**System Control Flow & Processes**

**System Control Flow**

- From startup to shutdown, a CPU reads and executes a sequence of instructions, one at a time; this sequence is the control flow

- Up to now, control flow can only be changed by jumps/procedures; both are reacting to changes in *program state*

- Also needs to react to changes in *system state*

- Unix/Linux user hits CTRL-C on keyboard

- User clicks on different application’s window on screen

- Instruction divides by zero

- Data arrives from disk or network adapter

- System timer expires

- Jumps and procedure calls can’t achieve this – the system needs mechanisms for “*exceptional”* control flow

- Exceptional control flow exists at all levels of a computer system

- Low level mechanisms

- Exceptions: change in processor’s control flow in response to a system event; implemented using a combination of hardware and OS software

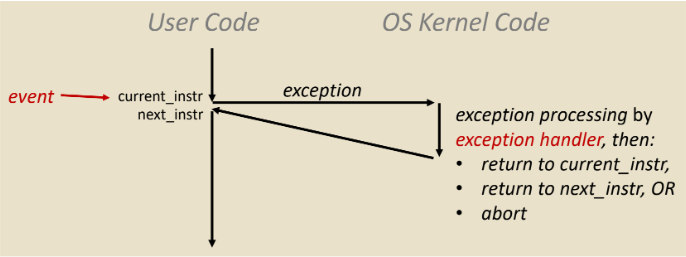
- Higher level mechanisms

- Process context switch: implemented by OS software and hardware timer

- Signals: implemented by OS software

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

- Kernel is memory-resident part of OS

 - Ex: Division by 0, page fault, Ctrl-C, I/O request completes

- System knows where to jump to in the OS using an exception table – jump table for exceptions (*Interrupt Vector Table*)

- Each type of event has a unique exception number *k,* which is the index into exception table

- Asynchronous Exceptions (Interrupts) are caused by events *external* to the processor

- After interrupt handler runs, handler returns to “next” instruction

- Examples are I/O interrupts (Ctrl-C, clicking mouse button) and timer interrupts (every few ms, an external timer chip triggers an interrupt used by the OS kernel to take back control from user programs)

- Synchronous Exceptions are caused by events that occur as a result of executing an instruction

- Traps: Intentional to transfer control to OS to perform some function; returns control to “next” instruction

- Examples: System calls, breakpoint traps, special instructions

- Faults: Unintentional but possibly recoverable; either re-executed faulting instruction or aborts

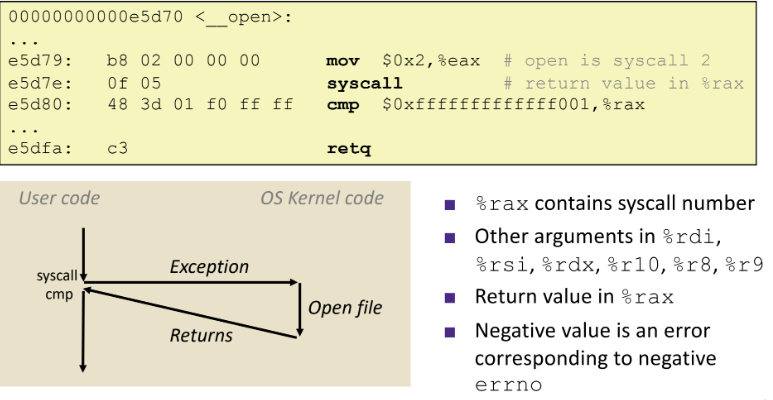
- Examples: Page faults, segment protection faults, divide-by-zero exceptions

- Aborts: Unintentional and unrecoverable; aborts current program

- Examples: Parity error, machine check (hardware failure detected)

- Each system call has a unique ID number

- Traps example: opening file

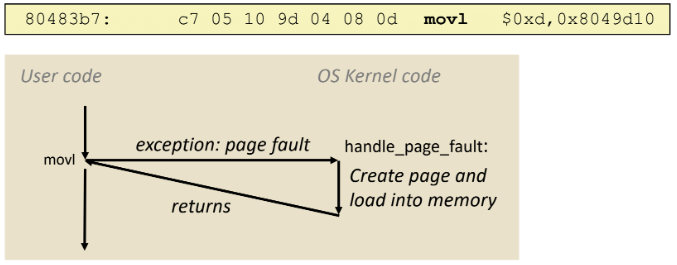
 - User calls open(filename, options); calls \_\_open function, which invokes syscall

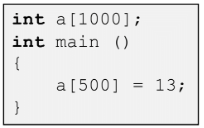
- Fault example: page fault

- User writes to memory location; that portion of memory is currently on disk

- Page fault handler must load page into physical memory

- Returns to faulting instruction: *mov* is executed again and is successful on second try

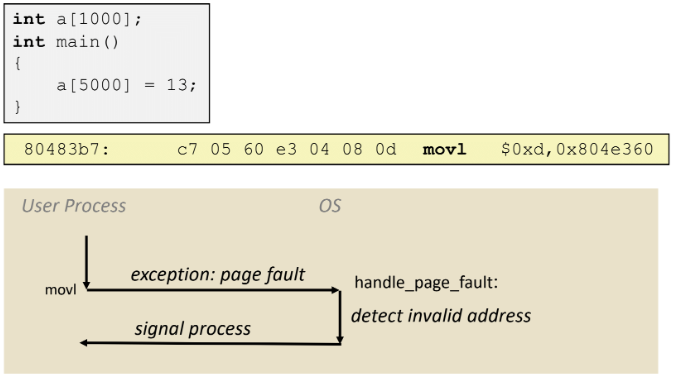




- Fault example: invalid memory reference

- Page fault handler detects invalid address and sends SIGSEGV signal to user process

- User process exits with segmentation fault



- Key points for exceptions

- Events that require non-standard control flow

- Generated externally (interrupts) or internally (traps and faults)

- After it is handled, one of three things may happen: re-execute current, execute next, abort

**Processes and Context Switching**

- A process is another abstraction in the computer system

- Provided by the OS, which uses a data structure to represent each process

- Maintains interface between program and underlying hardware

- Exceptional control flow is the *mechanism* the OS uses to enable multiple processes to run on the same system

- A process is an instance of a running program

- Process provides each program with two key abstractions:

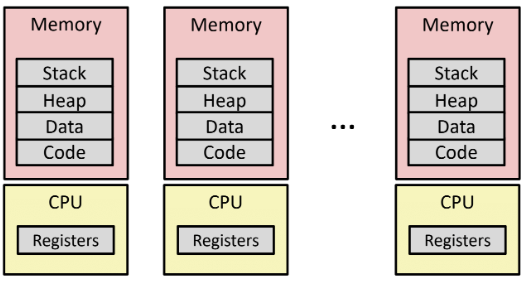
- Logical control flow: each program seems to have exclusive use of the CPU; provided by kernel mechanism called context switching

- Private address space: each program seems to have exclusive use of main memory; provided by kernel mechanism called virtual memory

- Multiprocessing: The illusion

- Computer runs many processes simultaneously

- Applications for one or more users; background tasks

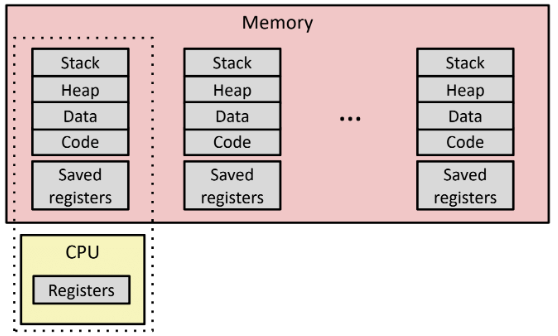


- Multiprocessing: The reality

- Single processor executes multiple processes concurrently

- Process executions interleaved; CPU runs one at a time

- Address spaces managed by virtual memory system

 - Execution context (register values, stack) for other processes saved in memory

- Context Switch: 1) Save current registers in memory; 2) Schedule next process for execution; 3) Load saved registers and switch address space

- Multiprocessing: The modern reality

- Multicore processors: multiple CPUs on single chip

- Share main memory and some of the caches

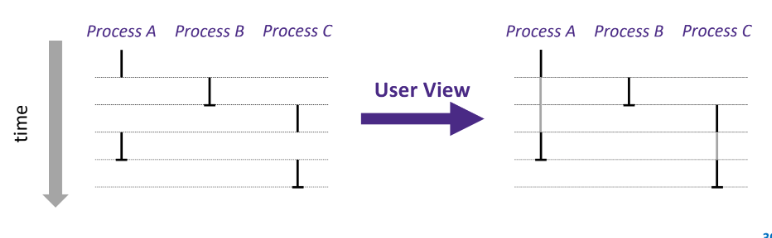
- Each can execute a separate process

- Concurrent Processes

- Each process is a logical control flow

- Two processes run concurrently if there instruction executions overlap in time; otherwise they are sequential

- (Assuming only one CPU) Control flows for concurrent processes are physically disjoint in time; however, the user can think of concurrent processes as executing at the same time, in *parallel*

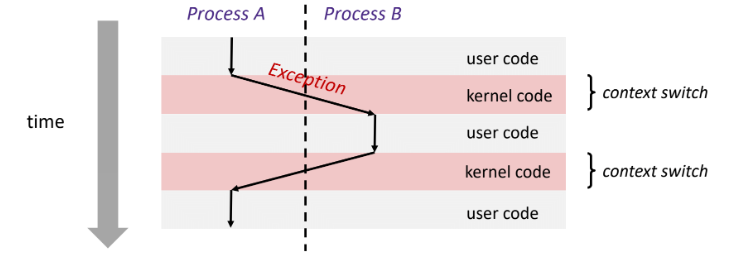


- Processes are managed by a shared chunk of OS code called the kernel

- The kernel is not a separate process, but rather runs as part of a user process

- Kernel virtual memory is memory invisible to user code (very top of stack)

- Context switch passes control flow from one process to another and is performed using kernel code



- Fork-exec model used to create new processes and programs (Linux)

- fork ( ) creates a copy of the current process (make copy of Slack process)

- exec\* ( ) replaces the current process’ code and address space with the code for a different program (turn Slack process into Google Chrome process)

- fork ( ) and execve ( ) are system calls

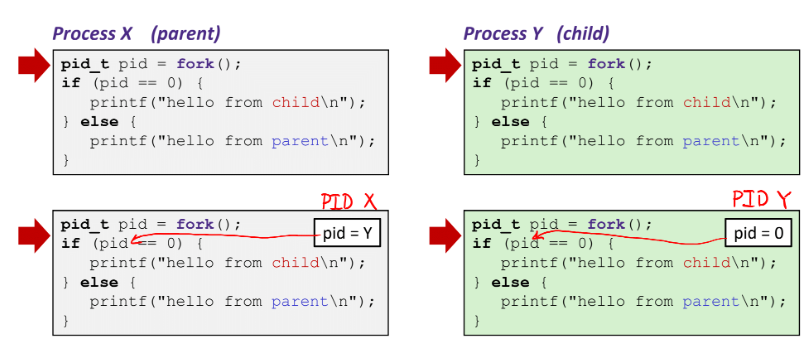
- System calls are traps (intentional exceptions)

- Fork: Creating new processes

- **pid\_t** fork (**void**): creates a new “child” process that is identical to the calling “parent” process, including all state (memory, registers, etc.); returns 0 to the child process and returns child’s process ID (PID) to parent process

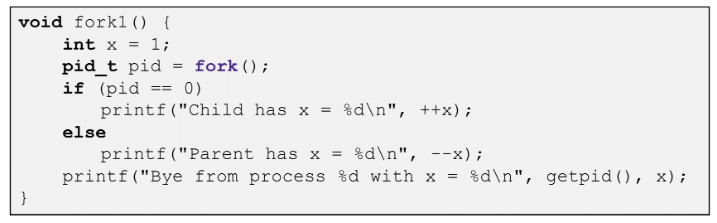
- Child is *almost* identical to parent: child gets an identical (but separate) copy of the parent’s virtual address space; child has a *different PID* than the parent

- fork is unique because it is called once but returns “twice”



- You can’t predict the execution order of parent and child as the two processes are running simultaneously

- Both processes continue/start execution after fork; child starts at instruction after the call to fork (storing into pid)



- This example will print (in no particular order *between* processes):

- “Child has x = 2”

- “Parent has x = 0”

- “Bye from process 0 with x = 2”

- “Bye from process [parent pid] with x = 0”

- Fork can be modeled with process graphs, which capture the partial ordering of statements in a concurrent program

- Each vertex is the execution of a statement

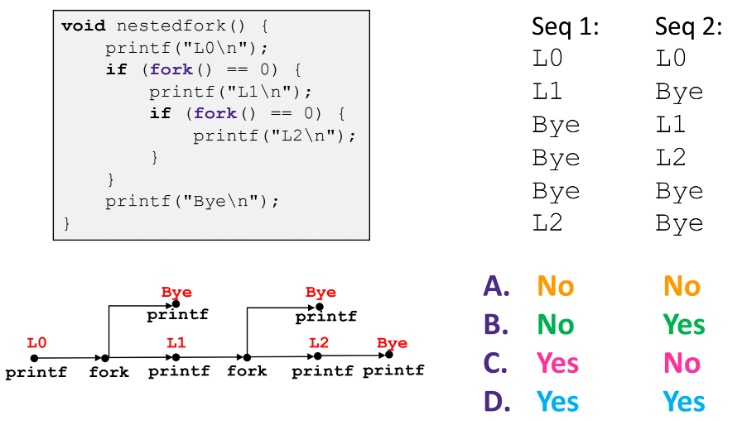
- a 🡪 b means a happens before b

- Edges can be labeled with current value of variables

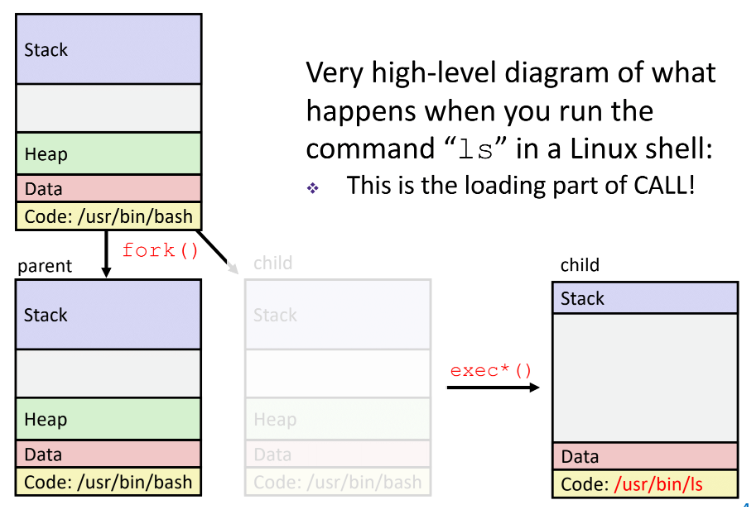
- printf vertices can be labeled with output

- Each graph begins with a vertex with no inedges

- A topological sort of the graph is a total ordering of vertices where all edges point from left to right

 - Any topological sort of the graph corresponds to a feasible total ordering

- Exec-ing a new program example



- Void exit (int status) explicitly exits a process

- 0 is used for a normal exit, nonzero for abnormal exit

- A terminated process still consumes system resources; called a “zombie”

- *Reaping* is performed by parent on terminated child – parent is given exit status information and kernel then deletes zombie child process

- If a parent doesn’t reap, then the orphaned child will be reaped by init process (pid == 1)

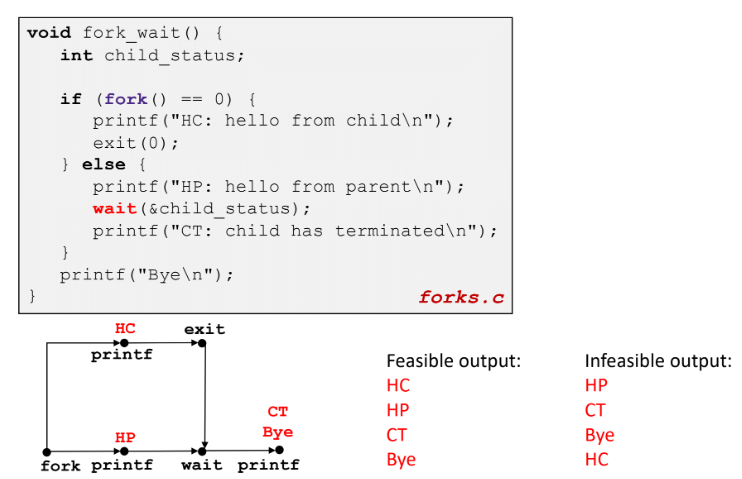
- In long-running processes (shells, servers), we need explicit reaping

- Int wait (int \*child\_status) suspends current process (parent) until one of its children terminates

- Return value is the PID of the child process that terminated; on successful return, the child process is reaped

- If child\_status != NULL, then the \*child\_status value indicates why the child process terminated

- If parent process has multiple children, wait will return when any of the children terminates; *waitpid* can be used to wait on a specific process



- If parent process is in an infinite loop, *ps* shows child process as “defunct”; killing parent allows child to be reaped by init; child is a zombie

- If child is in an infinite loop, it will still be active even after parent has been terminated; must kill explicitly, or else will keep running indefinitely

- Process hierarchy can be viewed using Linux *pstree* command

**Signals**

- A signal is a message that notifies a process that an event has occurred in the system

- Sent from the kernel to a process and is identified by small integer ID’s; akin to exceptions and interrupts

- Kernel sends a signal to a destination process by updating some state in the context of the destination process

- Sends signal either because kernel has detected system event such as divide-by-zero, or another process has invoked the kill system call to request the kernel send a signal

- Destination process receives a signal when it is forced to react to the delivery of the signal by the kernel

- Possible ways to react: ignore the signal, terminate the process, catch the signal by executing function called signal handler