ECE 3574: Introduction to Concurrency: Processes and Threads

Changwoo Min

Introduction to Concurrency: Processes and Threads

- Today we are going to introduce the notion of concurrency.
  - Why concurrent/parallel programming is important?
  - Operating Systems and Concurrency
  - Examples

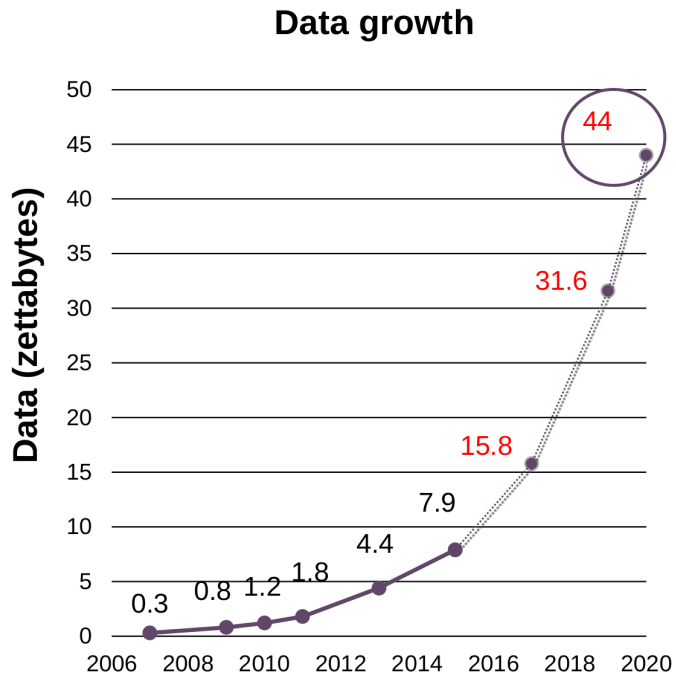
Milestone 2: FAQ

```
#include <iostream>
#include <string>
using namespace std;

int main(void)
{
    string str = "3735928559"; // 0xdeadbeef
    unsigned int num = stol(str);
    unsigned char *p = (unsigned char *)&num;

    printf("num = %x\n", num);
    for (auto i = 0; i < sizeof(num); ++i) {
        printf("p[%d] = %x\n", i, p[i]);
    }
    printf("*((unsigned int*)p)= %x\n",
           *((unsigned int*)p));
    printf("*((unsigned short*)(p+2))= %x\n",
           *((unsigned short*)(p+2)));
    return 0;
}
```

Data growth is already exponential

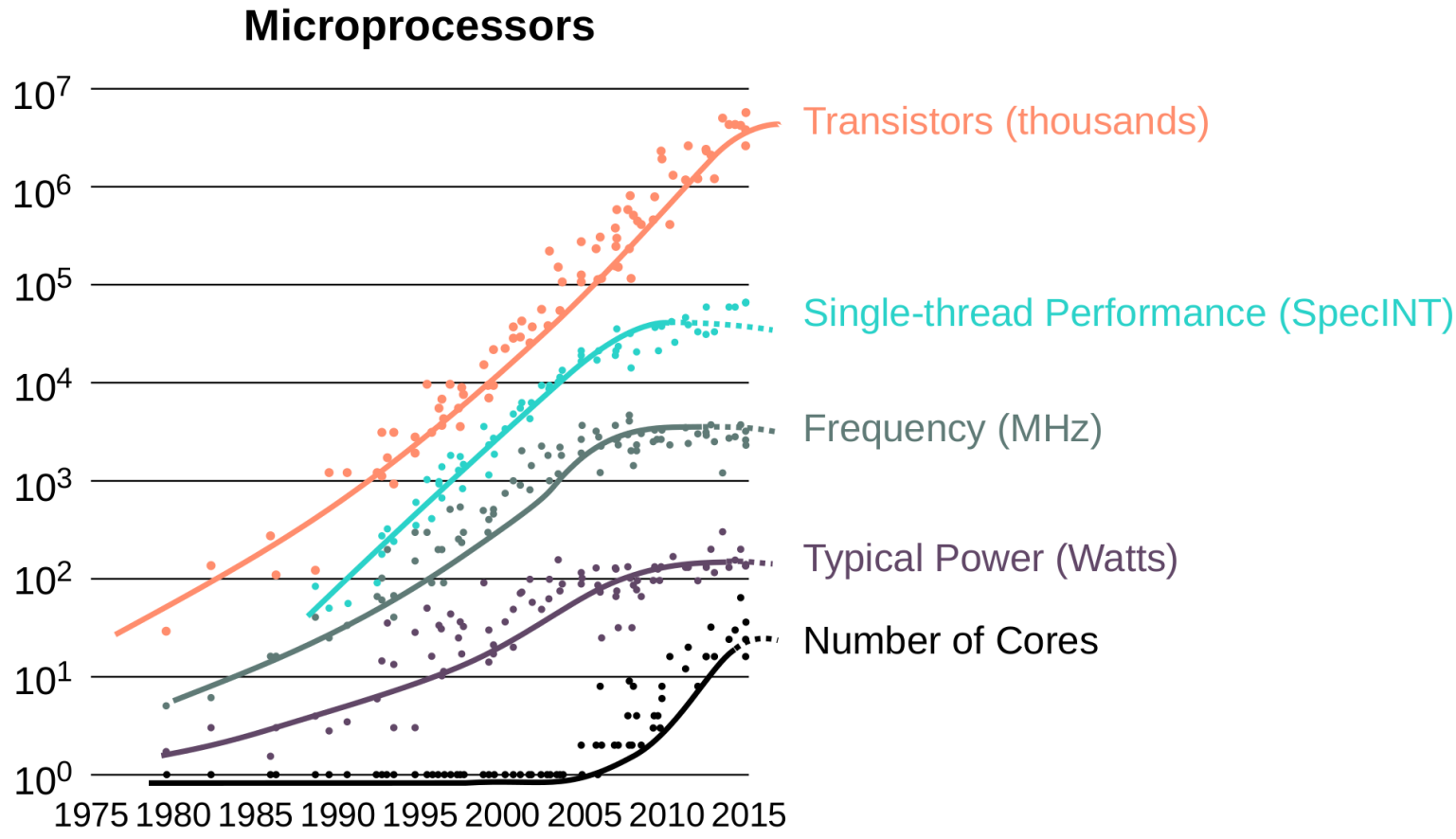


- 1 zettabytes = 10^9 terabytes

Data growth is already exponential

- **Data nearly doubles every two years (2013-20)**
- By 2020
  - 8 billion people
  - 20 billion connected devices
  - 100 billion infrastructure devices
- Need more processing power

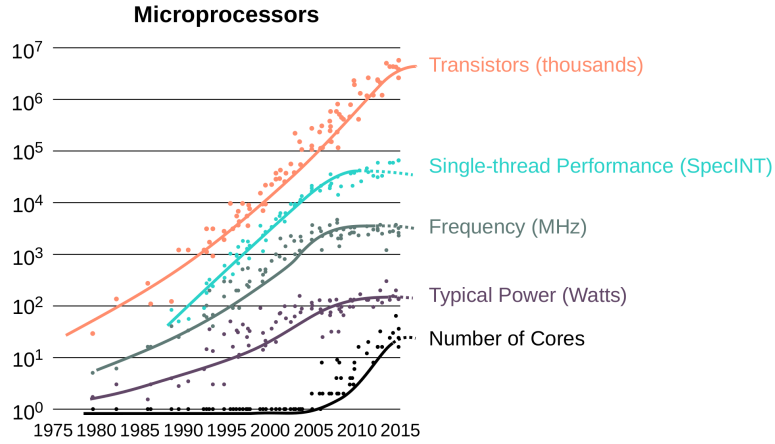
Single-core performance scaling stopped



Single-core performance scaling stopped

- **Increasing clock frequency is not possible anymore**
  - Power consumption: higher frequency → higher power consumption
  - Wire delay: range of a wire in one clock cycle
- **Limitation in Instruction Level Parallelism (ILP)**
  - 1980s: more transistors → superscalar → pipeline
    - 10 CPI (cycles per instruction) → 1 CPI
  - 1990s: multi-way issue, out-of-order issue, branch prediction
    - 1 CPI → 0.5 CPI

The new normal: multi-core processors



The new normal: multi-core processors

- **Moore's law**: the observation that the number of transistors in a dense integrated circuit doubles approximately every two years
- **Q: Where to use such a doubled transistors in processor design?**
- **~ 2007: make a single-core processor faster**
  - deeper processor pipeline, branch prediction, out-of-order execution, etc.
- **2007 ~: increase the number of cores in a chip**
  - multi-core processor

The new normal: multi-core processors

Example: Intel Xeon 8180M processor

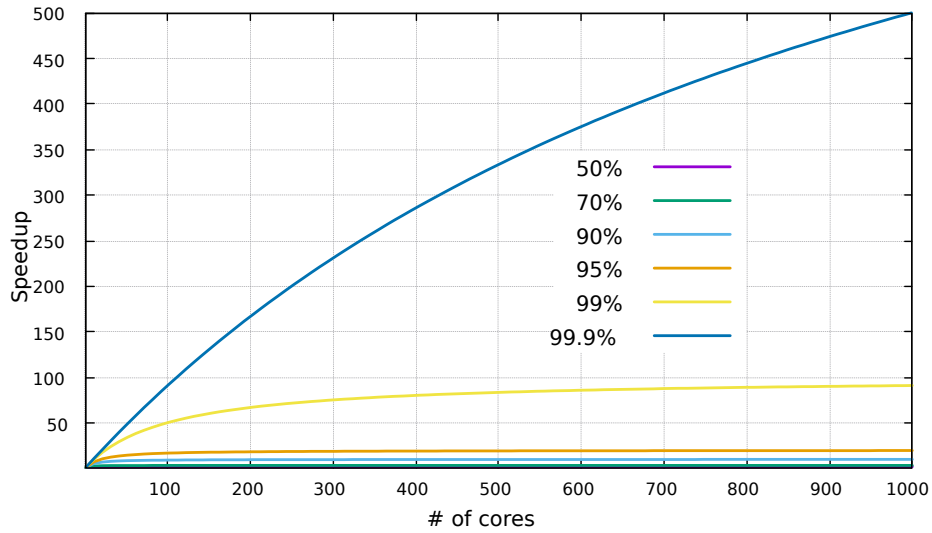
Essentials	
Product Collection	Intel® Xeon® Scalable Processors
Code Name	Products formerly Skylake
Vertical Segment	Server
Processor Number	8180M
Status	Launched
Launch Date	Q3'17
Lithography	14 nm
Recommended Customer Price	$13011.00

Performance	
# of Cores	28
# of Threads	56
Processor Base Frequency	2.50 GHz
Max Turbo Frequency	3.80 GHz
Cache	38.5 MB L3
# of UPI Links	3
TDP	205 W

- Support up to 8 sockets: 28*8 = 224 cores (or 448 H/W threads)

Small sequential part does matter

- **Amdhal's Law: theoretical speedup of the execution of a task**
  - Speedup = 1 / (1 - p + p/n)
  - `p` : parallel portion of a task, `n` : the number of CPU core



Where are such sequential parts?

- Applications: sequential algorithm
- Libraries: memory allocator (buddy structure)
- Operating system kernel
  - Memory managment: VMA (virtual memory area)
  - File system: file descriptor table, journaling
  - Network stack: receive queue
  - **Your application may not scale even if its design and implementation is scalable**

Concurrent Programming

- Concurrency extends the sequential programming model to include multiple executing computations or processes.
  - `Do A then B then if C Do D Else Do E`
  - `----------------------------------> time`
- A concurrent model is a much more realistic model of the world.
- Concurrency is a way to structure software, particularly as a way to write clean code that interacts well with the real world. – Rob Pike

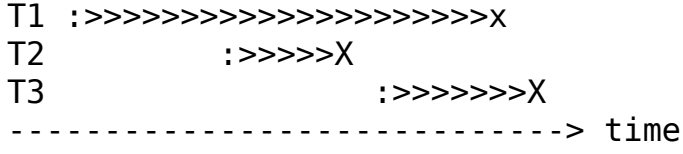
Examples

- Your operating system (even on a single CPU)
- A web server responding to multiple requests.
- A graphical program responding to user input
- Numerical simulations
- Robot making multiple complex movements
- Control system in a car

Examples

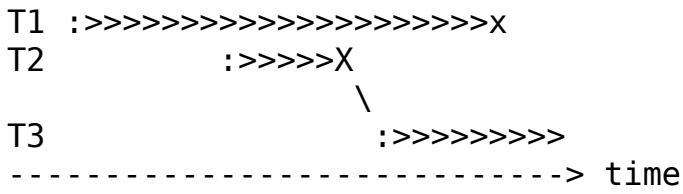
- Note: *concurrent* (virtual time) does not (necessarily) imply *parallelism* (real time)
- On a single CPU a program may be concurrent, but cannot be parallel.
- A concurrent program *might* take advantage of multiple CPUs though.

Concurrency is essentially an abstraction of time, or an ordering of events



Concurrency is essentially an abstraction of time, or an ordering of events

- These events may be linked so there is a dependency
- The input to T3 depends on the output of T2.



Two views of concurrency

- Engineering view: how do you implement concurrency
- CS Theory view: how do you think and reason about concurrent programs

Basic of Operating Systems: The process

- A process is an abstraction by operating systems that allow it to *virtualize* a CPU.
- This allows more than one program to *run*, that is appearing to *execute*, even with a single processor, a.k.a *multi-tasking*.
- Thus, OS's were the first place concurrent programming was encountered.

(greatly) simplified execution on bare hardware (no OS)

- After power-on set PC to a specific address
- starts the fetch-decode-execute cycle from there
- continues until a halt instruction is executed, or the power is cycled.

An operating system is a program that creates a virtual representation (abstraction) of both hardware and time.

- The OS is the code that starts running at power-up (or shortly thereafter in the case of BIOS) and
  - Virtualizes CPU
  - Virtualizes Memory
  - Virtualizes IO Devices
  - Virtualizes Time

The core of the operating system is the *kernel*

- The kernel executes for a while, then might allow some non-kernel code (a program) to run by entering it's main function. When main returns the kernel picks up and keeps on going until another program starts. This is how I started programming (Sinclair,Commodore, Apple II)

The core of the operating system is the *kernel*

- Of course this requires the programs be really short, so instead programs were written to give up control periodically, or **yield**. The OS then put these programs in different blocks of virtual memory, execute one until it yielded, then execute another until it yielded and so on. This is called **cooperative multi-tasking**. The earliest operating systems worked this way: DOS, Windows before Windows95, Mac OS <- 9,

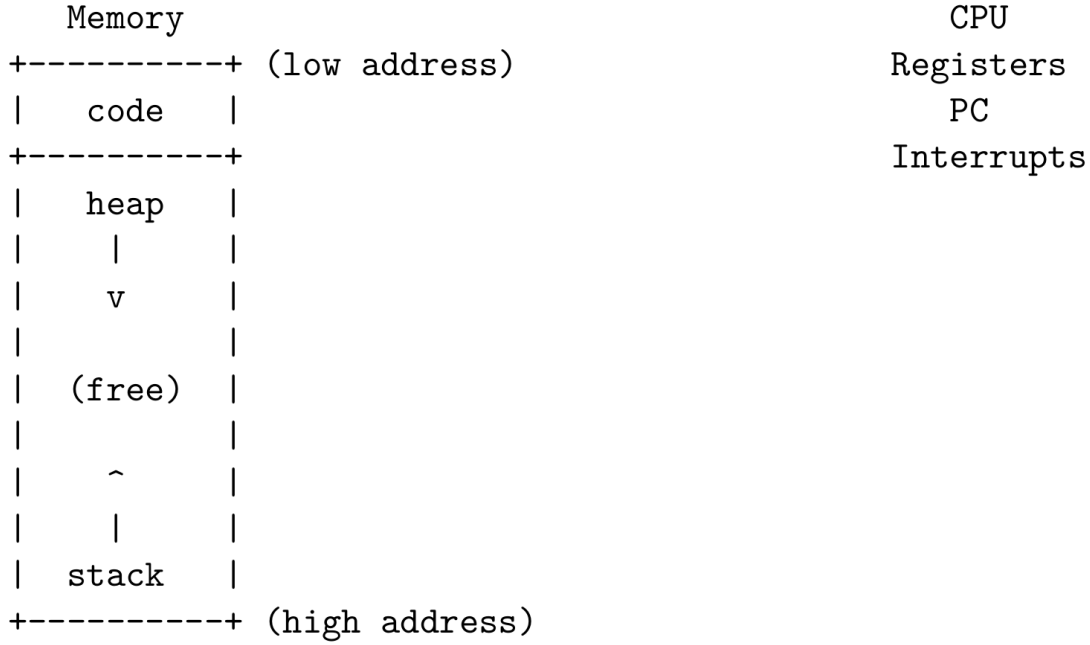
Preemptive Multi-tasking

- Cooperative multi-tasking required cooperation among **all** programs running. One bad program failing to yield could lock up a computer, requiring a power-cycle.
- So kernels evolved to enable them to interrupt another program in order to either execute itself, or to execute another program, possibly the one previously interrupted. This happened transparently to the program.

Preemptive Multi-tasking

- This is called **preemptive multi-tasking** and is the dominant form of operating systems.
- The interruption and change of executing code is called a **context switch**.
- Note this removes the requirement of cooperation from individual programs but places increased responsibility on the OS to share time fairly.

The abstraction of a running program in an OS is a Process

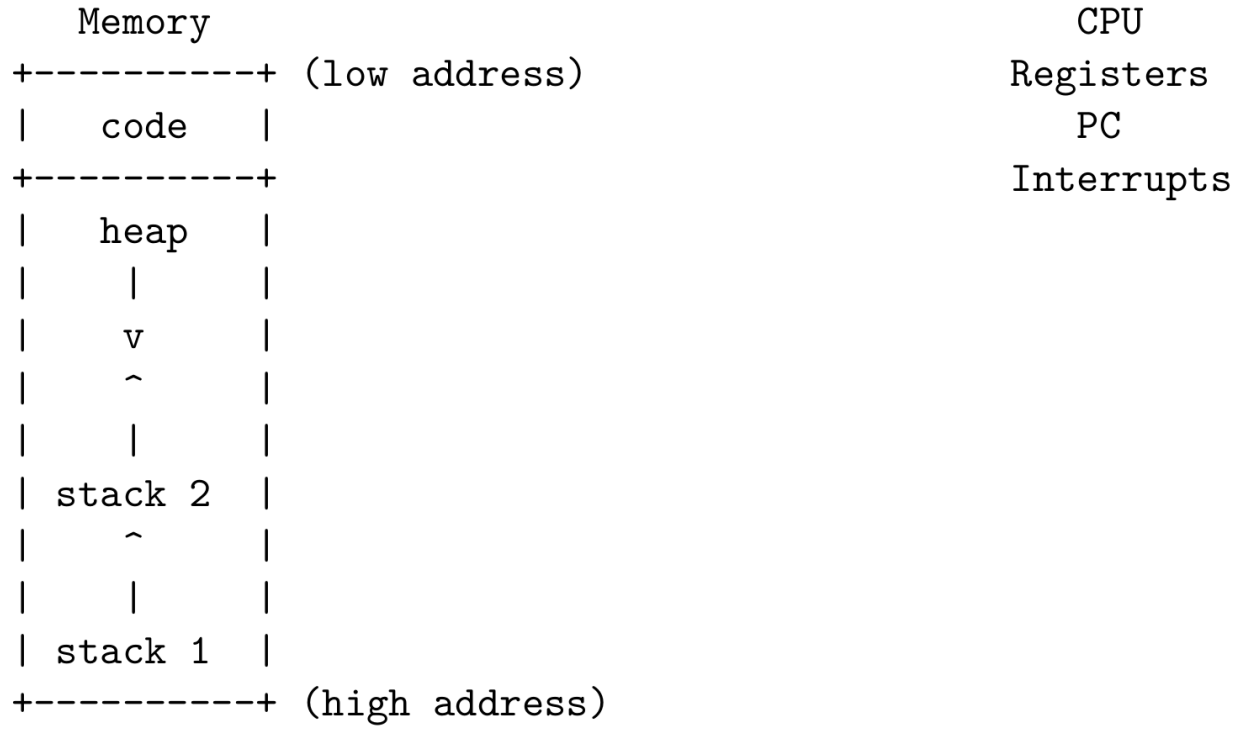


- Each process has its own memory region.

Processes are relatively expensive to create, the solution?

- *threads*, sometimes called light-weight processes
  - multiple threads of execution that have separate stacks, but share the heap
  - much faster to start and stop threads because it just is a re-partitioning of the existing process memory
- Side Note: the kernel may also be threaded, so there is (sometimes) a distinction between kernel threads and user threads.

The process abstraction changes to (two threads)



The OS schedules both process and threads

- The main kernel loop (in essence)

```
do{
    // choose a new process or thread to run
    // save running process state (start context switch)
    // set a timer to interrupt (e.g. 8254 or HPET)
    // load new process state (end context switch)
    // it runs until timer interrupts
    // enters OS code and loops
} while(true);
```

Process versus thread

- The essential difference between a process and a thread is how they communicate.
- Process may communicate using
  - Pipes / Sockets
  - Message Queues
  - Shared Memory
- This is OS specific, but see `boost::interprocess`

Process versus thread

- Threads communicate using a shared heap, over which message queues and other forms of communication can be written. As of C++11 this is now standardized, but it used to be OS specific (win-threads versus pthreads).

An important aspect of concurrency at the OS level is the scheduler.

- The scheduler is the part of the kernel that decides who runs during a given time-slice.
  - Based on a priority system
  - Based on what a process is doing
  - Schedules both processes and threads within them.

Next Actions and Reminders

- Read about Qt inter-process communication

Milestone 2: FAQ

```
#include <iostream>

#include <string>

using namespace std;



int main(void)

{

    string str = "3735928559"; // 0xdeadbeef

    unsigned int num = stol(str);

    unsigned char *p = (unsigned char *)&num;



    printf("num = %x\n", num);
```

```
    for (auto i = 0; i < sizeof(num); ++i) {
        printf("p[%d] = %x\n", i, p[i]);
    }
    printf("*((unsigned int*)p)= %x\n",
           *((unsigned int*)p));
    printf("*((unsigned short*)(p+2))= %x\n",
           *((unsigned short*)(p+2)));
    return 0;
}
```