Deadlocks in Java!

- What should you do if you have made a very complex Lab 1 which has deadlocks?
As we will see in Lecture 6, we can do as follows in C11:

```c
_Atomic int count;

/* Thread 1 */
count += 1;

/* Thread 2 */
count += 1;
```
Cache memories are extremely important on uniprocessors and even more important on multicores.

If all threads would perform each memory access from system RAM, programs would simply be too slow. For example, the bus would be overloaded.

Unfortunately, it’s even more difficult on multicores than on sequential machines to exploit cache memories.

Put simply, on sequential machines we get cache misses because the data does not fit (or was never accessed before) in the cache.

On multicores, we also get cache misses when new values are communicated between different caches, called **true sharing misses**.

In addition, we can also get cache misses due to some other processor wrote to a variable we are not interested in! These are called **false sharing misses**.
Caches in Multicores

- Suppose processor $P_1$ has read the value of a variable $X$.
- A copy of $X$ will be in the cache of $P_1$.
- Assume next $P_2$ writes a new value to $X$.
- That new value, $X'$, will be in the cache of $P_2$.
- What happens if $P_1$ wants to read $X$ again???
- Without caches $P_1$ would probably see the new value (if it has reached memory).
- Unless we take special actions, depending on the time between the write by $P_2$ and the second read by $P_1$, it is more or less likely that $P_1$ will read an obsolete value of $X$ from its cache!
- There are two special actions that the programmer needs to use:
  - $P_2$ must have released a lock $L$ after writing $X$, and
  - $P_1$ must have acquired the lock $L$ before reading $X$.
- The release by $P_2$ must happen before the acquire by $P_1$.
- We will go into details about this in Lecture 3 on memory consistency models.
The simplest model of a multicore machine has a shared memory and no private caches.

Without cache memories we do not have to be concerned about whether all threads see a consistent memory.

Of course we must still avoid data races and deadlocks!

Unfortunately, as we know, such a machine becomes too slow due to the huge memory access times.

However, the first multicore machine, from 1962, is similar to this, but the relative cost of accessing memory was less at that time.
A Second Model of a Multicore Machine

- A simple (and fastest for some applications) multicore machine is the Cell processor used e.g. in the Playstation 3, weapons and medical equipment
- The Cell processor was developed by IBM, Sony, and Toshiba
- It has a normal 64-bit POWER processor and a number of special compute nodes called **Synergistic Processing Units** or SPUs.
- Each SPU has a small private memory of 256 KB, called the **local store**.
- View the local store as a software managed cache of system RAM.
- The SPUs are programmed in C or C++ and on Linux the kernel schedules the SPU threads as it wishes... i.e. you can have any number of SPU threads regardless of how many SPUs your machine actually has.
- Normally the POWER processor controls the SPU threads and tells them what to do.
To process data on the SPUs we do the following:

1. Copy input data from system RAM to an SPU’s local store.
2. Let the SPU compute a partial result.
3. Copy the partial result to system RAM.

Simple isn’t it?
Yes, the Cell is simple and easy to understand

- The Cell is simple to understand for the programmer.
- The Cell is simple to make in hardware.
- It’s possible to write very fast software on the Cell due to its clever architecture (not because it’s simple, of course)
- In addition the Cell uses relatively little energy — the most energy efficient supercomputers were long all built with Cell processors.
- Here are some Gflops/s performance numbers: (flop = floating point operation)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cell</th>
<th>Cray X1E</th>
<th>AMD Opteron</th>
<th>Intel Itanium2</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense float matrix multiply</td>
<td>204.7</td>
<td>29.5</td>
<td>7.8</td>
<td>3.0</td>
</tr>
<tr>
<td>sparse float matrix multiply</td>
<td>7.68</td>
<td>-</td>
<td>0.80</td>
<td>0.83</td>
</tr>
<tr>
<td>2-D FFT</td>
<td>40.5</td>
<td>8.27</td>
<td>0.34</td>
<td>0.15</td>
</tr>
</tbody>
</table>
There must be a catch...or?

- Being simple to understand is **not** the same as being simple to use.
- The copying of data to/from the local stores must be done using messages created by the programmer.
- You are all used to programming machines with cache memories, and the local store of an SPU can be seen as a cache memory, except that you must manually transfer the data!
- So why are there no hardware managed cache memories for the SPU’s then?
- The researchers at IBM concluded that a sufficiently large number of applications can execute faster if we use the transistors for computing and not for cache memories — i.e. instead of only **one huge cache** we can have a number of small SPUs with their local stores.
- By letting the programmer manage the data transfers using a clever feature, very good performance can be achieved on the Cell.
An SPE contains an SPU and a local store of 256 KB.
On the Sony PS3 you can only use six of eight the SPU’s.
The third model of a multicore machine consists of a number of nodes which can send control and data messages to each other through a general interconnection network.

Each node has a processor, e.g. two levels of caches and a portion of the RAM memory.

Each node implements in hardware a protocol for transferring data between caches and memory.

The purpose of this protocol is to fetch data from a remote node at a read cache miss, and when writing to tell the other nodes which have a copy of that data that their copies have become obsolete.

The protocol is called a **cache coherence protocol**.

It is completely implemented in hardware.
Consider a parallel program with two threads, $T_1$ and $T_2$, and two variables $a$ and $f$, where $f$ is a flag.

Is the following code correct (in some sense)? Both variables are initially zero.

```c
int a, f;

// called by $T_1$
void v(void)
{
    a = u();
    f = 1;
}

// called by $T_2$
void w(void)
{
    while (!f)
    {
        printf("a = \%d\n", a);
    }
}
```

Can $T_2$ print out zero? (unfortunately, yes)

$T_2$ might in fact not print anything!
The compiler can legally reorder the writes to \( a \) and \( f \).

The compiler can allocate both \( a \) and \( f \) to registers (across multiple functions — not common but HP has done so for more than 20 years) and may never have to write back \( f \) to memory if \( v \) is called in an infinite loop...

Even if both of \( T_1 \)'s writes are performed in source code order, \( a \) and \( f \) may be located in RAM in different nodes and it may take longer time for the write to \( a \) to reach its node.

What is needed are strict rules about what may or may not happen. We will look at such in Lecture 3 but until then, the following rule should be used:

**Don’t use normal variables for synchronization — instead use locks!**
More about Unlock/Lock

Essentially, an unlock by $T_1$ will have the side effect of forcing the processor to wait until all previous writes have been noticed by all other processors.

A lock has the side effect of performing all pending incoming invalidations so the old value of $a$ is removed.

Then the code will work. Unlock and lock execute special instructions for this — see Lecture 5.
Parallel Programming in Java

- Creating Java threads
- Java synchronization
- The Java memory model and volatile attributes.
Creating Java Threads

- Either extend the Thread class or implement the Runnable interface.
- Your thread needs a public `void run()` method.
- Don’t call `run()` — you should instead call `start()`.
- To wait for a thread to terminate, you use `join` in a try-catch block.
A program which uses synchronization properly to avoid data races is said to be **thread safe**.

Every Java object has a lock. Before entering a method declared `synchronized`, the JVM checks if the calling thread is the owner of the lock. If the lock is owned by another thread, the calling thread is blocked and is put into an **entry set** for the object.

When the lock is released, some thread in the entry set is resumed and becomes the new lock owner.

If no thread owned the lock the calling thread becomes the owner at once.

Using `synchronized` we can avoid data races but we may end up with a deadlock, as illustrated on the next page.
public synchronized void insert(Object item)
{
    while (count == BUFFER_SIZE)
    {
        Thread.yield();
        ++count;
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
    }
}

- The remove and insert methods use the same lock to avoid data races.
- Suppose the producer calls insert, finds the buffer full, and calls yield() which lets another thread run.
- When the consumer calls remove it is blocked since the producer still owns the lock and a deadlock has occurred.
There are two methods `wait()` and `notify()` which are used to avoid the previous deadlock problem.

In addition to the entry set there is also a **wait set**.

Calling `wait()` releases the lock, blocks the thread, and puts it into the wait set.

The purpose is to let some other thread fix the object so it becomes usable for the thread which calls `wait()`.

Calling `notify()` wakes up one thread in the wait set if there was any there, puts it it into the entry set and sets its thread state to Runnable.

Calling `notify()` does **not** release the lock.

One can also call `notifyAll()` which wakes up all threads in the wait set.
The Java object lock is a so called **recursive lock** which means that we can call other synchronised methods for the same object without being blocked.

Of course, a thread may own multiple object locks (ie, call synchronised methods for different objects).

A notification for an object with an empty wait set has no effect (ie, the object does not remember the number of notify calls, as it does remember the number of V calls for a semaphore).
A thread waiting in the wait set can be interrupted, ie another thread can call its interrupt() method. This results in the InterruptedException being thrown:

```java
try {
    wait();
}
```catch (InterruptedException ie) { /* ignore */ }

The Java Memory Model

- The Java Memory Model has been changed after William Pugh identified problems with it several years ago.
- A relatively new keyword is `volatile`. It’s meaning has changed since it was introduced.
- Initially, the accessing of a particular volatile attribute of an object was serialized, i.e. all threads saw these accesses in one total order.
- These accesses were, however, unrelated to accesses of other variables.
- If you updated some variables and then set a volatile flag to indicate you were done, your program was buggy since the accesses were not ordered, i.e. the updates may be seen after the new value of the flag!
In more recent versions of Java an access to a volatile attribute is ordered with respect to other accesses in the same way as synchronized blocks:

- When entering a synchronized block, or reading a volatile attribute, the cache is conceptually made invalid.
- When leaving a synchronized block or writing a volatile attribute the cache is, again conceptually, copied to memory.

With this semantics using a volatile int flag as in the previous C code works as expected.
**Introducing Pthreads**

- Pthreads stands for POSIX Threads and is available on all UNIX machines, including Linux and MacOS X.
- POSIX stands for Portable Operating System Interface and is an API for UNIX programmers.
- Pthreads are quite similar to Java threads so nothing dramatic will happen to you...
- In the next slides we will show you how to start a thread and synchronize threads.
#include <pthread.h>

pthread_t is the type of a thread.

Create threads using pthread_create().

Wait for a thread using pthread_join().

Terminate a thread using pthread_exit().
int pthread_create(
    pthread_t* thread, // output.
    const pthread_attr_t* attr, // input.
    void* (*work)(void*), // input.
    void* arg); // input.

struct { int a, b, c } arg = { 1, 2, 3 };
status = pthread_create(&thread, NULL, work, &arg);

- The thread identifier is filled in by the call and attributes are optional.
- The created thread runs the work function and then terminates.
- A class is called a struct in C — no methods and everything public.
- Typically multiple arguments are passed in a struct as above.
A zero return value from `pthread_create()` indicates success, while a nonzero describes an error printable with `perror`. The `work` function can return a void pointer. Calling `pthread_join` waits for the termination of another thread and also gives access to the returned void pointer. A thread can only be joined once.
pthread_join()

```c
int pthread_join(
    pthread_t thread, // input.
    void** result);   // output.
```

- The call causes the caller to wait for the termination of a thread.
- If non-NULL, the terminated thread’s return value is stored in result.
- A thread can only be joined by one thread.
void pthread_exit(void*); // return value from work.

- Either use this or a return from the work function to terminate a thread.
- At termination of the main thread using exit or return, all other threads are killed, so use pthread_exit by the main thread instead.
- After a thread has terminated, the Pthreads system waits until some other thread joins with it. Then the terminated thread’s resources are recycled.
- If a thread will never be joined, it should have been detached so that the system can recycle resources at termination.
void pthread_detach(pthread_t thread); // recycle at exit.

- A thread can be detached from the beginning by specifying an attribute saying so at pthread_create.
- Or, any thread can call pthread_detach.
- A detached thread cannot be joined.
Next we will look at two programs:

- `sum.c` demonstrates a Pthread program.
- You will parallelize `Dataflow.java` in Lab 1.

You will later translate your `Dataflow.java` to Scala and then to C.
Parallelization of an Application

- Quite difficult usually!
- The programmer must identify parallel parts of a program, eg:
  - One outer loop iteration, or
  - one function call.
- The parallel parts which read or write the same data must communicate and be synchronised:
  - Either using message passing to communicate and wait for each other,
    or
  - communicate data through shared memory and synchronise with eg locks.
- Communication and synchronisation should be minimised to improve performance.
The Three Main Issues

- Ignore for the moment that memories are slow and caches are useful.
- Now the two main issues are:
  - Program correctness, and
  - Load-imbalance: some processors might have less work to do and must wait for others before proceeding.
- Unfortunately, memories can be hundreds of times slower than microprocessors:
  - Problem 1: communication usually creates new cache misses, and
  - Problem 2: caches in multiprocessors can introduce obscure bugs.
- All multiprocessors have some form of caches: optimizing performance usually means exploiting them better and this is the third main issue.
Programming on a Multiprocessor

- Global variables and data allocated with malloc/calloc are shared.
- Data allocated on the stack is private to a thread (but nothing prevents you from giving away a pointer to stack variables but doing so is uncommon).
- The most convenient way to parallelize a program is to use a good parallelizing compiler.
- The second most convenient way to parallelize a program is to do it **incrementally**, one loop at a time.
- There is a standard for this which is an extension to C/C++/Fortran called OpenMP.
- The extensions are specified by the programmer as comments in Fortran and as pragma (preprocessor directives) for C/C++.
- When these are not applicable, we parallelize manually.