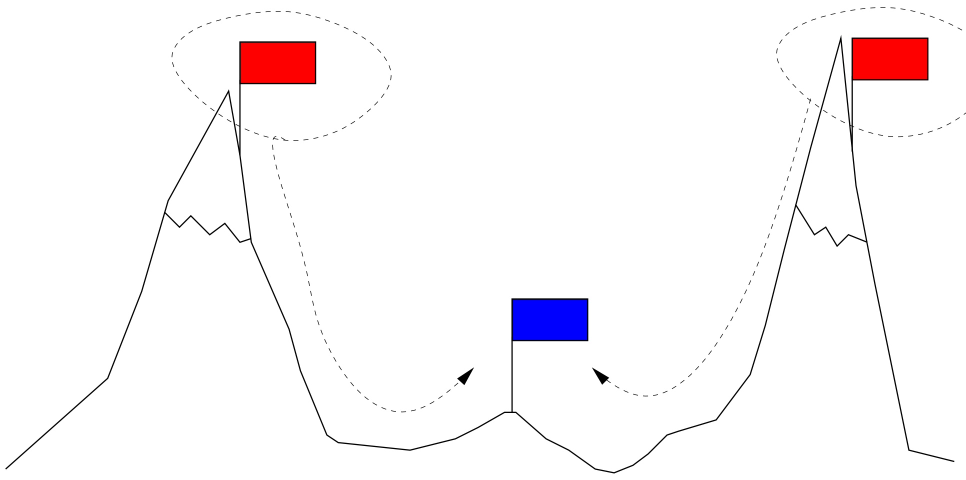
In Asynchronous Message Passing, once P sends, it only knows that `x` may be 0 or may be 23.

In Synchronous Message Passing, P gets more info from sending (i.e. that $x == 23$).

In synchronous Message Passing environments, we can actually achieve common knowledge.

Definition: Common knowledge is a special kind of knowledge for a group of agents. There is common knowledge of p in a group of agents G when all the agents in G know p, they all know that they know p, they all know that they all know that they know p, and so on *ad infinitum*.[8]

32.2.1 The 2-Army Problem



If just one Red army attacks, blue wins. If both red armies attacks, red wins.

Message are asynchronous and unreliable. One army sends message. No guarantee of delivery.

2^{nd} must send on acknoledgement. No guarantee of delivery.

1^{st} must send an acknoledgement of the acknoledgement.

In an asynchronous system, coordination is not (finitely) possible (and this is provable). This is true even if delivery is guaranteed (just not bounded).

32.3 Process Algebra

The goal is to isolate the 'concurrency' from the actual computation. Let us understand the concurrent behaviour better.

Need a formalism. Can we figure out whether concurrent programs are equivalent?

Process algebra are formalisms that try and do this.

- CSP (Communicating Sequential Processes) [Hoare '78] (this one is the closest to a real language)
- CCS
- Meije
- ACP
- π-caculus
- Ambient calculus

[8]Source: Wikipedia

32.3.1 CSP

Definition: CSP is a formalism which became a language. There are lost of variations in the formalism/syntax that occured over time.

Synchronous Message-Passing

Interleaving Semantics (mostly used to study process equivalence)

Notation:

- **Message-Passing**
  - Sending:

    ```
    P::   Q!e      // e is some expression
                   // P:: is the definition of process P
                   // P evaluates e, then sends the result to Q
                   // this is blocking
    ```

  - Receiving:

    ```
    Q::   P?x      // Q:: is the definition of process Q
                   // P?x receive from process P some data and
                   //     store it in local variable x
    ```

    In order for the send/receive to match up, matching can be very simple or type-based.

- **Sequential Operations**

  ```
  P:: Q!17; R?y  // the ';' symbol marks sequential composition.
  ```

- **Parallel Composition**

  ```
  P || Q         // P in parallel with Q
  ```

33 Lecture: March 28th, 2008

33.1 Last Time

- Process Algebra
  - Synchronized communication
  - Interleaving
  - Focus on the 'concurrency'
  - Message Passing (patterns of message communication)
  - CSP Formalism
    - Sending procedure: `S!17`
    - Receiving procedure: `R?x`
    - Sequential Composition: `;`
    - Parallel Composition: `R||S`

33.2 Process Algebra (continued)

33.2.1 CSP (continued)

- **Guarded Commands**

```
G   →   C

----     -----
guard    command
 /\        |
v  v       ---> list
T  F
```

We check whether G would evaluate to true. If so, then we can execute C. If not, we don't.

Booleans are easy:

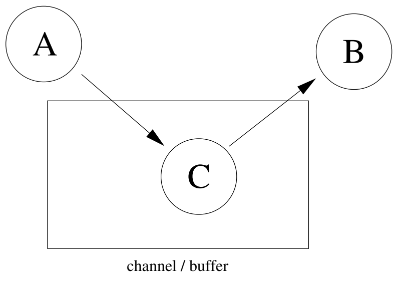
```
x > 3 → x++
```

can also have receive's (and sends)

```
S::R?x → x++
```

If this could succeed, then we do it and execute `x++`.

e.g. A single buffer



```
 C::  int x; A?x → B!x;
--------------------------------
             A?x; B!x
```

- **Iteration**

```
C::  *[int x; A?x → B!x]
```

This repeats what's in the brackets as many times as possible. When does it Stop? When the process inside is surely deadlocking.

- **Recursion**

```
C::  int x; A?x → B!x → C
```

These are equivalent but the recursion is less cumbersome.

Our comm. is blocking. Guards help, but work better if there's some way to choose. Call it **guarded choice**.

- **External choice (guard)**
  Notation: □

  e.g. Vending machine, tea/coffee

```
V: *[inslot? $1.00 → makeTea() □
     inslot? $1.10 → makeCoffee()]
```

  With this;

```
inslot:: V!$1.00
inslot|| V             // Here, we have external choice happening
```

  An interesting thought: What if the price are the same?

```
V: *[inslot?$1.00 → makeTea() □
     inslot?$1.00 → makeCoffee()]
```

  Here, the environment does not help make the choice. The process V makes the choice. i.e. makes an internal choice.

- **Internal choice** Internal choice actually a different operation.

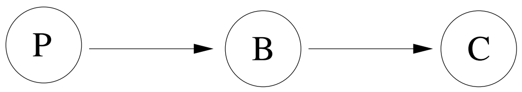
  Note: Internal choice do not care about the environment

```
V'::  *[ $\boxed{\texttt{inslot? \$1.00}}$ → makeTea() ⊓
          inslot? $1.10 → makeCoffee() ]
```

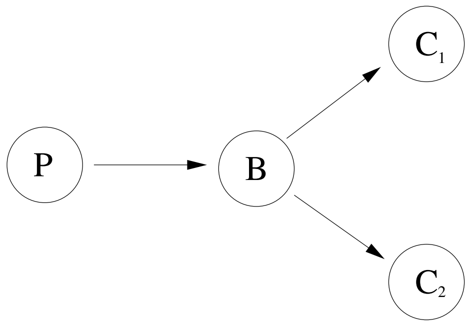
  This can actually deadlock.

```
inslot?V':$1.00
inslot||V'
```

  Easy to add multiple consumers in a 1-cell buffer environment



```
P::  *[ int x; x = produce(); B!x]
C::  *[ int x; B?x; consume(x) ]
B::  *[ int x; P?x → C!x]
```



```
C1::  [int x; B?x → consume(x)]
C2::  [int x; B?x → consume(x)]
B::  [int x; P?x → (C1!x □ C2!x)]
```

Note: Having external choice is better.

Using iterated guards, it is easy to build state-machine.

e.g. Elevator

```
int floor = 1, target = 0
 E::  declarations,[
         (b1?pressed && target == 0) →
             target = 1 □
          b2?pressed && target == 0) →
             target = 2 □

         target != 0 && target > floor → floor + 1 □
         target != 0 && target < floor → floor - 1 □
         target == floor → door!open → door?closed;target = 0
      ]
 B::  *[P1?press → E! pressed]
```

34 Lecture: March 31st, 2008

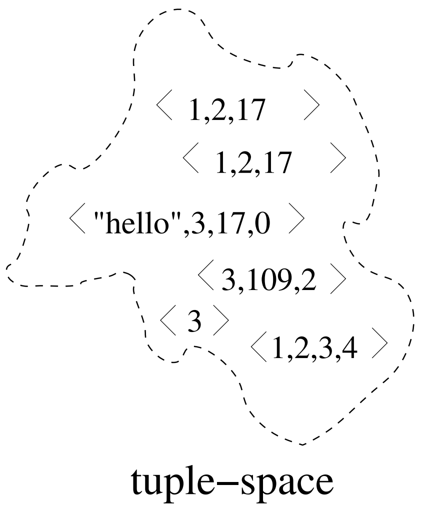
34.1 Last Time

- C.S.P. (a process algebra)
  - Message passing (Synchronous)
  - Sequential Composition, Parallel Composition, Guards, Iteration, Recursion, External Choice, Internal Choice, Arithmetics, Variables (local only), etc.
  - Elevator example (can you change it so people can exit the elevator too)?

34.2 Linda

Definition: Linda is a very simple paradigm for coordination and communication among several parallel processes operating upon objects stored in and retrieved from shared, virtual, associative memory. Linda was developed by David Gelernter and Nicholas Carriero at Yale University in the mid 80's.[9]

[9]Source: Wikipedia



Properties of this model:

- The **tuple-space** is a large, shared set of tuples.
- Multiple threads access the tuple-space.
- Processes communicate through the tuple-space.

34.2.1 The 4 Operations on Tuple-Space

Note that these opeations are all atomic with respect to the tuple-space.

1. `out(t)`. This operation (atomically) writes `t` into the tuple-space.
   `t` may be defined using expressions

   e.g.

   `out(<3,17+19,fact(4)>)`
   In fact, `out(t)` evaluates `t` to `t'` and then atomically puts `t'` in the tuple-space.

2. `in(t)` (atomically) removes and returns a tuple from tuple-space. It uses the template to decide which tuple to remove.

   ```
    in(t)
       |
       ---> t is a template
   ```

   `in(<1,2,17>)` removes tuple `<1,2,17>` from tuple-space.

   If there is no `<1,2,17>` in tuple space, then this call blocks until it can succeed.

   can also specify patterns:

   `in(<?i,?j,17>)`

   This looks for an (arbitrary) matching tuple and binds arguments accordingly.

3. `rd(t0)` (similar to `in`)

   The `rd` operation is very similar to `in`, templates, blocks, etc. The difference is that it does not remove the tuple.

   The tuples we've seen so far are passive tuples (i.e. data). To create new threads, we need active tuples.

4. `eval(t)`. (similar to `out`).

   `eval(t)` will actually launch a new thread to compute `t` into `t'`. When the thread is done, it deposits the resulting passive tuple into the tuple-space.

   e.g.

   `eval(<1,17+19,fact(4)>)`

   This launches a thread to compute `17+19 = 46` and it also compute `fact(4) = 24` and deposits the tuple `<1,46,24>`.

   Note: We can create threads that are infinitely running using the "eval" operation on an expression that never finishes to evaluate

e.g. Semaphore in Linda:

`s1.up()` can be implemented using `out(<"s1">)`.
`s1.down()` can be implemented using `in(<"s1">)`.

This is a general counting semaphore.

e.g. For-all Loops in Linda

```
for(i = 0; i < n; i++) {
    f(i);
}

for(i = 0; i < n; i++) {
    eval(<"loop",f(ii)>)
}
for(i = 0; i < n; i++) {
    in(<"loop",?x>)
}
```

e.g. Producer/Consumer Problem in Linda:

```
eval(<produce()>)
eval(<consume()>)

int producer() {
    i = 0;
    while(true) {
        x = produce_data();
        out(<"pc", x, i>)
        i++;
    }
}

consume() {
    i = 0;
    while(true) {
        in(<"pc",?x,i>
        consume(x);
        i++;
    }
}
```

34.2.2 Drawbacks of Linda

The tuple-space need every efficient implementation because it gets very large very fast. Also, fills up with garbage-tuples due to programmer mistakes.

It's not possible to always figure out if a tuple will be used. We cannot clean up. Limited to small programs.

If you want to play with Linda, this paradigm has been built on top of Java by Sun (called Java space) and by IBM (called TSpaces).

34.2.3 Leasing the tuples

tuples time-out in tuple-space and get cleaned up.

eval uses normal Java threads.

+non-blocking

35 Lecture: April 2nd, 2008

- Today's class is taught by Chris Pickett, Ph.D. student, supervised by Clark Verbrugge.
- cpicke@sable.mcgill.ca
- http://www.sable.mcgill.ca/~cpicke/spmt.pdf

Slides used with permission. The relevant papers on Chris' website provide details.

35.1 Speculative Multithreading (SpMT)

Speculative Multithreading is a subfield of Concurrent Programming. Its objective is to parallelize a sequential piece of code dynamically in an "unsafe manner". This allows a subtential speed-up in the execution of a sequential program, given that multiple processors are available.

Also, many people argue that concurrent programming is so hard that the paradigm should not be used. However, Speculative Multithreading provides us with the benefits of concurrent programming minus all its risks.

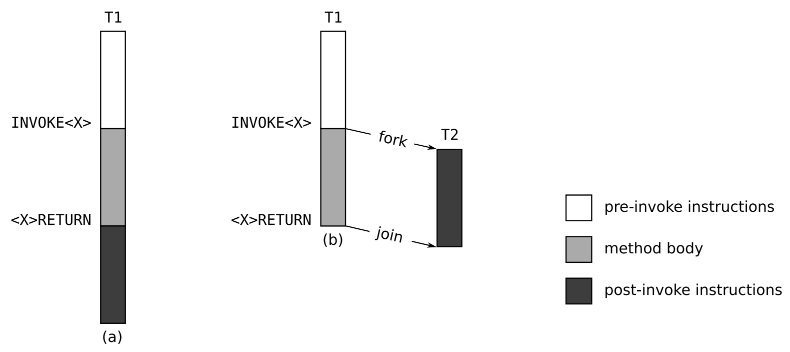
35.1.1 Introduction

Given the code below,

```
 r = foo();
```

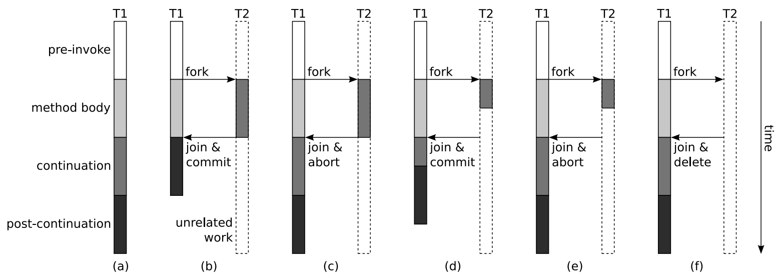
We expect that a sequential flow of execution will result in (a). But if, on the other hand, we use Speculative Multithreading, we can achieve (best case) an execution such as (b).

Method Level Speculation



However, the best-case (b) shown in the above slide does not always occur. We might be wrong in our speculation.

Detailed Method Level Speculation Execution Model



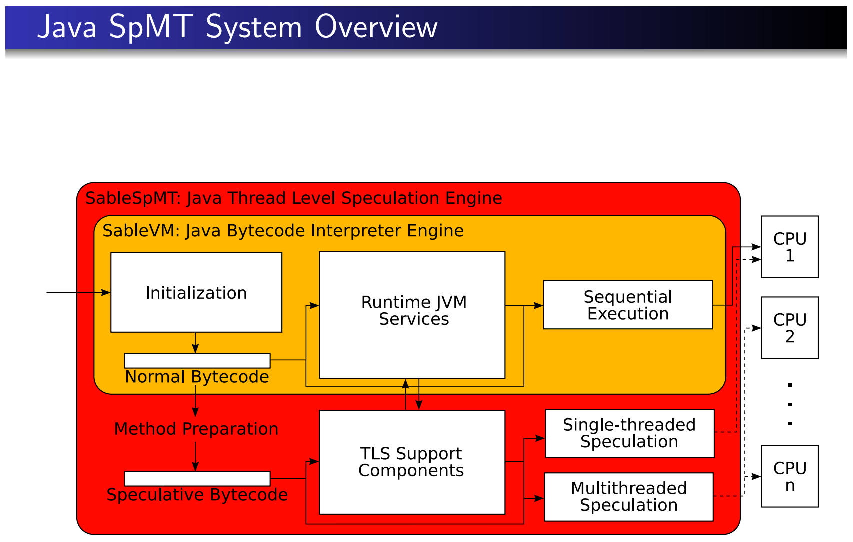
In (f) we fork the thread but T2 haven't started executing.

Threads in this figure are not really PThreads. Instead, we have a pool of processors. Some are speculative and some are non-speculative.

Note: There is one active PThread per processor. Ordinary Java threads are non-speculative PThreads whereas the other ones are speculative. If the number of non-speculative threads is greater or equal to the number of processors then no speculative PThreads will execute. The speculative PThreads dequeue speculative child "threads" or "tasks" from a priority queue and execute them.[10]

We're trying to have best case (b).

35.1.2 Design


Java SpMT System Overview
SableSpMT: Java Thread Level Speculation Engine
SableVM: Java Bytecode Interpreter Engine
Initialization
Runtime JVM Services
Sequential Execution
Normal Bytecode
Method Preparation
Speculative Bytecode
TLS Support Components
Single-threaded Speculation
Multithreaded Speculation
CPU 1
CPU 2
CPU n

Speculative bytecode may be executed by any processor at any time in any order.

Note: **TLS** stands for **T**hread **L**evel **S**peculation

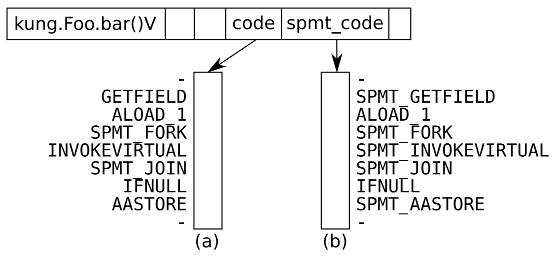
e.g.

```
// execute foo non-speculativel
r = foo(a,b,c);

// execute past return point
// speculatively in parallel with foo()
```

In this example, we're waiting for the function foo to return a value. We can either wait for the method to return or we can also try to guess a value for r. When foo has finished executing, we can compare the returned value by the guessed value and if they match, the speculative execution becomes non-speculative. If, on the other hand, the guessed value is not equal to the actual returned value, then we have to discard the speculative execution we compute what we've done speculatively so far.

[10] Comment by Chris Pickett

Method Preparation



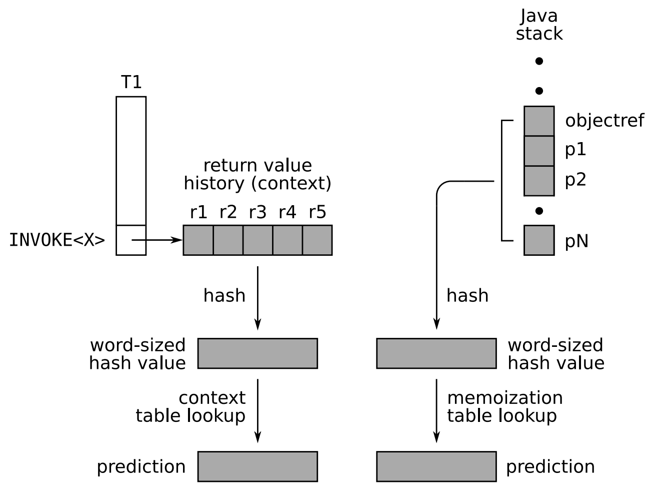
We need to buffer heap reads and writes.

Looking again at the piece of code above, if we see that the code following the call to the foo function uses r only to verify whether r is greater than 10, then we only need to predict whether this value is greater than 10 (or not). That is, we don't need to predict its exact value (i.e. don't need to discard the speculative execution).

Kinds of Predictors:

- Last-Value Predictor (1,1,1,1, $\longrightarrow$ 1)
- Stride Predictor (1,2,3,4, $\longrightarrow$ 5)
  This works for datastructures like arrays.
- 2-Delta Stride Predictor (1,2)
  This predictor updates the stride after it is the same twice in a row.
- Parameter Stride Predictor. This looks for a constant difference for stride between the returned value and one of the parameter.

Return Value Prediction



(a) Context Predictor (b) Memoization Predictor

Trade-off with memoization and context predictor is that you need a lot of memory to keep track of all the table lookups.

36 Lecture: April 4th, 2008

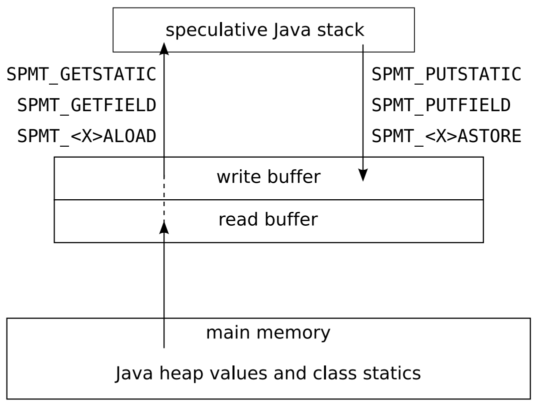
- Today's class is taught by Chris Pickett, Ph.D. student, supervised by Clark Verbrugge.
- cpicke@sable.mcgill.ca
- http://www.sable.mcgill.ca/~cpicke/spmt.pdf

Slides used with permission. The relevant papers on Chris' website provide details.

36.1 Speculative Multithreading (continued)

36.1.1 Design (continued)

Dependence Buffering



In the slide above, X is meant to be one of the 8 primitive types in Java (i.e. boolean, byte, char, short, int, float, long, double, and one reference type: a pointer to an object or a class).

writes		reads		heap	
addr	value	addr	value	addr	value
7	42	7	40	7	40

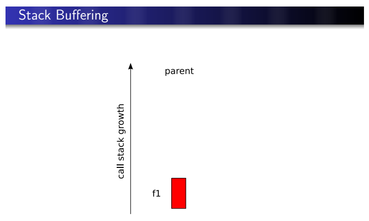
Let's suppose we want to load address 7. We...

1. ...look in the table of writes. (If it's not there, we go to the next step)
2. ...look in the table of reads (If it's not there we go to the next step)
3. ...retrieve from heap and store it in the read buffer.

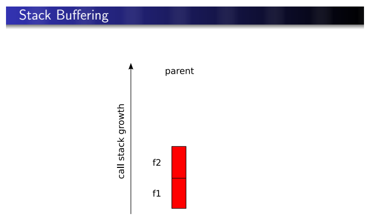
Note: If we want to write a value, we just look for it in the write buffer and create the entry if it's not there. If it is there, we just overwrite the old value. When the non-speculative parent thread returns from its call, all reads in the read buffer are compared against the values on the heap. If there are no dependence violations, then all writes in the write buffer are written to the heap.[11]

There are other schemes for implementing this (such as an undo-system) but these are more complex.

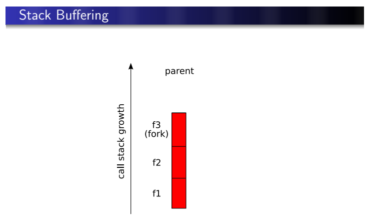
[11]Comment by Chris Pickett



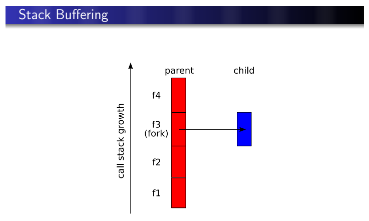
Stack Buffering



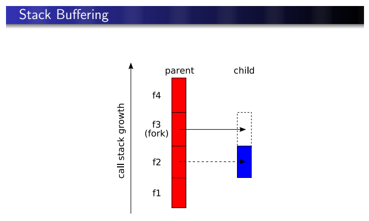
Stack Buffering



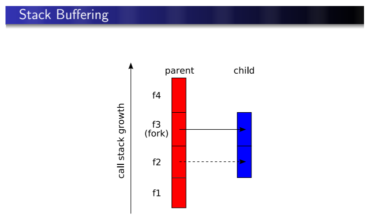
Stack Buffering



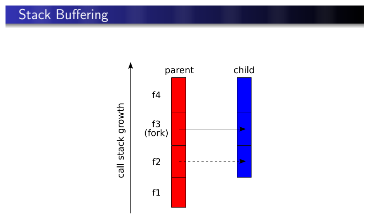
Stack Buffering



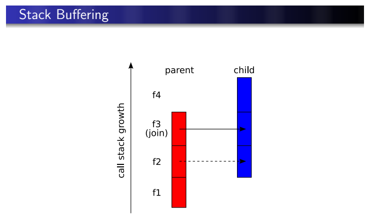
Stack Buffering



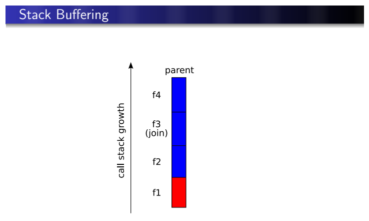
Stack Buffering



Stack Buffering



Stack Buffering

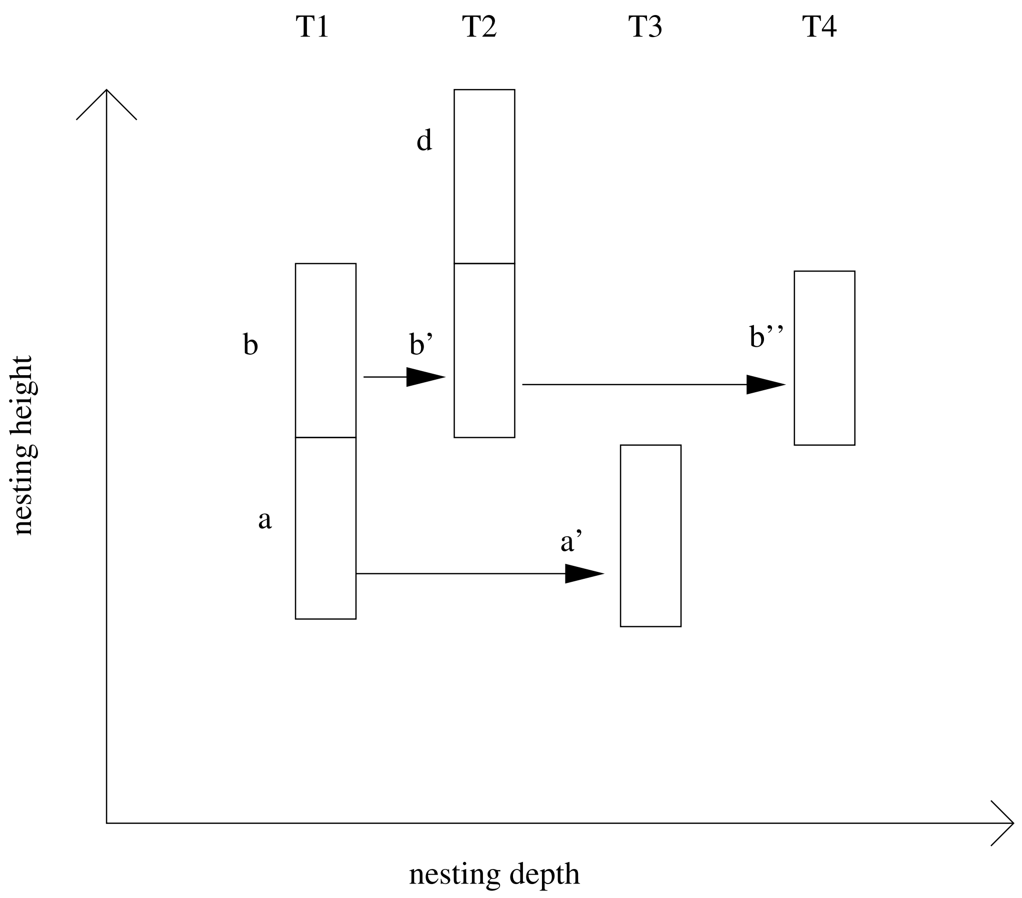


Stack Buffering

Note: In the slides above, *f1*,*f2*,... stand for *frame 1*, *frame 2*,...

Rules for joining threads:

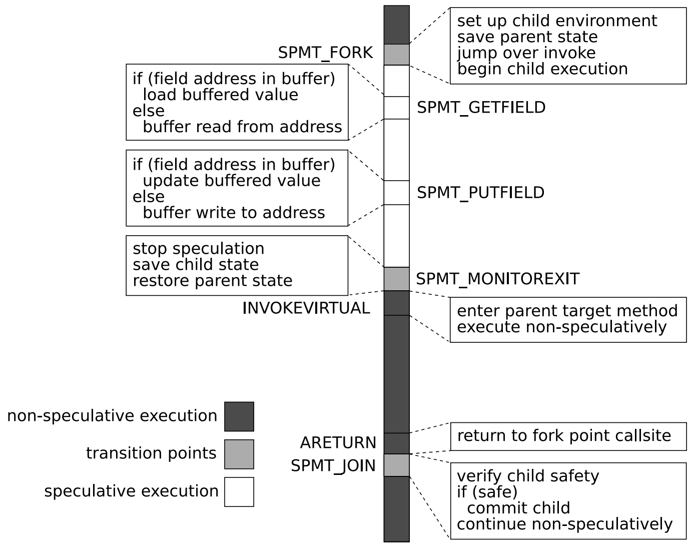
- A speculative thread cannot join another speculative thread
- can't return to a frame with an "elder sibling"
- aborts are recursive



```
// execute foo non-speculatively
r = foo (a, b, c);
// execute past return point
// speculatively in parallel with foo()
if (r > 10)
{
    s = o1.f; // buffer heap reads
    o2.f = r; // buffer heap writes
}
// invoke bar() speculatively
r.bar();
// stop speculation
synchronized (o4) { ... }
```

Some applications use a paradigm called "transactional memory" in order to have big critical sections while allowing every thread to enter their critical sections at the same time. However, with Speculative Multithreading, we don't allow speculative threads to enter/exit critical sections.

Single-threaded Mode



Note: In the slide above, buffer is meant to include both the read and the write buffer.

Make debugging easier because we have a sequential program (even though we can execute it using several threads)

37 Lecture: April 7th, 2008

- Today's class is taught by Chris Pickett, Ph.D. student, supervised by Clark Verbrugge.
- cpicke@sable.mcgill.ca
- http://www.sable.mcgill.ca/~cpicke/lock.pdf

Slides used with permission. The relevant papers on Chris' website provide details.

37.1 Component-Based Lock Allocation

- Critical section: piece of code that accesses shared state exclusively
- Lock: object that guards access to a critical section
- Lock allocation: mapping locks to critical sections

Note: A little vocabulary: we **acquire** a lock, but we **enter** a critical section

Sounds straightforward, but manual approaches are tricky!

Performance Degradation

```
class T1 extends Thread                      class T2 extends Thread
{                                            {
 public static Object a;                      public static Object b;

 run ()                                       run ()
 {                                            {
  synchronized (T1.a)                          synchronized (T1.a)
    {                                            {
     synchronized (T2.b)                          synchronized (T2.b)
       {                                            {
        t1Work();  <----performance---->             t2Work();
       }                degradation!                }
    }                                            }
 }                                            }
}                                            }
```

which lock protect which critical section is a tricky problem

deciding where critical section should start and end is not a trivial problem

Goal

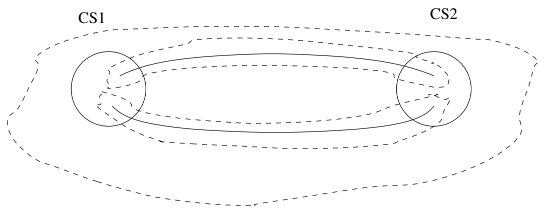
Our approach: *automatic* lock allocation

Goal: simplify concurrent programming

- Remove burden of manual allocation from programmer
- Aim to be *strictly* simpler: no extra language constructs
- Ideal result: automatic allocation performance matches or exceeds manual allocation performance

Automatic lock allocation: we want to match or exceed the performance

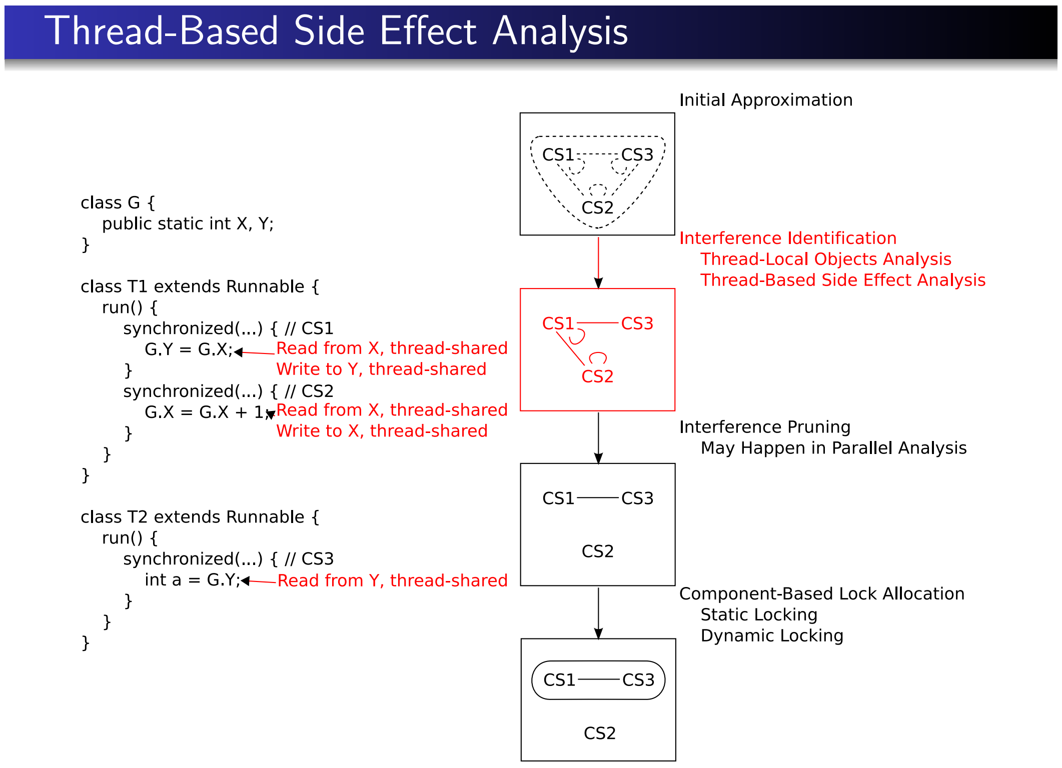
Interference graph:



each edge is an interference, i.e. a shared variable that is read by one critical section and writen by another.

reduce to graph coloring and graph coloring is NP-hard

and k-colouring (fixed number of locks k) is NP-complete.

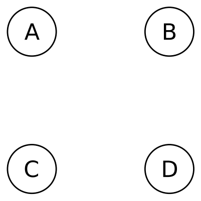


Thread-Based Side Effect Analysis

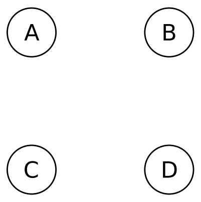
```
class G {
    public static int X, Y;
}

class T1 extends Runnable {
    run() {
        synchronized(...) { // CS1
            G.Y = G.X;
        }
        synchronized(...) { // CS2
            G.X = G.X + 1;
        }
    }
}

class T2 extends Runnable {
    run() {
        synchronized(...) { // CS3
            int a = G.Y;
        }
    }
}
```

Read from X, thread-shared
Write to Y, thread-shared
Read from X, thread-shared
Write to X, thread-shared
Read from Y, thread-shared
Initial Approximation
CS1
CS3
CS2
Interference Identification
Thread-Local Objects Analysis
Thread-Based Side Effect Analysis
Interference Pruning
May Happen in Parallel Analysis
Component-Based Lock Allocation
Static Locking
Dynamic Locking

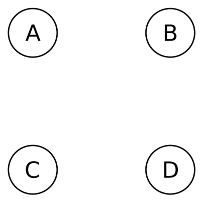
Lock Allocation Worksheet



Interference Graph



MHP Information



Pruned Interference Graph

	A	B	C	D
A				
B				
C				
D				

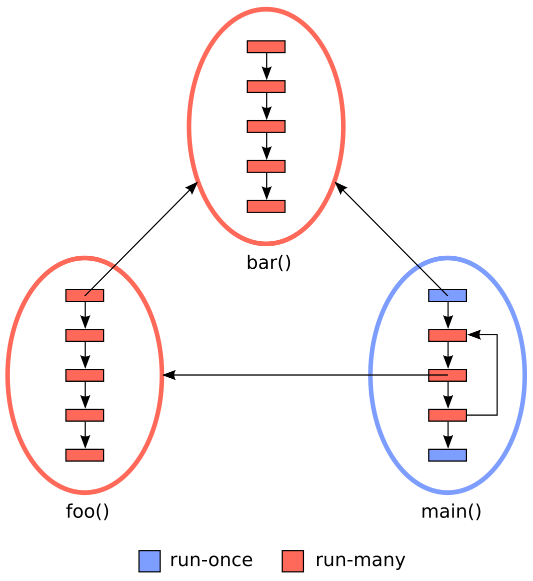
	A	B	C	D
A				
B				
C				
D				

	A	B	C	D
A				
B				
C				
D				

May Happen in Parallel Analysis

- MHP analysis finds methods that execute concurrently
- Several distinct steps:
  1. Identify run-once and run-many statements
  2. Identify run-once and run-many threads
  3. Categorize run-many threads as run-one-at-a-time or run-many-at-time
  4. Find methods that may happen in parallel based on thread reachability
- Critical sections that may not happen in parallel cannot interfere!

Run-Once Run-Many Analysis



This is a call graph

Anything with more than one incoming edge is a run-many

We have run-many statements inside loop, so foo itself is run-many

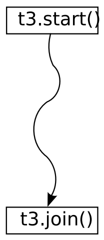
Note: Look up Fixed-Point Analysis

Initial approximation is run-once, at the method level and the statement level. alternate between analyzing methods and statements until fixed point is reached.

Note: If like this stuff, take grad course COMP621 Optimizing Compilers

t2 is run-once because you cannot start a thread more than once.

Finding run-one-at-a-time threads



- For each start, consider all joins:
  - Any valid join receiver must alias start receiver
  - Any valid join must post-dominate start
- If join is valid, check method validity:
  - Method must not be called recursively
  - Method must not happen in parallel with itself

Definition: B **post-dominates** A $\Leftrightarrow$ B is always executed after A

A receiver is just an object that receives a message from another function. In this case, the `Thread` object receives a `start()` message.

Three kinds of component-based lock allocation:

- **Singleton:** a single static lock protects all components
- **Static:** one static lock per component
- **Dynamic:** attempt to use per-data structure locks for each component, otherwise static

Finally, isolated vertices with no self loops are unlocked

38 Lecture: April 9th, 2008

Class is cancelled.

39 Lecture: April 11th, 2008

39.1 Last Time

- SpMT
  - Automatic Parallelization
  - Better models for concurrency

39.2 Dataflow

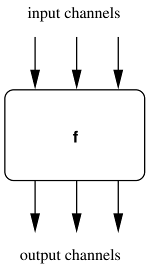
Functions don't conflict with each other: i.e. they are independant. Except for their input/output (through channel).

This model looks like cicuit. And this is convenient because there has been a lot of research on circuit.

Channels are FIFO, lossless and have capacity. Ideally, if you have a fixed capacity =¿ realistic: can build it.

Two ways to build it:

- Static (This is the model in assignment #4.)
- Dynamic (or stream-based dataflow)

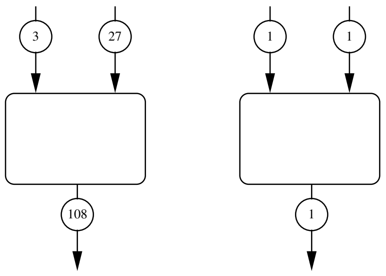


e.g. f here, reads/consume 1 token from each input channel and writes 1 token to each output channel.

This is a **regular actor** because it consumes a fixed number of tokens for each channel and writes fixed number.

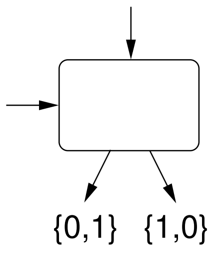
39.3 Homogenous actors

The fixed number is 1 for each input/output channel

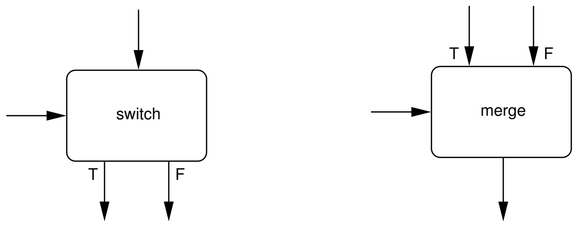


Regular actors make the outcome easy to analyse. However, regular actors limit the expressiveness.

39.4 Irregular Actors

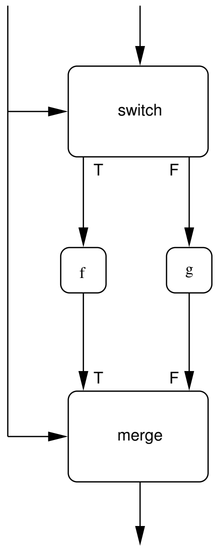


A new paradigm with irregular actors allow for the following.



Static Dataflow: regular actors + switch and merge.

IF-schema:



Abstraction: Actor looks regular from the outside

39.5 Stream-based dataflow

Let actors be arbitrary.

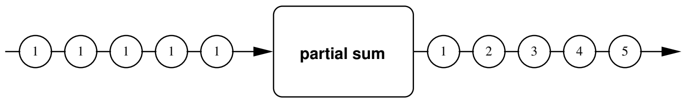
Consume/emit same/different numbers of tokens.

Analysing can be more difficult.

Not quite completely arbitrary.

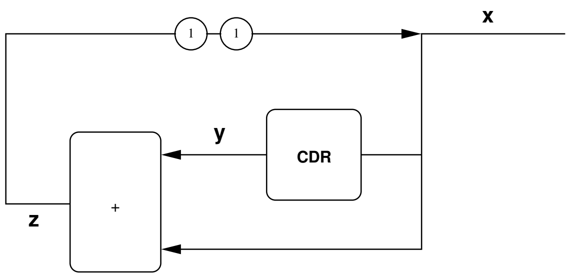
- continuity property
- order-preserving property on functions

e.g. What is not allowed:
An function f that detects whether its input is infinite or not and emits T or F accordingly



Even though the actor above is regular, it is not functional (i.e. for a given input, it does not yield the same output)

The one following, on the other hand, is functional. On the steam-level, it inputs $\mathtt{1}^{\infty} \rightarrow \mathscr{N}$



This circuit variables `x`, `y` and `z`.

To figure out what this circuit is doing, we will formulate dataflow functions that describe our variable.
Linda `x:: 1; 1; z`
`y:: CDR(x)`
`z::` x $\oplus$ y

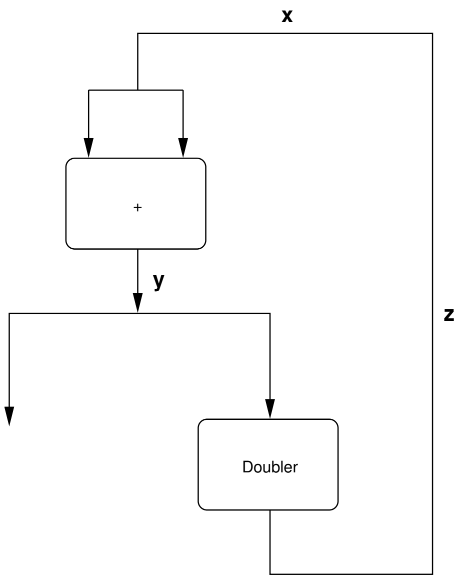
Now, to see the behaviour. We assume all channels are empty to start. and then we compute iterations.

At the end of this, we will get a nice description of the circuit.

This table expresses what tokens each channels x,y,z hold over iteration 0,1,2,3,4,5.

	0	1	2	3
x	-	1:1	1:1:2	1:1:2:3
y	-	1	1:2	1:2:3
z	-	2	2:3	2:3:5

We see that this process converges toward a description (not necessarily a finite one) of the temporal history of our entire network.



```
x::  1:z
y::  x ⊕ x
z::  Doubler(y) // Doubler outputs two tokens given one token input
```

	0	1	2	3
x	-	1	1:2:2	1:2:2:4:4:4:4
y	-	2	2:4:4	2:4:4:8:8:8:8
z	-	2:2	2:2:4:4:4	2:2:4:4:4:4:8:8:8:8:8:8:8:8

Index