These overheads and other things can be found at:

http://www.cs.auckland.ac.nz/~robert-s/415.340/lectures_1997.htm

Processes and Threads

What is a process?

What is a thread?

Textbook - Ch 4.1, 4.5 (Do read 4.5.2 it is interesting)

Java threads

Getting work done - performing some useful task

Not merely the same as running a program

- More than one person running the same program
- More than program to perform the useful task
- Running different parts of the program simultaneously on multiple processors
- Coordinating several programs (possibly on several processors) to fulfil the task

A process is:

- *an instance of a program in execution*

This is an old pragmatic definition

- it doesn't really fit anymore (but we will use it to start with)

What does a process need to run?

Code & data - memory

other resources

Processor

This leads to two different views

Things the process owns - sometimes shared

code & data

files

resource limitations

security information - rights, capabilities

access to devices, including windows

What the process does

stream of execution - current location

That is how it was for a long time

a process both owned *resources* and had a *current location*

Resource part

called a task in Mach

Location part

commonly called a thread of execution

Why not have more than one location?

Threads (lightweight processes)

Share common resources of the process

Easier to create than processes - why?

very little resource allocation - a thread data structure and some stack space

Easier to switch between - why?

few resource changes are required

page tables

open files

and all other process information remains the same

save current thread registers, pc and sp

load the next threads registers

But what about a thread switching between a thread in one process to a thread in another process?

this takes just as long as previously

Use of threads

true parallelism - if more than one processor (multiprocessor)

added responsiveness - especially for the user interface

server applications

can simplify program abstraction (can also complicate) e.g. Explorer windows in Win95

Different implementations

user level - the process itself creates and controls the scheduling

advantages:

works even if the OS doesn't support threads

even easier to create - no system call required

easier to swap between - no processor mode changing

system level - OS knows about the threads, schedules them, can start and stop them

advantages:

when a thread blocks in the kernel other threads in the same process can continue

can start different threads on different processors

Different implementations of Java use both these approaches, but so far only the latest version for Solaris allows different threads to run on different processors.

Since all current computers have several computational units system level threads are a good idea

Process Control Blocks & Process Tables

PCB - Process Control Block

place from where the OS can find all the information we need to know about a process

Process Table - collection of PCBs

commonly an array of pointers to PCBs

List the information you think should be in a PCB

memory

open streams/files

devices, including abstract ones like windows

processor registers

process identification

process state - including waiting information

priority

owner

which processor

links to other processes (parent, children)

resource limits/usage

Doesn't have to be kept together

Different information is required at different times

Linux PCBs in the same order as above (hunt and seek in the real thing)

```
/* memory management info */
  struct mm_struct *mm;

/* open file information */
  struct files_struct *files;

/* tss for this task */
  struct thread_struct tss;

  int pid;

  volatile long state;     /* -1 unrunnable, 0 runnable, >0
stopped */

  long priority;

  unsigned short uid,euid,suid,fsuid;

#ifdef __SMP__
  int processor;
#endif

struct task_struct *p_opptr, *p_pptr, *p_cptr,
*p_ysptr, *p_osptr;

/* limits */
  struct rlimit rlim[RLIM_NLIMITS];
  long utime, stime, cutime, cstime, start_time;
```

Java threads

Run in the virtual machine - sometimes with OS support.

```
public class Thread implements Runnable
```

can construct our own by extending this class

```
public class Consumer extends Thread {

  private Buffer buffer;
  private TextArea output;

  public Consumer(TextArea output, Buffer buffer) {
    this.output = output;
    this.buffer = buffer;
  }

  public void run() {
    int data;
    for (int i = 1; i <= 20; i++) {
      data = buffer.retrieve();
      output.appendText("\n" + data);
    }
  }

}
```

The run() method is essential. It is where the new thread starts running after calling start as in:

```
    Consumer consumer = new Consumer(consumerText,
buffer);

    consumer.start();
```

The thread finishes when the run method finishes or the thread is stopped with stop().

All have a priority

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

Priority can be changed with

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

10 is the maximum priority, 1 is the minimum.

Default priority is 5.

Threads of higher priority always run before threads of lower priority.

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

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

The current thread can go to sleep for a while with:

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

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

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

We can make one thread wait for another to finish with join().

```
consumer.join(); // until the consumer thread is finished
```

This also throws an InterruptedException

See the Java API for more methods to do with threads

http://www.cs.auckland.ac.nz/JavaCache/API/api/javab1.htm

Synchonization

It is commonly useful to force threads to wait so that they don't muck up shared objects. The way to do this is with synchronized methods.

```
public class Buffer {
  …
  public Buffer(int size) {
  …
  }
  public synchronized void insert(int data) {
    while (bufferIsFull) {
      try  {wait();}
      catch (InterruptedException e) {
        …
      }
    }
    …
    notify();
  }

  public synchronized int retrieve() {
    while (bufferIsEmpty) {
      try  {wait();}
      catch (InterruptedException e) {
        …
      }
    }
    …
    notify();
    return data;
  }
}
```

The synchronized methods only allow one thread at a time to be running inside this object.

wait()

Stops the running thread. Can only be called inside a synchronized method.

Should always be used inside a while loop.

notify()

Starts up one of the threads which was waiting. Can only be called inside a synchronized method.

notifyAll()

Starts up all of the threads waiting inside this object.

Thread groups?