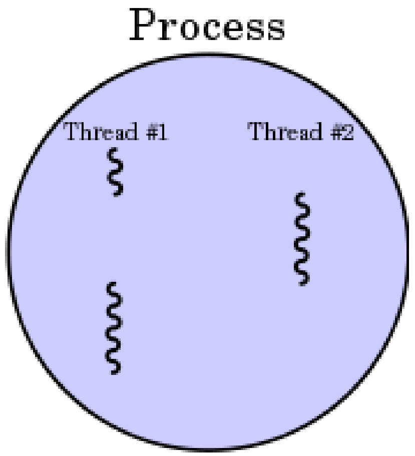
Threads



Process States

- Recall the different Process States
  - New, Ready, Running, Waiting, Terminated

- A process will only perform computations (use the cpu) when it is running on the Processor
  - It needs to be in the **Running State**

- I/O Events (e.g. writing to disk) can reduce the time a process is in the running state, increasing the amount of time it will take to perform all the computations

Process Performance

- So … we would like to maximise the amount of time a process can remain in the running state
  - i.e. so that it will finish more quickly

- As we know, processes perform <u>two</u> main operations
  - Computations
  - I/O Interaction
    - And .. Computations can be performed while I/O is being waited for

Process Performance, continued

- We need a means to perform these 2 main category of operations at the same time

  while a process is in the running state

- To achieve this, need to divide a process into sub (mini) processes (called … threads !)
  - which perform specific operations of the overall program, e.g.
    - Sub Process 1 reads data from the user
    - Sub Process 2 increments a counter / performs large computations

Now… sub process 2 can continue all the time while we wait for the user

Threads

- Threads provide this functionality
  - Threads are mini-processes that operate somewhat independently of other threads in the **same** process
  - Threads share resources allocated to its parent process including primary storage, files and I/O devices, (this is a core difference to 2 processes)
  - A thread, sometimes called a **lightweight process** (LWP), is a basic unit of CPU utilisation.

But ….

- Why do we need threads?
- Why not just use another process?

Threads vs. Many Processes

- Threads DO provide similar approach to computation as using many processes
- However, they
  require fewer Operating System resources
- Many threads can be running
  but OS only **cares about a single process**

This contrasts with the use of many processes which must be moved between the different queues

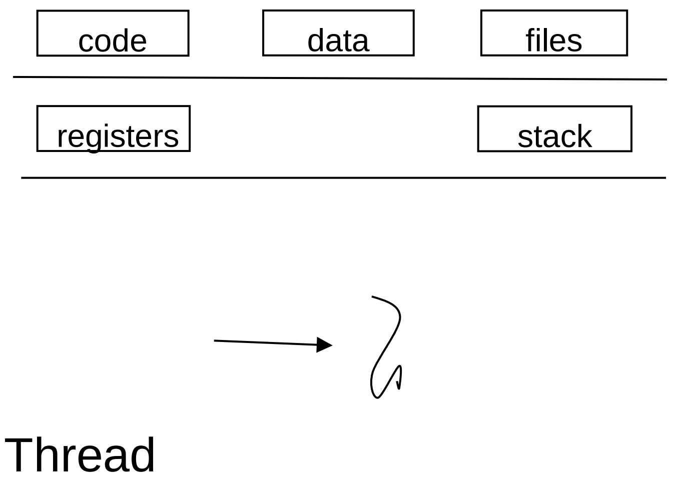
Recall the overhead of a context switch

What do Threads share?

- A thread is a basic unit of CPU utilisation and it comprises of:
  - A thread ID
  - A program counter
  - A register set
  - A stack

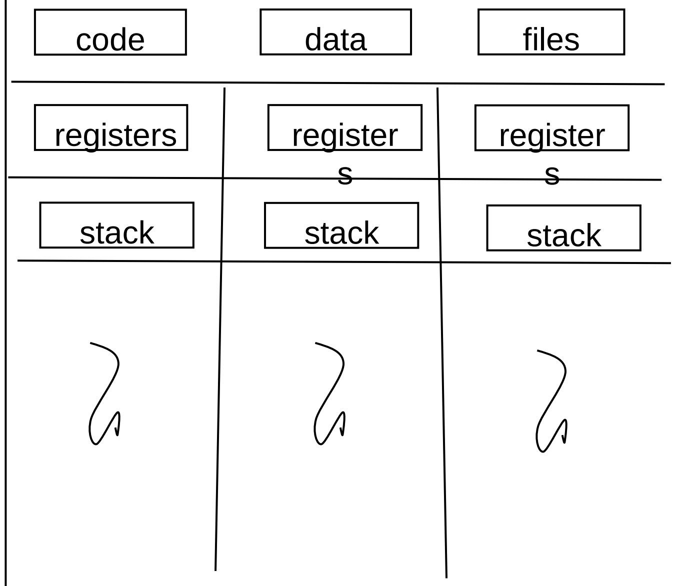
- A thread shares with other threads belonging to the same process:
  - It’s code section
  - It’s data section
  - Other OS resources such as open files, etc

Threads vs. Process



Single-threaded process

A traditional process has a <u>single thread</u> of control.



Multithreaded process

If a process has multiple threads of control, it can perform more than one task at a time

Why Threads?

- Threads can share common data,
  therefore inter-process communication (IPC) is not required.

- Because of their nature, threads can take advantage of multiprocessors.

- They only need a stack and storage for registers so threads are cheap to create.

- Threads use very little resources of an operating system in which they execute.
  - i.e. threads do not need new address space, global data, program code
    or operating system resources.

- Context switching is fast when working with threads.
  - Need only save and/or restore PC, SP and registers set

Advantages of Threads over Processes (1)

- ***Responsiveness:***
  If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

- ***Faster context switch:***
  Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

- ***Effective utilization of multiprocessor system:***
  If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

Advantages of Threads over Processes (2)

- ***Resource sharing:***
  Resources like code, data, and files can be shared among all threads within a process. Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

- ***Communication:***
  Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process

- ***Enhanced throughput of the system:*** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

User-level Vs Kernel-level Threads

User-level threads

All **code** and **data structures** for the library exist in user space.

Invoking a function in the API results in a **local function call** in user **space** and not a system call.

user mode

Kernel-level threads

All **code** and **data structures** for the library exists in **kernel space**.

Invoking a function in the API typically results in a **system call** to the kernel.

kernel mode