02. Exceptional Control Flow

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s control flow (or flow of control)

Physical control flow

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

Altering the Control Flow

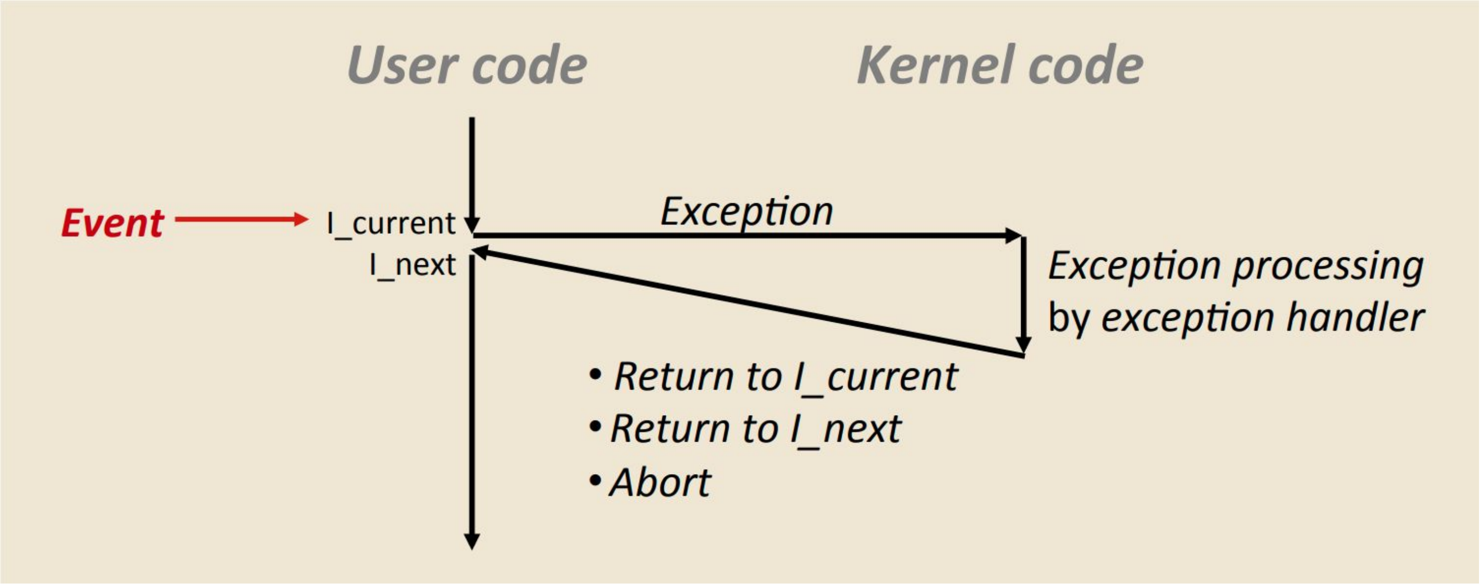
- Two mechanisms for changing control flow in **program state**:
  - Jumps and branches
  - Call and return
- Insufficient for an OS (difficult to react to changes in **system state**):
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - User hits Ctrl-C at the keyboard
  - System timer expires
- OS needs mechanisms for “exceptional control flow”

Exceptional Control Flow

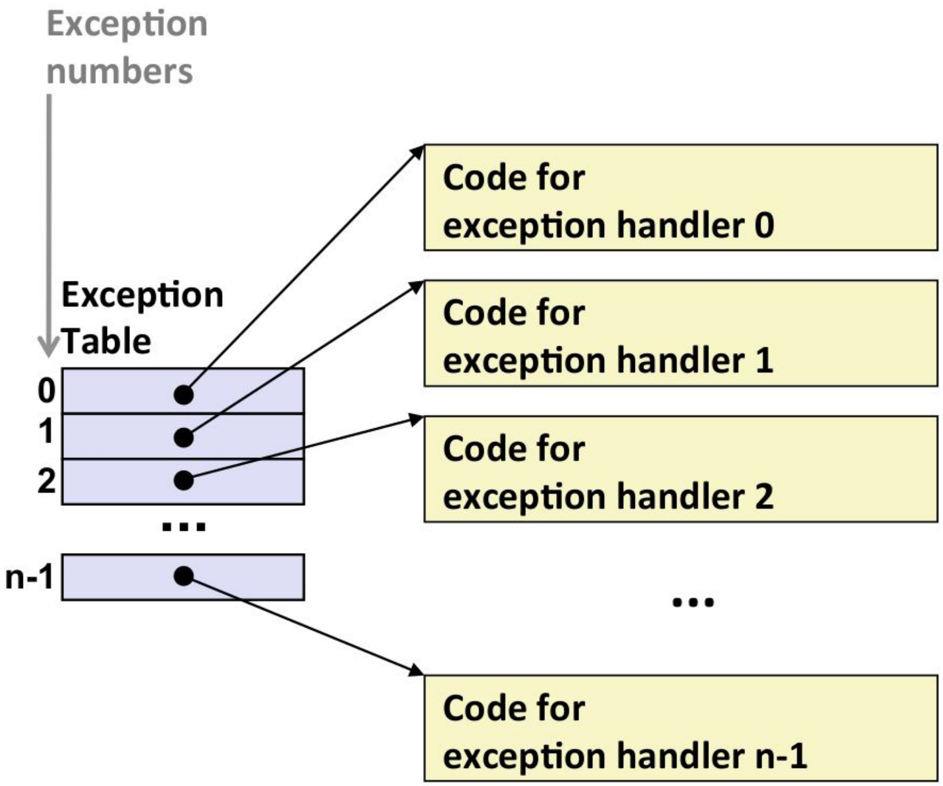
- Exists at all levels of a computer system
- Low level mechanisms
  - 1. Exceptions
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS
- Higher level mechanisms
  - 2. Process context switch
    - Implemented by OS and hardware timer
  - 3. Signals
    - Implemented by OS
  - 4. Nonlocal jumps: setjmp() and longjmp()
    - Implemented by C runtime library

Exceptions

- An exception is a transfer of control to the OS kernel in response to some ev



Exception Tables

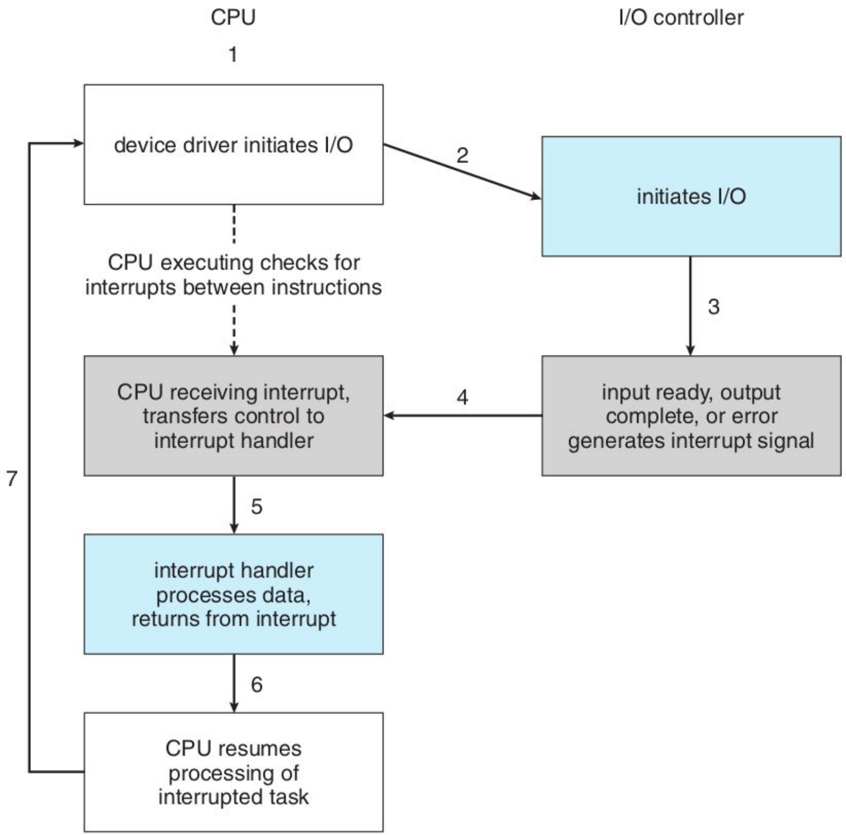


- On x86, they are called interrupt vector tables
    - Although not all are interrupts
- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Exception handler k is called each time exception k occurs

Asynchronous Exceptions (Interrupts)

- Caused by events **external** to the processor
  - Indicated by setting the processor's **interrupt pin**
  - Handler returns to "next" instruction
- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network
    - Arrival of data from a disk

Interrupt-Driven I/O Cycle



Synchronous Exceptions

- Caused by events that occur **as a result of executing an instruction** (also called software-generated interrupts):
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - Aborts
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

System Calls

- System
  - Wha

- Each sy
  - The ... umbers

Number	Name	Description
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Invoking a System Call

- A user program runs in user mode, and a system call runs in kernel mode
  - When making a system call in user mode, a special **trap** instruction is executed to switch from user mode to kernel mode
  - The trap handler examines the system call number and dispatches to the corresponding system call handler
- Some system calls have one or more parameters
  - Parameters can be passed in registers
  - Parameters can be stored in a block, and the address of the block is passed in a register
  - Parameters can be passed through the stack
  - Block and stack methods do not limit the number or length of parameters being passed

x86 Example

```
movl $20, %eax # Get PID of current process
int $0x80 # Invoke system call!
# Now %eax holds the PID of the current process
```

- X86 (32 bit and 64 bit) has several methods for system call
  - **int $0x80** is the classical way to make the system call
    - 0x80 of the system's exception table gives the trap handler
  - Parameters for Linux system call are passed using registers
    - %eax is for system call number, and %ebx, %ecx, %edx, %esi, %edi, %ebp are used for passing 6 parameters to the system call
    - After the system call, register %rax contains the result (return value)
- There is a faster way to make 32-bit system calls: using the **sysenter**
  - Parameter passing is the same as for int $0x80
- On x86-64, we normally use the **syscall** instruction for fast system calls
  - System call number → %rax
  - Return value → %rax
  - 6 parameters → %rdi, %rsi, %rdx, %r10, %r8, %r9

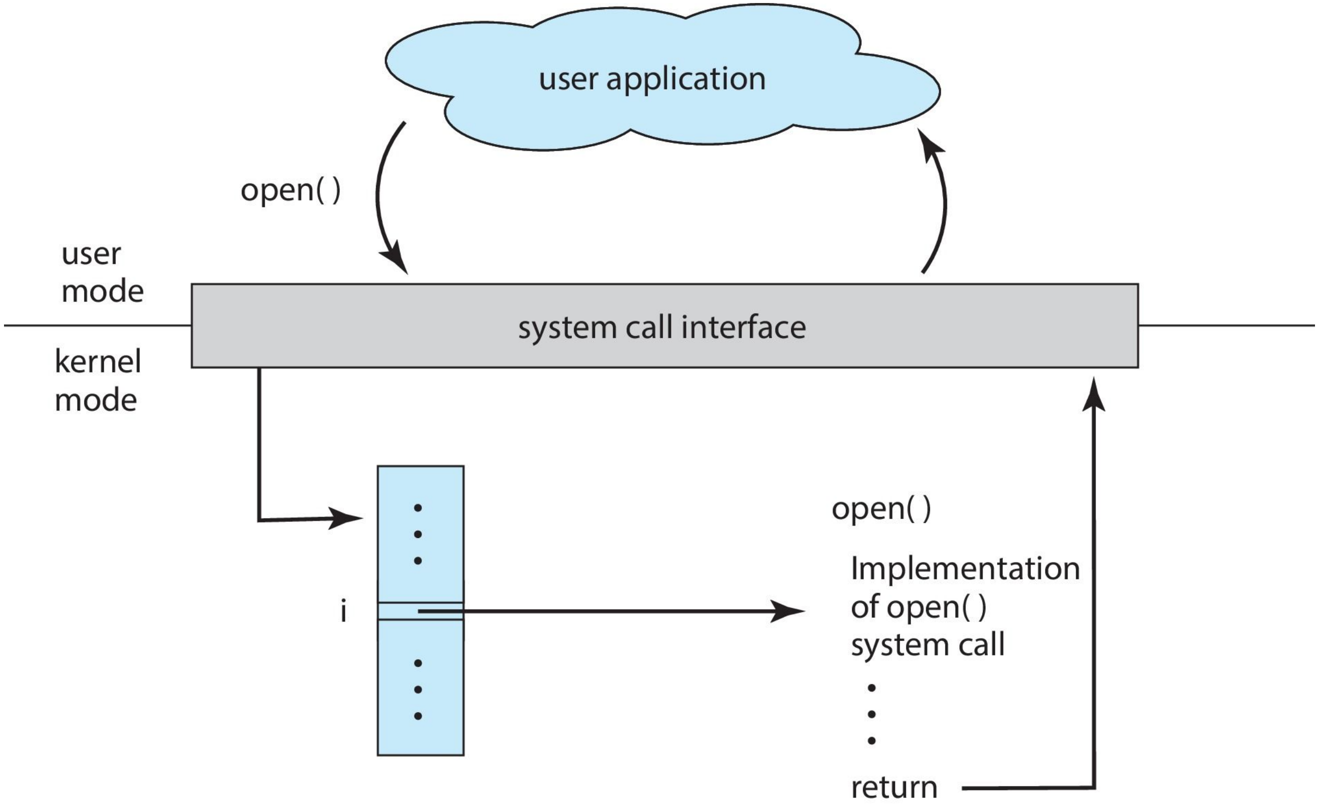
Different Architectures

Arch/ABI	Instruction	System call #	Ret val	Ret val2	Error
alpha	callsys	v0	v0	a4	a3
arc	trap0	r8	r0	-	-
arm/OABI	swi NR	-	r0	-	-
arm/EABI	swi 0x0	r7	r0	r1	-
arm64	svc #0	w8	x0	x1	-
blackfin	excpt 0x0	P0	R0	-	-
i386	int $0x80	eax	eax	edx	-
ia64	break 0x100000	r15	r8	r9	r10
m68k	trap #0	d0	d0	-	-
microblaze	brki r14,8	r12	r3	-	-
mips	syscall	v0	v0	v1	a3
nios2	trap	r2	r2	-	r7
parisc	ble 0x100(%sr2, %r0)	r20	r28	-	-
powerpc	sc	r0	r3	-	r0
powerpc64	sc	r0	r3	-	cr0.SO
riscv	ecall	a7	a0	a1	-
s390	svc 0	r1	r2	r3	-
s390x	svc 0	r1	r2	r3	-
superh	trap #0x17	r3	r0	r1	-
sparc/32	t 0x10	g1	o0	o1	psr/csr
sparc/64	t 0x6d	g1	o0	o1	psr/csr
tile	swint1	R10	R00	-	R01
x86-64	syscall	rax	rax	rdx	-
x32	syscall	rax	rax	rdx	-
xtensa	syscall	a2	a2	-	-

Wrappers

- The program invokes a wrapper function in the C library
  - If you've ever noticed files like "libc.so", that's the C library
  - But, why?
- The wrapper function puts any arguments to the system call in the registers **expected by the OS kernel**
- The wrapper function executes a trap instruction to invoke the system call (as mentioned above -- system call number is in %rax)
- The wrapper function checks if the service returned an error, and if so, sets a global variable named ***errno*** with this error value
- The wrapper function then returns to the caller and provides an integer return value to indicate success or failure

API – System Call – OS Relationship



System Call Example: Opening File

- User calls: **open(filename, options)**
- Calls **__open()** function, which invokes system call instruction **syscall**
  - %rax contains syscall number -- 0x2
  - Return value in %rax
  - Negative value is an error corresponding to negative errno

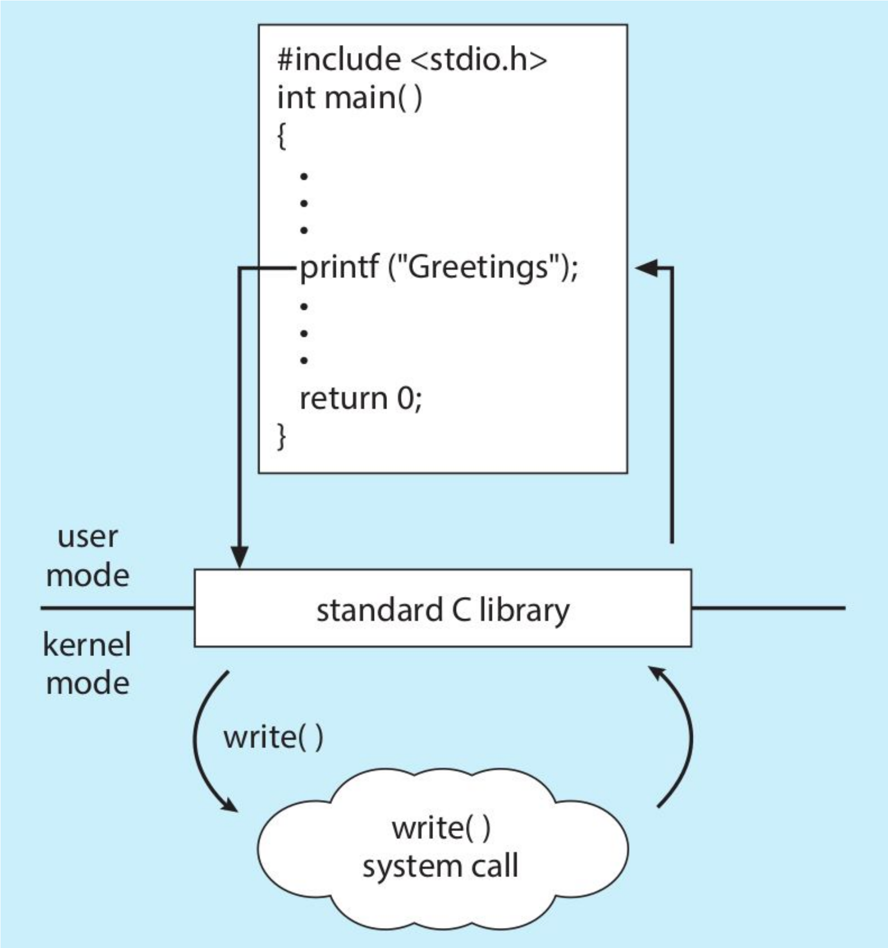
```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%eax  # open is syscall #2
e5d7e:   0f 05               syscall         # Return value in %rax
e5d80:   48 3d 01 f0 ff ff   cmp  $0xfffffffffffff001,%rax
...
e5dfa:   c3                  retq
```

Tracing System Calls

- The strace command lets you see the system calls that are made by a process
  - Example: type “strace ls” at a terminal to see all the system calls that the ls (which lists files in the current directory) makes
  - You might see calls like:
    - execve -- this loads a new program and starts running it
    - open -- open a file
    - read -- read from a file
    - close -- close a file
    - fstat -- get information about a file
- See details by typing “man strace”

Standard C Library Example

- C program invoking printf() library call, which calls write() system call

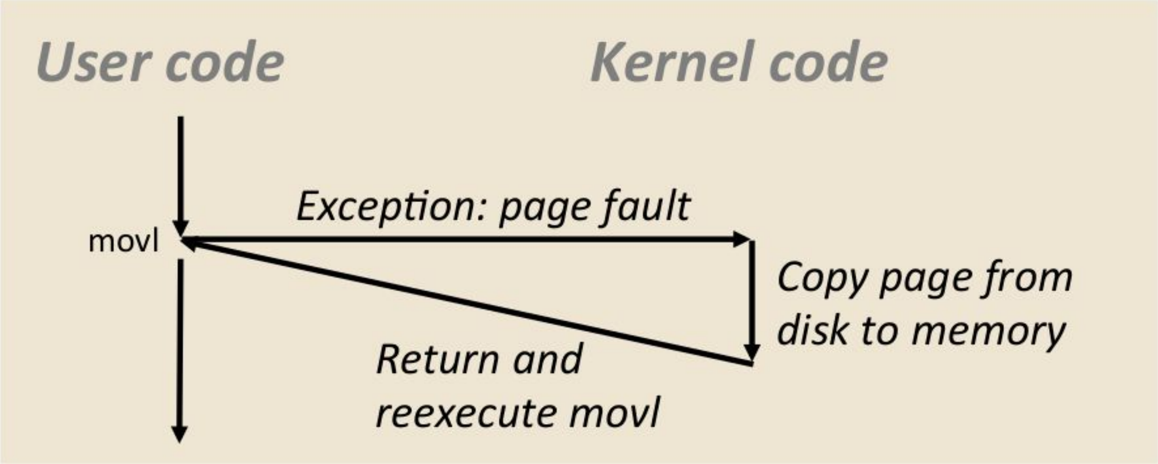


Fault Example: Page Fault

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

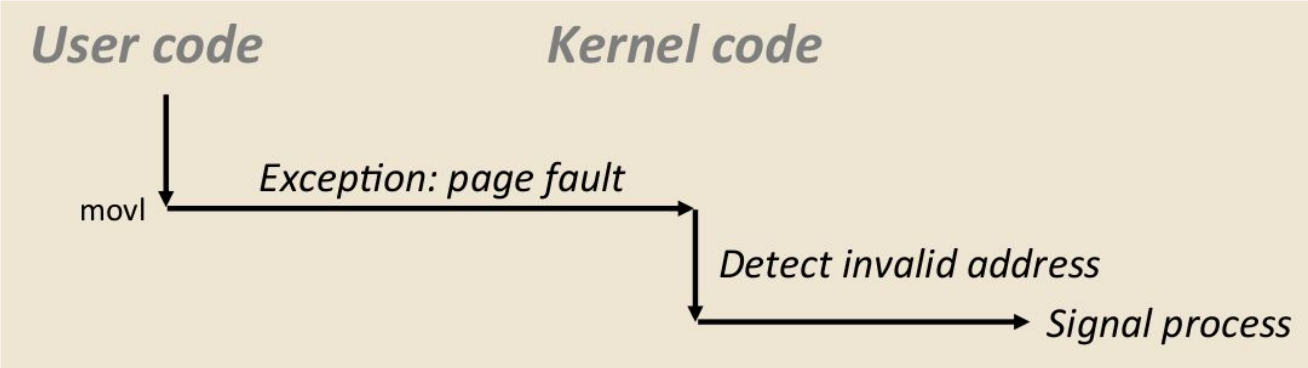
- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
80483b7:   c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```




```
80483b7:   c7 05 60 e3 04 08 0d    movl    $0xd,0x804e360
```

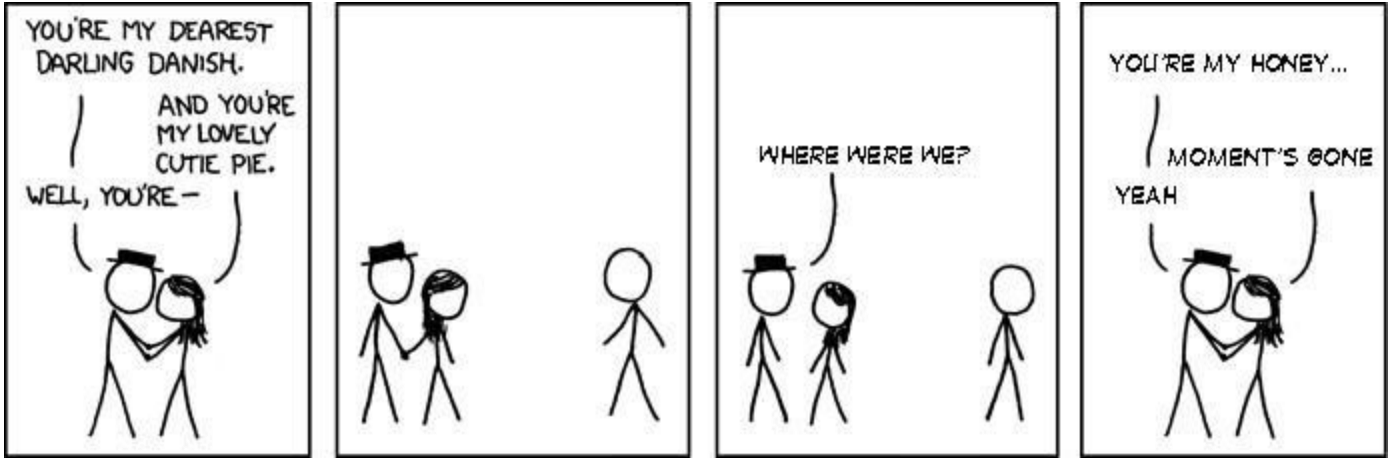
- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”

Homework

- Read Chapter 3
- Mini-project 1 is due on Sep. 11th

Next Lecture

- We will start learning process management



Credit: http://xkcdsw.com/873