Process Control

Philipp Koehn

23 April 2018

Control Flow

- The CPU executes one instruction after another
- Typically, they are next to each other in memory
  (unless jumps, branches, and returns from subroutine)
- Exceptional Control Flow, triggered by
  - hardware exception
  - software exception

Exceptions

- Interrupts
  - signal from I/O device
  - also: timer interrupts for multi-tasking

Exceptions

- Interrupts
  - signal from I/O device
  - also: timer interrupts for multi-tasking

- Traps and system calls
  - intentional
  - triggered by instruction ("syscall")

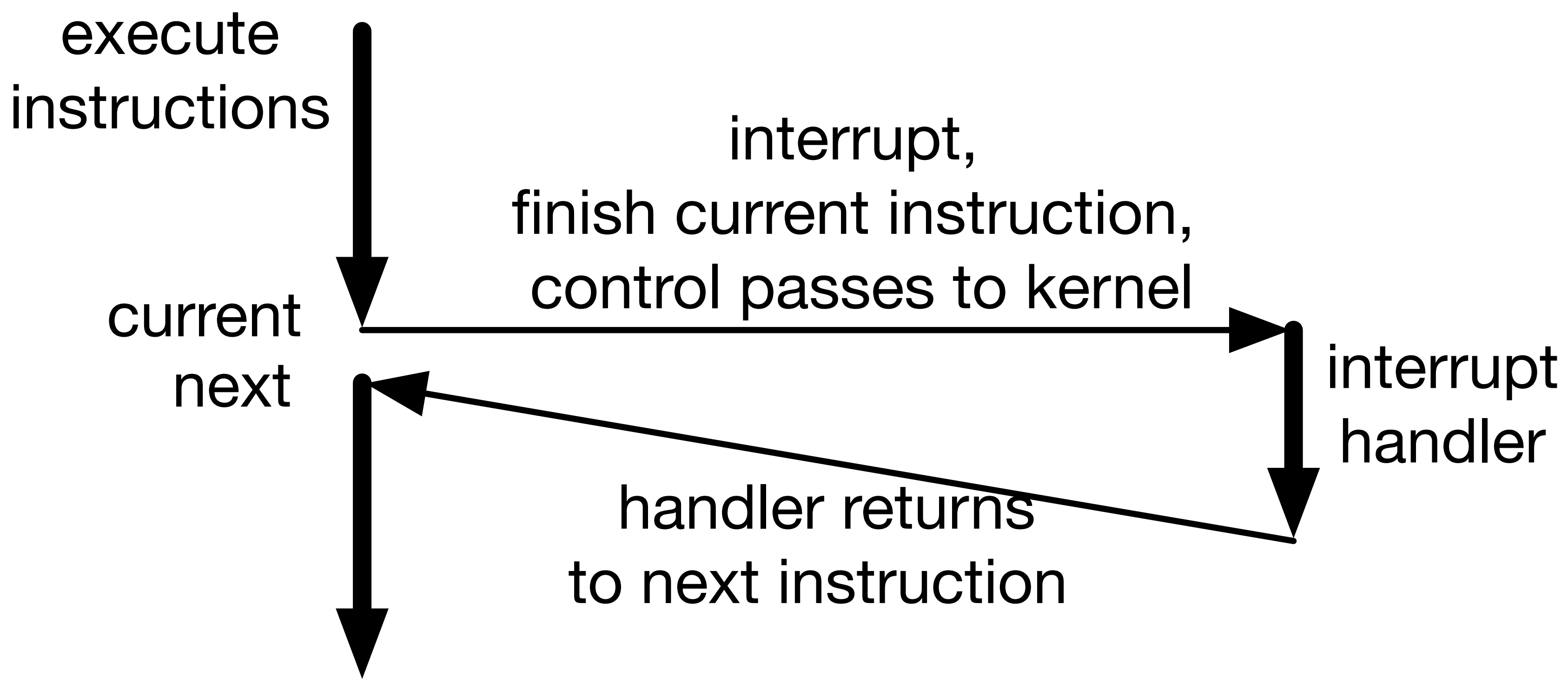
Exceptions

- Interrupts
  - signal from I/O device
  - also: timer interrupts for multi-tasking
- Traps and system calls
  - intentional
  - triggered by instruction ("syscall")
- Faults
  - maybe recoverable, e.g., swapped out memory ("page fault")
  - if recovered, return to regular control flow

Exceptions

- Interrupts
  - signal from I/O device
  - also: timer interrupts for multi-tasking
- Traps and system calls
  - intentional
  - triggered by instruction ("syscall")
- Faults
  - maybe recoverable, e.g., swapped out memory ("page fault")
  - if recovered, return to regular control flow
- Aborts
  - unrecoverable fatal error, e.g., memory corrupted
  - application process is terminated

Abrupt Change in Control Flow



processes

Process

- Exceptions basic building block for processes

- Modern computers seem to run several things at once
  - retrieve and display web pages
  - play music in the background
  - accept emails and alert you to them

Process

- Exceptions basic building block for processes

- Modern computers seem to run several things at once
  - retrieve and display web pages
  - play music in the background
  - accept emails and alert you to them

- Process := a running program
  - appears to have full access to memory
  - appears to run without interruptions

Process

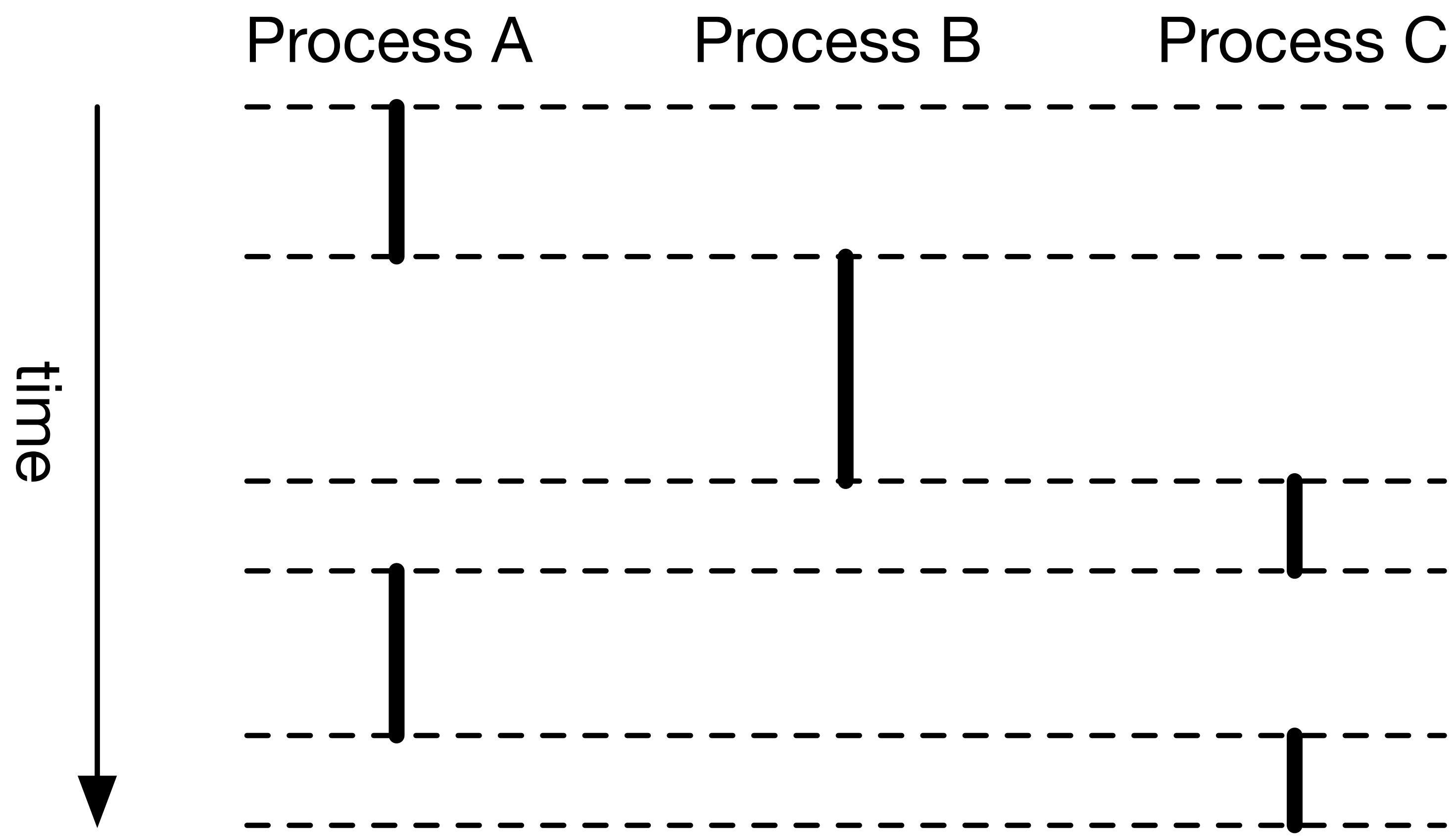
- Exceptions basic building block for processes

- Modern computers seem to run several things at once
  - retrieve and display web pages
  - play music in the background
  - accept emails and alert you to them

- Process := a running program
  - appears to have full access to memory
  - appears to run without interruptions

- Multi-tasking: modern OS that allow multiple processes at once

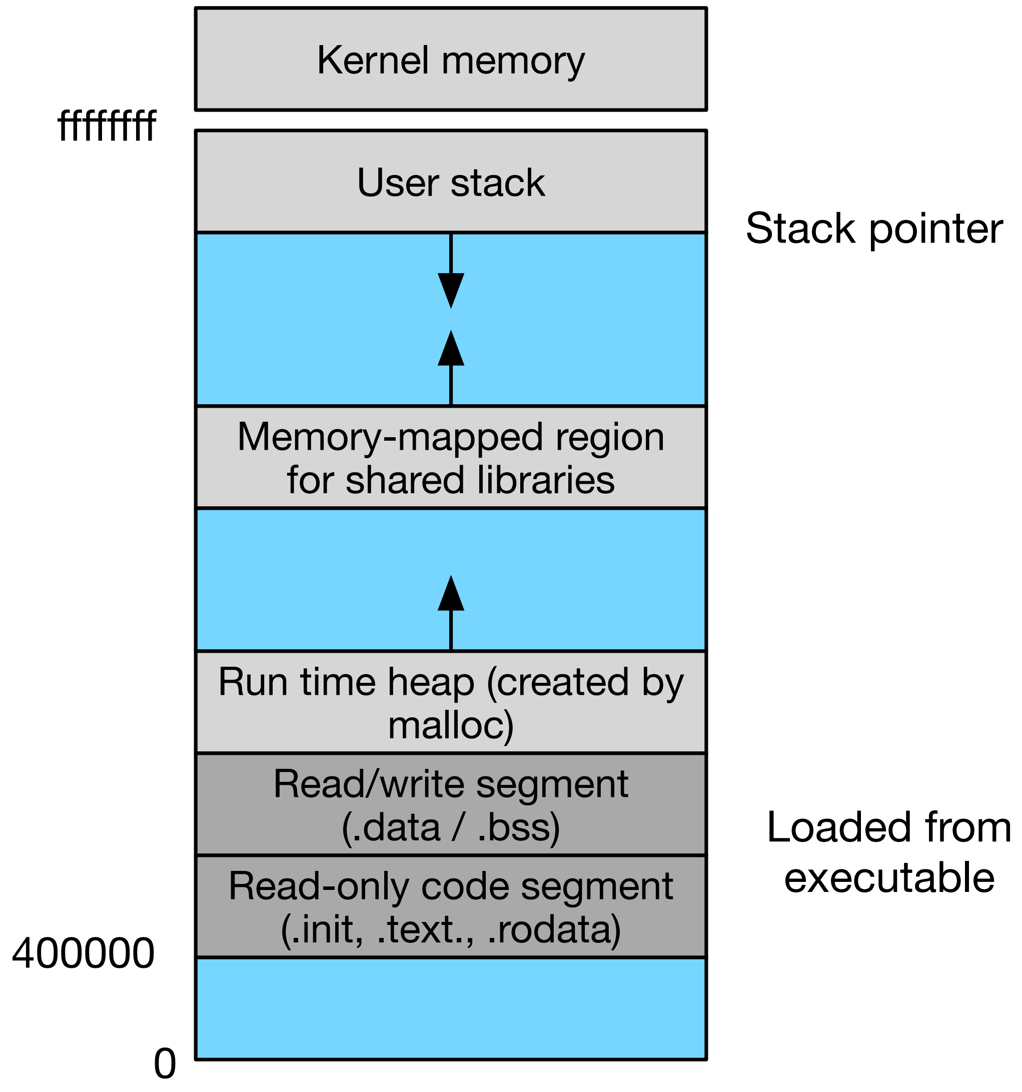
Logical Control Flow



User and Kernel Mode

- Mode bit in control register
- Kernel mode: may execute any instruction, access any memory
- User mode: limited to private memory
- Switch from user to kernel mode
  - voluntary (sleep)
  - triggered by interrupt
  - system call

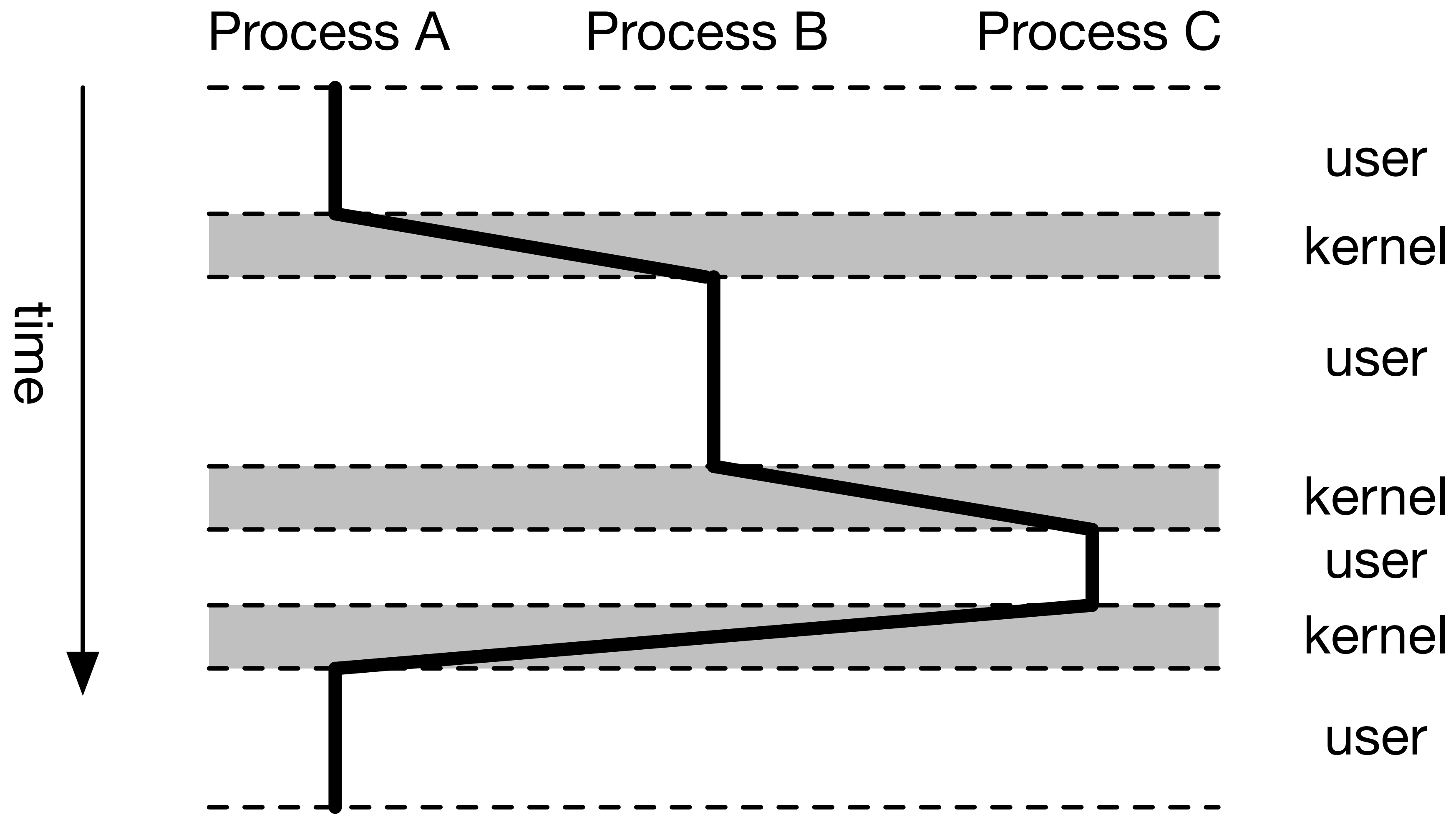
Private Address Space


Kernel memory
ffffffff
User stack
Stack pointer
Memory-mapped region for shared libraries
Run time heap (created by malloc)
Read/write segment (.data / .bss)
Read-only code segment (.init, .text., .rodata)
Loaded from executable
400000
0

Process Context

- Kernel maintains context for each process
- Context
  - program counter
  - register values
  - address table (more on that next lecture)
  - opened files
  - various meta information (e.g., process name)
- In Linux, each process context viewable in /proc "file" system

Context Switches



system calls

Examples

Number	Name	Description
0	read	read from file
1	write	write to file
2	open	open file
3	close	close file
33	pause	suspend process until signal arrives
39	getpid	get process id
57	fork	create new process
60	exit	end process
61	wait4	wait for a process to terminate
62	kill	kill another process

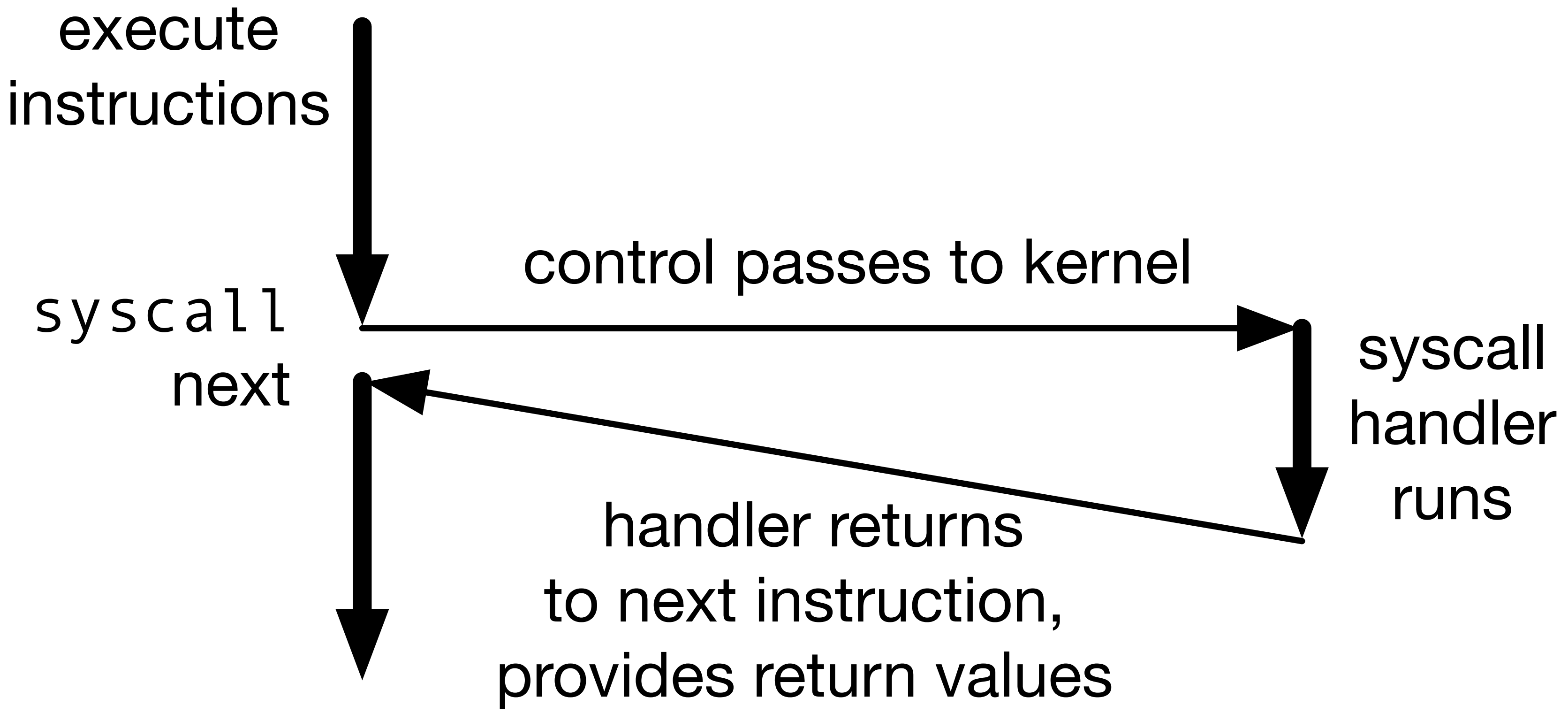
Assembly Example

```
.section .data
string:
    .ascii "hello, world!\n"
string_end:
    .equ len, string_end - string

.section .text
.globl main
main:
    movq $1, %rax           ; write is system call 1
    movq $1, %rdi           ; arg1: stdout is "file" 1
    movq string, %rsi       ; arg2: hello world string
    movq len, %rdx          ; arg3: length of string
    syscall

    movq $60, %rax          ; exit is system call 60
    movq $0; %rdi           ; exit status
    syscall
```

System Call Control



process control

Creating New Processes

- C code than spawns a child process

```
int main() {
  int x = 1;
  pid_t  pid = fork();

  if (pid == 0) {
    printf("child x=%d", ++x);
    exit(0);
  }
  printf("parent x=%d", --x);
  exit(0);
}
```

- When run, it returns

```
parent x=0
child x=2
```

Syscall 57: Fork

- fork() creates a child process
- Call once, return twice
  - in child process: return value 0
  - in parent process: return value is process id of child

Syscall 57: Fork

- fork() creates a child process
- Call once, return twice
  - in child process: return value 0
  - in parent process: return value is process id of child
- Concurrent exception
  - parent and child processes run concurrently
  - no guarantee which proceeds first (and for how long)

Syscall 57: Fork

- fork() creates a child process
- Call once, return twice
  - in child process: return value 0
  - in parent process: return value is process id of child
- Concurrent exception
  - parent and child processes run concurrently
  - no guarantee which proceeds first (and for how long)
- Duplicate by separate address space
  - initially memory is identical
  - each process makes changes to its private copy

Another Example

- Multiple forks

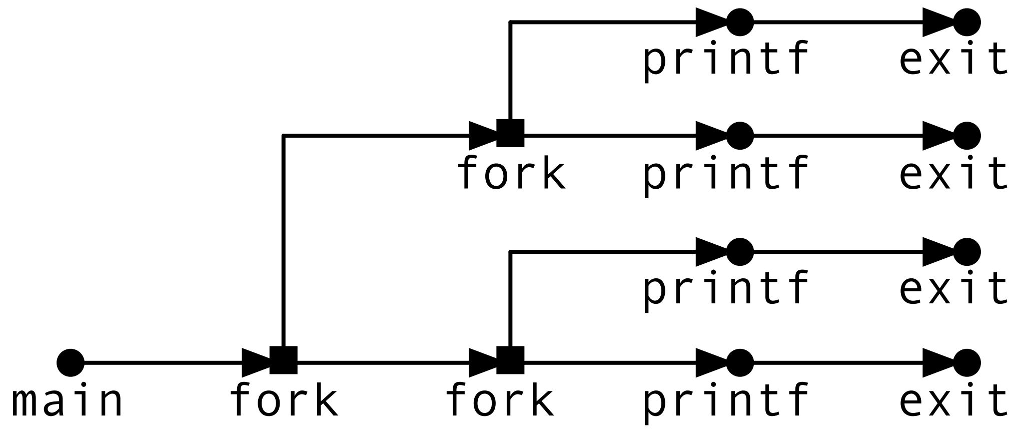
```
int main() {
  fork();
  fork();
  printf("hello\n");
  exit(0);
}
```

Another Example

- Multiple forks

```
int main() {
  fork();
  fork();
  printf("hello\n");
  exit(0);
}
```

- Outputs "hello" 4 times



Death in the Family

- What happens when what dies when?
- Child process dies
  - process still in kernel's process table
  - waiting for parent to read exit status
  - "zombie": dead, but still active
- Parent process dies
  - children processes become orphaned
  - orphan killing: terminate all orphaned processes
  - re-parenting: make init process (pid: 1) parent
    ($\rightarrow$ a "daemon" process)

Waiting for Child to Die

1. Parent spawns child process
2. Both processes running
3. Parent waits for child to complete
   - C: waitpid()
   - Assembly: syscall 61
4. Parent stalls
5. Child dies (zombie)
6. Parent receives exit status of child
7. Child dies completely

Exec

- Parent process may execute another program
  - C: execve(filename, argv, envp)
  - Assembly: syscall 59
- Passes environment variables (envp)
- Executed command takes over
- If both should run: fork first

`signals`

Signals

- Software-level communication between processes

- Sending the signal from one process

- Receiving the signal by another process
  - ignore
  - terminate
  - catch signal

- Handled by kernel

Examples

Number	Name	Default	Corresponding Event
1	SIGHUP	terminate	Terminate line hangup
2	SIGINT	terminate	Interrupt from keyboard
3	SIGUIT	terminate	quit from keyboard
4	SIGILL	terminate	illegal instruction
5	SIGTRAP	terminate & dump core	trace trap
9	SIGKILL	terminate*	kill process
18	SIGCONT	ignore	continue process if stopped
19	SIGSTOP	stop until SIGCONT*	stop signal not from terminal
20	SIGTSTP	stop until SIGCONT	stop signal from terminal

* = SIGKILL and SIGSTOP cannot be caught

Sending Signals

- From shell with command

```
linux> /bin/kill -9 2423
```

- From shell with keystroke to running process

```
linux> start-my-process
CTRL+C
```

  - CTRL+C: sends SIGINT
  - CTRL+Z: sends SIGTSTP

- There is also a C function and an Assembly syscall

Receiving Signals

- When kernel about to continue process, checks for signals
- If there is a signal, forces process to receive signal
- Each signal has a default action
  - ignore
  - terminate
  - terminate and dump core
  - stop
- Process can also set up a signal handler for customized response

Signal Handler

- Signal handler in C

```
#include "csapp.h"

void sigInt_handler(int sig) {
  printf("Caught SIGINT\n");
  exit(0);
}

int main() {
  signal(SIGINT, sigint_handler);
  pause();
  return 0;
}
```

- Now, process writes "Caught SIGINT" to stdout before terminating