outlineL25-w13TR-student

Saturday, December 3, 2022 2:11 PM

outlineL25-
w13TR-st...

CS 354 - Machine Organization & Programming
Tuesday Nov 29, and Thursday Dec 1 , 2022

Homework hw7: DUE on or before Monday Dec 5
Homework hw8: DUE on or before Monday Dec 12
Project p5: DUE on or before Friday Dec 2
Project p6: Available Friday and due on last day of classes.

Last Week

(from Week 11) Alignment Alignment Practice Unions Pointers Function Pointers Buffer Overflow & Stack Smashing	Flow of Execution Exceptional Events

This Week

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

CS 354 (F22): L25 - 1

Transferring Control via Exception Table

* *Exceptions transfer control to the Kernel.*

Transferring Control to an Exception Handler

1. push return addr (I_{curr} or I_{next})
2. push interrupted processor's state so it can be restarted

→ What stack is used for the push steps above?

The kernel stack

3. do indirect function call which runs the appropriate exception handler

indirect function call uses exception table to determine what function to execute

EHA = M[R[ETBR] + ENUM]

ETBR is for exception table base reg like pointer to array's base

ENUM is for exception number like index

EHA is for exception handler's address like element's value

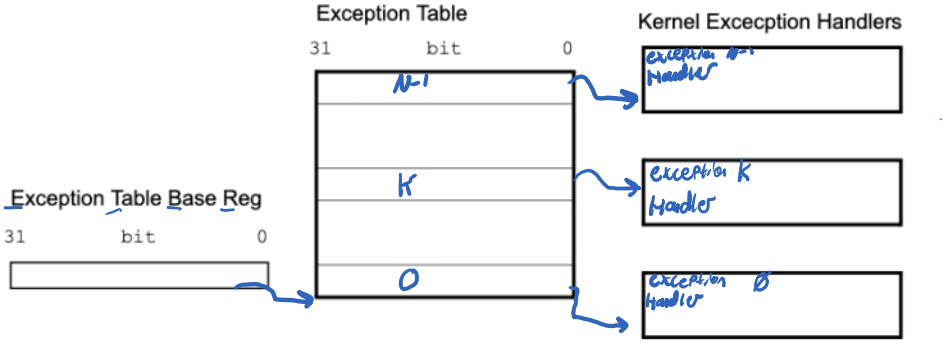
Exception Table

is a jump table for exceptions that's allocated in memory by the OS on system boot

array of function pointers

exception number a unique non-negative integer associate with each type of exception

index to table



Exceptions/System Calls in IA-32 & Linux

Enum

Exception Numbers and Types

0 - 31 are defined by processor

Enums

0 divide by zero
13 protection fault — seg fault
14 page fault — mem access
18 machine check

32 - 255 are defined by OS

128 ($0x80) trap makes system call

System Calls and Service Numbers

1 exit terminate process
2 fork create a new process
3 read file
4 write file
5 open file
6 close file
11 execve

File I/O

Making System Calls

where OS will look

1.) put service number in %eax

2.) put sys call args in remaining regs EXCEPT %esp

3.) `int $0x80` trap to start system call handler

System Call Example

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

main.c

Assembly Code:

.section .data. x are assembler directives — .data section of Data Segment

```
   .section .data
   string:
      .ascii "hello world\n"
   string_end:
      .equ len, string_end - string
   .section .text
   .global main
   main:
      movl $4, %eax
      movl $1, %ebx
      movl $string, %ecx
      movl $len, %edx
      int $0x80
      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

.text section of CODE Segment

① sys num #
arg 0 in ebx
arg 1 in ecx
arg 2 in edx
② — SysCall
③ trap to start sys call handle

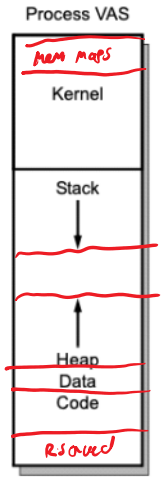
① ② ③ → go

Processes & Context

Recall, a *process*

- is an instance of an executing program
- has *context* the information needed to restart the process

Why? - easier to treat processes as a single entity running by itself



Key illusions process exclusively uses

1. CPU
2. Main memory
3. Devices

→ Who is the illusionist?

The OS

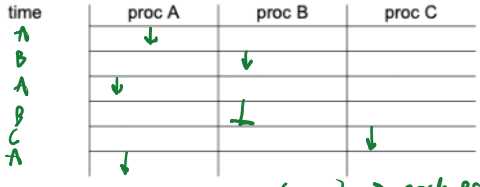
Concurrency (multi-threading, multiprocessing, multitasking)

Combined execution of 2+ processes "at the same time"

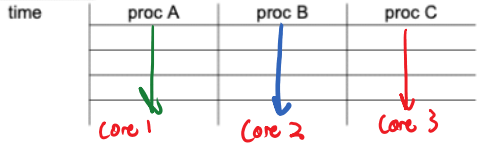
scheduler Kernel code to switch executions among processes

interleaved execution One CPU used among multiple processes

"*time slice*" interval that a particular process runs in



parallel execution Multiple CPUs (cores) → each processor gets one



User/Kernel Modes

What? Processor *modes* are
different privilege levels a process can run at

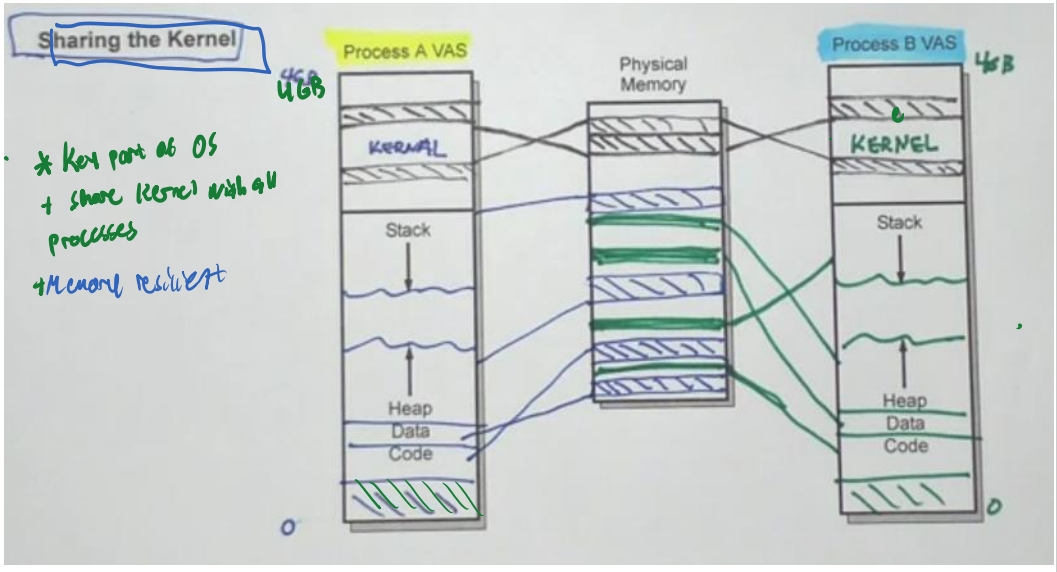
mode bit indicates current mode, 1 = kernel/supervisor, 0 = user

1 kernel mode Can access any instruction + any memory + any device

0 user mode Can access some instructions (execute) Can access devices but only through OS.
memory access is limited

flipping modes

- STARTS in user mode
- only exceptions can switch user mode to kernel mode
- Kernel exception handler can switch to user mode



Context Switch

What? A *context switch*

- is when the OS switches out one running process for another process
- requires preservation of processes context so it can be restarted

1. CPU state
2. User stack %ebp %esp
3. Kernel stack " %ebp %esp "
4. Kernel data structures
   a. page table
   b. process table
   c. fd table

When?
Happens as a result of an exception when Kernel needs to execute another process.

Why? enables exceptions to be handled "processed"

How?

1. **save** context of current process
2. **restore** restore context of another process
3. **transfer** control to restored process

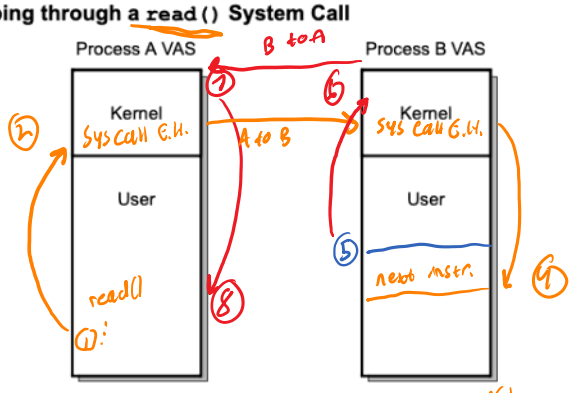
* *Context switches are very* expensive !

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

BAD! "cache pollution"

Context Switch Example

Stepping through a `read()` System Call



1. In user mode running process A ... get read()
2. Switch to kernel mode, run sys call exception handler for SOC #3
   arrange direct memory access
3. In kernel mode → time for context switch (save process A, restore B)
4. Switch to user mode for process B
5. In user mode, running process B
6. INT occurs from Disc Controller
   ↳ Switch to kernel mode, run exec handler
7. In kernel mode, running our interrupt handler → Do a context switch back to
   Save B
   Restore A
8. Switch to user, Resume process A

Meet Signals

* *The Kernel uses signals to notify User processes of exceptional events.*

What? A *signal* is

a small message sent to the process via the kernel

Linux: 30 standard signal types each given a unique positive int

`$kill -l` see signal names and numbers

signal(7) `$man 7 signal`

Why?

- so Kernel can notify user processes about
  1. Low level hardware events (exec 0-31) } P6
  2. high-level sw in Kernel or user process signals
- to enable user processes to communicate with eachother } P6
- to implement a high-level software form of exception handling

Examples

1. divide by zero

exception #0 interrupts to kernel handler

- kernel signals user proc with SIGFPE #8 (floating point exception)

2. illegal memory reference

exception #13 interrupts to kernel handler

- kernel signals user proc with SIGSEGV #11

3. keyboard interrupt

- ctrl-c interrupts to kernel handler which signals SIGINT #2 to foreground process
- ctrl-z interrupts to kernel handler which suspends running process → puts in background

fix: 'fg' Signals SIGSTP #20 to suspend foreground process

Three Phases of Signaling

Sending

- when the kernel exception handler runs in response to exceptional event or from signal to user process
- is directed to a dest. process

Delivering

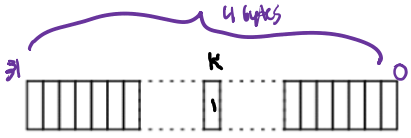
when the kernel records a sent signal for its destination process

pending signal is delivered but not received

pending signal is delivered but not received

- each process has a bit vector for recording pending signals

bit vectors are kernel data structures where each bit has a distinct meaning



- bit k set to 1 when a signal is delivered

Receiving

when the kernel causes dest process to react to pending signals

- happens when The kernel transfers control back to a process
- mult pending signals are done in order from low to high signal number 0 highest priority

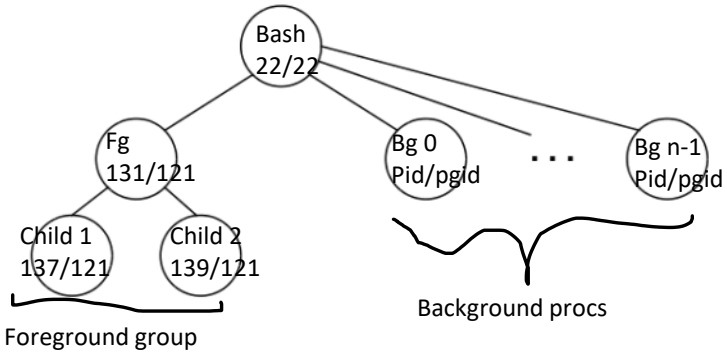
blocking prevents a signal from being received

- enables a process to Control which signal it pays attention to
- each process has a second bit vector for blocked signals (1=block)

Process IDs and Groups

What? Each process

- is identified by a PID — Process ID #
- belongs to exactly 1 group identified by a — Process group id number (pgid)



Why?

Numbers are easier to manage than names

How?

Recall:pslists processes

```
ps -u
ps -al
     jobs
```

Processes that user started
-al shows all running processes
Lists all processes using a simple number

getpid(2)getpgrp(2) `$man 2 getpid` Man 2 getpgid

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

pid_t getpid(void)
```

Returns calling processes' ID

```
pid_t getpgrp(void)
```

Returns calling processes' gpID

Sending Signals

What? A signal is sent by the kernel or a user process via the kernel Or via the keyboard
from the command line or in a program using system calls

How? Linux Command

kill(1) `$man 1 kill` Linux manual section 1 - user commands

Sends a signal from the command shell to a specified process.

```
kill -9  <pid>
```

9 is SIGKILL, 2 is SIGINT, 20 is SIGTSTP, 19 SIGSTOP, 18 SIGCONT

→ What happens if you kill your shell? "SAVE YOUR WORK FIRST"

How? System Calls

kill(2) `$man 2 kill` Linux manual section sys call

killpg(2)

```
#include <sys/types.h>
#include<signal.h>
int kill (pid_t pid, int sig)pid of process to signal, sig to send SIGKILL
```

returns 0 on success, -1 on error

alarm(2) Man 2 alarm

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

Seconds until alarm is sent

Returns # secs remaining if
prev. alarm is still running

Receiving Signals

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

Or by executing a program
specified signal handler

How? Default Actions

- Terminate the process eg. SIGINT #2 CTRL-C
- Terminate the process and dump core SIGSEV #11
- Stop the process SIGTSTP #20
- Continue the process if it's currently stopped SIGCONT #18
- Ignore the signal SIGWINCH #28

How? Signal Handler

1. Code a Signal Handler
   - Looks like a regular function but its called by KERNEL
   - Should NOT make unsafe syscalls

     "printf" (ok, for p6 where instructed)

2. Register the Signal Handler
   - Catches one or more signals

~~signal(2)~~ Don't use this
sigaction(2) Use this instead - POSIX for examining and changing a signal action

Code Example

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

void handler_SIGALRM() { ... }
int main(...) {
```

Stdlib.h stdio.h

//possible int args used (exception handler func)

```
Struct sigaction sa;
Memset(&sa, sizeOf(sa)); //sa.sa_flags = 0; zeroes out struct
Sa.sa_handler = handler_SIGALRM; //func pointer to our handler code
If (sigaction(SIGALRM, &sa, NULL) != 0) {
 printf("Error: binding SIGALRM handler\n");
        Exit(1);
}
```