# CS 354 - Machine Organization & Programming
# Tuesday Nov 28, and Thursday Nov 30 , 2023

**CS Annual Climate Survey Reminder: Data Buddies "slides"**
**Homework hw7:** DUE on or before Monday Nov 27
**Homework hw8:** DUE on or before Monday Dec 5

**Project p6:** Available and due on last day of classes.

**Learning Objectives**

- Describe and explain how computers transfer control to other processes
- Diagram and describe Exception Table and its use.
- Identify by name, number, and use several common exception types.
- Identify by name, number, and use several common system call operations.
- Describe and trace assembly for system calls.
- Describe and explain a process'es context.
- Diagram and describe interleaved processes and parallel processes
- Describe and explain the role of the Kernel's scheduler.
- Compare and constrast kernel mode vs user mode.
- Identify and describe the steps and state changes in a context switch.

**This Week**

| | |
|---|---|
| Kinds of Exceptions (from Week 12)<br>Transferring Control via Exception Table<br>Exceptions/System Calls in IA-32 & Linux<br>Processes and Context<br>User/Kernel Modes<br>Context Switch<br>Context Switch Example | Meet Signals<br>Three Phases of Signaling<br>Processes IDs and Groups<br>Sending Signals<br>Receiving Signals |
| **This Week and Next Week**: Signals, and multifile coding, Linking and Symbols<br>B&O 8.5 Signals Intro, 8.5.1 Signal Terminology<br>8.5.2 Sending Signals<br>8.5.3 Receiving Signals<br>8.5.4 Signal Handling Issues, p.745 | |

# Transferring Control via Exception Table

✵ *Exceptions transfer control* to the kernel

### Transferring Control to an Exception Handler

1. push ret addr (I curr / I next)
2. push Interrupted proc state

→ What stack is used for the push steps above? Kernel's stack

3. do indirect function call

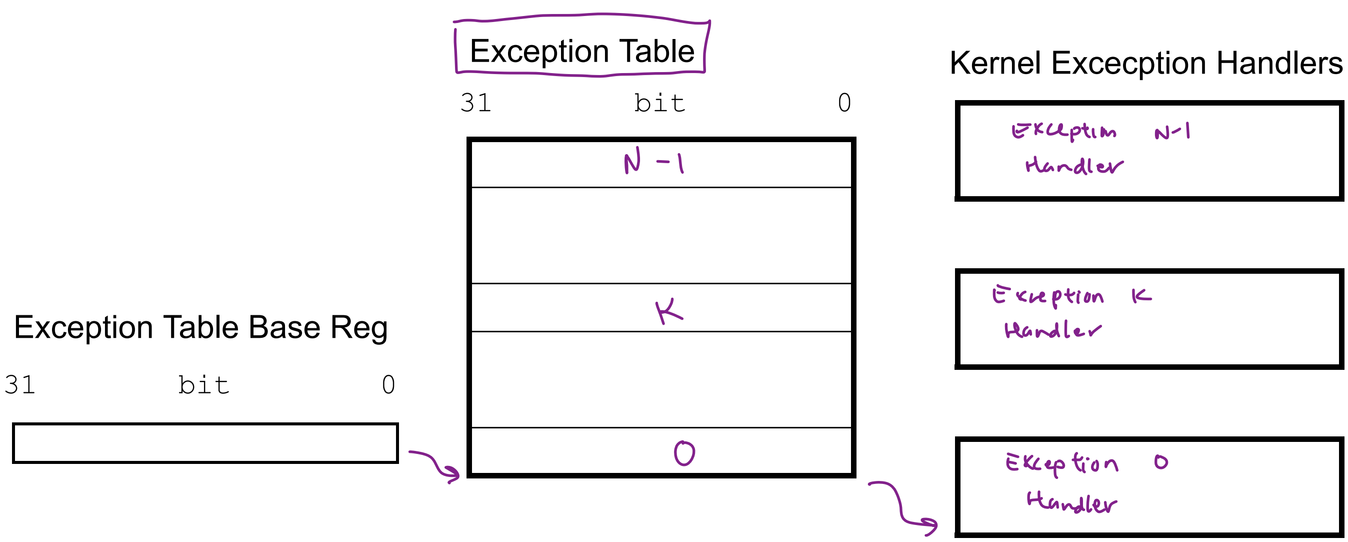
*indirect function call* $EHA = m[R[ETBR] + ENUM]$

ETBR is for exception table base reg
ENUM is for exception number
EHA is for exception handler's address

} jump table

**Exception Table** - unique non neg ints associated w/ each exception type

*exception number*



# Exceptions/System Calls in IA-32 & Linux

### Exception Numbers and Types

| | | |
|---|---|---|
| 0 - 31 are defined by processor | 0 | div by 0 |
| | 13 | general prot fault - SEG fault! |
| | 14 | page fault - handled by OS |
| | 18 | mach check - hardware error |
| 32 - 255 are defined by OS | 128 ($0x80) | ← trap to system call |

### System Calls and Service Numbers

1 exit
2 fork
3 read file    4 write file    5 open file    6 close file    file I/O
11 execve

### Making System Calls

1.) put svc num in %eax

2.) put sys call args in registers : %ebx, %ecx, %edx, %esi, %edi

3.) `int $0x80` trap or system call

### System Call Example

```
#include <stdlib.h>
int main(void) {
→ write(1, "hello world\n", 12);
  exit(0);
}
```

**Assembly Code:**

```
   .section .data
   string:
      .ascii "hello world\n"
   string_end:
      .equ len, string_end - string
   .section .text
   .global main
   main:
svc num — movl $4, %eax            put 4 in eax  "write file"     write to
          movl $1, %ebx                                            std out
   args   movl $string, %ecx       args to write
          movl $len, %edx
sys call - int $0x80
svc num — movl $1, %eax            "exit"
    arg - movl $0, %ebx            arg = 0        exit (0)
sys call - int $0x80
```

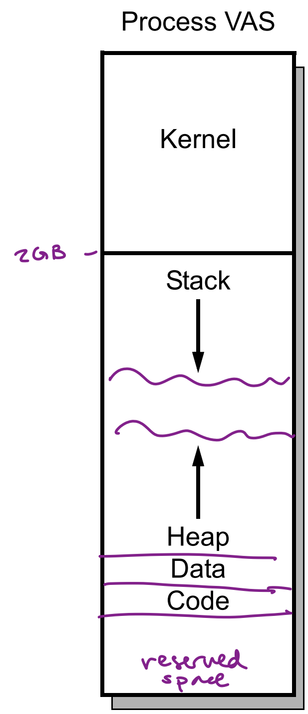
# Processes & Context

**Recall,** a <u>process</u>

- an instance of an exec program (running)
- has "context" info needed to restart process

**Why?**

easier to treat process as a single entity




running by itself

Key illusions – OS

1. CPU
2. memory
3. Devices

→ Who is the illusionist? OS

**Concurrency**

combined execution of 2 or more proc.

*<u>scheduler</u>* kernel code that switches btw proc

*<u>interleaved execution</u>* one CPU that is shared w/ all proc that take turns exec

*<u>time slice</u>* interval that a proc runs in

| time | proc A | proc B | proc C |
|---|---|---|---|
| 0 | ↓ | | |
| 1 | | ↓ | |
| 2 | | | ↓ |
| 3 | ↓ | | |
| 4 | | ↓ | |
| 5 | ↓ | | |
| 6 | | ↓ | |

*<u>parallel execution</u>* each process gets a core

| time | proc A | proc B | proc C |
|---|---|---|---|
| | ↓ | ↓ | ↓ |
| | | | |
| | | | |
| | | | |

# User/Kernel Modes

**What?** Processor *modes* are diff privilege level that a process can run on

*mode bit* indicate curr mode 1= kernel 0= user

kernel mode
- can exec any inst
- " access any mem location
- " " any device

user mode
- can exec some inst
- " access some mem
- " " some devices

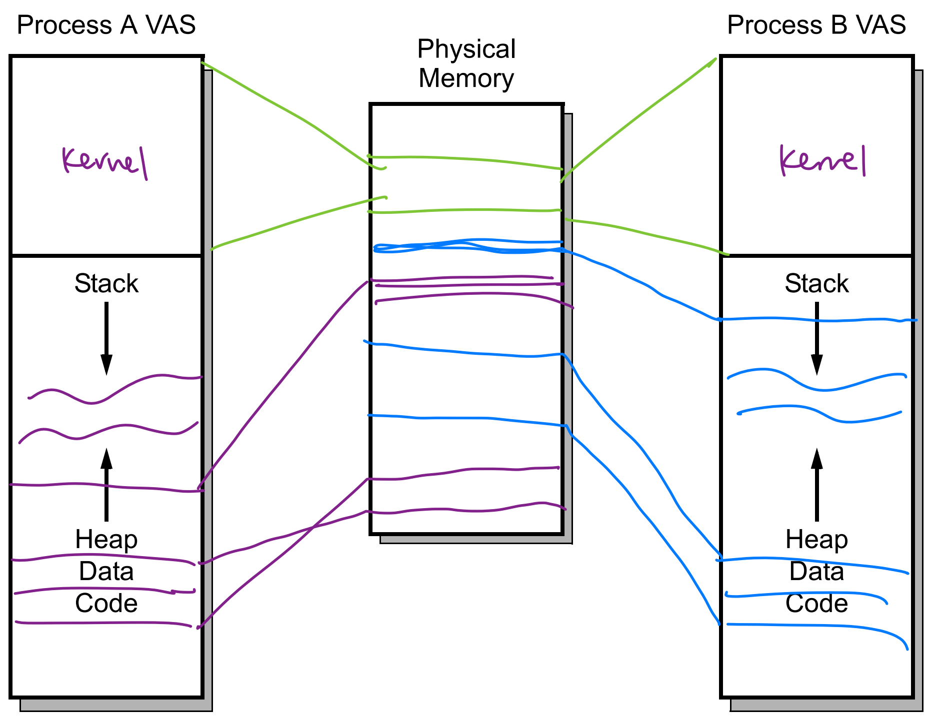
flipping modes

- start in user mode
- only exception switch to kernel
- kernel's E.H. can switch to user

**Sharing the Kernel**

key parts of OS:
+ shared by all proc
+ mem resident

pages of mem 4K



# Context Switch

stepping through a read call() system call

**What?** A *context switch*

- when OS switches from one process to another
- req preservation of proc context so it can restart
  1. CPU state
  2. user's stack esp ebp
  3. kernel's stack esp ebp
  4. kernel's data structure
     a. page table
     b. process table
     c. File table

**When?** happens as result of exception when kernel execute another process

ex) scheduler runs after timer interrupts to swap proc

**Why?** enables exceptions to be process

**How?**

1. Save context of curr process
2. restore context of some other process
3. transfer control to restored process

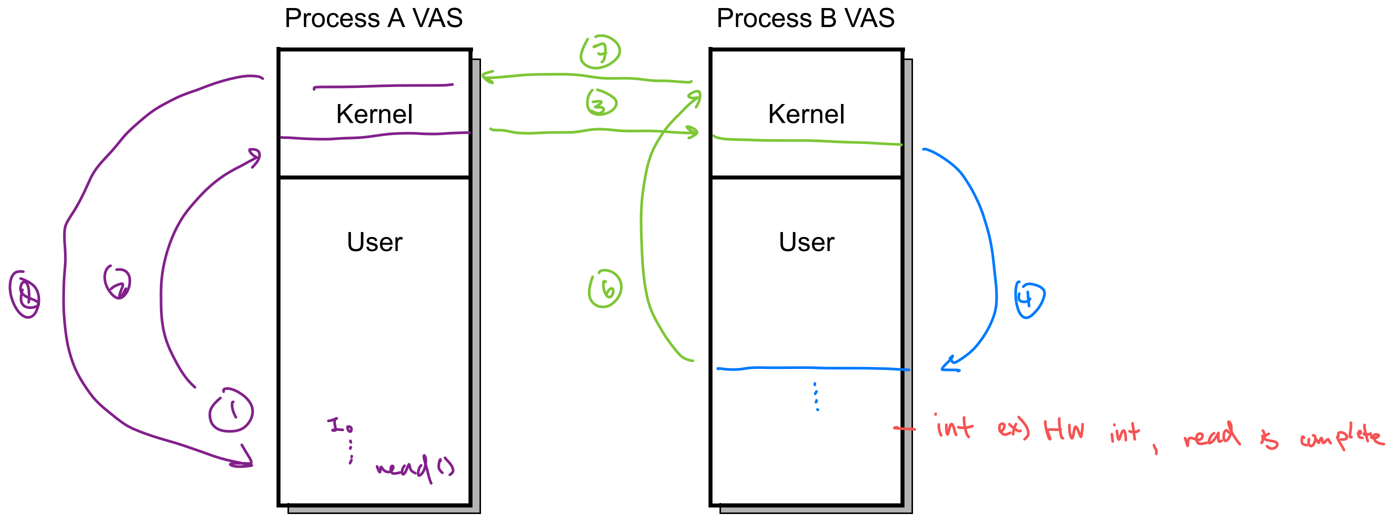
✵ *Context switches* are very expensive!

→ What is the impact of a context switch on the cache? negative

– "cache pollution"

# Context Switch Example

### Stepping through a `read()` System Call




1. process A is running in user mode ... get to read (i)

2. switch to kernel mode, run EH for sys call svc num (3)

3. In kernel mode do context switch
   - save context
   - restore B context
   - transfer control to B

4. switch to user mode

5. In user mode in proc B - int occurs
   - finish I curr

6. Switch to kernel mode

7. In ker mode do context switch - save B, restore A, transfer to A

8. switch user control - cont. A

# Meet Signals

✵ *The Kernel uses signals* to notify user proc. of exceptional events

**What?** A *signal* is small msg sent to proc via kernel

Linux: has 30 std sig. types, each w/ unique non-neg ID

**$kill -l** lists signal names and numbers

signal(7) man 7 signal

**Why?**

- so kernel can notify processes
  1. low-level HW exceptions
  2. high-level SW events (kernel) or from user processes

- to enable user proc to comm. w/ each other

- to implement a higher-level software form of Exceptional control flow

**Examples**

1. divide by zero

exception 0 interrupts to kernel handler

- kernel signals user proc with SIGFPE #8

2. illegal memory reference

exception 13 interrupts to kernel handler

- kernel signals user proc with SIGSEGV #11

3. keyboard interrupt irq #1

- ctrl-c interrupts to kernel handler which signal SIGINT #2
  terminate foreground process by default
- ctrl-z interrupts to kernel handler which signal SIGSTP #20
  suspends foreground process by default

# Three Phases of Signaling

### Sending

- when the kernel 's E.H. runs in response to Exception Event
- is directed to destination (proc)

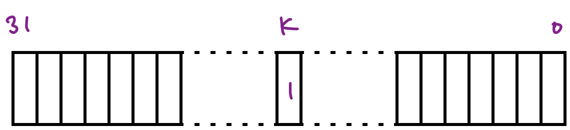
### Delivering

when the kernel records a sent signal for its destination proc

<u>*pending signal*</u> delivered but not recieved

- each process has a bit vector to record pending signals

<u>*bit vectors*</u>




- bit K is set to 1 when signal K is delivered

### Receiving

when the kernel causes dest proc to react to pending signal

- happens when kernel transfers control back to process
- multiple pending signals are recieved in order low to high signal

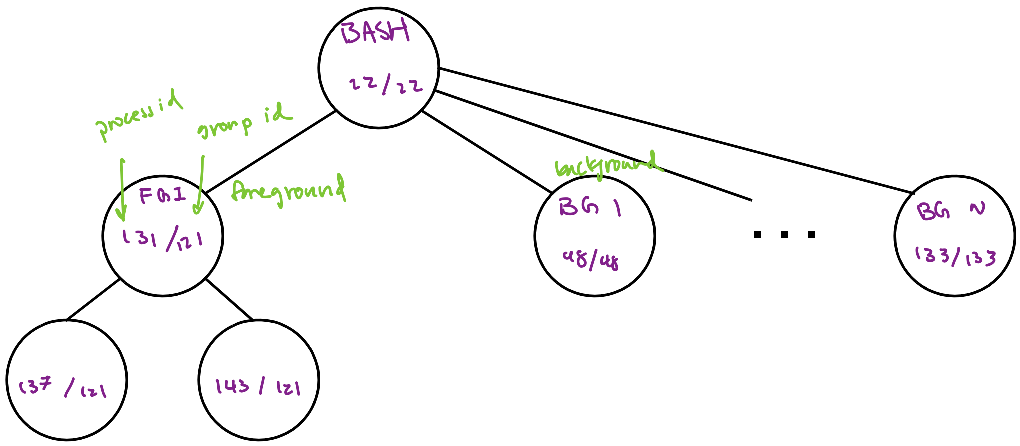
<u>*blocking*</u> prevents a signal from being rec'd

- enables a process to control which signal it pays attention to
- each process has a second bit vector for blocking signal

# Process IDs and Groups

**What?** Each process

- is identified by a process id
- belongs to exactly one process group




**Why?**

#s are easier to manage than names

**How?**

Recall:ps list process ps -u your processes

ps -al all in long forms

jobs list process using simple #s

getpid(2)getpgrp(2) man 2 get pid

`#include` <unistd.h>

`pid_t getpid(void)` returns process pid

`pid_t getpgrp(void)` " " gpid

# Sending Signals

**What?** A signal is sent by the kernel or a user process via the kernel or from cmd line

or in program using sys calls

## How? Linux Command

kill(1) man 1 kill - send signal from cmd line to a specific proc

**`kill -9 <pid>`** 9 SIGKILL 2 SIGINT ctr-c 20 SIGTSTP ctrl-z

terminate all proc.

→ What happens if you kill your shell?

logout

proc -> kernel

## How? System Calls

kill(2) man 2 kill - send from calling proc to callee proc

killpg(2) - send signal to all members of pgid

```
#include <sys/types.h>
#include <signal.h>

   int kill (pid_t pid, int sig)
```

proc that is target of signal

the signal being sent

returns 0 on success

alarm(2) man 2 alarm

sets alarm that will deliver SIGALRM after specified # of seconds

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds)
```

# secs until SIGALRM is sent to you

rets # secs. remaining if prev set alarm is running

otherwise, ret 0

# Receiving Signals

**What?** A signal is received by its destination process by doing default action

or by executing code specified by sig handler

### How? Default Actions

- Terminate the process SIGINT #2 ctrl -C
- Terminate the process and dump core SIGSEGV ctrl - c segfault
- Stop the process SIGTSTP #20 ctrl -Z suspend
- Continue the process if it's currently stopped SIGCONT #18 resume
- Ignore the signal SIGWINCH #28

### How? Signal Handler

1. code a signal handler (func)
   - looks like a regular func but it's called by the kernel
   - should not make unsafe system calls like printf (file I/O) (except in p6)
2. Register the signal handler
   - catch 1 or more signals

~~signal(2)~~
sigaction(2) POSIX examing and changing a signals default behavior

### Code Example

```
#include <signal.h>
#include ...
#include <string.h>

void handler_SIGALRM() { ... }   code to handle sig

int main(...) {

    // 2. Register sig alarm handler
    struct sigaction sa;
    memset (&sa, 0, sizeof (sa));      sa gets null
    sa.sa_handler = handler_SIGALRM;   no parenthesis!
    if (sigaction (SIGALRM, &sa, NULL) != 0) {
        printf ("Error binding SIGALRM handler\n");
        exit(1);
    }
```

different

for each sig handler