Contents of Lecture 3

- Threads in Java
- Synchronized methods and blocks
- Lock and Trylock in Java
- `volatile` in Java vs in C
- Pthreads

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`.
- If `main` calls `run`, `main` has to do the work itself...
- To wait for a thread to terminate, you use `join` in a try-catch block.

The Work class

```
class Work extends Thread {

        Node            excess; // nodes with excess preflow
        Node            s;      // source node
        Node            t;      // sink node

        public void run()
        {
                while (excess != null) {
                }
        }
}
```

An incomplete preflow function

```
int preflow(int s, int t, int nthread)
{
        work                = new Work[nthread];
        for (i = 0; i < nthread; ++i)
                work[i] = new Work(node[s], node[t]);

        for (i = 0; i < nthread; ++i)
                work[i].start();

        for (i = 0; i < nthread; ++i) {
                try {
                        work[i].join();
                } catch (Exception e) {
                        System.out.println("" + e);
                }
        }
}
```

Java synchronization

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

Synchronized blocks

- Not only methods can be synchronized.

```
void push(Node u, Node v, Edge e)
{
        synchronized (u) {
        }
}
```

- The behaviour is the same, that the object's lock is attempted to be taken

Locking a useless object

- Suppose a thread succeeds in taking an object but finds it useless
- For instance a queue which is empty
- We would then like to wait for another thread to "fix" the object — such as putting something in the queue
- As it is, we have the lock and nobody can fix it
- We want to unlock and wait for the fix
- And be told when when we should check it out again
- A Java object has a condition variable for this

The Java Object Wait-Set

- There are two methods `wait()` and `notify()` for this
- 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.

More Details

- The Java object lock is a so called **recursive lock** which means that we can call other synchronized methods for the same object without being blocked by ourselves.
- A thread may own multiple object locks (i.e., call synchronized methods for different objects).
- A notification for an object with an empty wait set has no effect.

Using `wait()`

- 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:

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

ReentrantLock

- Sometimes it is more convenient to use an explicit lock and not the "implicit" / "internal" lock that is used with `synchronized`

```
import java.util.concurrent.locks.ReentrantLock;

ReentrantLock               x = new ReentrantLock();

x.lock();
x.unlock();
if (x.tryLock()) {
        x.unlock();
}
```

- If we cannot see a simple lock-order rule we can sometimes use tryLock
- Say you want to take some locks in a random order. Try one at a time and restart with the first if any fails.
- Can you see a simple rule to avoid deadlocks and tryLock for Lab 2?

`volatile` in Java

- The definition of `volatile` 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 the same 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!

`volatile` in Java

- Now `volatile` is more similar to `synchronized`
- When entering a synchronized block, or reading a `volatile` attribute, everything in the cache is conceptually made invalid.
- It is not in reality but think of it as all variables must be fetched from memory with their most recent values that have previously been written to memory.
- When leaving a synchronized block or writing a `volatile` attribute everything in the cache is, again conceptually, written to memory.
- The same concept applies to locks in Java (and C/C++):
  - Taking a lock fetches every new variable from memory
  - Unlocking it writes everything back to memory
- In addition, unlocking must wait for OK from memory that every thread knows about the writes
- Note that this is only conceptual otherwise extremely slow

`volatile` in C

- `volatile` in C tells the compiler that accesses to this variable should never be optimized in any way
- It is not at all used for multithreading
- Normal variables can be allocated a register in the CPU but never volatile variables

Introducing Pthreads

- POSIX stands for Portable Operating System Interface and is an API for UNIX programmers
- Pthreads, or POSIX Threads, is available on all UNIX machines, including Linux and MacOS X
- Pthreads are quite similar to Java threads
- They are enabled by:
  - `#include <pthread.h>` in the C source file
  - compiling with: `gcc file.c -pthread`
  - or: `clang file.c -pthread`

Getting started

- `#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()`

pthread_create() 1(2)

```
int pthread_create(
        pthread_t*                thread,          // output.
        const pthread_attr_t*     attr,            // input.
        void*                     (*work)(void*), // input.
        void*                     arg);            // input.

pthread_t                 thread;
int                       status;
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

pthread_create() 2(2)

```
int pthread_create(
        pthread_t*                  thread,          // output.
        const pthread_attr_t*       attr,            // input.
        void*                       (*work)(void*), // input.
        void*                       arg);            // input.
```

- A zero return value from `pthread_create()` indicates success, and a nonzero describes an error printable with `perror`.
- `void*` is a void pointer and is similar to Java's `Object`
- So the `work` function can return any data.
- Calling `pthread_join` waits for the termination of another thread and also gives access to the returned void pointer.

pthread_join()

```
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.
- In Lab 2 you don't need to return any value from a thread and can just use:

```
pthread_t       thread[t];
for (i = 0; i < t; i += 1)
   if (pthread_create(&thread[i], NULL, work, arg) != 0)
      error("pthread_create failed");

for (i = 0; i < t; i += 1)
   if (pthread_join(thread[i], NULL) != 0)
      error("pthread_join failed");
```

pthread_exit()

```
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.
- 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.

pthread_detach()

```
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.

Terminating a Thread

- There are three ways to terminate a thread:

1. Return from the work function.
2. The thread can call the function `void pthread_exit(void* value)`. The Standard C Library function `void exit(int status)` should normally not be used since it terminates the entire program.
3. Calling the function `int pthread_cancel(pthread_t thread)` makes a request to terminate the specified thread. See cancellation below.

- The first two are used by the thread itself and the third is used to stop another thread.
- Stopping another thread can be useful e.g. when a user has hit a "Cancel" button or another thread already has found a winning chess move.

Thread Cancellation Overview

- Simply terminating a thread can be disastrous if for example it has locked a mutex and is modifying shared data.
- Therefore the `pthread_cancel` simply requests that the thread should terminate. To actually know that the thread has terminated, it must be joined with.
- A thread that received a cancellation request is informed about this fact at certain points in the program, called **cancellation points**.
- The termination of the thread is started when it comes to such a cancellation point, if it has a **pending** cancellation request.
- A thread can install a function that is executed before the thread actually terminates.
- It is possible to allow cancellation at any time — see below.

Cancellation State and Type

- For cancellation, a thread has two variables, each with two possible values.
- The variables cannot be accessed directly but only through function calls.
- They are:
  - **State** — cancellation is either enabled or disabled.
  - **Type** — cancellation is either asynchronous or deferred.
- These result in three different cancellation modes:
  1. **Disabled** — any cancellation request received is saved until cancellation is enabled in the future.
  2. **Deferred** — cancellation is started at a cancellation point if there is a pending cancellation request.
  3. **Asynchronous** — cancellation can start at any time.

Modifying the Cancellation Mode

- The functions to modify the cancellation mode returns the thread's old value of the respective variable.
- `int pthread_setcancelstate(int state, int* old);` The state must be one of:
  - `PTHREAD_CANCEL_ENABLE`
  - `PTHREAD_CANCEL_DISABLE`
- `int pthread_setcanceltype(int type, int* old);` The type must be one of:
  - `PTHREAD_CANCEL_DEFERRED`
  - `PTHREAD_CANCEL_ASYNCHRONOUS`
- The default mode for new threads is **deferred**.

Cancellation Points

- A number of functions are cancellation points, including

```
pthread_cond_wait          pthread_cond_timedwait
pthread_testcancel         pthread_join
close                      creat
open                       read
system                     wait
waitpid                    write
```

- POSIX guarantees that the above (and some others) are cancellation points.
- Another list contains possible cancellation points, including

```
printf                     scanf
fopen                      fclose
```

- ISO C and POSIX functions not on any of those lists are guaranteed not to be cancellation points.

Receiving a Cancellation Request

- Thus if a cancellation request is received while cancellation is disabled, the request is simply blocked until it is enabled again.
- A pending request is delivered when the thread comes to a function which is a cancellation point.
- Changing the cancellation mode is **not** a cancellation point.
- At a cancellation point the thread first executes any installed cleanup handler (see below) and then terminates the thread.
- The return value from a cancelled thread is `PTHREAD_CANCELED` which thus is a valid value of a void pointer.

Installing Cleanup Handlers

- Each thread has a stack of cleanup handlers.
- They are installed with the function:
  `void pthread_cleanup_push(void (*func)(void*), void* arg);`
- The argument `arg` will be passed to `func` when it is executed.
- To remove a cleanup handler, use the function:
  `void pthread_cleanup_pop(int execute);`
- If the argument `execute` is nonzero, the cleanup handler will first be executed and then popped.
- When a thread is about to be terminated, all cleanup handlers on the stack are executed, starting with what is on the top of the stack.

Execution of Cleanup Handlers

- There are three situations when one or all cleanup handlers are executed:
  1. When a thread is being terminated due to a cancellation.
  2. When a thread is being terminated due to it has called `pthread_exit`.
  3. When it has called `pthread_cleanup_pop` with a nonzero parameter.
- In the last case, only one cleanup handler is executed, as we just saw.

Synchronization in Pthreads

- Pthreads has three main primitives for synchronization:
  - mutex
  - condition variable
  - barrier

Avoid synchronization!

- Ideally a parallel program needs no synchronization.
- Synchronization and therefore data communication between threads/caches take time.
- Some problems can be divided into suitable tasks statically.
- However, a common problem if T tasks are statically assigned to P threads is that some tasks take more time and therefore there becomes an inbalance in the work load, i.e. some threads take much longer time than the others.

Mutex

- A POSIX Threads **mutex** is a lock with a sleep queue
- The type is `pthread_mutex_t` and the most important functions related to it are:
- `pthread_mutex_init`
- `pthread_mutex_destroy`
- `pthread_mutex_lock`
- `pthread_mutex_trylock`
- `pthread_mutex_unlock`
- All five take a pointer to a `pthread_mutex_t`, and `pthread_mutex_init` also takes a pointer to attributes, which may be `NULL`.

Lock a mutex twice

- Trying to lock the same mutex multiple times does not work by default:

```
pthread_mutex_t A;

pthread_mutex_init(&A, NULL);
pthread_mutex_lock(&A);
pthread_mutex_lock(&A);
```

Recursive Mutex

- We must initialize the mutex as follows for this:

```
pthread_mutex_t A;
pthread_mutexattr_t     attr;

pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&A, &attr);

pthread_mutex_lock(&A);
pthread_mutex_lock(&A);
```

Worklists

- Assume we have a number of tasks to be processed.
- We put the tasks in lists and create threads which take tasks from the lists and process them.
- Concurrently adding or removing of items in the lists means the lists must be protected.
- Two alternatives:
  1. Put a mutex lock in each list head, i.e. protect the data.
  2. Use a common mutex for all lists, i.e. protect the code.
- Which is best depends on the application. There can be more concurrency if each list head has its own lock, at the cost of memory...

Condition variables in Pthreads

- As in Java, a condition variable lets a thread wait for something to happen in the future, and another thread to inform it that it has happened.
- For example: a worker thread can wait for a task being inserted in the list and another thread can signal any waiting thread that it just has inserted a new task.
- The condition variable type is: `pthread_cond_t`.
- In addition to initialization and destruction functions the main functions are:
  - `pthread_cond_wait` — causes calling thread to wait
  - `pthread_cond_signal` — wakes up one waiting thread
  - `pthread_cond_broadcast` — wakes up all waiting threads

Pthread mutex and condition variable

- Suppose you have locked a mutex and want to wait
- You need to both unlock and wait
- So that is done in one function atomically

```
pthread_cond_wait(&cond, &mutex);
```

- If they were two separate functions we would have problems
- Unlocking first could miss a signal
- (and waiting first could not unlock the mutex...)

More Details

- One mutex may be used for multiple condition variables
- Two threads wanting to wait for the same condition variable must use the same mutex.
- It is OK to signal a condition variable without having locked the corresponding mutex, but not so common.

Predicate, Condition Variable, and Mutex

- The logic expression in the C code which decides whether a thread should wait on the condition variable is called the **predicate** associated with the condition variable.
- The predicate is computed from shared data which different threads can modify, and therefore that data must be protected using a mutex.
- For example, the predicate may be computed by a boolean function:

```
bool empty(buffer_t* buffer);
```

- The predicate should always be tested in a loop and not in an if-statement.

Why You Need A Loop

- You should write your code like this.

```
pthread_mutex_lock(&mutex);
while (!predicate())
        pthread_cond_wait(&cond, &mutex);
/* do something... */
pthread_mutex_unlock(&mutex);
```

- There are at least three reasons for doing so:
  1. **Intercepted wakeups:** Another thread might have locked the mutex before.
  2. **Loose predicates:** This is a kind of predicate that says "the predicate may be true (but check before relying on it)."
  3. **Spurious wakeups:** This is very uncommon and is essentially an error that a thread was woke up without any good reason.

Intercepted Wakeups

- Should you signal a condition variable before or after you unlock its associated mutex (in case you have it locked) ???
- If you signal first, then the woke up thread will immediately try to lock the mutex and find it locked and wait again, now instead on a mutex, causing unnecessary synchronization overhead, both in the form of instructions and cache misses
- There may already be another thread waiting for the mutex which then later gets the mutex first
- If you unlock first, there may be a higher chance another thread takes the lock before the thread you wake up.
- It means the predicate might not longer be true after the other thread has unlocked the lock and it is your turn.
- This is called an intercepted wakeup.
- Due to intercepted wakeups, you must check the condition in a loop.

Loose Predicates

- It may be more convenient and/or efficient to say "you might have something interesting to check out" rather promising something.
- The woke up thread then must itself determine if there really was something for it, or whether it should continue waiting.
- This is called a loose predicate.

Spurious Wakeups

- When you hit CTRL-C to terminate a program you send a so called UNIX signal to it. This use of the word signal has nothing to do with the signal function of condition variables.
- When a thread receives a UNIX signal and was in the UNIX kernel waiting for a system call to complete, that system call is terminated and returns with the error code `EINTR`.
- Some UNIX signals are sent to all threads of a running program (called a **process**) and a thread waiting on a condition variable, i.e. in a system call on UNIX will thus be interrupted to handle the signal.
- An interrupted system call is not resumed but the application proceeds after it has returned.
- In principle a system call used by `pthread_cond_wait` could be interrupted and result in a spurious wakeup, but that is not the behaviour on Linux which uses the **futex** system call described below.

Implementation of Linux Native Pthreads Library

- Should a mutex lock involve the Linux kernel?
- Preferably not because it takes a lot of time
- On Linux there is a low-level synchronization primitive called **futex** which is used to implement the Pthreads library.
- Two good C libraries: gnu libc (glibc) and musl
- Linux command to show system calls: `strace -f ./a.out`
- `-f` tells it to trace threads and other programs it has *forked* (created)

User Level Locking with Futex in Linux

- Originates from IBM Research and the IBM Linux Technology Center.
- Implemented in the GNU C Library and in the Linux kernel, since version 2.5.7.
- The lock variable is in user space in shared memory and there is a corresponding wait queue for a lock in the kernel.
- The fast case is when there is no contention for the lock and therefore the kernel needs not be involved.
- The lock is manipulated in user space with atomic instructions (or the equivalent).

Initialization of a Pthread Mutex

```
pthread_mutex_t        mutex = PTHREAD_MUTEX_INITIALIZER;
```

- This initializes the mutex with default attributes.
- `PTHREAD_MUTEX_INITIALIZER` is a constant expression, meaning we can initialize a mutex like this at file scope (static storage, ie a `static` or global variable).
- If allocated by eg malloc, then `pthread_mutex_init()` should be called.
- After usage, `pthread_mutex_destroy` should be called for a mutex, and then its memory should be deallocated using `free`, if appropriate.

pthread_cond_timedwait

- To wait on a condition variable with a time out, use `pthread_cond_timedwait`.

```
int pthread_cond_timedwait(
        pthread_cond_t*,
        pthread_mutex_t*,
        struct time_spec*);
```

- The time is absolute time and to wait eg for at most 3 seconds, one can use:

```
timeout.tv_sec = time(NULL) + 3;
timeout.tv_nsec = 0;
```

- If there is a time out, the return value is `ETIMEDOUT`.

Pthreads barriers

- A barrier is used to let all threads work in a more synchronous way.
- All threads must reach `pthread_barrier_wait` before any can proceed beyond it.

```
int pthread_barrier_init(
        pthread_barrier_t*       barrier,
        pthread_barrierattr_t*   attr,
        unsigned int             count);

int pthread_barrier_destroy(pthread_barrier_t* barrier);

int pthread_barrier_wait(pthread_barrier_t* barrier);
```

Initialization in sequential programs

- The usual sequential way to initialize is to do like this:

```
#include <stdbool.h>
void f(void)
{
        static bool initialized = false;
        if (!initialized) {
                init();
                initialized = true;
        }
        ...
}
```

- Might not work in a multithreaded program!

Initialization in multithreadthreaded programs

```
#include <stdbool.h>
pthread_mutex_t  init_lock;
void f(void)
{
        static bool initialized = false;
        pthread_mutex_lock(&init_lock);
        if (!initialized) {
                init();
                initialized = true;
        }
        pthread_mutex_unlock(&init_lock);
}
```

pthread_once

- One can write `pthread_once_t once = PTHREAD_ONE_INIT;`.
- The `once` variable can have static storage duration.
- The function

  ```
  int pthread_once(pthread_once_t*, void (*)(void));
  ```

  is used to call a function once. It takes a pointer to a "once" variable and a function to execute for the initialization.
- If another thread executes the same call after the first is done, nothing will happen.
- If it instead calls it during the first call, the second thread will wait until the first call is done.

Thread attributes

- Examples of attributes which can be set:
  - Whether another thread can join with a particular thread (portable).
  - Stack address
  - Stack size (not portable)
- A thread which is not joinable is "detached" which means the resource used by the thread are recycled immediately when the thread terminates.
- The joinable attribute can be set to one of
  - `PTHREAD_CREATE_JOINABLE`, or
  - `PTHREAD_CREATE_DETACHED`.
- An initially joinable thread can make itself detached but not vice versa.