Processes & ECF

Science Computer Science

CS 351: Systems Programming
Melanie Cornelius

Slides and course content obtained with permission
from Prof. Michael Lee, <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

Reminders

1. Look for emails from Fourier
2. Look for emails from IIT VPN
3. Look for emails from GitHub
4. Solutions to Quiz 1 posted!
5. **Read CH 8 in CS:APP**

Agenda

- Definition & OS responsibilities
- Exceptional control flow
  - synch vs. asynch exceptions
  - exception handling procedure

- Definition & OS responsibilities

a **process** is a *program in execution* **-** it is the foundational unit of computation

- **programs describe** what we want done,
- **processes carry out** what we want done

a process has:

- code (program)
- runtime data (global, local, dynamic)
- PC, SP, FP & other registers

a process also:

- **exists on some subset** of system hardware
- can **communicate** with other processes
- can **use** other static materials (files, etc)

```
main() {
  fnA();
}

fnA() {
  fnB();
}

fnB() {
  loop {

  }
}
```

essential to program execution is *predictable, logical control flow*

which requires that nothing disrupt the program mid-execution easiest way to guarantee this is for a process to "own" the CPU for its entire duration (i.e., no-one else allowed to run)

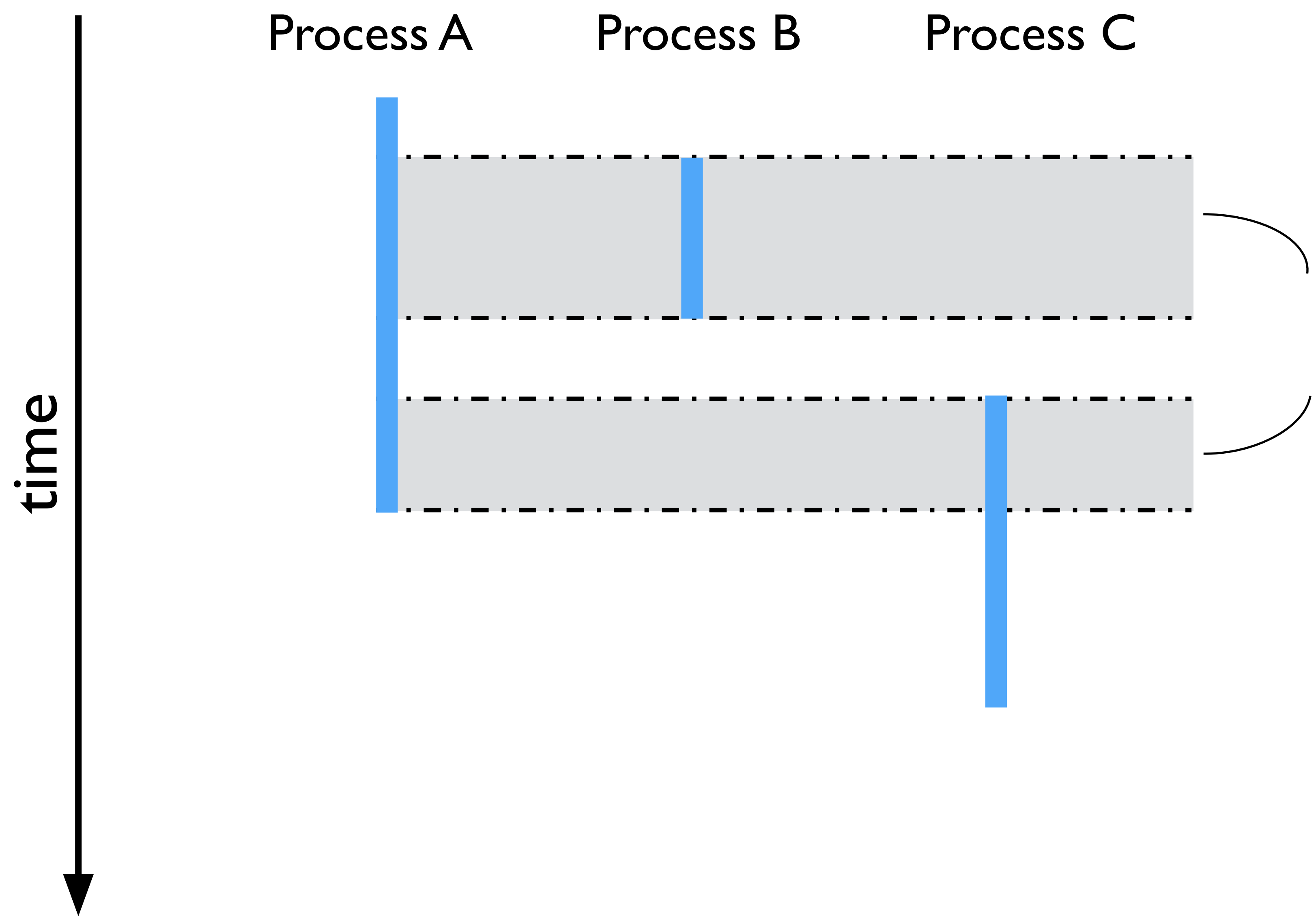
... downsides?

1. No multitasking!
2. A malicious (or badly written) program can "take over" the CPU forever
3. An idle process (e.g., waiting for input) will underutilize the CPU

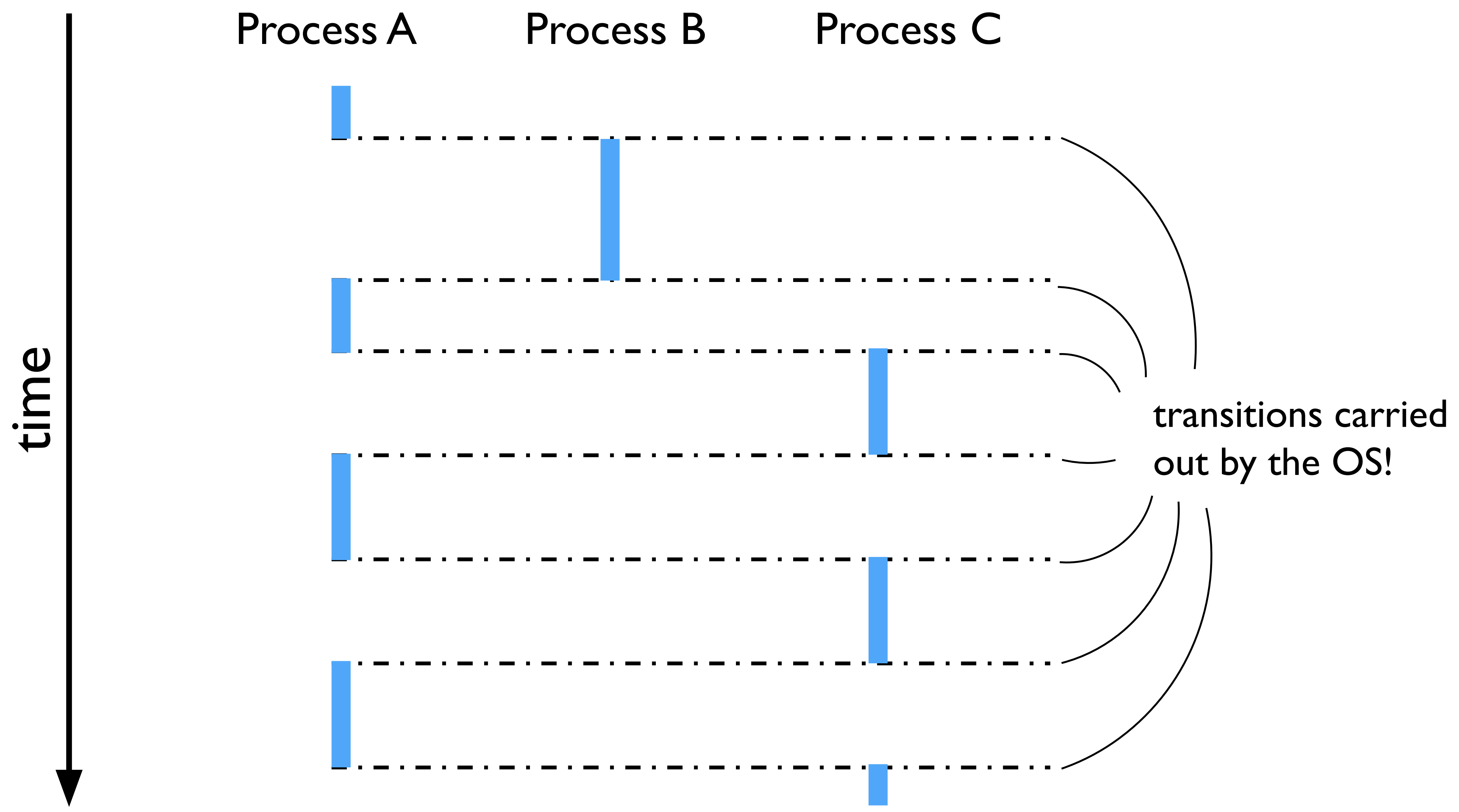
the operating system simulates a *seamless logical control flow* for each active process

many of which can be taking place *concurrently* on one or more CPUs

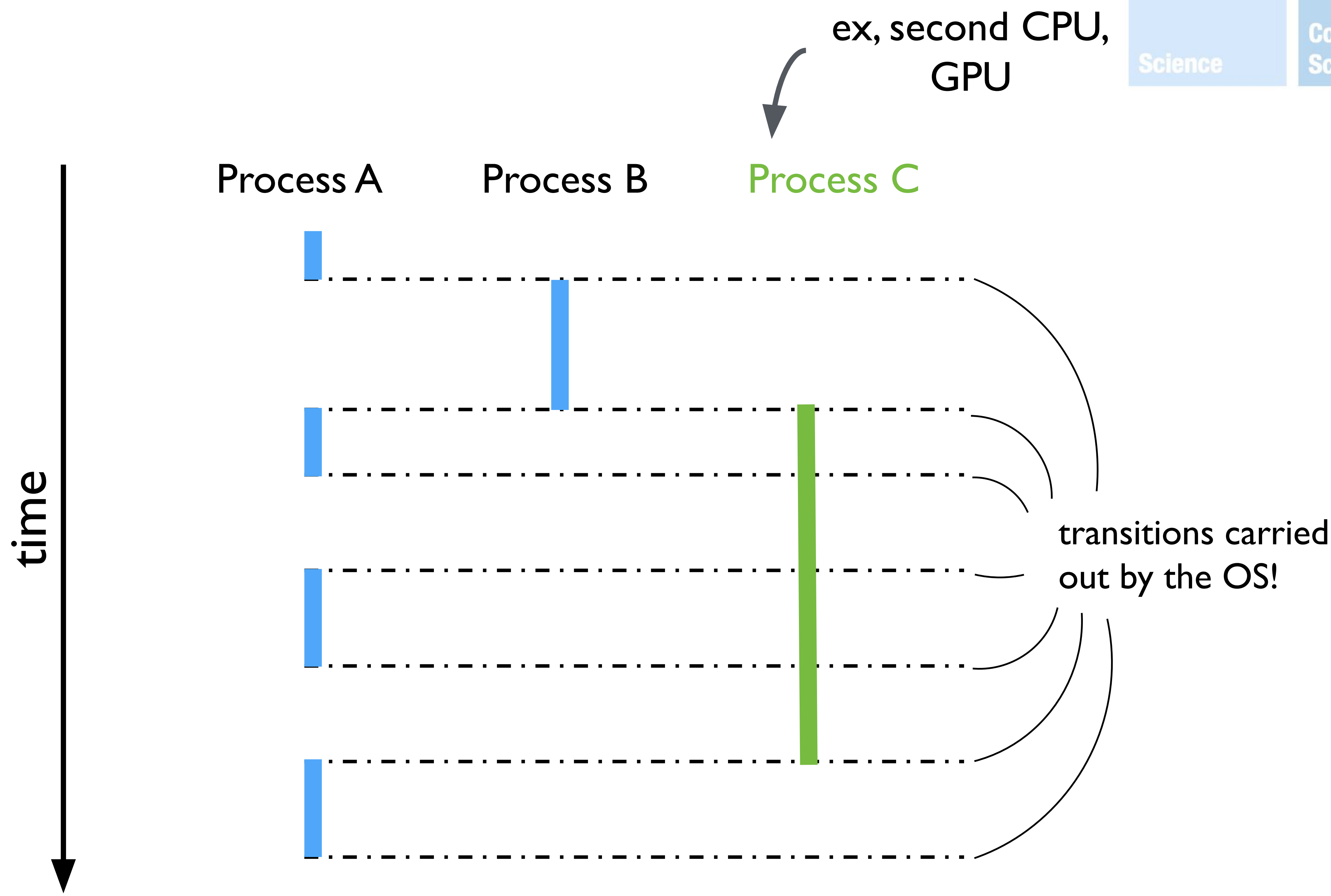
Discussions



Logical control flow



Physical flow (one compute)



Physical flow (two computes)

This is extremely elegant!

From the perspective of the program, the entire machine belongs to itself! (virtualization)

From the perspective of the process, it controls its runtime! (isn't interrupted by anything other than itself)

To implement this, we empower the OS.
We need:

periodic clock interrupt

1. a mechanism to periodically *interrupt* the current process to run the OS

scheduler

2. an OS module that *schedules* processes

context switch

3. a way to help seamlessly *switch* between processes

Discussions

to implement scheduling and carry out context switches, the OS must maintain a wealth of *per-process metadata* a process has:

- code (program)
- runtime data (global, local, dynamic)
- PC, SP, FP & other registers
- OS metadata, aka *process control block*
  - e.g., PID, mem/CPU usage, pending syscalls

a process also:

- **exists on some subset** of system hardware
- can **communicate** with other processes
- can **use** other static materials (files, etc)

actions that take place outside a process's logical control flow (e.g., context switches),
but may still affect its behavior
are part of the process's ***exceptional* control flow**

§ Exceptional Control Flow

Discussions

Two classes of exceptions:

I. synchronous
II. asynchronous

Two classes of exceptions:

I. **synchronous**

II. asynchronous

Synchronous exceptions are caused by the *currently executing* instruction

3 subclasses of synchronous exceptions:

1. traps **Intentional!**
2. faults **Usually unintentional - but might recover**
3. aborts **Unintentional - cannot recover**

1. traps

traps are intentionally triggered by a process

e.g., to invoke a system call

```
char *str = "hello world";
int len = strlen(str);
write(1, str, len);
...
```

```
# syscall num
# trap instr
```

return from trap (if it happens) resumes execution at the next instruction

i.e., looks like a function call!

2. faults

faults are usually unintentional, and may be recoverable or irrecoverable

e.g., segmentation fault, protection fault, page fault, div-by-zero often, return from fault will result in *retrying* the faulting instruction

— esp. if the handler "fixes" the problem

3. aborts

aborts are unintentional and irrecoverable

i.e., abort = program/OS termination

e.g., memory ECC error

Two classes of exceptions:

I. synchronous

II. asynchronous

Two classes of exceptions:

I. synchronous

II. asynchronous

Asynchronous exceptions are caused by events *external to* the current instruction

```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```

```
hello world!
hello world!
hello world!
hello world!
^C
$
```

hardware initiated asynchronous exceptions are known as ***interrupts***

e.g., ctrl-C,
ctrl-alt-del,
power switch

interrupts are associated with specific processor (hardware) pins

- checked after every CPU cycle
- associated with handler functions via the "interrupt vector" - array of pointers to handlers in the OS code

Typical interrupt procedure:

1. save process context
2. load OS
3. run handler & scheduler
4. load process context (might not match process from #1!)
5. return

Discussions

important: after switching context to the OS (for exception handling), there is **no guarantee a process will be switched back in!**

Cost of metadata saving, loading new code, no guarantee of return --

switching context to the kernel is **potentially very expensive**

— but it's often the only way to do what needs doing! (sys calls, IO, etc)

Moral:

use system calls as **sparingly** and as **efficiently** as possible!