Exceptional Control Flow: Exceptions and Processes

CSE4100: Multicore Programming

Sungyong Park (PhD)
Data-Intensive Computing and Systems Laboratory (DISCOS)
https://discos.sogang.ac.kr
Office: R908A, E-mail: parksy@sogang.ac.kr

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

- **Up to now: two mechanisms for changing control flow:**
  - Jumps and branches
  - Call and return

  React to changes in ***program state***

- **Insufficient for a useful system:**
  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

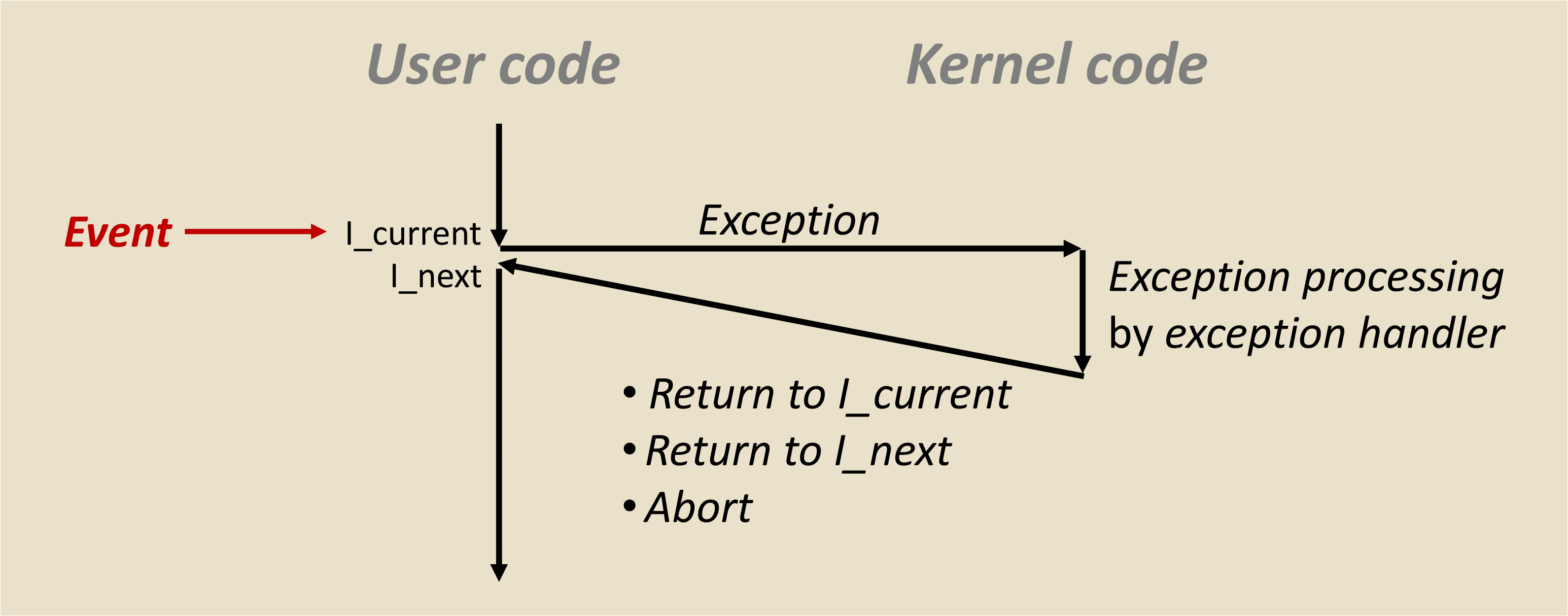
- **System needs mechanisms for "exceptional control flow"**

Exceptional Control Flow

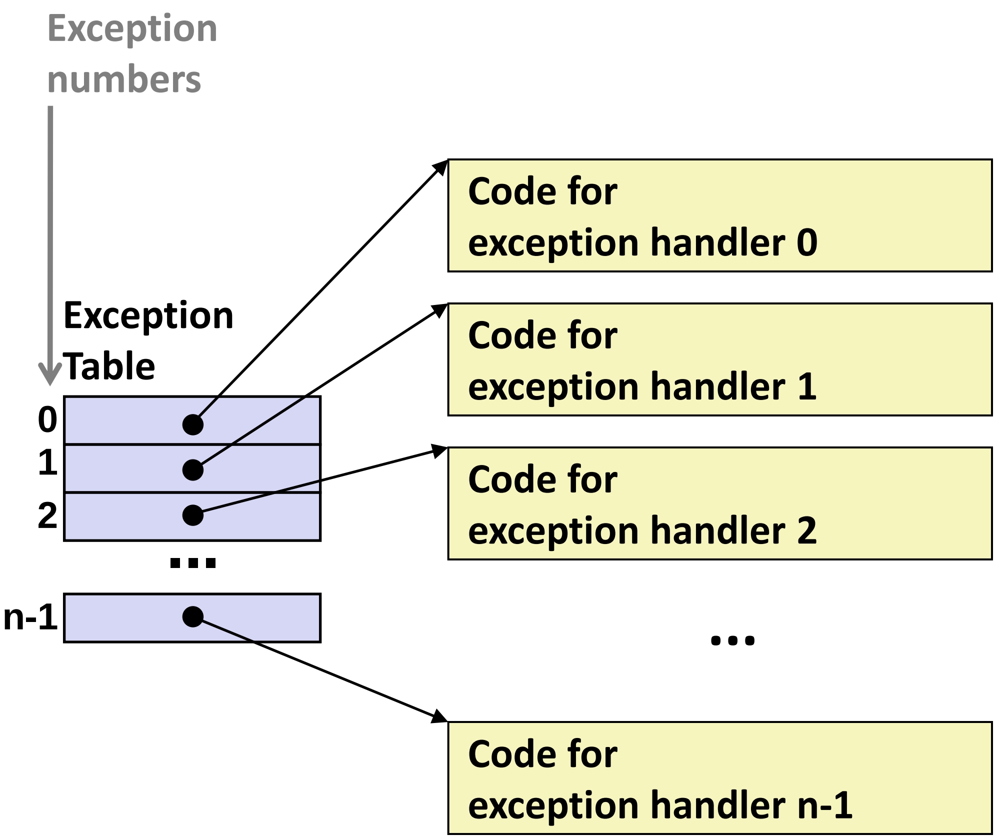
- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context switch**
    - Implemented by OS software and hardware timer
  - 3. **Signals**
    - Implemented by OS software
  - 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 *event* (i.e., change in processor state)**
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table
(a.k.a. interrupt vector)
- Handler k is called each time
exception k occurs

Classes of Exceptions

- Interrupts, Traps, Faults, and Aborts

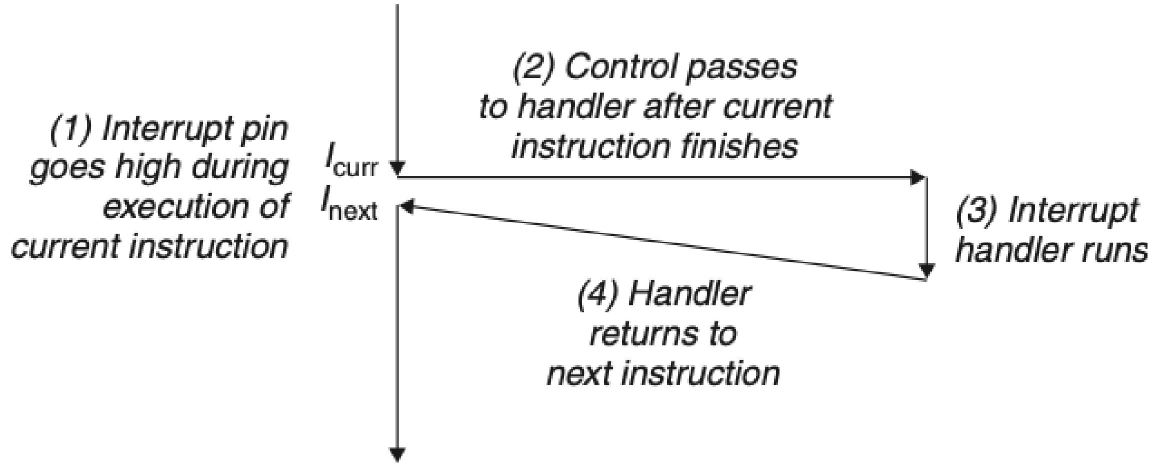
Class	Cause	Async/sync	Return behavior
Interrupt	Signal from I/O device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	Always returns to next instruction
Fault	Potentially recoverable error	Sync	Might return to current instruction
Abort	Nonrecoverable error	Sync	Never returns

Figure 8.4 **Classes of exceptions.** Asynchronous exceptions occur as a result of events in I/O devices that are external to the processor. Synchronous exceptions occur as a direct result of executing an instruction.

Asynchronous Exceptions (Interrupts)

- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

Figure 8.5
Interrupt handling. The interrupt handler returns control to the next instruction in the application program's control flow.



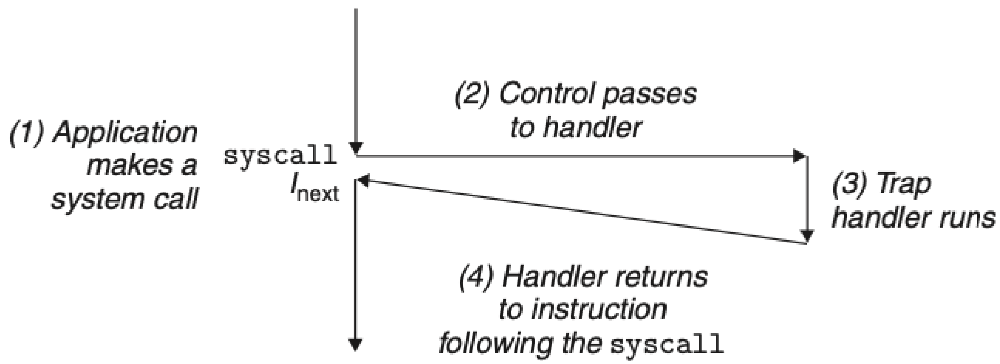
Asynchronous Exceptions (Interrupts)

- **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
    - Arrival of a packet from a network
    - Arrival of data from a disk

Synchronous Exceptions

- **Caused by events that occur as a result of executing an instruction:**
  - ***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

Figure 8.6
Trap handling. The trap handler returns control to the next instruction in the application program's control flow.



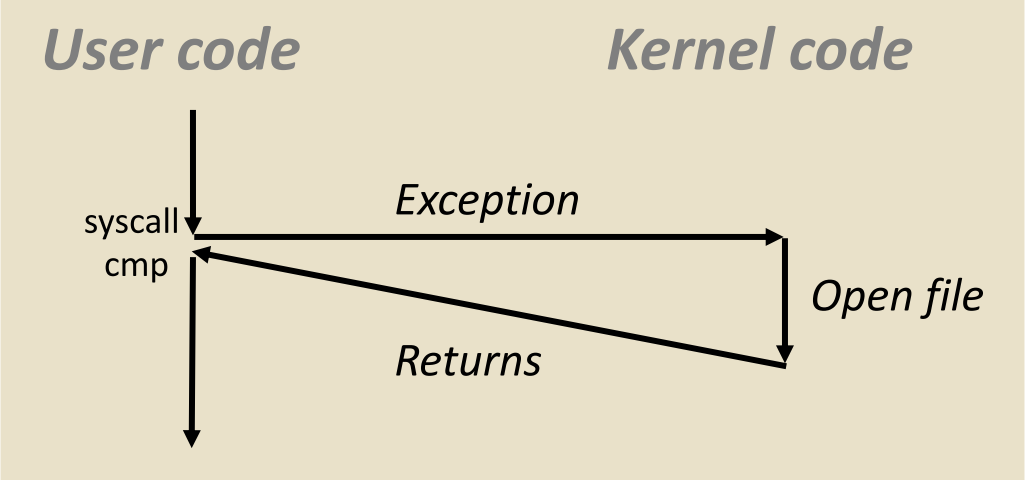
Synchronous Exceptions (Traps)

Examples: System calls

System Call Example: Opening File

- User calls: **`open(filename, options)`**
- Calls **`__open`** function, which invokes system call instruction **`syscall`**

```
00000000000e5d70 <__open>:
...
e5d79:   b8 02 00 00 00      mov  $0x2,%rax  # 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
```



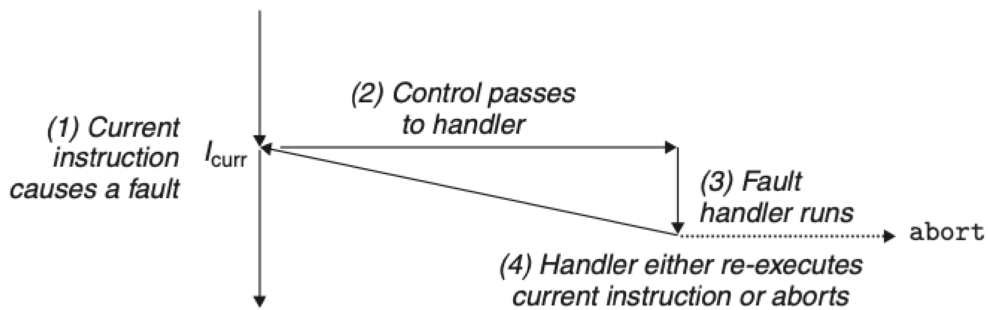
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`

System Call Number

- Each x86-64 system call has a unique ID number
- Examples:

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

Figure 8.7
Fault handling.
Depending on whether the fault can be repaired or not, the fault handler either re-executes the faulting instruction or aborts.



Synchronous Exceptions (Fault Handling)

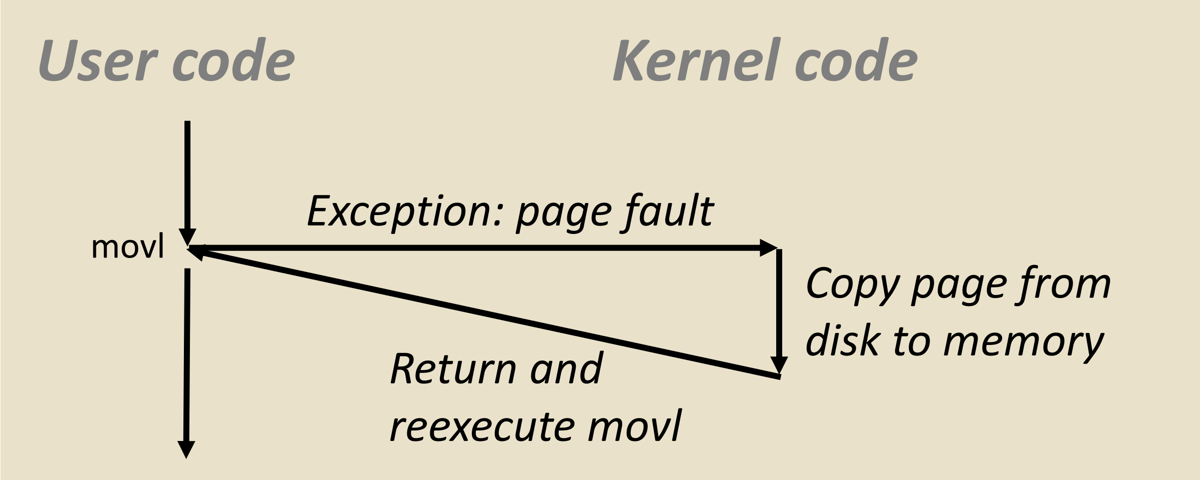
Examples: Page faults

Fault Example: Page Fault

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

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

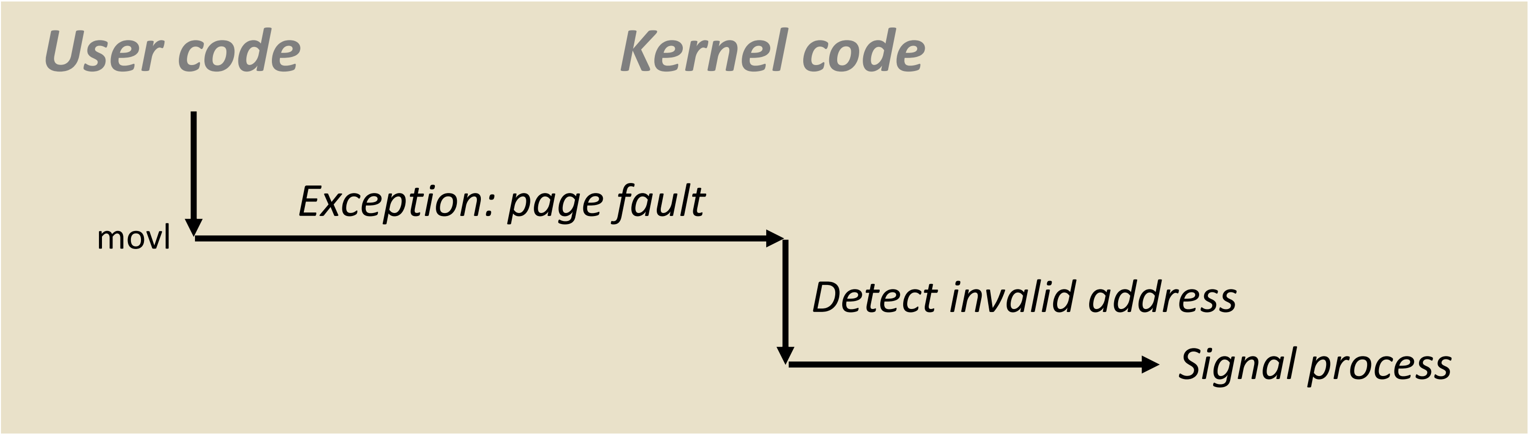
```
80483b7:      c7 05 10 9d 04 08 0d    movl    $0xd,0x8049d10
```



Fault Example: Invalid Memory Reference

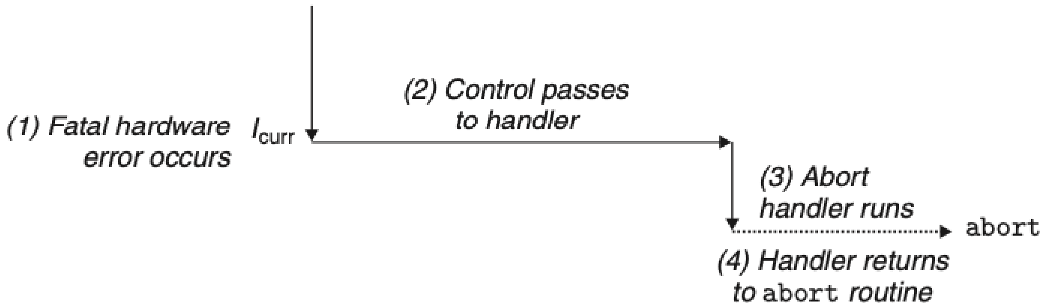
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

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



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

Figure 8.8
Abort handling. The abort handler passes control to a kernel **abort** routine that terminates the application program.



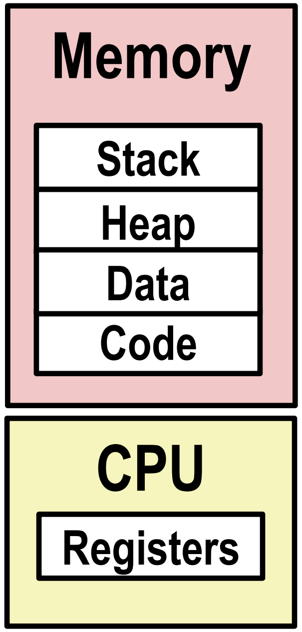
Synchronous Exceptions (Abort Handling)

Examples: Hardware errors such as parity errors that occur when DRAM or SRAM bits are corrupted.

Processes

- **Definition: A *process* is an instance of a running program.**
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- **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*



Control Flow

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

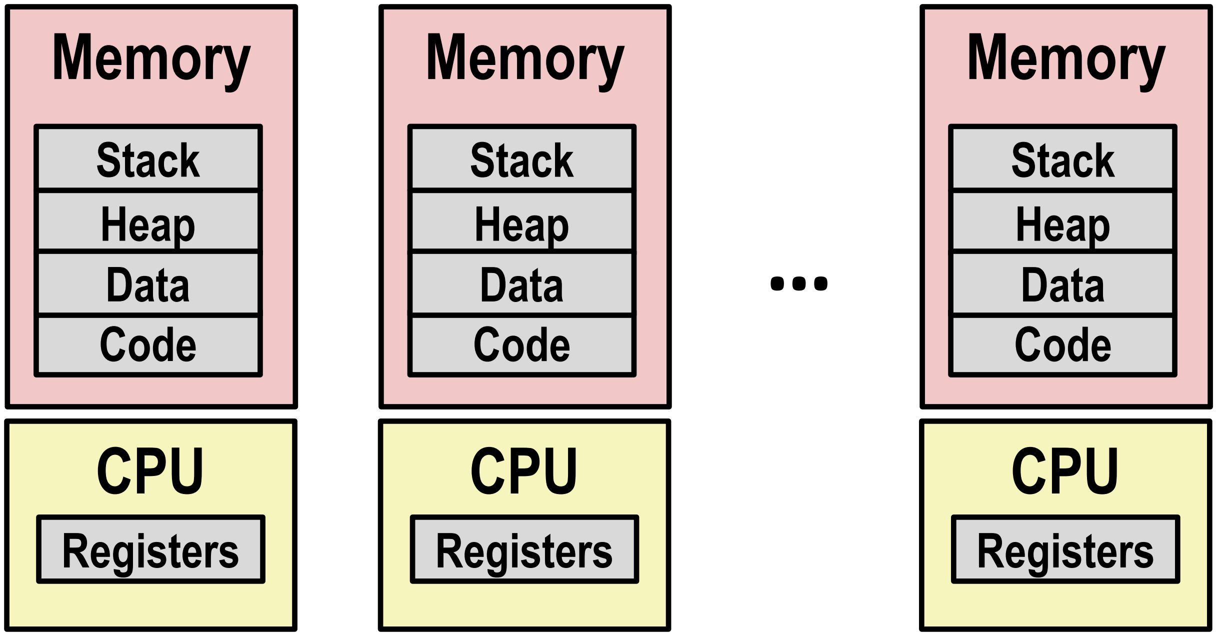
Physical control flow

Time

<startup>
inst_1
inst_2
inst_3
…
inst_n
<shutdown>

* many modern CPUs execute several instructions at once and/or out of program order, but this is invisible to the programmer

Multiprocessing: The Illusion



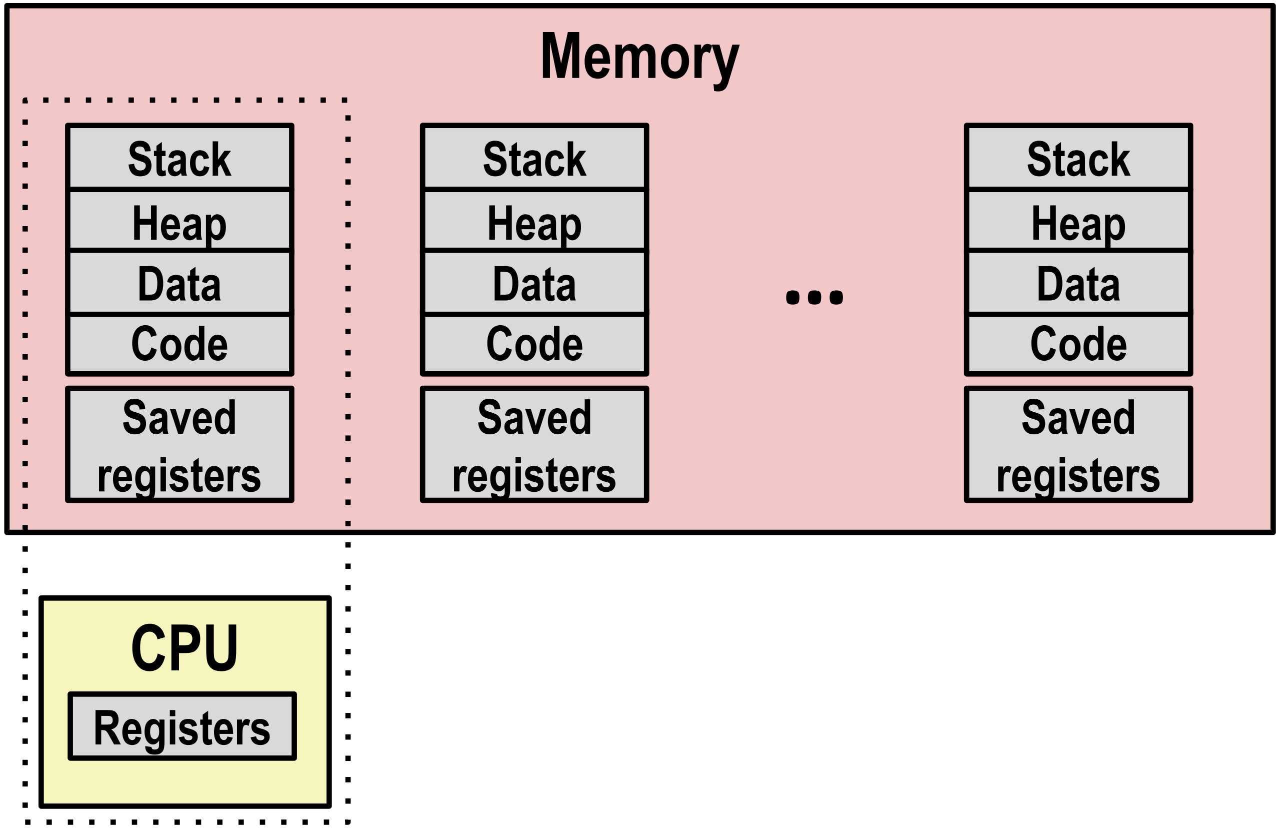
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example

```
xterm
Processes: 123 total, 5 running, 9 stuck, 109 sleeping, 611 threads          11:47:07
Load Avg: 1.03, 1.13, 1.14  CPU usage: 3.27% user, 5.15% sys, 91.56% idle
SharedLibs: 576K resident, 0B data, 0B linkedit.
MemRegions: 27958 total, 1127M resident, 35M private, 494M shared.
PhysMem: 1039M wired, 1974M active, 1062M inactive, 4076M used, 18M free.
VM: 280G vsize, 1091M framework vsize, 23075213(1) pageins, 5843367(0) pageouts.
Networks: packets: 41046228/11G in, 66083096/77G out.
Disks: 17874391/349G read, 12847373/594G written.

PID    COMMAND      %CPU TIME     #TH  #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE
99217- Microsoft Of 0.0  02:28.34 4    1    202   418   21M    24M    21M    66M    763M
99051  usbmuxd      0.0  00:04.10 3    1    47    66    436K   216K   480K   60M    2422M
99006  iTunesHelper 0.0  00:01.23 2    1    55    78    728K   3124K  1124K  43M    2429M
84286  bash         0.0  00:00.11 1    0    20    24    224K   732K   484K   17M    2378M
84285  xterm        0.0  00:00.83 1    0    32    73    656K   872K   692K   9728K  2382M
55939- Microsoft Ex 0.3  21:58.97 10   3    360   954   16M    65M    46M    114M   1057M
54751  sleep        0.0  00:00.00 1    0    17    20    92K    212K   360K   9632K  2370M
54739  launchdadd   0.0  00:00.00 2    1    33    50    488K   220K   1736K  48M    2409M
54737  top          6.5  00:02.53 1/1  0    30    29    1416K  216K   2124K  17M    2378M
54719  automountd   0.0  00:00.02 7    1    53    64    860K   216K   2184K  53M    2413M
54701  ocspd        0.0  00:00.05 4    1    61    54    1268K  2644K  3132K  50M    2426M
54661  Grab         0.6  00:02.75 6    3    222+  389+  15M+   26M+   40M+   75M+   2556M+
54659  cookied      0.0  00:00.15 2    1    40    61    3316K  224K   4088K  42M    2411M
53818  mdworker     0.0  00:01.67 4    1    52    91    7628K  7412K  16M    48M    2438M
50878  mdworker                                   91    2464K  6148K  9976K  44M    2434M
                                                  73    280K   872K   532K   9700K  2382M
50078  emacs        0.0  00:06.70 1    0    20    35    52K    216K   88K    18M    2392M
```

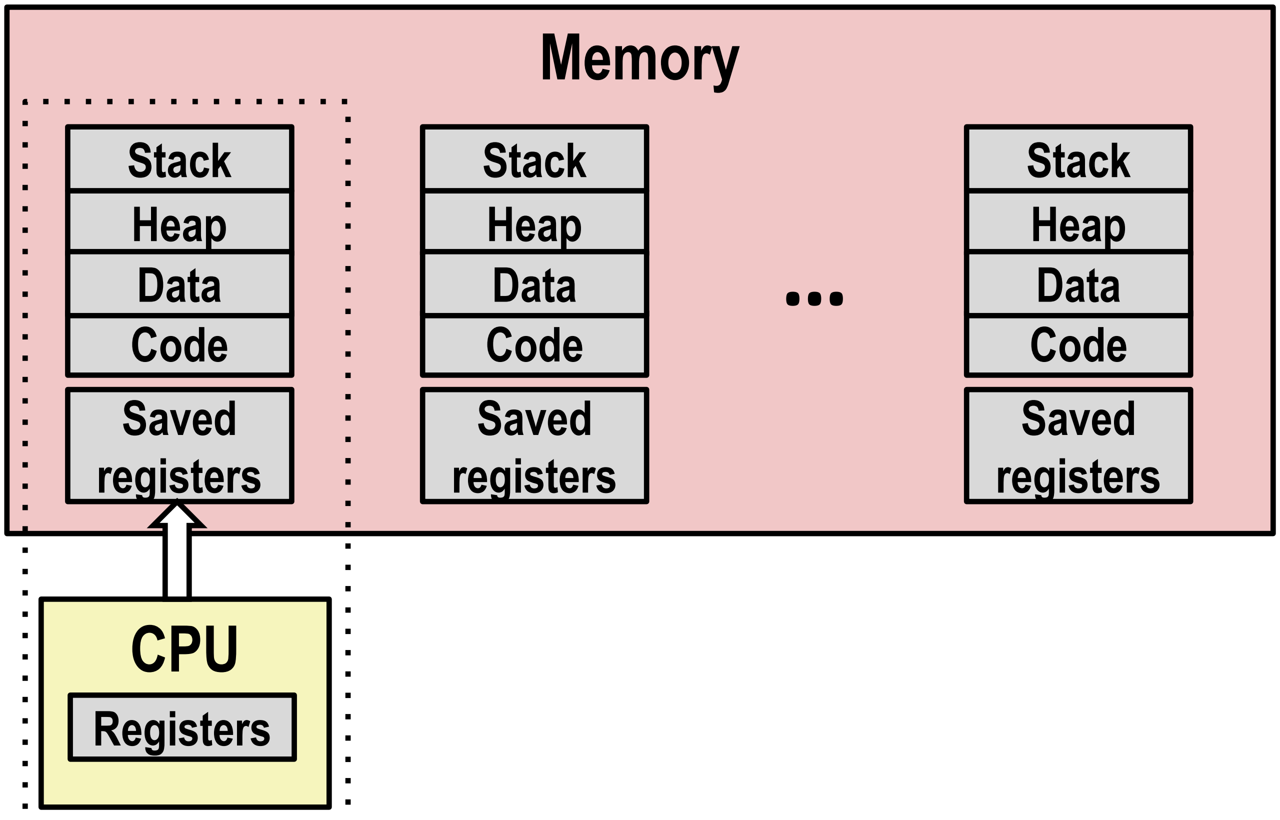
- **Running program "top" on Mac**
  - System has 123 processes, 5 of which are active
  - Identified by Process ID (PID)

Multiprocessing: Single CPU



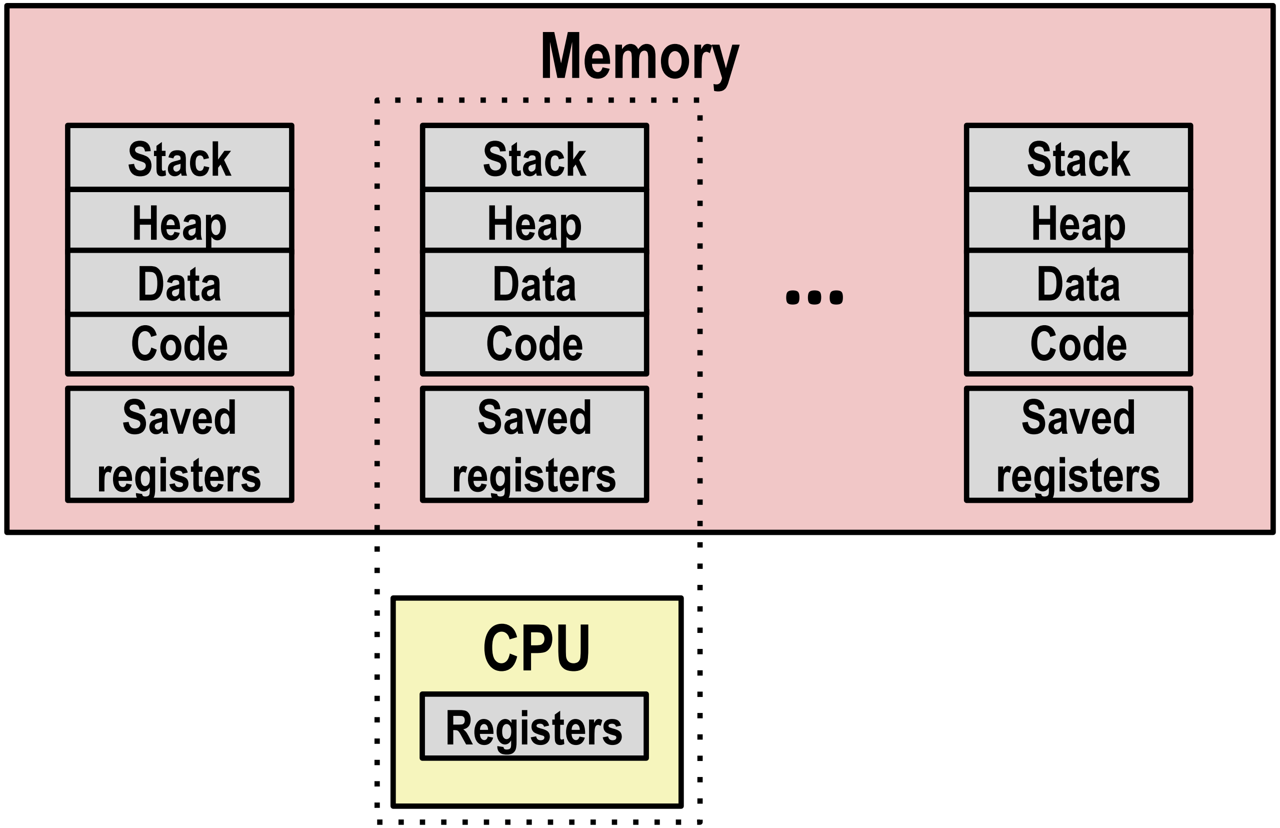
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system
  - Register values for nonexecuting processes saved in memory

Multiprocessing: Single CPU



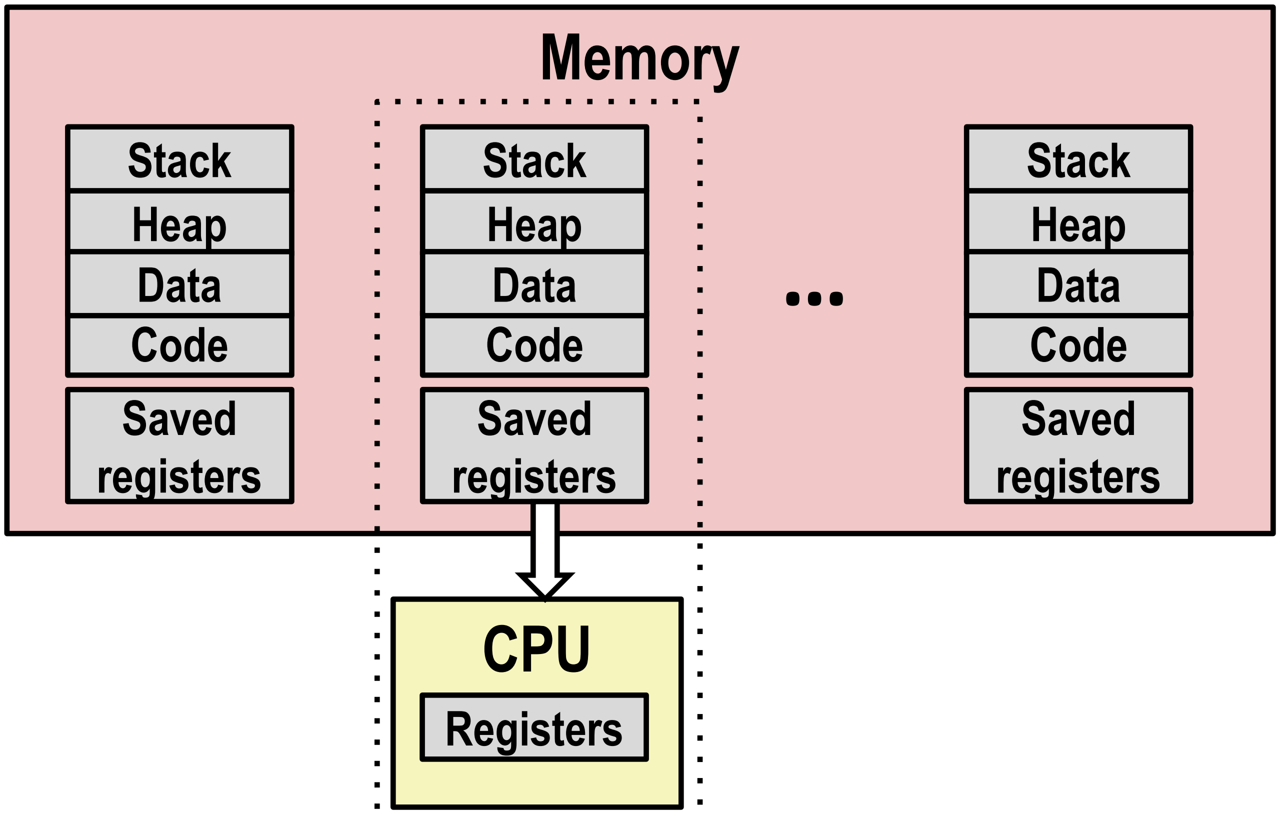
- Save current registers in memory

Multiprocessing: Single CPU



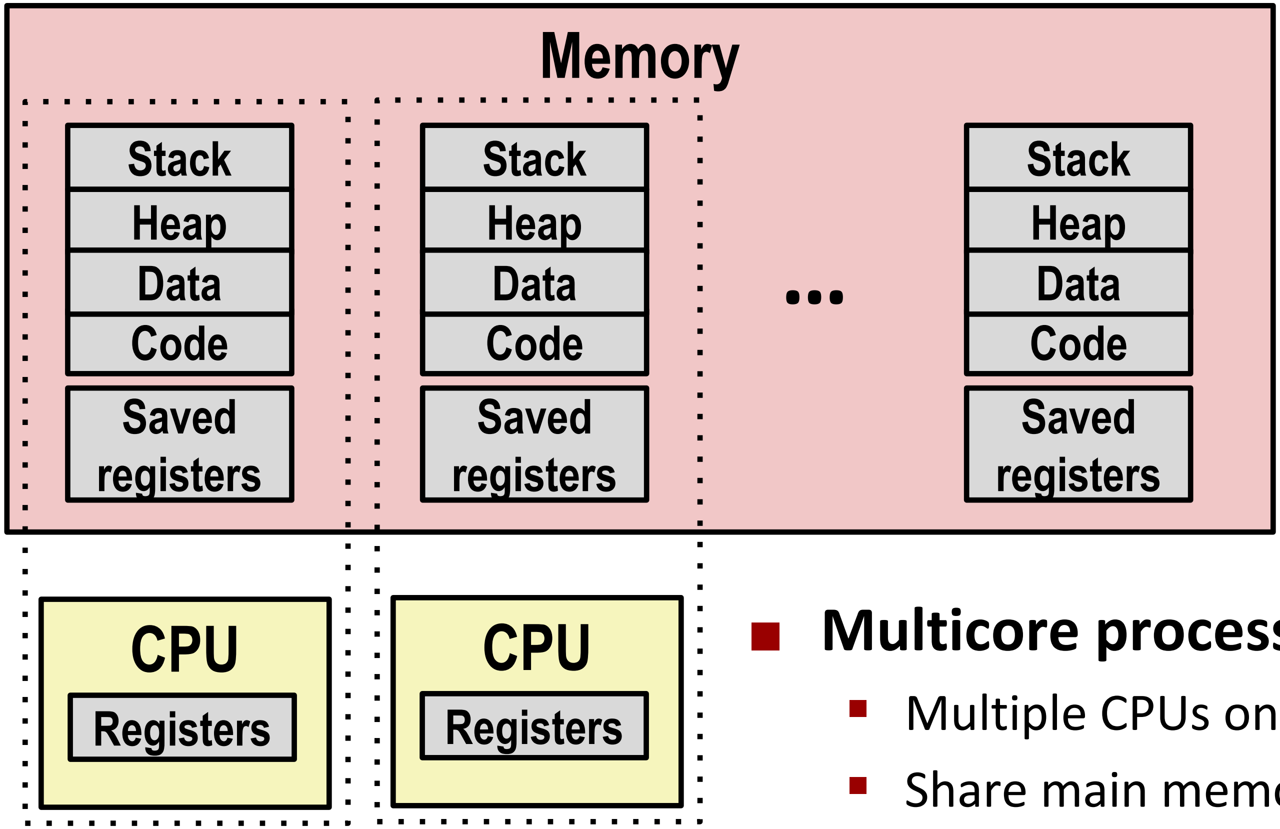
- Schedule next process for execution

Multiprocessing: Single CPU



- Load saved registers and switch address space (context switch)

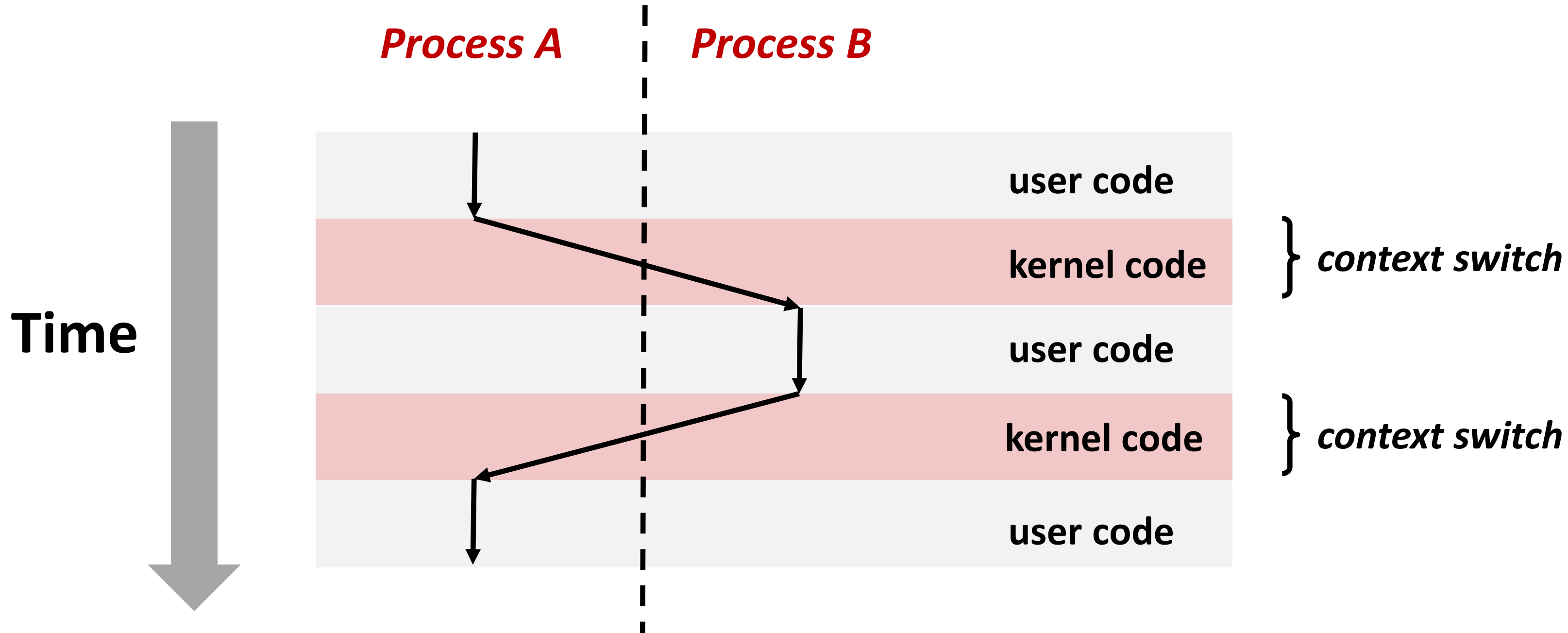
Multiprocessing: Multiple CPUs (Core)



- **Multicore processors**
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processes onto cores done by kernel

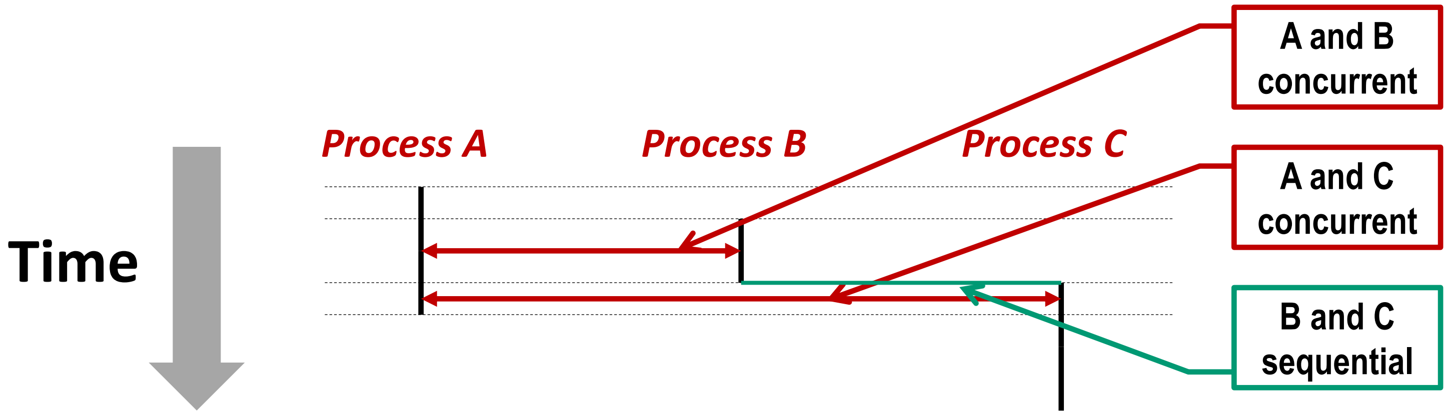
Context Switching

- **Processes are managed by a shared chunk of memory-resident OS code called the *kernel***
  - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- **Control flow passes from one process to another via a *context switch***



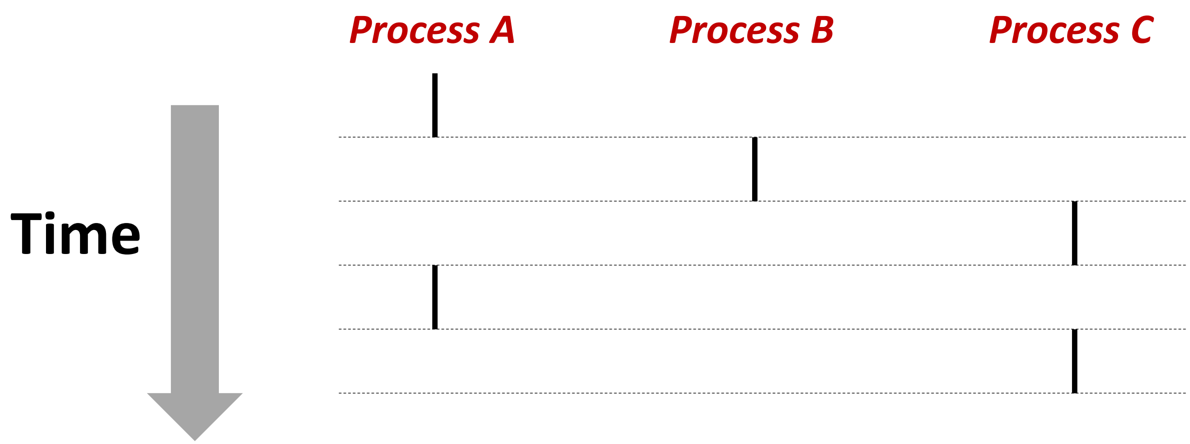
User View of Concurrent Processes

- Two processes *run* ***concurrently*** (*are concurrent)* if their execution overlaps in time
- Otherwise, they are ***sequential***
- Appears as if concurrent processes run in parallel with each other
  - This means they can interfere with each other (e.g., synchronization) (more on that in a couple weeks)



Traditional Reality: Single CPU Case

- Only one process runs at a time
- A and B execution is *interleaved*, not truly concurrent
- Similarly for A and C
- Still possible for A and B / A and C to interfere with each other



System Calls (Revisted)

- **Whenever a program wants to cause an effect outside its own process, it must ask the kernel for help**
- **Examples:**
  - Read/write files
  - Get current time
  - Allocate RAM (sbrk)
  - Create new processes

```
// fopen.c
FILE *fopen(const char *fname,
            const char *mode) {
  int flags = mode2flags(mode);
  if (!flags) return NULL;
  int fd = open(fname, flags,
                DEFPERMS);
  if (fd == -1) return NULL;
  return fdopen(fd, mode);
}

// open.S
    .global open
open:
    mov $SYS_open, %eax
    syscall
    cmp $SYS_error_thresh, %rax
    ja  __syscall_error
    ret
```

All the System Calls

accept
accept4
acct
add_key
adjtimex
bind
bpf
brk
capget
capset
chdir
chroot
clock_adjtime
clock_getres
clock_gettime
clock_nanosleep
clock_settime
clone
clone3
close
close_range
connect
copy_file_range
delete_module
dup
dup3
epoll_create1
epoll_ctl
epoll_pwait
epoll_pwait2
eventfd2
execve
execveat
exit
exit_group
faccessat
faccessat2
fallocate
fanotify_init
fanotify_mark
fchdir
fchmod
fchmodat
fchown
fchownat
fdatasync
fgetxattr
finit_module
flistxattr
flock
fremovexattr
fsconfig
fsetxattr
fsmount
fsopen
fspick
fsync
futex
futex_waitv
get_mempolicy
get_robust_list
getcpu
getcwd
getdents64
getegid
geteuid
getgid
getgroups
getitimer
getpeername
getpgid
getpid
getppid
getpriority
getrandom
getresgid
getresuid
getrlimit
getrusage
getsid
getsockname
getsockopt
gettid
gettimeofday
getuid
getxattr
init_module
inotify_add_watch
inotify_init1
inotify_rm_watch
io_cancel
io_destroy
io_getevents
io_pgetevents
io_setup
io_submit
io_uring_enter
io_uring_register
io_uring_setup
ioctl
ioprio_get
ioprio_set
kcmp
kexec_file_load
kexec_load
keyctl
kill
landlock_add_rule
landlock_create_ruleset
landlock_restrict_self
lgetxattr
linkat
listen
listxattr
llistxattr
lookup_dcookie
lremovexattr
lsetxattr
madvise
mbind
membarrier
memfd_create
memfd_secret
migrate_pages
mincore
mkdirat
mknodat
mlock
mlock2
mlockall
mount
mount_setattr
move_mount
move_pages
mprotect
mq_getsetattr
mq_notify
mq_open
mq_timedreceive
mq_timedsend
mq_unlink
mremap
msgctl
msgget
msgrcv
msgsnd
msync
munlock
munlockall
munmap
name_to_handle_at
nanosleep
nfsservctl
open_by_handle_at
open_tree
openat
openat2
perf_event_open
personality
pidfd_getfd
pidfd_open
pidfd_send_signal
pipe2
pivot_root
pkey_alloc
pkey_free
pkey_mprotect
ppoll
prctl
pread64
preadv
preadv2
prlimit64
process_madvise
process_mrelease
process_vm_readv
process_vm_writev
pselect6
ptrace
pwrite64
pwritev
pwritev2
quotactl
quotactl_fd
read
readahead
readlinkat
readv
reboot
recvfrom
recvmmsg
recvmsg
remap_file_pages
removexattr
renameat
renameat2
request_key
restart_syscall
rseq
rt_sigaction
rt_sigpending
rt_sigprocmask
rt_sigqueueinfo
rt_sigreturn
rt_sigsuspend
rt_sigtimedwait
rt_tgsigqueueinfo
sched_get_priority_max
sched_get_priority_min
sched_getaffinity
sched_getattr
sched_getparam
sched_getscheduler
sched_rr_get_interval
sched_setaffinity
sched_setattr
sched_setparam
sched_setscheduler
sched_yield
seccomp
semctl
semget
semop
semtimedop
sendmmsg
sendmsg
sendto
set_mempolicy
set_mempolicy_home_node
set_robust_list
set_tid_address
setdomainname
setfsgid
setfsuid
setgid
setgroups
sethostname
setitimer
setns
setpgid
setpriority
setregid
setresgid
setresuid
setreuid
setrlimit
setsid
setsockopt
settimeofday
setuid
setxattr
shmat
shmctl
shmdt
shmget
shutdown
sigaltstack
signalfd4
socket
socketpair
splice
statx
swapoff
swapon
symlinkat
sync
sync_file_range
sync_file_range2
syncfs
sysinfo
syslog
tee
tgkill
timer_create
timer_delete
timer_getoverrun
timer_gettime
timer_settime
timerfd_create
timerfd_gettime
timerfd_settime
times
tkill
umask
umount2
uname
unlinkat
unshare
userfaultfd
utimensat
vhangup
vmsplice
wait4
waitid
write
writev

System Call Error Handling

- **Almost all system-level operations can fail**
  - You must explicitly check for failure
- **On error, most system-level functions return –1 and set global variable `errno` to indicate cause.**
- **Example:**

```
pid_t pid = fork();
if (pid == -1) {
    fprintf(stderr, "fork error: %s\n", strerror(errno));
    exit(1);
}
```

Error-reporting Functions

- Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(1);
}
```

```
  pid_t pid = fork();
  if (pid == -1)
    unix_error("fork error");
```

- Not always appropriate to exit when something goes wrong.

Error-handling Wrappers

- We simplify the code we present to you even further by using Stevens[1]-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid = fork();

    if (pid == -1)
        unix_error("Fork error");
    return pid;
}
```

```
  pid = Fork(); // Only returns if successful
```

- NOT what you generally want to do in a real application

[1]e.g., in “UNIX Network Programming: The sockets networking API“ W. Richard Stevens

Obtaining Process IDs

- **`pid_t getpid(void)`**
  - Returns PID of current process
- **`pid_t getppid(void)`**
  - Returns PID of parent process

Process States

From a programmer's perspective, we can think of a process as being in one of three states

- **Running**
  - Process is either executing, or waiting to be executed and will eventually be *scheduled* (i.e., chosen to execute) by the kernel
- **Stopped**
  - Process execution is *suspended* and will not be scheduled until further notice (next lecture when we study signals)
- **Terminated**
  - Process is stopped permanently

Terminating Processes

- **Process becomes terminated for one of three reasons:**
  - Receiving a signal whose default action is to terminate (next lecture)
  - Returning from the `main` routine
  - Calling the `exit` function

- **`void exit(int status)`**
  - Terminates with an *exit status* of `status`
  - Convention: normal return status is 0, nonzero on error
  - Another way to explicitly set the exit status is to return an integer value from the main routine

- **`exit` is called once but never returns.**

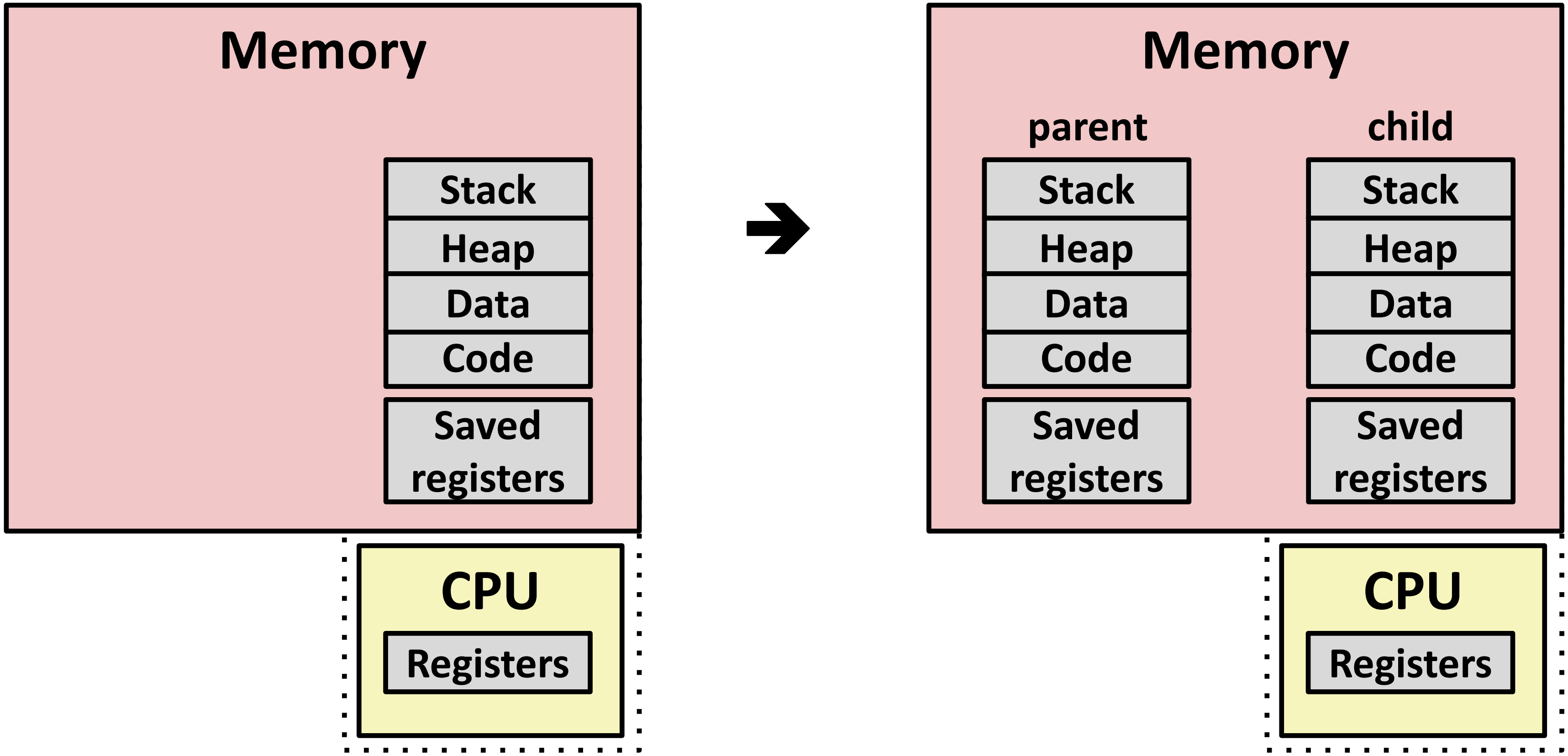
Creating Processes

- ***Parent process*** **creates a new running** ***child process*** **by calling `fork`**

- **`int fork(void)`**
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent

- **`fork` is interesting (and often confusing) because it is called *once* but returns *twice***

Conceptual View of `fork`



- **Make complete copy of execution state**
  - Designate one as parent and one as child
  - Resume execution of parent or child
  - (Optimization: Use copy-on-write to avoid copying RAM)

fork Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
parent: x=0
child : x=2
```

fork Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

```
linux> ./fork
parent: x=0
child : x=2
```

- Call once, return twice
- Concurrent execution
  - Can't predict execution order of parent and child
- Duplicate but separate address space
  - **x** has a value of 1 when fork returns in parent and child
  - Subsequent changes to **x** are independent
- Shared open files
  - `stdout` is the same in both parent and child

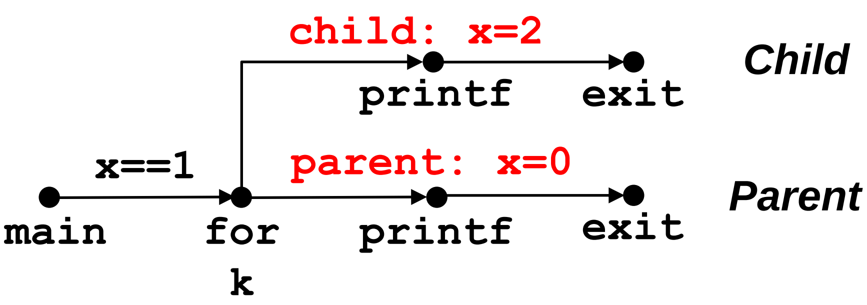
Modeling `fork` with Process Graphs

- **A *process graph* is a useful tool for capturing 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
- **Any *topological sort* of the graph corresponds to a feasible total ordering.**
  - Total ordering of vertices where all edges point from left to right

Process Graph Example

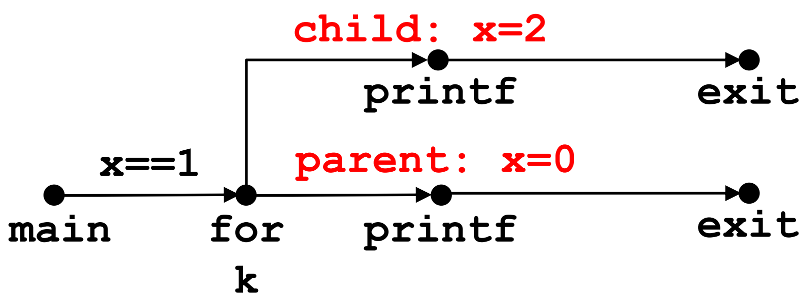
```
int main()
{
   pid_t pid;
   int x = 1;

   pid = Fork();
   if (pid == 0) {  /* Child */
      printf("child : x=%d\n", ++x);
            exit(0);
   }

   /* Parent */
   printf("parent: x=%d\n", --x);
   exit(0);
}
```

fork.c

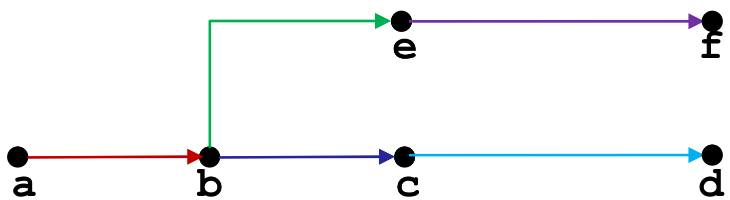


Interpreting Process Graphs

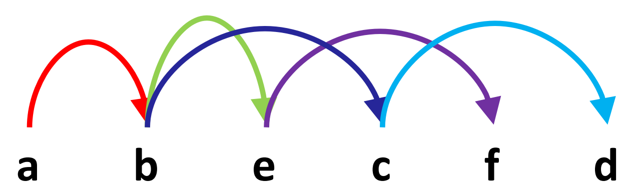
- Original graph:



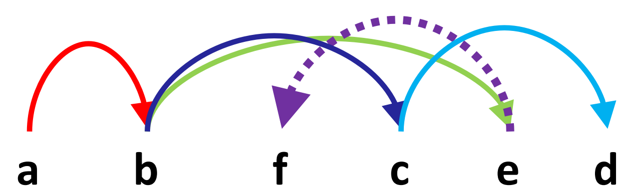
- Relabled graph:



Feasible total ordering:



Feasible or Infeasible?

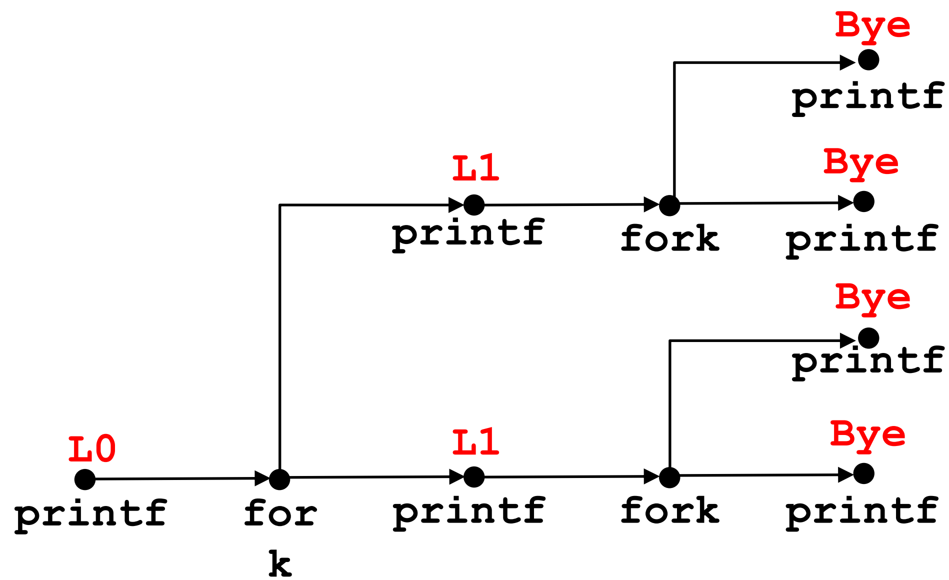


Infeasible: not a topological sort

fork Example: Two consecutive forks

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c



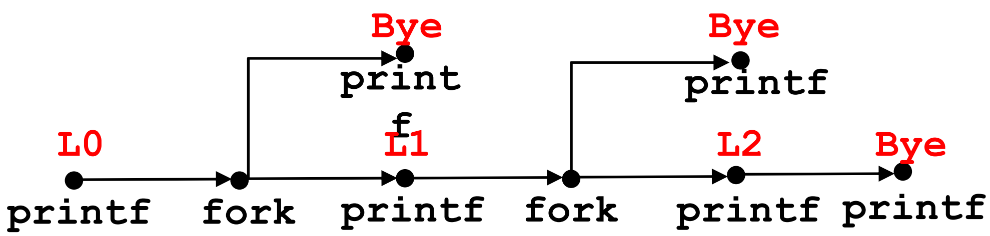
Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

fork Example: Nested forks in parent

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



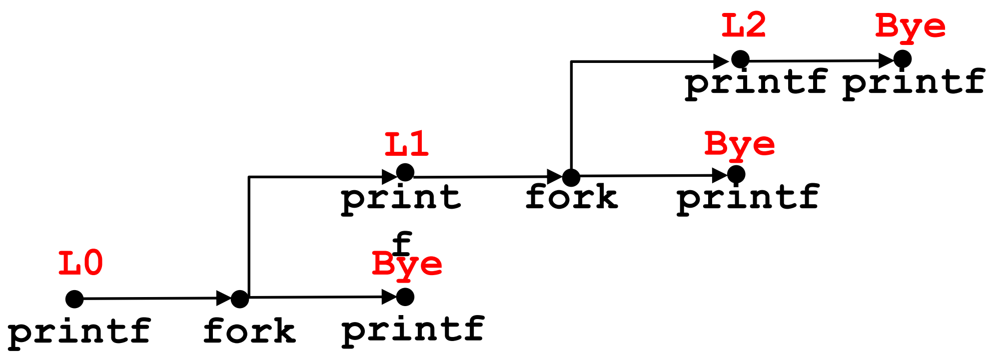
Feasible output:
L0
L1
Bye
Bye
L2
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

fork Example: Nested forks in children

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:
L0
Bye
L1
L2
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
Bye
L2

Reaping Child Processes

- **Idea**
  - When process terminates, it still consumes system resources
    - Examples: Exit status, various OS tables
  - Called a “zombie”
    - Living corpse, half alive and half dead
- **Reaping**
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel then deletes zombie child process
- **What if parent doesn’t reap?**
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Zombie Example

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

forks.c

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

- **ps** shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by **init**

Non-terminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
            getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
            getpid());
        exit(0);
    }
}
```

forks.c

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

- Child process still active even though parent has terminated
- Must kill child explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

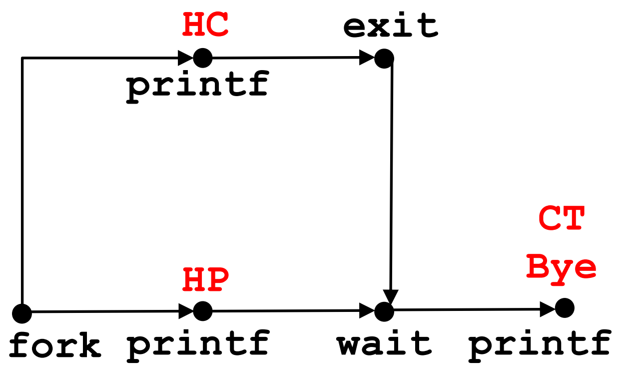
- **Parent reaps a child by calling the `wait` function**

- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - See textbook for details

wait: Synchronizing with Children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
            exit(0);
   } else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
}
```

forks.c



Feasible output:
HC
HP
CT
Bye

Infeasible output:
HP
CT
Bye
HC

Another wait Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

forks.c

`waitpid`: Waiting for a Specific Process

- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

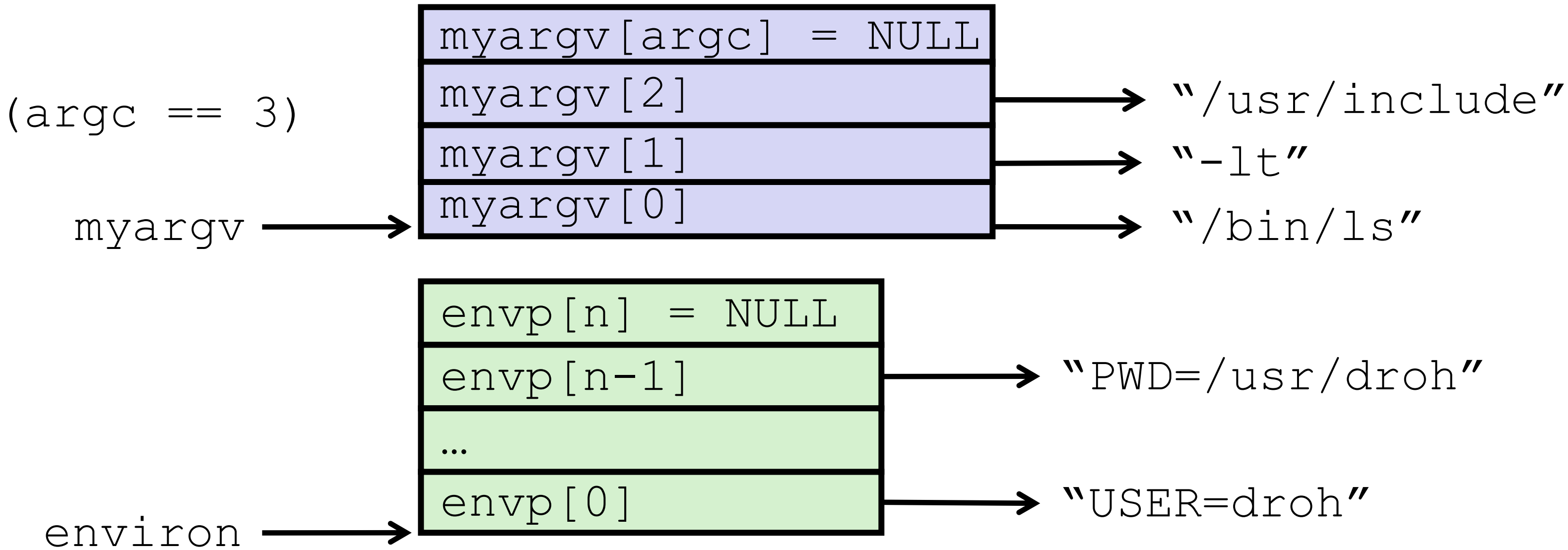
forks.c

`execve`: Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- **Loads and runs in the current process:**
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - …with argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - …and environment variable list **`envp`**
    - "name=value" strings (e.g., `USER=droh`)
    - `getenv, putenv, printenv`
- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context
- **Called once and never returns**
  - …except if there is an error

`execve` Example

- **Executes** `"/bin/ls -lt /usr/include"` **in child process using current environment:**




```c
if ((pid = Fork()) == 0) {   /* Child runs program */
   if (execve(myargv[0], myargv, environ) < 0) {
      printf("%s: Command not found.\n", myargv[0]);
      exit(1);
   }
}
```

Summary

- **Exceptions**
  - Events that require nonstandard control flow
  - Generated externally (interrupts) or internally (traps and faults)

- **Processes**
  - At any given time, system has multiple active processes
  - Only one can execute at a time on a single core, though
  - Each process appears to have total control of processor + private memory space

Summary (cont.)

- **Spawning processes**
  - Call `fork`
  - One call, two returns
- **Process completion**
  - Call `exit`
  - One call, no return
- **Reaping and waiting for processes**
  - Call `wait` or `waitpid`
- **Loading and running programs**
  - Call `execve` (or variant)
  - One call, (normally) no return