

# Processes

The thing which represents our work to the system.

Sometimes referred to as a heavyweight process.

*An instance of a program in execution.*

*An instance...*

May be more than one version of the same program running at the same time.

(Hopefully sharing the code.)

Each instance has resource limitations, security information - rights, capabilities etc.

*...of a program ...*

So it includes code, data, connections (to files, networks, other processes), access to devices.

*... in execution.*

It needs the processor to run. But it doesn't run all the time.

So it needs information about what it is up to stored somewhere.

## Two parts to a process

1. Things the process owns (may be shared), resources. Also information about the process.
2. What the process is doing - the stream of execution.

Traditional processes had *resources* and a *single current location*. e.g. traditional UNIX.

The resource part is called a *task* or a *job*.  
The location part is commonly called a *thread*.

Most operating systems now provide support to keep these parts separate, e.g. Solaris, Windows NT/XP (or 95/98/ME for that matter), Mach (basis of MacOS X).

# Threads

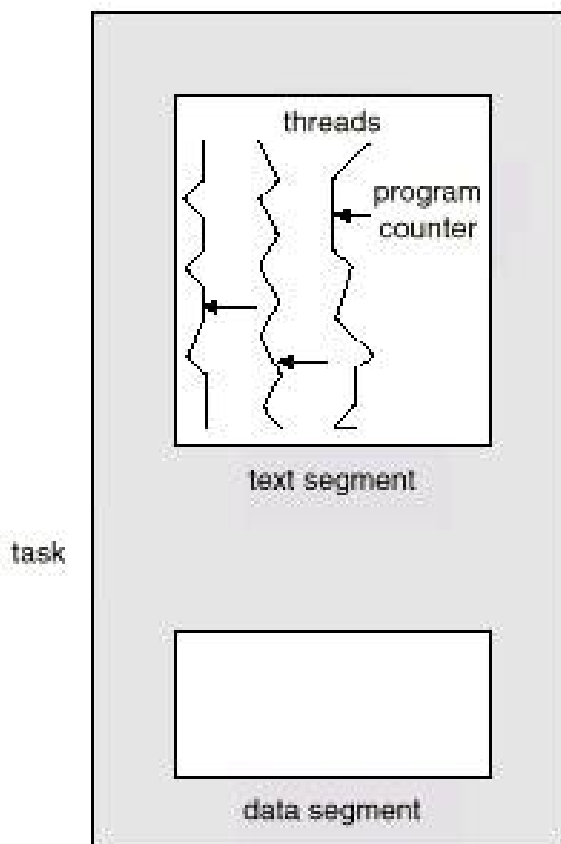
Sometimes referred to as lightweight processes.

Sometimes we want to share data as well as code.  
(Could just share files or memory and not use threads.)

Easier to create than a process.

They provide a nice encapsulation of a problem within a process rather than multiple processes.

Easier to switch between threads than between processes.



# Typical uses

- splitting work across processors (shared memory multiprocessor)
- added responsiveness (handle user input while still finishing another function)
- controlling and monitoring other threads
- server applications
- can help program abstraction
- 340/370 assignments

# Thread implementation

## User-level

- The OS only sees one thread per process.
- The process constructs other threads by user-level library calls or by hand.
- User-level control over starting and stopping threads.
- Usually a request is made to the OS to interrupt the process regularly (an alarm clock) so that the process can schedule another thread.
- The state of threads in the library code does not correspond to the state of the process.

## System-level

- The OS knows about multiple threads per process.
- Threads are constructed and controlled by system calls.
- The system knows the state of each thread.

# User-level thread advantages

Works even if the OS doesn't support threads.

Some implementations of Java have user-level threads because the underlying OS doesn't.

Ruby provides threads even on DOS.

Easier to create - no system call.

Just a normal library procedure call.

No switch into kernel mode (this saves time).

Control can be application specific.

Sometimes the OS doesn't give the type of control an application needs.

e.g. precise priority levels, changing scheduling decisions according to state changes

Easier to switch between - saves two processor mode changes.

Can be as simple as saving and loading registers (including SP, PSW and PC).

So why would anyone want to use system-level threads?

# System-level thread advantages

Each thread can be treated separately.

Rather than using the timeslice of one process over many threads.

Should a process with 100 threads get 100 times the CPU time of a process with 1 thread?

A thread blocking in the kernel doesn't stop all other threads in the same process.

With the user-level threads if one thread blocks for IO the OS sees the process as blocked for IO.

On a multiprocessor different threads can be scheduled on different processors.

This can only be done if the OS knows about the threads.

# Jacketing

The major problem with user-level threads is the blocking of all threads within a process when one blocks.

A possible solution is known as jacketing.

- A blocking system call has a user-level jacket.

- The jacket checks to see if the resource is available, e.g., device is free.

- If not another thread is started.

- When the calling thread is scheduled again (by the thread library) it once again checks the state of the device.

So there has to be some way of determining if resources are available to accept requests immediately.



# The best of both worlds?

Solaris (versions < 9) has both user-level and system-level threads.

LWP – light-weight process (what we have been calling system-level threads)

Kernel threads – active within the kernel

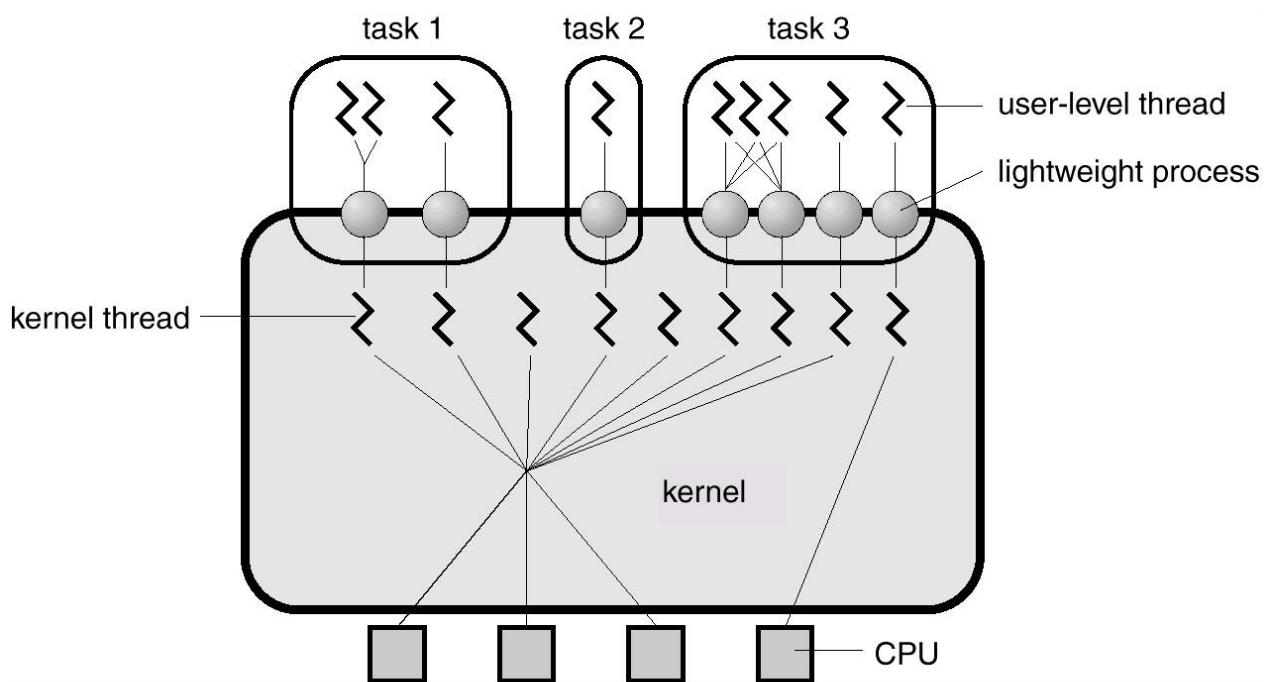
Each LWP is associated with one kernel thread.

One or more user threads can be multiplexed on each LWP.

A process can have several user and several LWPs.

The number of LWPs per process is adjusted automatically to keep threads running.

# Solaris < 9 process thread system



From version 9 onwards, Solaris uses one-to-one mapping of user-level and kernel-level threads.

# PThreads

## User-level thread library

### POSIX 1003.1c threads

`pthread_create` - creates a thread, one of the parameters is a function to execute

`pthread_exit` - finishes, threads can also finish by returning from the function passed in create

`pthread_join` - wait until another thread has finished

`pthread_cancel` - request that another thread exits - can be ignored, acted on immediately, or deferred to a *cancellation point*

## Clean up handlers

These are run when a thread exits or is cancelled - so the thread can release resources such as locks.

Can be added and deleted as the thread runs.

## TSD - Thread Specific Data

Global or static variables which can be different in each thread.

A block of memory for each thread.

## Communication and synchronization

Mutexes and condition variables (more on these later in the course)

When a thread starts another program all other threads are supposed to die. Why?

# Linux threads

Clone - makes a new process (more on that later)

Shares - memory, open files (actually descriptors),  
signal handlers

From one point of view Linux threads are processes -  
but they share all resources and hence the advantages  
of threads.

**Linux threads and POSIX – latest versions probably  
fix some of these.**

Can't be set to schedule threads according to priority  
within a process - each thread is scheduled  
independently across all threads/processes in the  
system.

Ordinary system calls e.g. read, are not cancellation  
points.

Starting a new program in one thread doesn't kill the  
other threads in the same process.

When a Linux thread blocks doing IO do all other  
threads in the same process stop? Why/why not?

# Java threads

May be implemented either with system-level or user-level thread systems.

The Thread class represents threads.

You control threads via a Thread object.

```
public class Thread implements  
    Runnable
```

Can construct our own by extending this class.

```
public class Consumer extends Thread {  
    public Consumer() {  
        ...  
    }  
    public void run() {  
        ...  
    }  
}
```

## Java threads (cont.)

Can also just implement the Runnable interface. Then construct a Thread passing the Runnable object as the parameter. The `run()` method is essential. It is where the new thread starts running after calling `start()` as in:

```
Consumer consumer = new Consumer();  
consumer.start();
```

The thread finishes when the run method finishes.

(It is not a good idea to use `stop` as the thread may be in the process of changing some shared data. It is deprecated.)

# Java thread management

`start()` – starts the thread, eventually calling the thread's `run()` method.

`sleep()` – pauses the thread for a number of milliseconds

N.B. this is notoriously inaccurate for use as a timer especially on Windows OSs – resolution is about 50msec on Win95/98 and about 15msec on NT.

This is actually a static method and can be called in any code

```
Thread.sleep(timeInMillis);
```

Since the sleeping thread could be interrupted we have to make sure we can catch this:

```
try {  
    Thread.sleep(50);  
}  
catch (InterruptedException e) {  
    System.err.println("Awoken from my  
    sleep");  
    return;  
}
```

`join()` – the current thread waits for the joined thread to finish.

`yield()` – the current thread volunteers control to other threads.

# Java thread priorities

Threads of higher priority run before threads of lower priority (sort of, may just be more often).

Can find the priority of a thread

```
consumer.getPriority();
```

Priority can be changed with

```
consumer.setPriority(newPriority);
```

10 is the maximum priority, 1 is the minimum.

Default priority is 5. Used by the event loop thread.

Not really enough levels – UNIX allows 255, XP has 6 (sort of).

If you want a reference to the current thread i.e. the one executing the statement at the moment:

```
Thread referenceToCurrentThread =  
    Thread.currentThread();
```



# Ruby threads

Almost always user-level threads, but can be compiled as pthreads.

Controlled by Thread objects.

Creating a Thread object also starts it running.

```
thread = Thread.new(parameterList) {  
    |parameters| ... }
```

The thread finishes when the block finishes.

`Thread.stop` - stops the current thread (makes it wait)

`thread.wakeup` - starts “thread” going again

`thread.join` - waits for “thread” to finish

`Thread.pass` - same as Java's `yield`

# Ruby threads

`sleep` – actually part of the built-in Kernel module

Current thread - `Thread.current`

## Priorities

The larger the priority the better the priority.  
Default 0.

`thread.priority` – method to return the priority

`thread.priority =` – method to change the priority (seems to take any integer, positive and negative)

# Before next time

## Read the Ruby API about threads

[http://www.cs.auckland.ac.nz/references/ruby/doc\\_bundle/ProgrammingRuby/book/ref\\_c\\_thread.html](http://www.cs.auckland.ac.nz/references/ruby/doc_bundle/ProgrammingRuby/book/ref_c_thread.html)

[http://www.cs.auckland.ac.nz/references/ruby/doc\\_bundle/ProgrammingRuby/book/tut\\_threads.html](http://www.cs.auckland.ac.nz/references/ruby/doc_bundle/ProgrammingRuby/book/tut_threads.html)

Read from the textbook

4.1 Process Concept

Chapter 5 Threads

5.1 Overview

5.2 Multithreading Models

5.3 Threading issues

5.4 Pthreads

5.5 Windows XP threads

5.6 Linux threads

5.7 Java Threads