DM510: Threads and Concurrency

Lars Rohwedder

Disclaimer

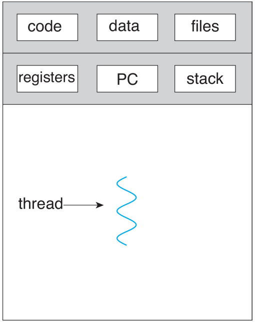
These slides contain (modified) content and media from the official Operating System Concepts slides: https://www.os-book.com/OS10/slide-dir/index.html

Today's lecture

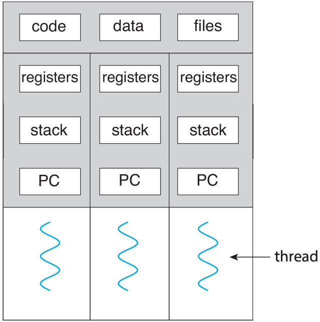
- Chapter 4 of course book

Multi-threaded process

- Last lecture: several processes (running simultaneously) can coorporate on tasks
- Instead of several processes, often several **threads** within one process are used
- Reasons of using several threads/processes: continuing during a blocking call or CPU intensive task (especially for responsiveness), parallelism (multicore processor)



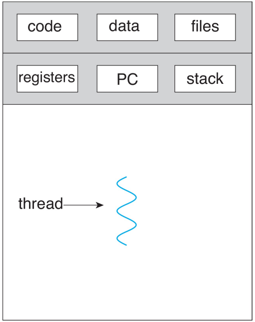
single-threaded process



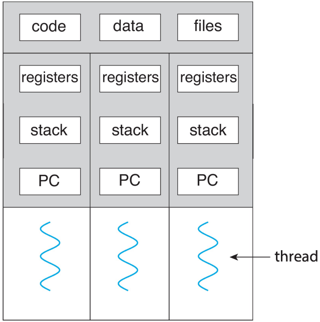
multithreaded process

Multi-threaded process

- Last lecture: several processes (running simultaneously) can coorporate on tasks
- Instead of several processes, often several **threads** within one process are used
- Reasons of using several threads/processes: continuing during a blocking call or CPU intensive task (especially for responsiveness), parallelism (multicore processor)



single-threaded process



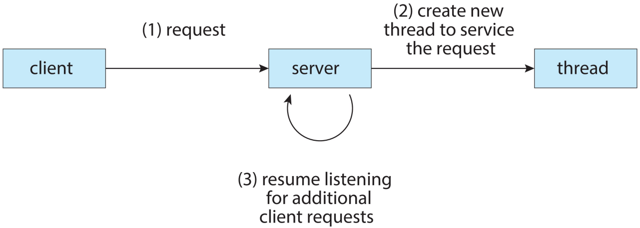
multithreaded process

Advantages of threads

- Lower resource overhead (especially memory)
- Faster thread creation and thread switches
- Communication via shared memory simpler (address space is shared)

Examples

- A web server responding to HTTP requests:



- A database needs to search a large table for a specific entry. It scans the first quarter on one thread (running on CPU core 1), the second quarter on another thread (running on CPU core 2), etc.

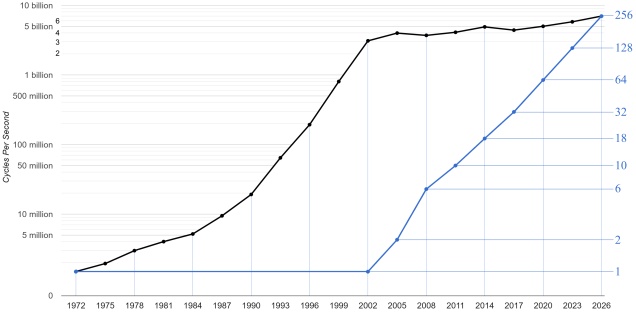
Parallelism

Background on parallelism

- Speed at which processors execute instructions (in million cycles per second: MHz) is no longer increasing due to physical limits
- Performance improvements are now mainly due to **parallelism** (more CPU cores)

Challenges

- Dividing activities
- Balancing
- Data splitting
- Data dependency
- Testing and debugging



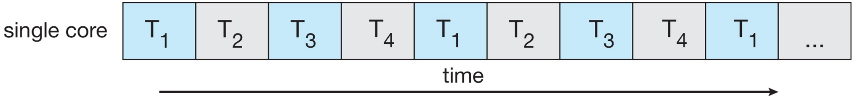
CPU Clock Rate + Core Count (Including "boost clock" and server core count, but no overclocking) (In Log)

Source: https://www.singularity.com/charts/page61.html 1976- 1998 "Microprocessor clockspeed" (modified) | 1999-2026 Manufacturer specifications
Method: Fastest / heighest (not coupled) core count CPU released (1999-2024) 2026 - Leaks + conservative prediction

source: Wikipedia

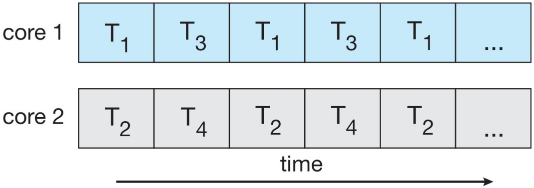
Concurrency and parallelism

Note that **concurrency** (simultaneous progress on different tasks) is also possible on single core (via preemptive scheduler):



single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

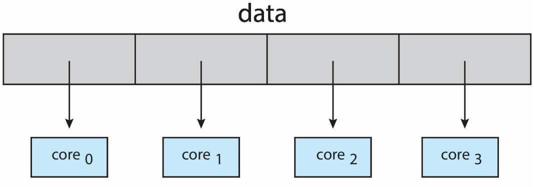
True **parallelism** on several cores:



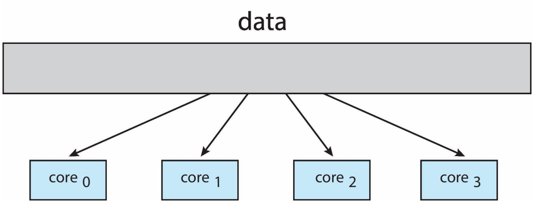
core 1	T_1	T_3	T_1	T_3	T_1	...
core 2	T_2	T_4	T_2	T_4	T_2	...

Forms of parallelism

Data parallelism: distribute data to cores performing the same task on each batch



Task parallelism: distribute different tasks to cores, working on same data



Amdahl's law

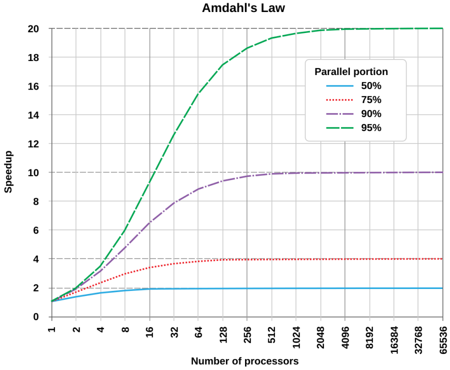
Possible speedup due to parallelism (even with many cores) is bounded. It depends highly on how sequential the program is

Theoretical speedup

Let S be the ratio of sequential operations to all operations. Then

$$\text{speedup} \leq \frac{1}{\mathrm{S} + \frac{1-\mathrm{S}}{\text{cores}}}$$

Optimistic estimate that does not account for communication overhead



Source: Wikipedia

Implementation Details

Types of threads

- Typically, libraries handle internals of thread implementation
- **pthreads** (POSIX threads) is a widespread API specification implemented in many libraries, especially on UNIX systems, see demo/course resources

Types of threads

- Typically, libraries handle internals of thread implementation
- **pthreads** (POSIX threads) is a widespread API specification implemented in many libraries, especially on UNIX systems, see demo/course resources
- Conceptionally, two implementations of threads can be distinguished:

Kernel threads

The kernel creates, schedules, terminates threads like processes, except resources (e.g. address space) are shared. Thus:

- threads may run on different cores
- other threads can continue when one is in a blocking system call

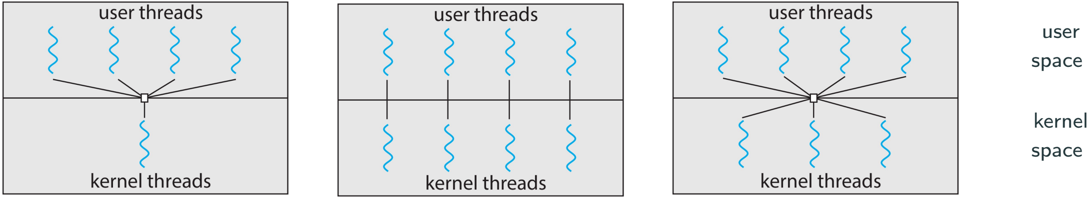
User threads

Threads implemented in user mode. Kernel possibly not aware of threads.

- Limited: process uses single core, blocking call suspends all threads
- Low on resources, applicable also to e.g. embedded systems without OS or with very limited OS

User-kernel thread mapping

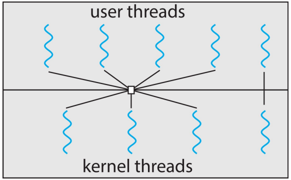
- In principle one could map user threads (of one process) to a smaller number of kernel threads in different ways:



Many-to-one One-to-one Many-to-many

- **one-to-one** model is by far the most popular and default option
- Possible motivation for not using one-to-one is that kernel threads are more resource heavy than user threads and we may not have perfect control over scheduling of kernel threads

Light-weight processes

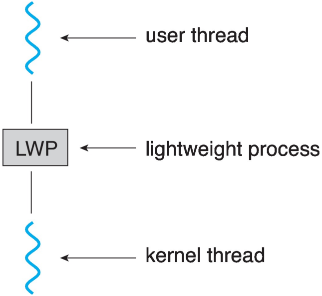


Two-level

- A variant of many-to-many is **two-level** model that allows user process to carefully decide how user threads are allocated to kernel threads
- This provides fine-grained control, but also requires a sophisticated kernel interface, usually achieved by **lightweight processes (LWP)**

Lightweight process

- A LWP is a wrapper for a kernel thread
- threads are allocated to LWPs by user process
- Kernel informs process via **upcall** of: user thread enters blocking call (so that LWP can be used for other user thread) or thread is no longer blocked



Cancelling threads

- We could cancel a threads execution at whatever point it currently is (**asynchronous cancellation**), but this might leave process in an inconsistent state
- Safer alternative: thread specifies points where it can be cancelled (**deferred cancellation**). Then it continues to run until it reaches such a point

Implicit Multi-Threading

Motivation of implicit multi-threading

Optimizing performance and ensuring correctness in multithreaded programs can be difficult:

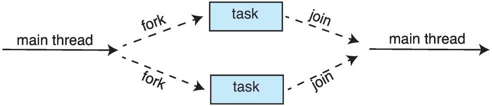
- how many kernel threads should be used? *Depends on CPU architecture*
- How to pass data between different threads? *Requires synchronization, which causes overhead*
- In general, writing optimized code for all architectures may be difficult.

At the same time, many similar programs face these challenges.

There are various libraries and frameworks, which take a computational task and parallelize it almost automatically

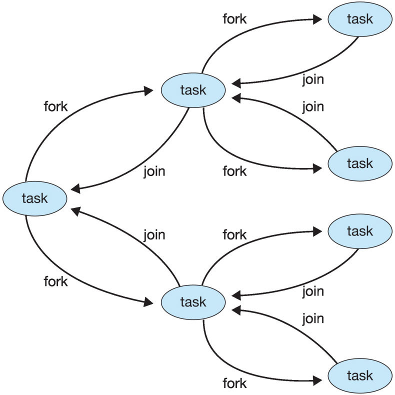
Fork-join model

- Structure: when task is large, split into subtasks, do both in parallel (fork) and wait for results (join)



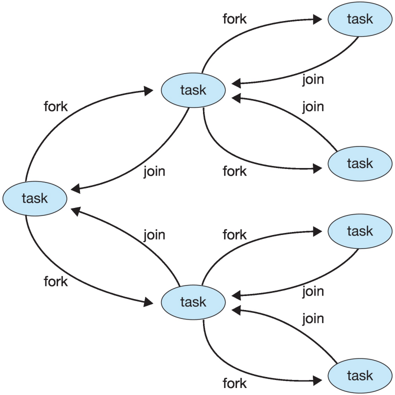
Fork-join model

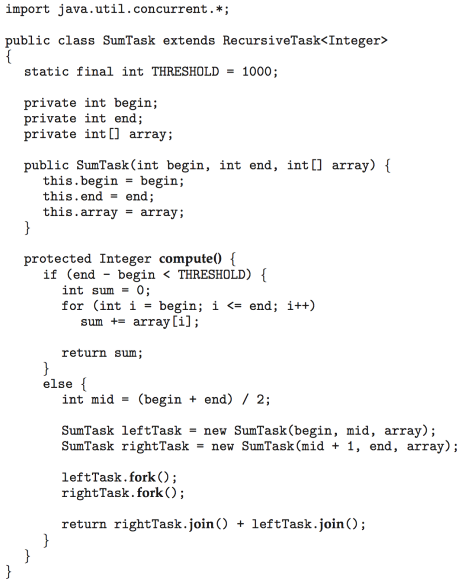
- Structure: when task is large, split into subtasks, do both in parallel (fork) and wait for results (join)
- Same can be done recursively



Fork-join model

- Structure: when task is large, split into subtasks, do both in parallel (fork) and wait for results (join)
- Same can be done recursively





```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

OpenMP

- Library for parallelizing C/C++ programs
- Parallelization of for-loops requires only minimal changes: adding "`#pragma omp parallel for`" before the loop
- See demo/course resources

Other parallelization

Single-instruction-multiple-data (SIMD)

- Modern processors come with special registers that can hold vectors, e.g. 4 integers (of 4 bytes each)
- SIMD instructions can perform parallel operations on vectors.
- They perform data-parallelism
- **Note:** this is not threading or an operating system aspect

General-purpose graphics processors (GPGPU)

- Graphic processors come with very many, sometimes 1000s, of (very simple) cores
- These traditionally perform computer graphics tasks, but are being used increasingly for other purposes (e.g. linear algebra) as well
- Accessed as a device via driver
- Not easy to program