MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Threads

Daniel Playne
d.p.playne@massey.ac.nz

Processes

Reminder:
Processes are programs being executed on a system. They consist of not only the program itself but also the associated memory, register values, etc

Multiple processes can be created and communicate with each other using a method of interprocess communication.

Threads

So far we have assumed that each process has a single **thread** of execution. That is, when the process is running there is on program counter executing a single sequence of instructions.

However, many applications may have multiple different activities all going on at once.

Consider a computer game - it may be rendering graphics on the screen, communicating across a network, playing sound effects on an audio device...

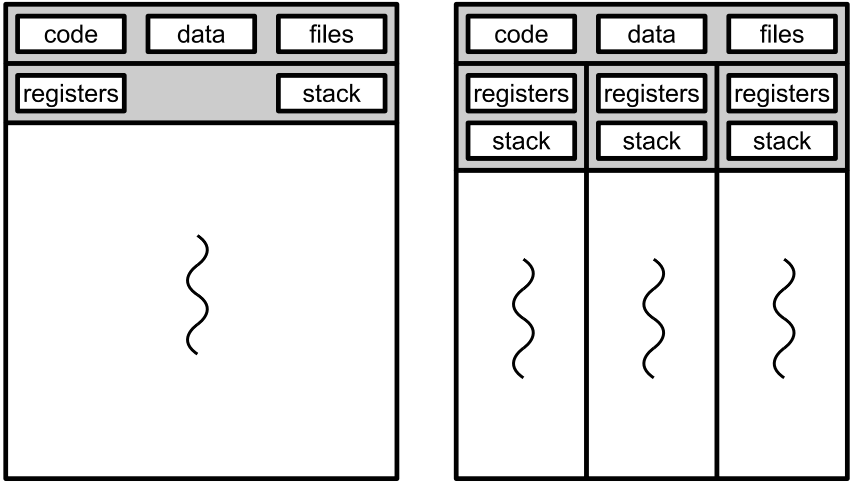
Threads

A thread is the basic unit of CPU utilisation - it consists of an id, a program counter, a register set and a stack.

Threads share code, data and other resources provided by the operating system (open files etc).

Traditionally processes have a single thread of execution, but we can write **multithreaded** processes.

Threads



Single-threaded vs multi-threaded process.

Threads

Many software applications designed for modern computers are multithreaded. For example, a web browser may have one thread to display an image while another thread is communicating on the network.

Most GUI applications are multithreaded applications that have a dedicated GUI thread that is responsible for managing user interaction and offloads longer tasks to other background threads.

Threads

Benefits of multithreaded programming:

- **Responsiveness** - prevent one part of an application being blocked by another performing a long operation (especially important for GUI applications).
- **Resource Sharing** - processes can communicate through shared memory and message passing but must be explicitly defined by the programmer. Threads share the same address space and are often easier to program.

Threads

Benefits of multithreaded programming:

- **Economy** - allocating memory and resources for a process is relatively costly. Threads share the resources and are generally more economical to create and switch between (in Solaris, for example, creating a process is roughly 30x slower than a thread and switching a process is roughly 5x more expensive).
- **Scalability** - multi-threaded processes can make use of all the resources of a multi-core processor (a single-threaded process can only run on one core).

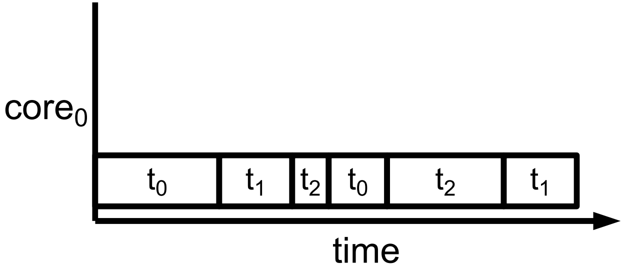
Threads

The benefit of **scalability** is becoming increasingly important, as the number of cores in CPUs increase programmers will have to write multi-threaded programs to make use of the available resources.

The are still benefits with writing multi-threaded applications for single-core CPUs but these are mostly to do with convenience rather than performance.

Concurrency & Parallelism

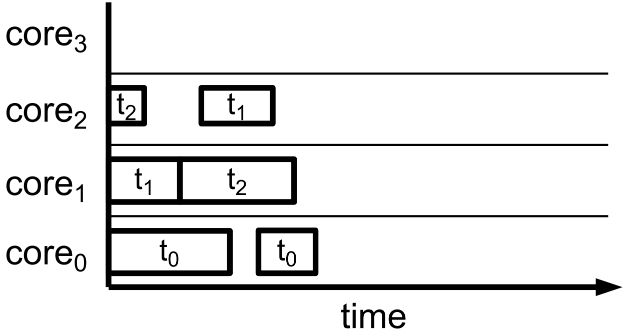
When a multi-threaded process executes on a single-core processor, the CPU can provide **concurrency**. Even though only one thread can execute at a time, their execution is interleaved so they all appear to be moving through their thread of instructions.



Concurrent execution of threads.

Concurrency & Parallelism

On a multi-core system, multiple threads can run in parallel as the threads can be assigned to different cores. Note the difference between **parallelism** and **concurrency**.



Parallel execution of threads.

Amdahl's Law

Amdahl's law determines the potential performance benefit of allocating N processing cores to an application which consists of a serial S portion and a parallel portion.

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

It is interesting to note that as $N \to \infty$ then the speedup converges to $\frac{1}{S}$, at a certain point assigning additional cores to a problem provides no benefit.

Multithreading Models

There are two general models for threading support - **user threads** and **kernel threads**.

User threads - are supported by a user library outside the kernel. From the view of the kernel, the process is a regular single-threaded process.

Kernel threads - are supported by the operating system kernel itself. The kernel must keep track of the threads in a user program and can schedule them independently.

User Threads

Implementing threads at the user level requires each process to have its own runtime system to keep track of its threads and store the information in a - **thread table** (similar to a process table just for threads).

The thread table must store each thread's program counter, stack pointer, registers, state etc. Threads will go through a similar set of states (ready, blocked) to processes and the next thread for execution will be chosen by a thread scheduler.

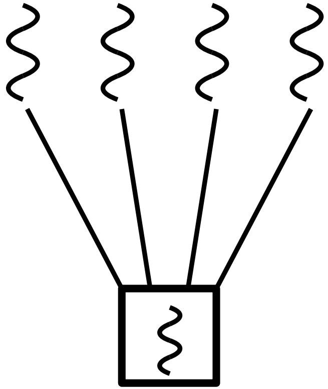
User Threads

The advantages of user threads is that switching between threads and calling the scheduler are all local procedures and are faster than making a system call to the kernel. Each process can use its own custom thread scheduler specifically designed for that process.

The downsides of user threads are how to deal with blocking system calls (and how to prevent all threads from blocking) and how the thread scheduler can stop one thread from running forever.

User Threads

This approach to multithreading is also called the **many-to-one** approach as many user-level threads are mapped onto one kernel-level thread.



Many-to-one - **user threads**

Kernel Threads

The other approach is for threads to be implemented by the kernel - with this approach no run-time system is necessary for the process to keep track of its threads. Instead the kernel is responsible for maintaining a thread table and scheduling the threads to execute.

Threads can only be created, destroyed etc through system calls to the kernel which is then responsible for creating the thread and storing its information in the thread table.

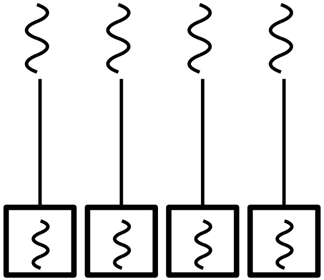
Kernel Threads

The advantages of kernel threads is that the kernel can schedule them properly onto different cores in a multicore CPU and do not require any changes to programs that involve blocking system calls.

One downside of kernel threads is the relatively higher overhead of creating a new thread (which requires a system call to do so).

Kernel Threads

This approach to multithreading is also called the **one-to-one** approach as each user-level thread maps directly to a kernel thread.


One-to-one - **kernel threads**

Threads

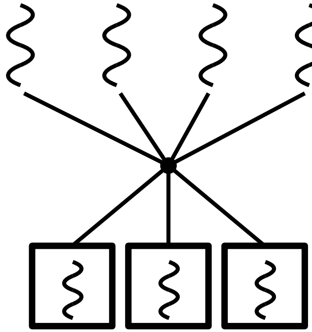
User threads (many-to-one) allow the programmer to create as many threads as they wish but does not result in true concurrency as the kernel will only schedule one thread at a time.

Kernel threads (one-to-one) allow for true concurrency but the programmer must be careful not to create too many threads.

The **many-to-many** model multiplexes many user-level threads together onto a smaller (or equal) number of kernel threads.

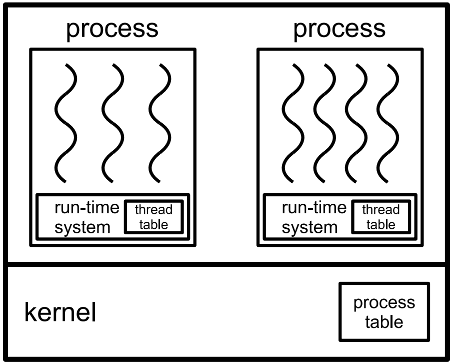
Threads

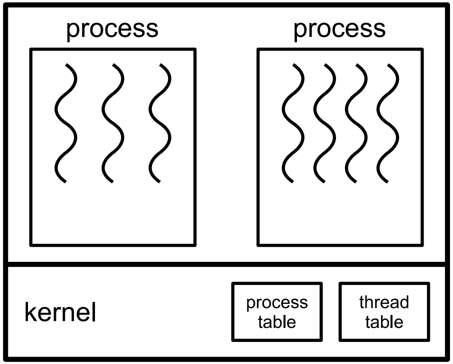
The advantage of the many-to-many model is that the user can create as many threads as they like, gives true concurrent computation (when supported by hardware) but does not overload the kernel scheduler.



Many-to-many threads

Multithreading Models





Multithreading model - **user threads** and **kernel thread**.

Asynchronous & Synchronous Threading

There are also two general strategies for how threads execute when they are created - **asynchronous** and **synchronous** threading.

Asynchronous Threading - once a parent thread creates a child thread, the parent resumes its own execution so that both the parent and child are executing concurrently.

The parent and child threads with this model will often perform independent tasks with little or no communication between them.

Asynchronous & Synchronous Threading

Synchronous Threading - involves a parent thread creating one (or more) children threads and then at some point waiting for all of those children to terminate before resuming execution.

This is the **fork-join** model and is usually used when the threads are cooperating on the same task with significant data sharing.

The parent must wait for the children to complete their tasks before the computation can continue.

Thread Libraries

Thread libraries are APIs that can be used to create and manage threads in user programs. These low-level APIs can be used to develop parallel code.

Examples:
POSIX pthreads
Windows Thread library
Java thread API

Thread Libraries

For both POSIX and Windows thread libraries, globally declared data is shared between all threads created by the same process.

Each thread maintains its own stack and will have its own copy (not shared) of local data.

As Java does not have global data, access to shared data must be explicitly defined.

POSIX threads

POSIX threads or Pthreads is an extension to the POSIX standard to introduce functions for working with threads - it is a specification rather than an implementation.

Operating system designers are free to implement the specification in anyway they wish. Pthreads is supported on most UNIX-type system - Linux, macOS, etc.

Windows does not officially support Pthreads but there are third-party implementations for Windows.

POSIX threads

The main types and important functions for Pthreads are as follows:

`pthread_t`	thread identifier type
`pthread_attr_t`	thread attributes type
`init()`	initialise default attributes
`pthread_create()`	create thread
`pthread_join()`	wait for thread to exit
`pthread_exit()`	terminate thread

POSIX threads

The `pthread_t` type is a data structure used to store the information for a thread. A new thread is created used `pthread_create()` and another thread can wait for a thread to terminate using `pthread_join()`.

The most common use of Pthreads is simply to create and run separate threads.

Examples:

`Example - Thread - Pthreads`
`Example - Threads - Pthreads`
`Example - pi - Pthreads`
`Example - Scheduling - Pthreads`

Windows threads

Windows threads are supported by the Windows API and provide a similar set of types and functions to POSIX.

`HANDLE`	thread identifier type
`CreateThread()`	create thread
`WaitForSingleObject()`	wait for thread to exit
`SetThreadPriority()`	set thread priority attribute
`ExitThread()`	terminate thread

Windows threads

The Windows API can be used to create a similar set of examples as POSIX.

Examples:

`Example - Thread - Windows`

`Example - Threads - Windows`

`Example - pi - Windows`

Summary

- Threads
- Multithreading models - user/kernel threads
- Asynchronous/synchronous Threading
- Thread Libraries - POSIX, Windows API