Overview

- We have seen that multitasking allows us to concurrently execute multiple processes
- We go one step further and examine how multiple activities or threads of execution within a single process can be supported
- We consider why adding threads to our applications may be a good idea
- We finish by looking at some of the problems that can arise when we move to a multithreaded programming model

Processes and Threads

- Imagine an employee locked in an office working on some problem
- To help her solve the problem she has access to a calculator (the office block has only one calculator which must be shared around) and the office contains a blackboard where she can note temporary results
- However, she is finding solving the problem to be a lot of work and wants some help
- She could ask the occupant of a neighbouring office to help but sharing information and the calculator between the two offices would be cumbersome
- Is there an easier way?

Processes and Threads

- Well a second employee in the same office would help
- The blackboard could be split in two and each person could agree to stick to their half
- Sharing data, coordinating efforts and passing the calculator around would be straightforward since both people are in the same office
- There is only one risk: one of the employees might go berserk and scribble over the work of the other or refuse to cooperate with her

Processes and Threads

- An office is a process address space
- A person is a thread
- The calculator is the CPU
- A blackboard is a stack
- More than one person in the office is a multithreaded process

Processes and Threads

- Transferring the calculator from a person in one office to someone in another office is a context switch between processes
- Transferring the calculator between two people in the same office is a context switch between threads

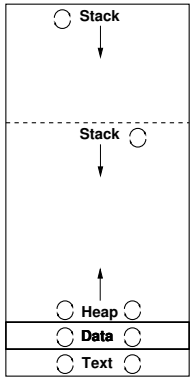
Processes and Threads

- We ended the last lecture by agreeing that processes had their merits but presented us with two problems:
  1. Sharing information between them was difficult
  2. Context switching between processes was a heavyweight operation and wasteful of CPU cycles
- Adding threads to our model has solved these two problems
- Threads give us lightweight concurrency within a process

Processes and Threads

- Like we needed a PCB to represent processes we need a thread control block (TCB) to track threads
- Since threads live inside a process it makes sense that a TCB is smaller than a PCB and switching between threads is intuitively faster than switching between processes
- A TCB tracks what's specific to each thread e.g. CPU register values (including stack pointer) and priority
- The thread lifecycle is identical to the process lifecycle we met earlier

Processes and Threads



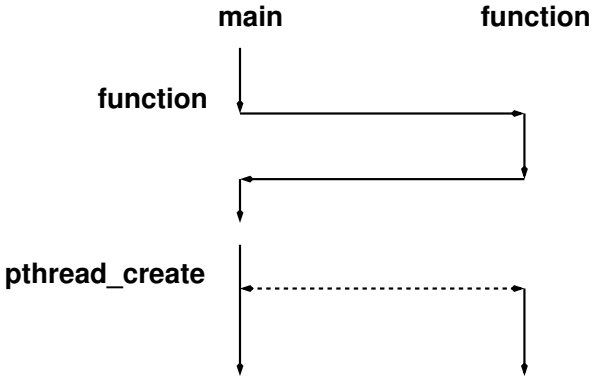
Advantages of Threads

- Lightweight compared to a full process
- Can help decompose a complex task into a set of simpler ones, each handled by a separate thread e.g. web server, MP3 player, database server (any application with multiple ongoing activities)
- If one thread is blocked another can execute allowing parallelism on a multiprocessor machine (if a single threaded process blocks it can make no progress irrespective of the number of cores on the machine)
- All threads have access to the full process address space so it is easy for them to share information and cooperate

Threads Programming in C

- We can use the C `pthread` library to create, manipulate and destroy threads on Linux
- New threads are created with a call to `pthread_create` (the call includes a pointer to the code to be executed as a thread and a pointer to any data the thread is to have access to)
- When a thread has finished executing we need to clean up after it by calling `pthread_join` on it
- Calling `pthread_create` is not the same as making a function call
- The difference can be depicted as follows. . .

Thread Creation vs. Function Call



Concurrency

- We now have two sources of concurrency within our machine:
  1. Multitasking: distinct processes may be context switched
  2. Multithreading: threads within a process may be context switched
- When switches (we concentrate on thread switches) occur cannot be predicted, they can occur in any order and with any interleaving of instructions
- If threads can be scheduled in any order, our multithreaded program must function correctly under all circumstances
- What could go wrong?

Concurrency

Going on holidays

- With one seat left your travel agent is booking your flight at a computer and executes the following code:

```
1. MOV SEATS, EAX; Load seats in EAX
2. DEC EAX;        Decrement if positive
3. MOV EAX, SEATS; Update available seats
```

- What happens if another travel agent tries to book the same flight and their booking thread is switched in after instruction 1 but before instruction 2?
- The update is not indivisible, it is not atomic and can be halted halfway through: one logical operation may be equivalent to many machine instructions

Concurrency

Another example

- What value will `x` have when both threads have finished:

```
Thread A          Thread B
x = 1;            y = 2;
x = y + 1;        y = y * 2;
```

- A correct program will produce the same result for all runs, across every possible scheduling ordering
- Bugs that arise only on particular runs, due to particular scheduling orderings are races or race conditions
- Difficult to detect: space shuttle launches delayed and have caused deaths due to faulty medical software

Concurrency

Mutual exclusion

- We have a number of threads each with a segment of code, called a critical section, that contains instructions that may affect data shared by other threads
- To maintain consistency and data integrity we must ensure only one thread is ever executing in the critical section at any instant: this means enforcing mutual exclusion
- We can implement our own mutual exclusion algorithms in software but it's easier to ask the OS for help
- The OS asks the hardware

Concurrency

Synchronization

- The solution to many problems in operating systems follows a producer-consumer pattern: one thread (the producer) feeds data to another thread (the consumer) through a shared buffer
- While one thread is adding to the buffer the other should not be concurrently removing from it (and vice versa)
- Producer should sleep while no room available
- Consumer should sleep while no data available
- Meeting these requirements involves both enforcing mutual exclusion and synchronizing threads