CS 1217

Lecture 8 – Context Switch, CPU Multiplexing

Logistics

- Lab 1 will be released today; latest tomorrow
- **No** extensions : Use your slack time wisely
- Attend the Tuesday Lab session: will help you get started with Lab 1
    - Reminder: Attendance is mandatory

Recap

- Hardware Interrupts
- Software Interrupts
- Software Exceptions

- Mechanisms to handle all of the above remain the same

- Interrupt Handlers, Interrupt Service Routines

Making syscalls

- To access the kernel system call interface an application:
  - arranges arguments to the system call in an agreed-on place where the kernel can find them, typically in registers or on its stack
  - loads a number identifying the system call it wants the kernel to perform into a pre-determined register, and
  - executes the `syscall` in MIPS (`int` in x86) instruction

- `libc` provides the wrappers around the `int` instruction that programmers are familiar with

Software Interrupts

- Applications need a mechanism for “asking” the operating system to carry out privileged operations on its behalf
- CPU ISAs provide a **special instruction** (`syscall` on MIPS, `int` on x86) that generates a software (or synthetic) interrupt.
- Rest of the interrupt handling path is unchanged

Software Interrupts vs. Software Exceptions

- **Interrupts** are **voluntary**.
  - For example, a process wants to read some data from disk that it needs to make forward progress

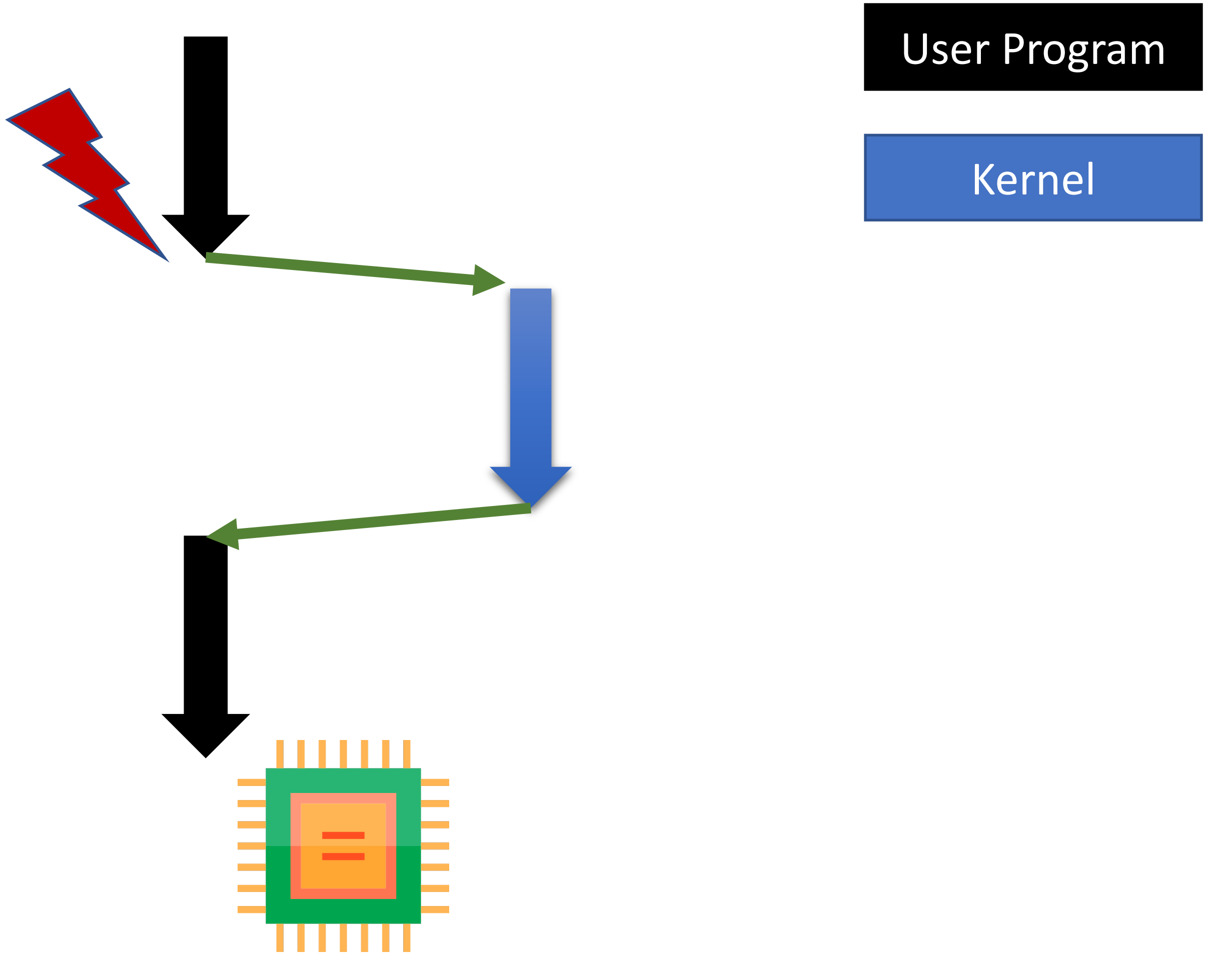
- **Exceptions** are **non-voluntary**.
  - Buggy code was written that has now resulted in a divide by zero exception; the process didn’t request help, but has to now be terminated

Software Exceptions

- A **software exception** indicates that code running on the CPU has created a situation that the processor needs help to address.

- Examples?
  - Attempt to use a privileged instruction—also probably kills the process.
  - Divide by zero—probably kills the process.
  - Attempt to use a virtual address that the CPU does not know how to translate

Execution Timelines



Interrupts Masking

- Hardware interrupts can be **asynchronous** or **synchronous**.

- **Asynchronous** interrupts can be masked
  - The processor provides an interrupt mask allowing the operating system to choose which interrupts will trigger the ISR.
  - If an interrupt is masked, it will not trigger the ISR.
  - If an interrupt is still asserted when it is unmasked, it will trigger the ISR at that point
- Some interrupts are **synchronous** and cannot be masked:
  - typically indicate very serious conditions which must be handled immediately.

Usefulness of Masking Interrupts

- Priority management

- What if servicing one interrupt causes more interrupts to be generated?

Interrupt Handlers

- Who ”manages” the interrupt handlers?

- What would happen if applications were allowed to modify the interrupt service routines?

- Interrupt handlers allow the operating system to **control access** to hardware devices and **protect them** from direct control by untrusted applications
- The memory that contains interrupt handlers is protected from access by user applications.

Steps for Interrupt Handling

- The CPU
  - Enters privileged mode
  - saves register state carefully
  - Jumps to a pre-determined location and memory and starts executing instructions (or the **handler**)

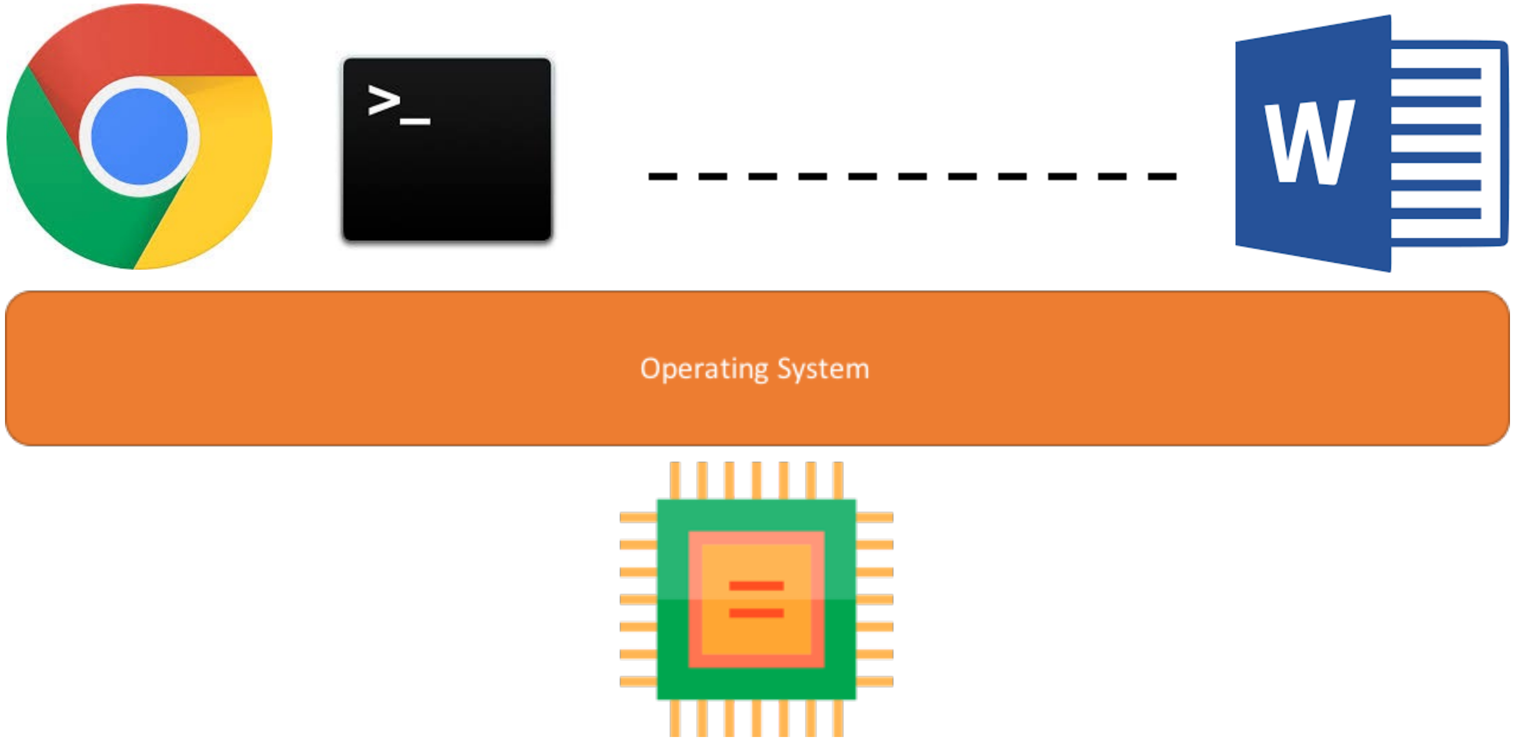
- The interrupt handler has now done its task : How do you get back to user (unprivileged) mode?

Aside: `exit()` vs `_exit()`

- `libc` wrappers preclude the system call, system call is implementation is in a separate ”function” (handler)

CPU Multiplexing

- The CPU executes many processes **concurrently**.
- Assuming there is only one CPU, what does **concurrently** mean?



You vs. the CPU

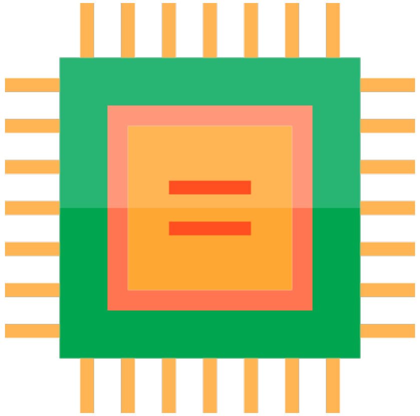
- Typical CPU frequency : 2.5 Ghz
  - How many µs per clock cycle?
  - 400

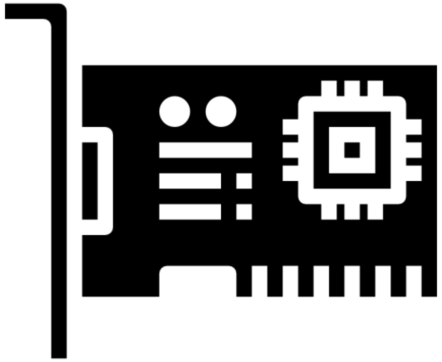
- Assume a typical integer instruction can be completed in 5 cycles: 2000 µs

- How long does it take for you to press a key?
  - Assume 2 seconds for fun

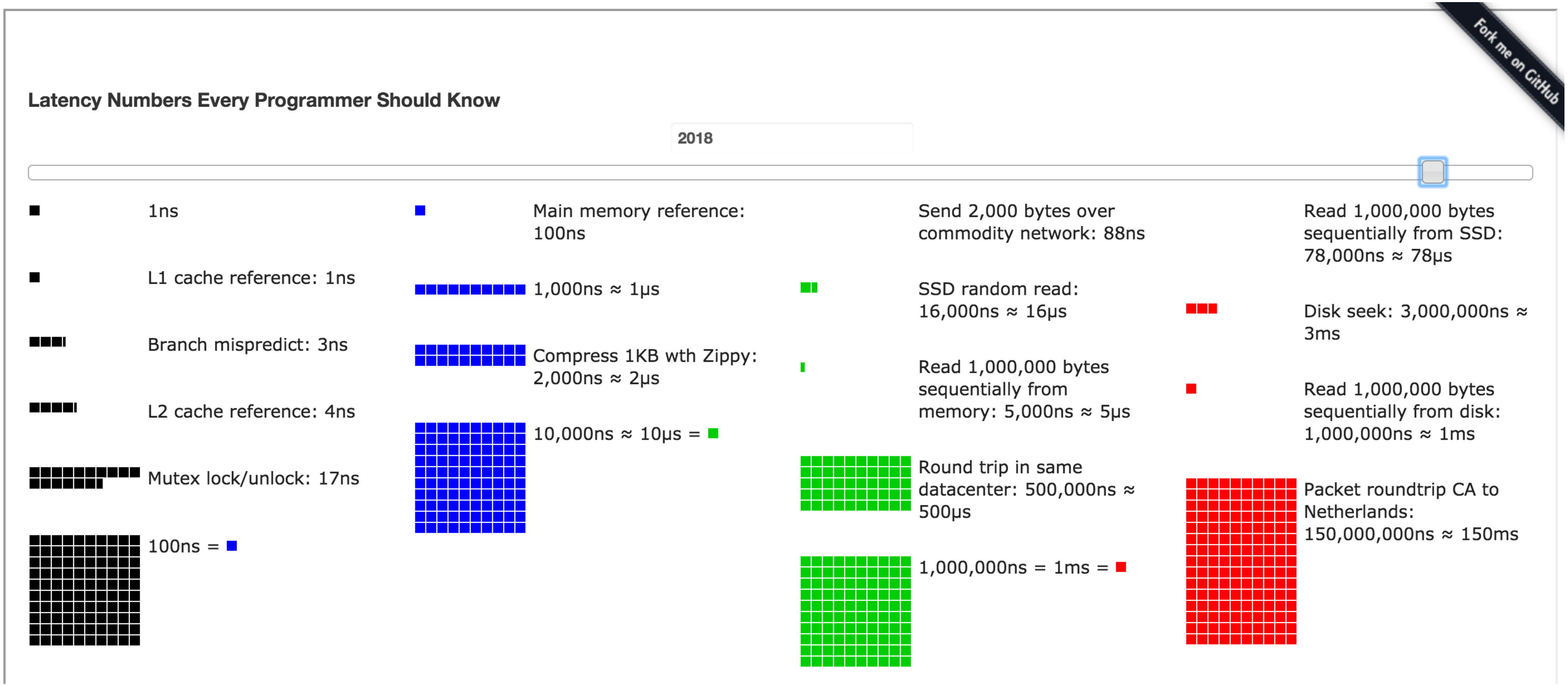
- How many integer additions could have been done in the time it takes for you to press a key?

The CPU vs the rest of the System



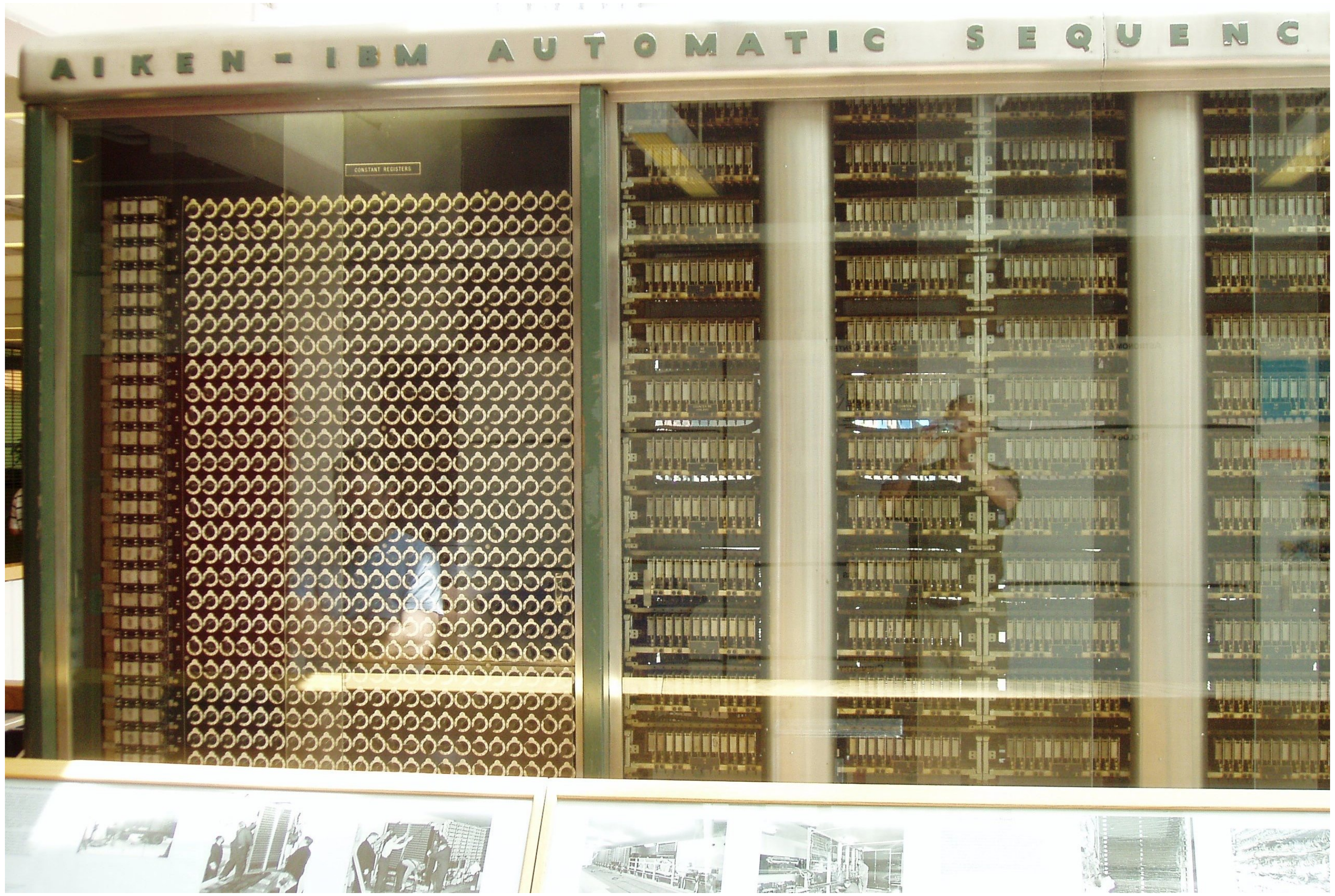


Latency #s Every Programmer Should Know

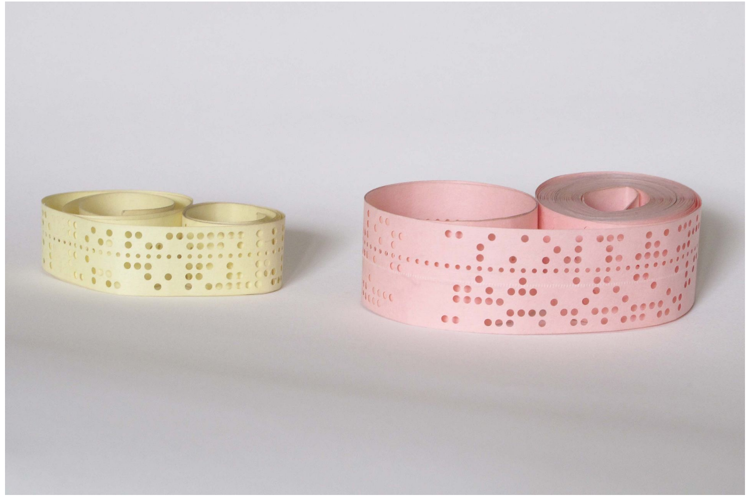


https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html

A lesson in history



- Harvard Mark I
- WW2 era general purpose electromechanical computer
- Could run only one program at a time; had **complete** access to the entire machine



Scheduling

- No interaction with users
- Scheduling the humans, not the machine

Time scale : Days