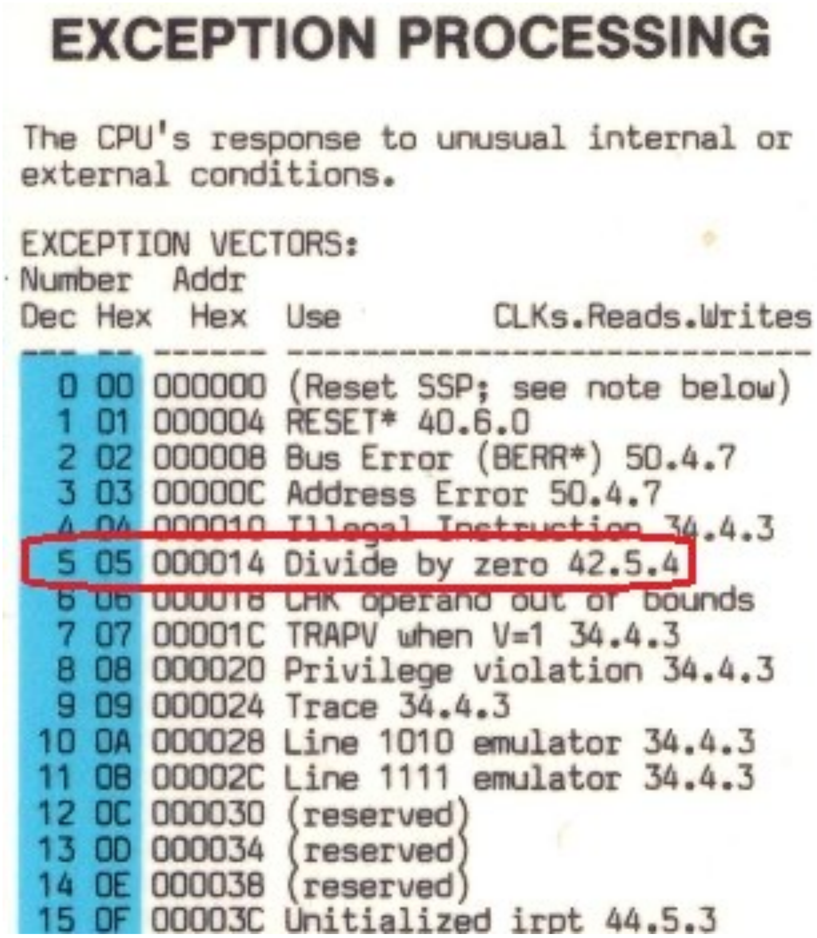
# CS 1217

Lecture 9 – Context Switch wrap; Process States, Intro to Scheduling

# Software exceptions

- Software exception types
  - **Faults**: These can be corrected, and the program may continue as if nothing happened.
  - **Aborts**: Some severe unrecoverable error; leads to code dumps/stack traces
- Divide instruction implementation raises exception if denominator = 0



EXCEPTION PROCESSING

The CPU's response to unusual internal or external conditions.

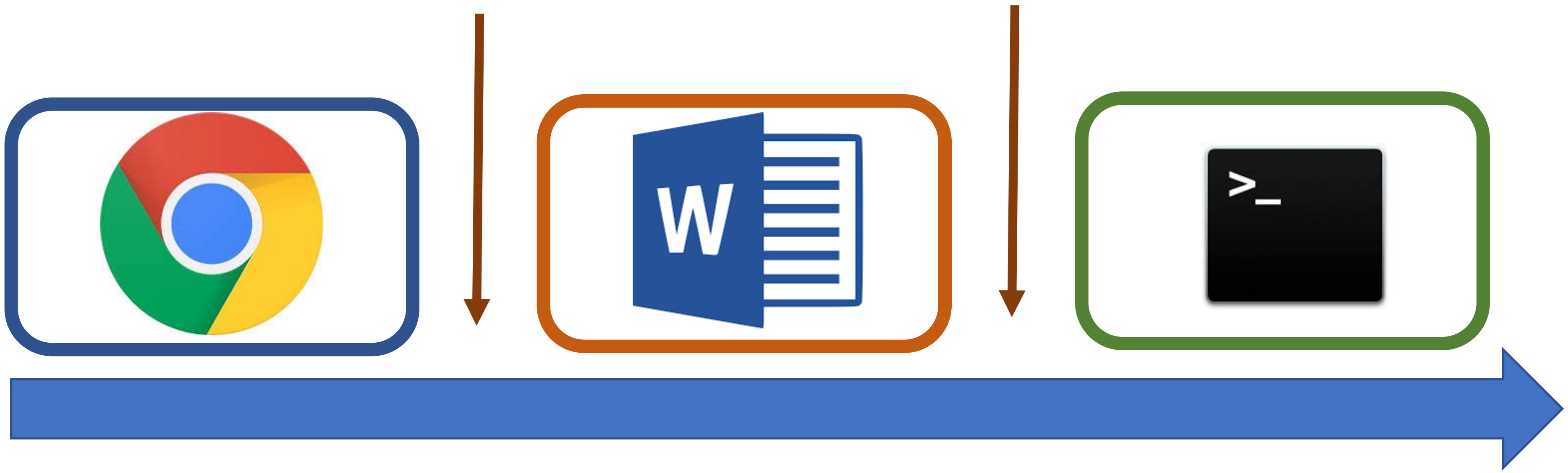
EXCEPTION VECTORS:

| Number Dec | Hex | Addr Hex | Use CLKs.Reads.Writes |
|---|---|---|---|
| 0 | 00 | 000000 | (Reset SSP; see note below) |
| 1 | 01 | 000004 | RESET* 40.6.0 |
| 2 | 02 | 000008 | Bus Error (BERR*) 50.4.7 |
| 3 | 03 | 00000C | Address Error 50.4.7 |
| 4 | 04 | 000010 | Illegal Instruction 34.4.3 |
| 5 | 05 | 000014 | Divide by zero 42.5.4 |
| 6 | 06 | 000018 | CHK operand out of bounds |
| 7 | 07 | 00001C | TRAPV when V=1 34.4.3 |
| 8 | 08 | 000020 | Privilege violation 34.4.3 |
| 9 | 09 | 000024 | Trace 34.4.3 |
| 10 | 0A | 000028 | Line 1010 emulator 34.4.3 |
| 11 | 0B | 00002C | Line 1111 emulator 34.4.3 |
| 12 | 0C | 000030 | (reserved) |
| 13 | 0D | 000034 | (reserved) |
| 14 | 0E | 000038 | (reserved) |
| 15 | 0F | 00003C | Unitialized irpt 44.5.3 |

Motorola 68000

# Batch Processing

- As time progressed: More users; multiple, interactive jobs
- Form a queue, send another job to execute when one finishes



# Problems with Batch Computing?

- **Inefficiency!**
  - Usage of slower parts of the system will cause the CPU to stall waiting for them to finish
- CPUs are very fast, everything else in the system is comparatively slower
  - How do you increase CPU utilization
- How do you support multiple, concurrent, interactive users?

# Supporting Multiple Users

Time scale : seconds

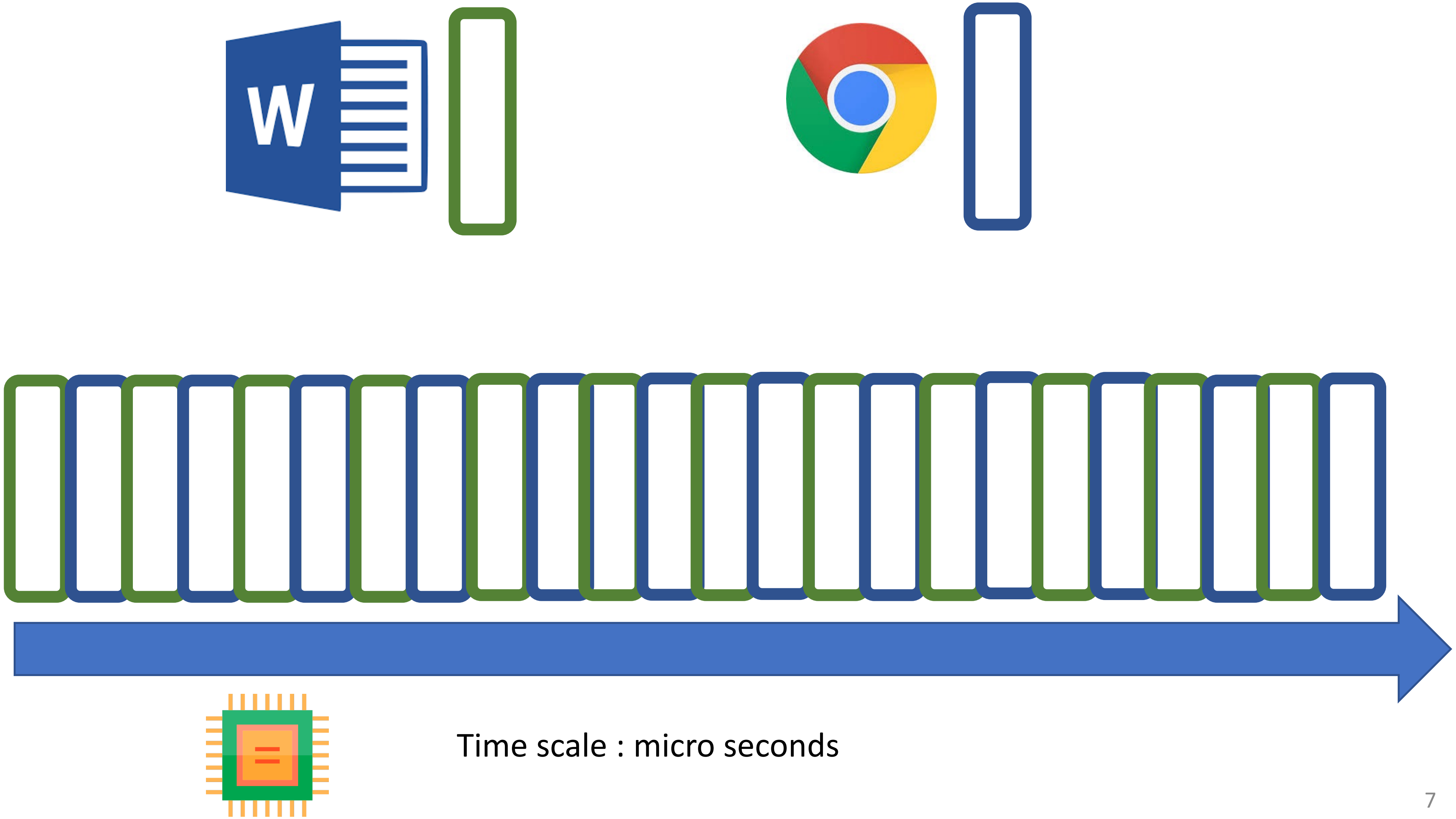
# Co-Operative Scheduling

- While a program is doing something slow (e.g. reading the disk), the kernel can find some other process to run, while this process is waiting for the operation to complete.

- What does this depend on?

- What can do wrong here?

# Context Switching



# Why does this work?

- Human perception has some limits
  - 15 ms (or so)

# Implementing Context Switches

- Who has to implement the switching between processes/threads?

- How does the OS get to run?

- What if none of the processes do any of those things?

# Timer Interrupts

- **Timer interrupts** generated by a timer device ensure that the operating system regains control of the system at regular intervals
- Form the basis of **pre-emptive scheduling**: the OS doesn’t have to wait for a process to let it run; can take control and preempt it
- Timer interrupts are handled the same way as other interrupts

# Saving Thread State

- A thread of the process is executing at any given time
- Threads can be pre-empted : when they restart, should have the exact state when they were pre-empted
  - It should appear to the thread that nothing has happened, the execution was not interrupted

- What does the thread state consist of?
  - Registers
  - Address space

- Who saves the state?

# Trap Frame

- Saving thread state is the **first thing** that happens when the interrupt service routine is triggered (**Why**?)

- Saved state is also known as a **trap frame**

- The kernel switches to a different stack – why?

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort gs;
  ushort padding1;
  ushort fs;
  ushort padding2;
  ushort es;
  ushort padding3;
  ushort ds;
  ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding6;
};
```

xv6 trap frame

# xv6 Context Switch

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save the current registers on the stack, creating
# a struct context, and save its address in *old.
# Switch stacks to new and pop previously-saved registers.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

# Context Switch Frequency

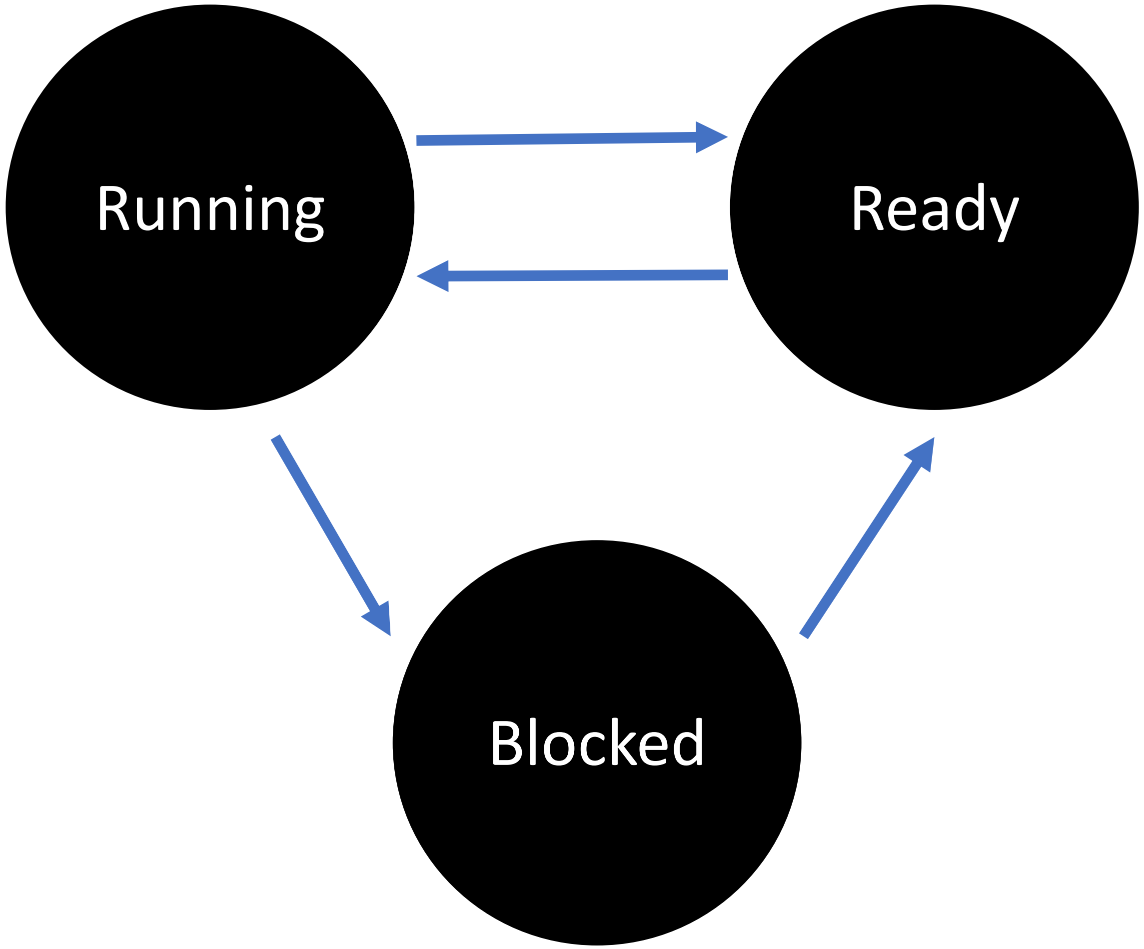
- Context switches have associated overhead, both in ***space*** and in ***time***
  - Why?

- How does that affect the rate at which context switches can happen?

# Process States

- Processes can be thought about as being in different “states”
- **Running** : Currently executing instructions on the CPU
- **Ready** : Not currently executing, but can; is capable of getting restarted
- **Waiting/Blocked/Sleeping**: not executing instructions and not able to be restarted until ***some*** event occurs

# Process States



# Scheduling

- **Scheduling** is the process of choosing the next process (or processes) to run on the CPU (or CPUs).

- Why is scheduling required?

# When to Schedule Processes?

- When a process **voluntarily** gives up the CPU by calling `yield()`
- When a process makes a blocking **system call** and must sleep until the call completes.
- When a thread **exits**
- When the **kernel decides** that a thread has run for long enough

- Which one is co-operative? Which one is pre-emptive?

# Quick note on yield()

- `yield()` can be a useful way of allowing a ***well-behaved*** thread to tell the CPU that it has no more useful work to do.