# [CSED211] Introduction to Computer Software Systems

## Lecture 13: Exceptional Control Flow - Exceptions and Processes

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.11.22

# Lecture Agenda

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control

# 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>**

$\texttt{inst}_1$

$\texttt{inst}_2$

⋮

$\texttt{inst}_n$

**<shutdown>**

# Altering the Control Flow

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

  Both 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
    - Control-flow change in response to a system event (i.e., system-state change)
    - Implemented using combination of hardware and OS software

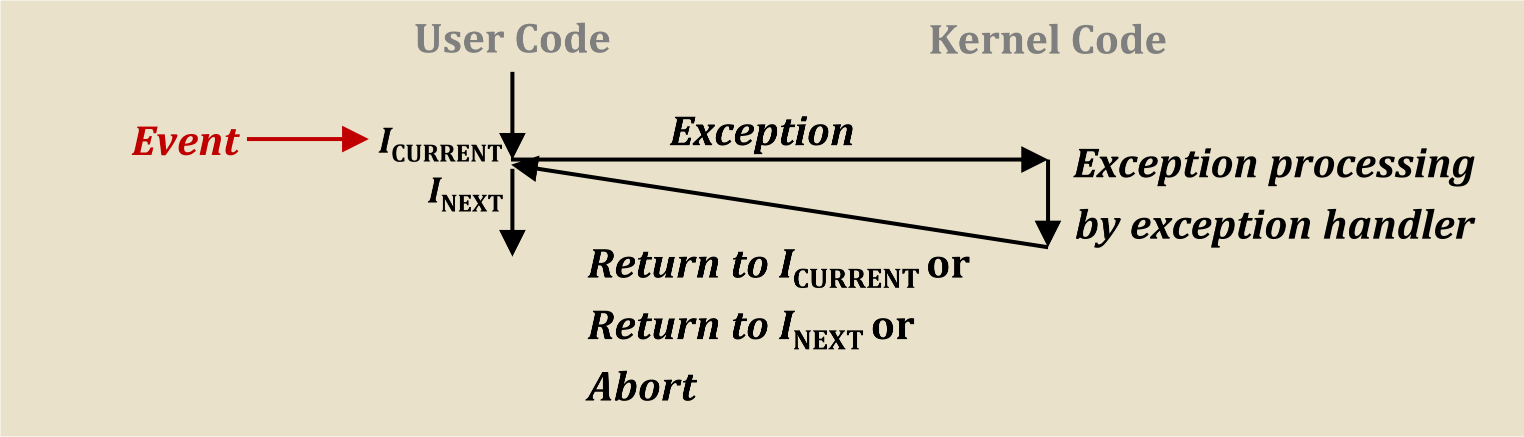
- Higher level mechanisms
  - 2. Process context switch
    - Implemented by OS software and hardware timer
  - 3. Signals
    - Implemented by OS software
  - 4. Nonlocal jumps: `setjmp()` and `longjmp()`
    - Implemented by C runtime library

# Lecture Agenda

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control

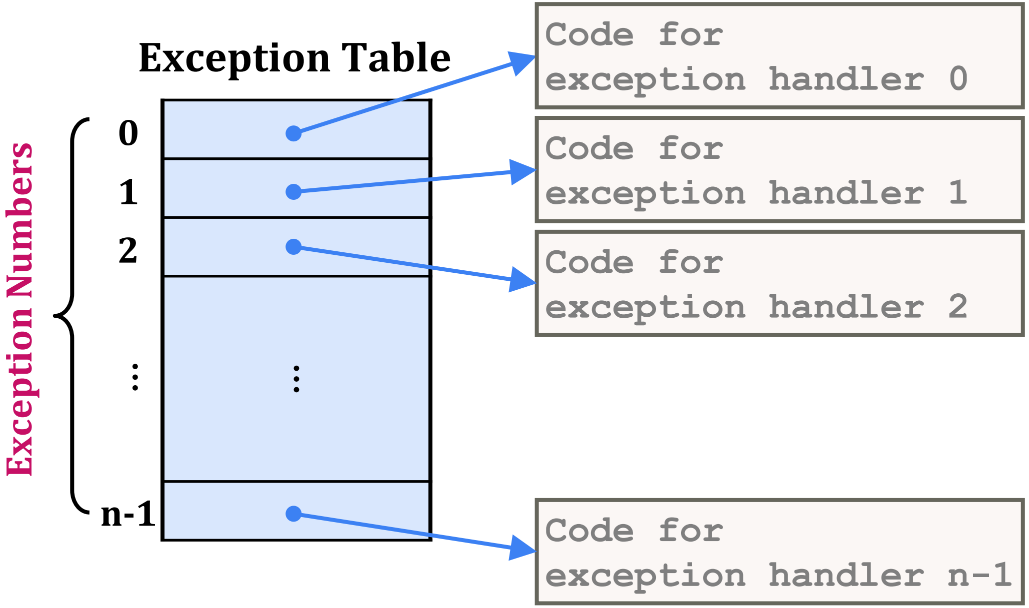
# Exceptions

- An exception requires a transfer of control to the OS kernel in response to a specific event (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: divide by zero, arithmetic overflow, page fault, I/O request completions, typing Ctrl-C, etc.



# Exception Table

- Data structure to point the handling routine for each type of event
  - Each type of event (i.e., exception) has a unique number *k* as its ID (set by either CPU or OS)
  - *k* = index into the exception table (a.k.a. interrupt vector)
  - Handler *k* is called each time exception *k* occurs

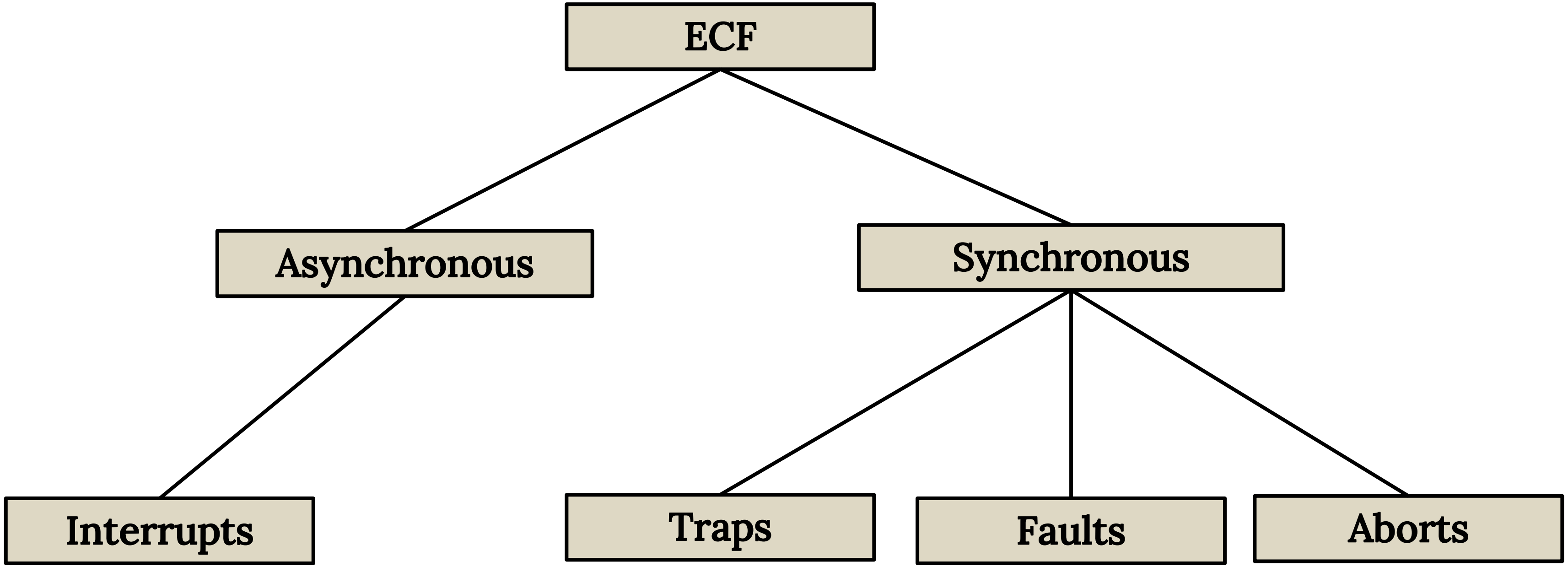


# Exception Table: IA32 (Excerpt)

- Check Intel manual for more information

| Exception Number | Description | Exception Class |
|---|---|---|
| 0 | Divide error | Fault |
| 13 | General protection fault | Fault |
| 14 | Page fault | Fault |
| 18 | Machine check | Abort |
| 32-127 | OS-defined | Interrupt or trap |
| 128 (0x80) | System call | Trap |
| 129-255 | OS-defined | Interrupt or trap |

# (Partial) Taxonomy



# Asynchronous Exceptions (a.k.a. Interrupts)

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

- Examples
  - Timer interrupt
    - Every few milliseconds, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external devices
    - Hitting Ctrl-C at the keyboard
    - 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 the next instruction
    - **Faults**
        - Unintentional but possibly recoverable
        - Examples: page faults (recoverable), protection faults (unrecoverable), floating-point exceptions
        - Either re-executes faulting (i.e., current) instruction or aborts
    - **Aborts**
        - Unintentional and unrecoverable
        - Examples: parity error, machine check
        - Aborts current program

# System Calls

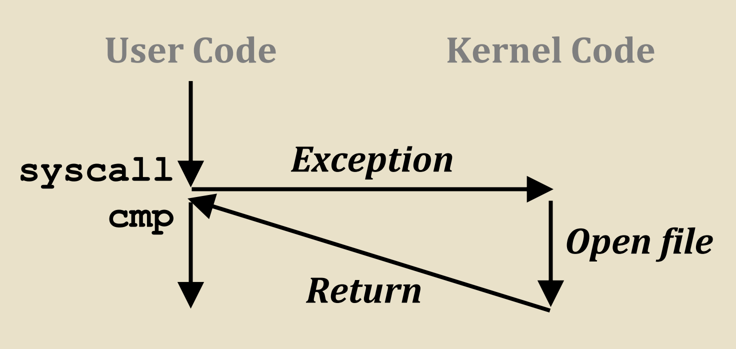
- Each x86-64 system call has a unique ID number

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

# System Call Example: File Open

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

```
0000000000e5d70 <__open>:
 …
 e5d79:   b8 02 00 00 00   sub  $0x2,%eax                # open is syscall #2
 e5d7e:   0f 05            syscall                       # Return value in %rax
 e5d80:   48 3d 01 f0 ff   cmp  $0xfffffffffffff001,%rax
 …
 e5dfa:   c3               retq
```

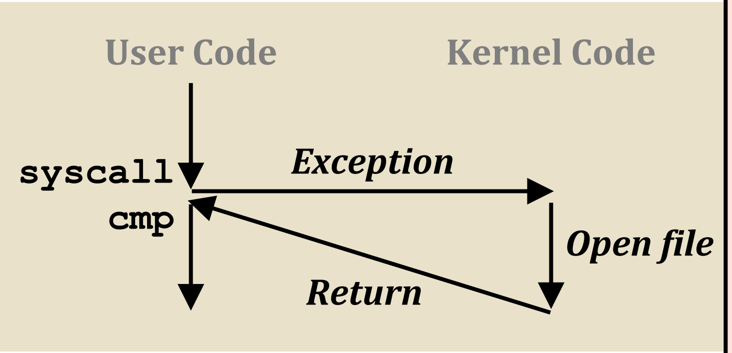



- **`%rax`** contains syscall number
- Other arguments in **`%rdi`**, **`%rsi`**, **`%rdx`**, **`%r10`**, **`%r8`**, and **`%r9`**
- **Return value in `%rax`**
- **Negative return value indicates an error corresponding to negative `errno`**

# System Call Example: File Open

- User calls: `open(filename`
  - Calls `__open` function, whi

```
0000000000e5d70 <__open>:
 …
 e5d79:    b8 02 00 00 00      sub   $
 e5d7e:    0f 05               syscal
 e5d80:    48 3d 01 f0 ff      cmp   $
 …
 e5dfa:    c3                  retq
```



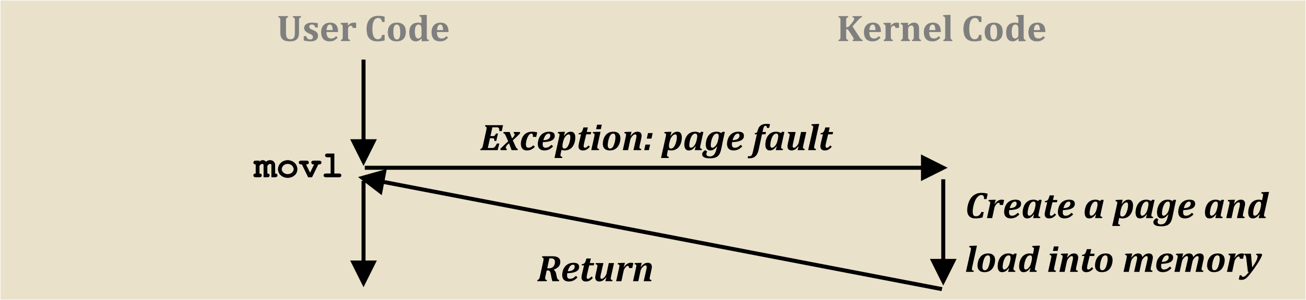

- Almost like a function call
  - Transfer of control
  - On return, executes the next instruction
  - Passes arguments using the same calling convention
  - Gets result in %rax
- One important exception
  - Executed by kernel
  - Different set of privileges
- And other differences
  - Function address is in %rax
  - Uses **`errno`**
  - etc.

# Fault Example: Page Fault

- User writes to a memory location
  - When portion (page) of user's memory is currently on disk. Not in memory.

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

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



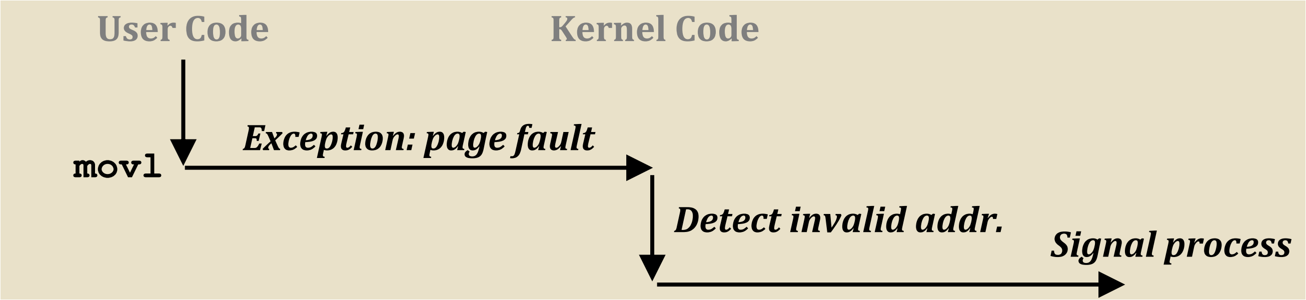

  - Page handler must load the page into physical memory
  - Returns to the faulting instruction
  - Successful on second try

# Fault Example: Invalid Memory Reference

- User writes to an invalid memory location

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

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




  - Page handler detects invalid address
  - Sends **SIGSEGV** signal to user process
  - User process exits with segmentation fault

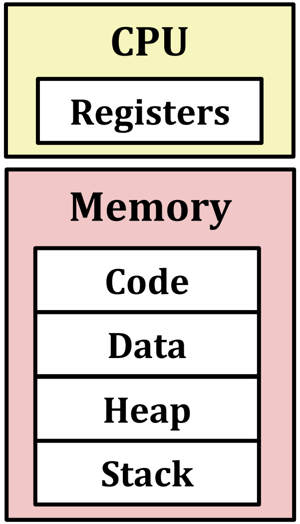
# Lecture Agenda

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control

# Processes

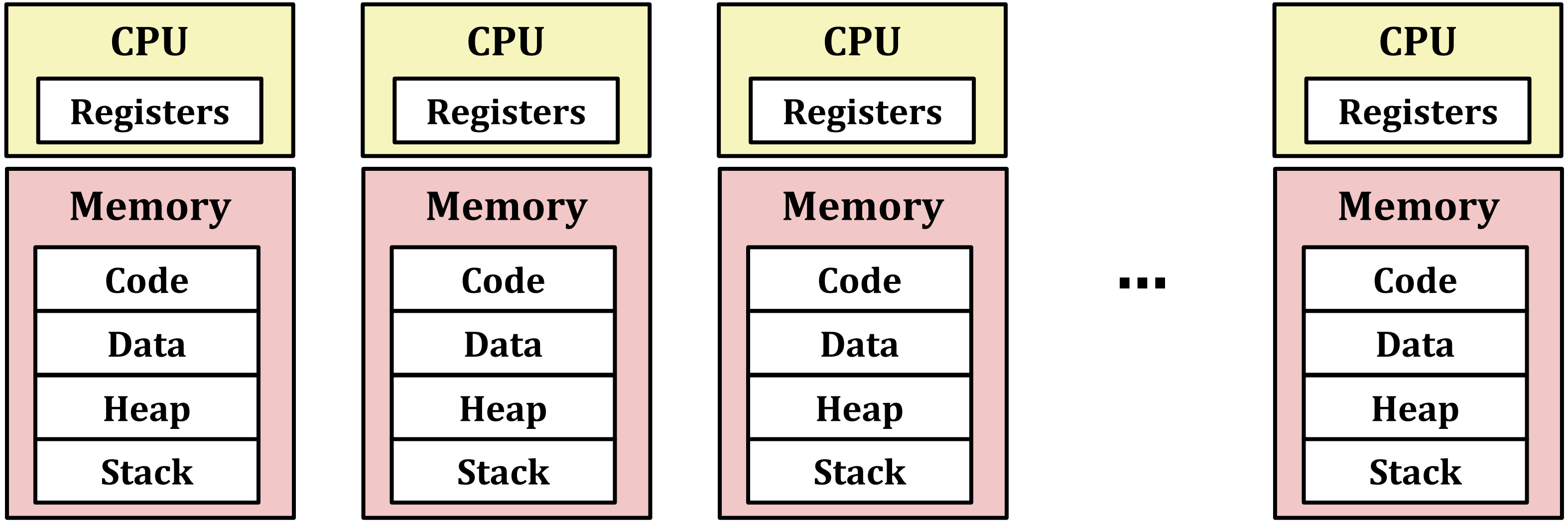
- An instance of running program
  - One of the most profound ideas in computer science
  - Not the same as program or processor

- 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



# Multiprocessing: The Illusion

- Computer runs many processes simultaneously
  - Applications for one or more users: e.g., web browsers, email clients, editors, etc.
  - Background tasks: e.g., monitoring network & I/O devices



# Multiprocessing Example

- Running program `top` on Mac
  - System has 810 processes, five of which are active
  - Identified by Process ID (PID)

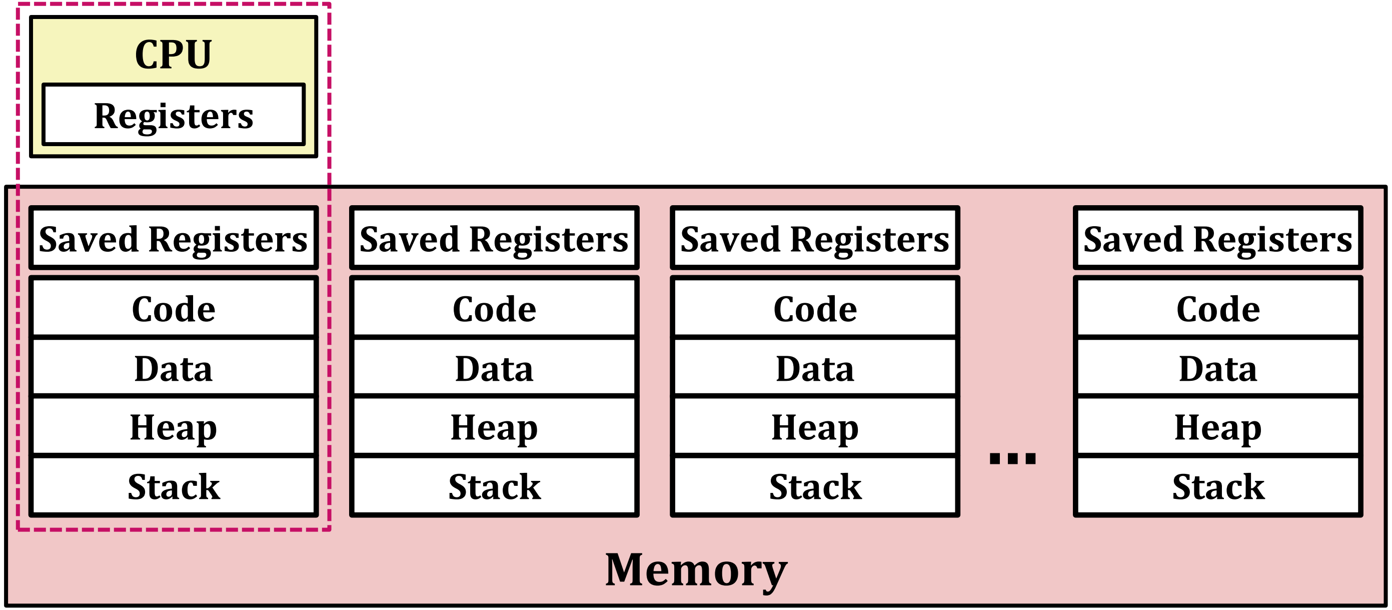
jisung — top — 111×33

```
Processes: 810 total, 5 running, 805 sleeping, 4116 threads                                          17:20:52
Load Avg: 2.29, 2.18, 2.38  CPU usage: 16.46% user, 2.62% sys, 80.90% idle
SharedLibs: 797M resident, 152M data, 156M linkedit.
MemRegions: 200354 total, 5896M resident, 500M private, 5721M shared.
PhysMem: 23G used (2968M wired, 2498M compressor), 604M unused.
VM: 310T vsize, 4729M framework vsize, 0(0) swapins, 0(0) swapouts.
Networks: packets: 3653323/3213M in, 2249336/882M out. Disks: 2466133/39G read, 1197430/23G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE    BOOSTS
1629   HwpMac2014   99.2 12:19:51 3/1   1    210   27M    0B     21M    1629  1     running  *1534[3]
561    WindowServer 7.3  20:03.43 17    6    4081- 2085M+ 324M-  70M    561   1     sleeping *0[1]
1975   Discord Help 5.6  36:02.48 44    2    715   269M+  0B     75M    1625  1625  sleeping *1[4]
53296  top          4.3  00:01.27 1/1   0    29    13M    0B     0B     53296 53280 running  *0[1]
0      kernel_task  3.5  28:37.85 569/8 0    0     16M    0B     0B     0     0     running   0[0]
1161   WindowManage 2.8  01:18.28 5     2    385   38M    0B     3776K  1161  1     sleeping *0[5185+]
1      launchd      2.4  21:44.08 3/1   2/1  4187  28M-   0B     4720K  1     0     running   0[0]
53315  screencaptur 2.2  00:00.14 5     3    170-  13M+   0B     0B     53315 1     sleeping *0[8+]
940    distnoted    1.5  10:17.13 2     1    650-  5025K  0B     832K   940   1     sleeping *0[1]
2221   nosmain      1.3  09:57.39 5     1    207   34M    0B     19M    2175  1     sleeping *0[1]
53318  screencaptur 1.2  00:00.05 7     6    74-   7346K+ 16K    0B     1633  1633  sleeping *0[105+]
554    bluetoothd   1.1  03:53.77 11    5    412   16M    240K   6912K  554   1     sleeping *0[1]
1620   Microsoft Po 0.8  13:33.01 33    8    1896  1260M  103M   200M   1620  1     sleeping *0[2730]
1619   AdobeAcrobat 0.8  06:45.49 28    4    388   326M   0B     159M   1619  1     sleeping  0[2335]
535    launchservic 0.7  01:37.05 5     4    887   9777K  0B     816K   535   1     sleeping *2550[318]
545    distnoted    0.6  04:03.04 2     1    236   1969K  0B     528K   545   1     sleeping *0[1]
493    logd         0.6  05:56.01 4/1   3    2594  14M    0B     13M    493   1     running  *0[1]
1792   Discord Help 0.6  04:55.87 13    2    213   337M   13M    35M    1625  1625  sleeping *1[5]
1753   Microsoft Te 0.5  04:38.59 12    1    239   226M   6160K  22M    1622  1622  sleeping *1[5]
753    BTLEServer   0.5  00:44.78 3     2    69    4290K  0B     1248K  753   1     sleeping *0[1]
1939   LogiTune     0.5  01:38.37 32    1    381   62M    0B     32M    1939  1     sleeping *0[1441]
1927   logioptionsp 0.4  02:12.43 260   1    832   95M    0B     29M    1927  1     sleeping *0[1]
1941   Dropbox      0.4  01:42.15 134   1    741   336M   80K    88M    1941  1     sleeping *0[2438+]
1616   Google Chrom 0.4  04:32.51 44    1    933   207M   0B     53M    1616  1     sleeping *0[1863]
```

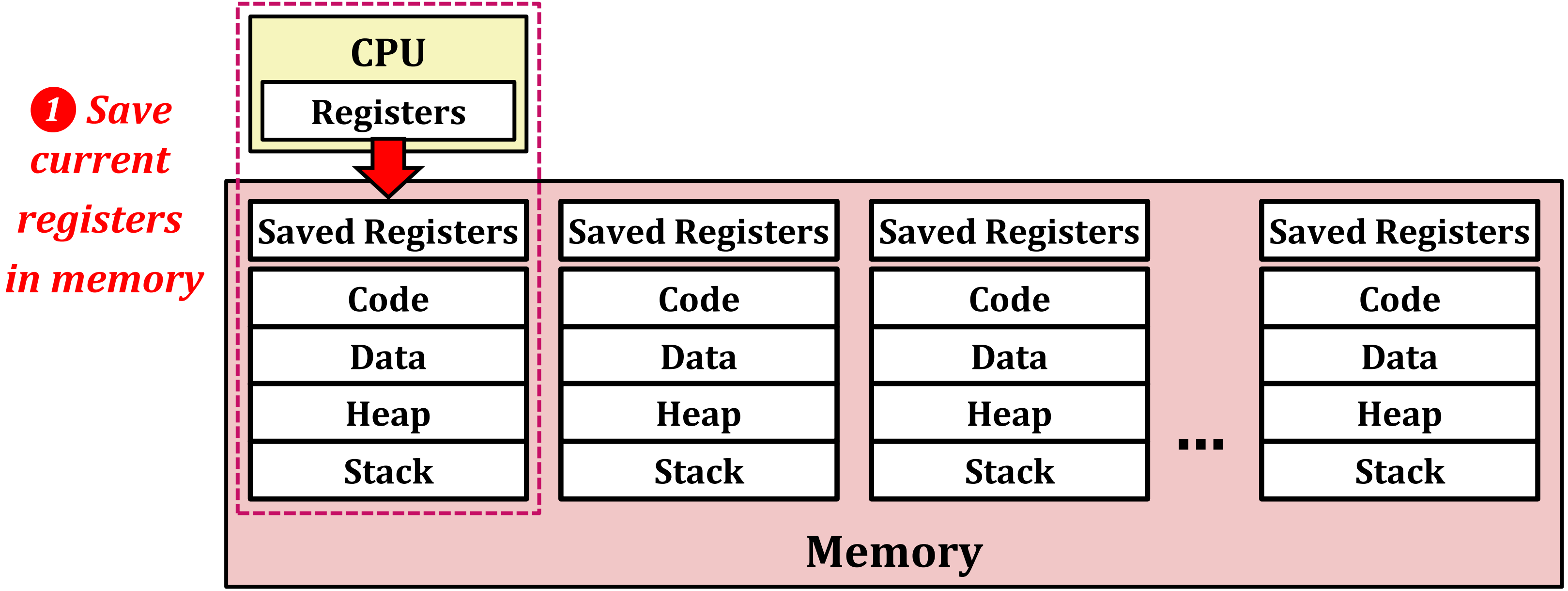
# Multiprocessing: The (Traditional) Reality

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



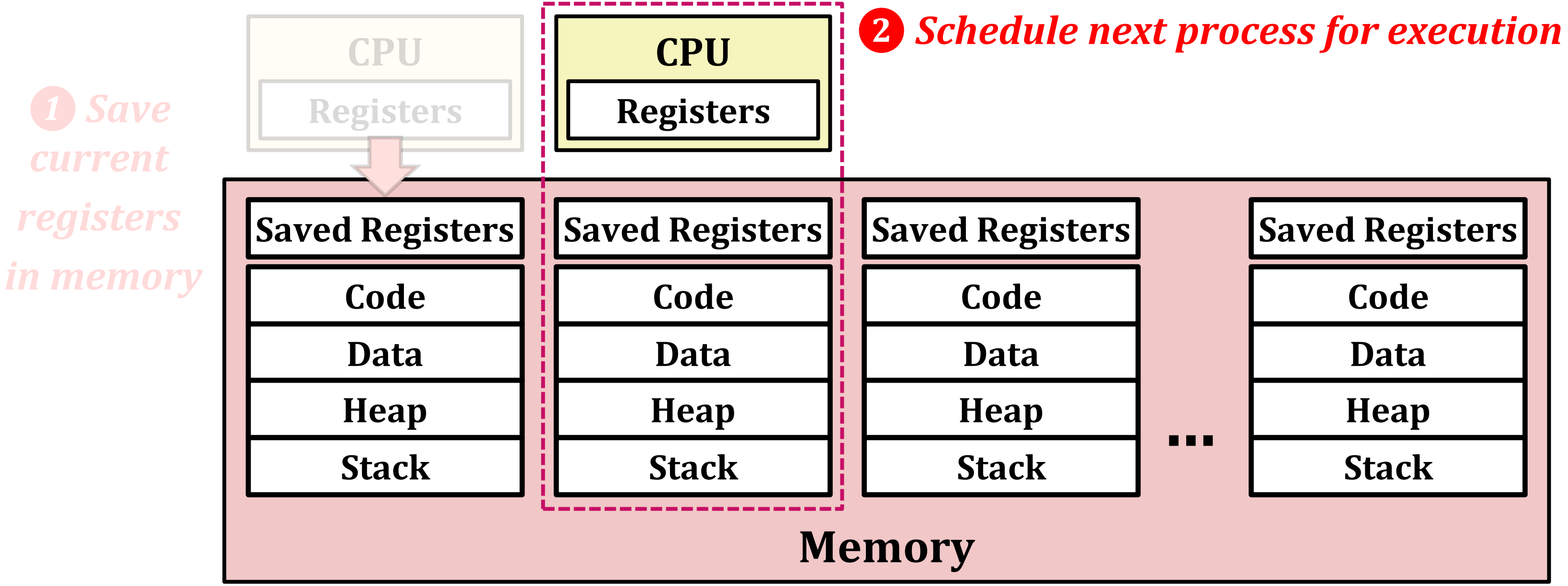
# Multiprocessing: The (Traditional) Reality

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



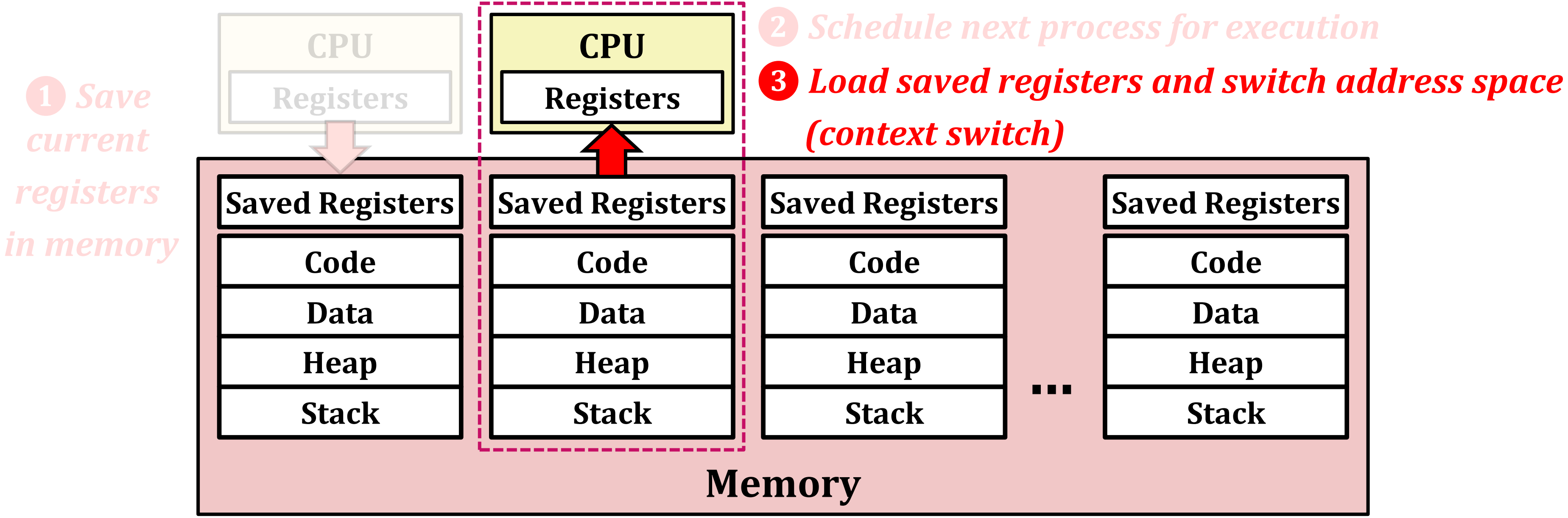
# Multiprocessing: The (Traditional) Reality

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



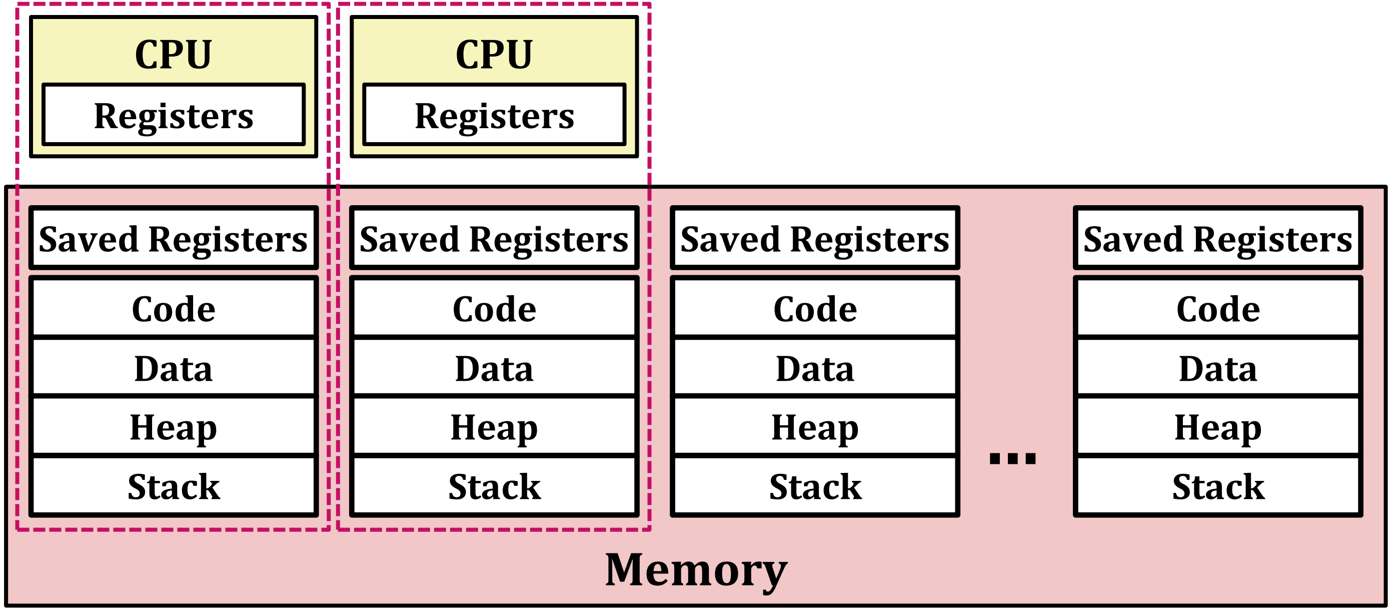
# Multiprocessing: The (Traditional) Reality

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



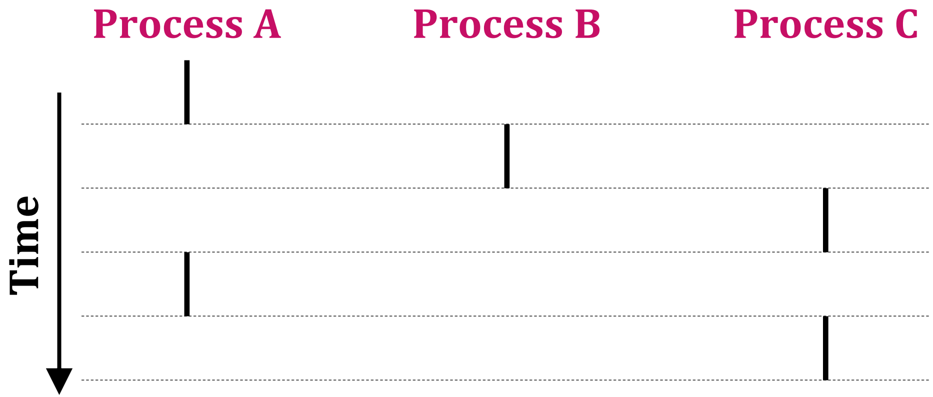
# Multiprocessing: The (Traditional) Reality

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



# Concurrent Processes

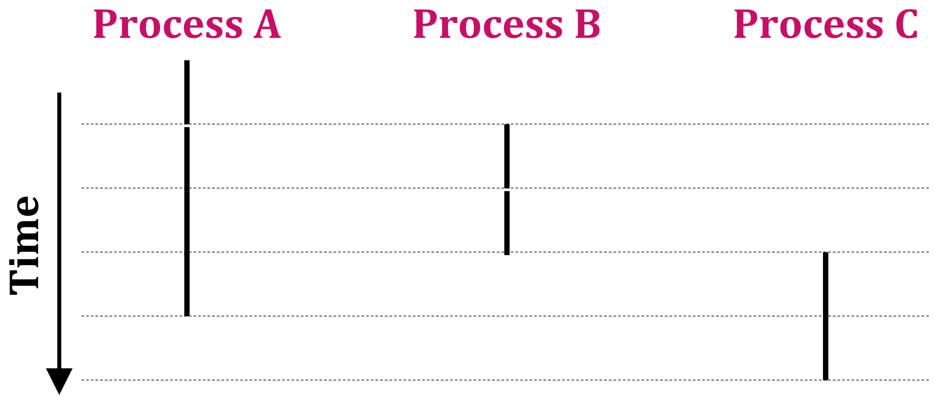
- Each process is a logical control flow
- Two processes run concurrently if their flows overlap in time
  - Otherwise, they are sequential
- Examples (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C



# Concurrent Processes: User View

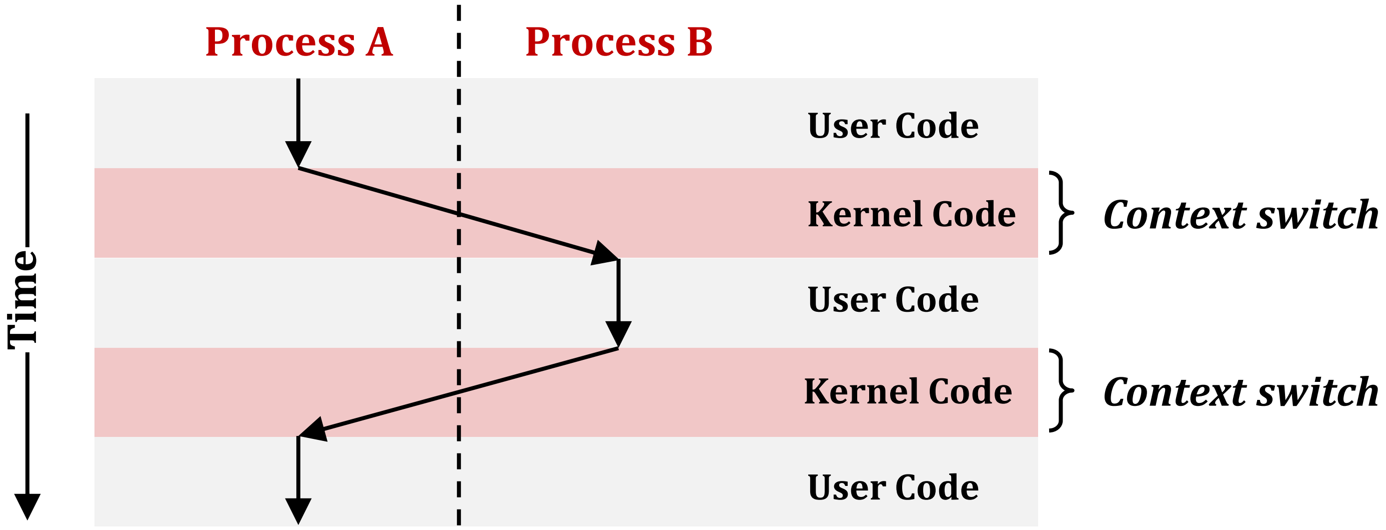
- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other



# Context Switching

- Processes are managed by a shared chunk of OS code called kernel
  - The kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via context switch



# Lecture Agenda

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control

# System Call Error Handling

- On error, Linux system-level functions typically return `-1` and set global variable `errno` to indicate cause.

- Hard and fast rule
  - Must check the return status of every system-level function
  - Only exception is the handful of functions that return `void`

- Example

```
if((pid = fork()) < 0){
  fprintf(stderr, "fork error: %s\n", strerror(errno));
  exit(0);
}
```

# Error-Reporting Functions

- Can simplify somewhat using an error-reporting function

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

```c
 if((pid = fork()) < 0)
   unix_error("fork error");
```

- But must think about application; not always appropriate to exit when something goes wrong.

# Error-Handling Wrappers

- Simplify the code even further using Stevens-style error-handling wrappers

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

    if((pid = fork()) < 0)
        unix_error("fork error");
    return pid;
}
```

```
pid = Fork();
```

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

# Obtaining Process IDs

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

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

# Creating and Terminating Processes

- From a programmer's perspective, a process is in one of three states
- Running: the process is either executing or waiting to be executed (and will eventually be scheduled (i.e., chosen to execute) by the kernel)
- Stopped: the process execution is suspended and will not be scheduled until further notice (next lecture when we study signals)
- Terminated: the pocess 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

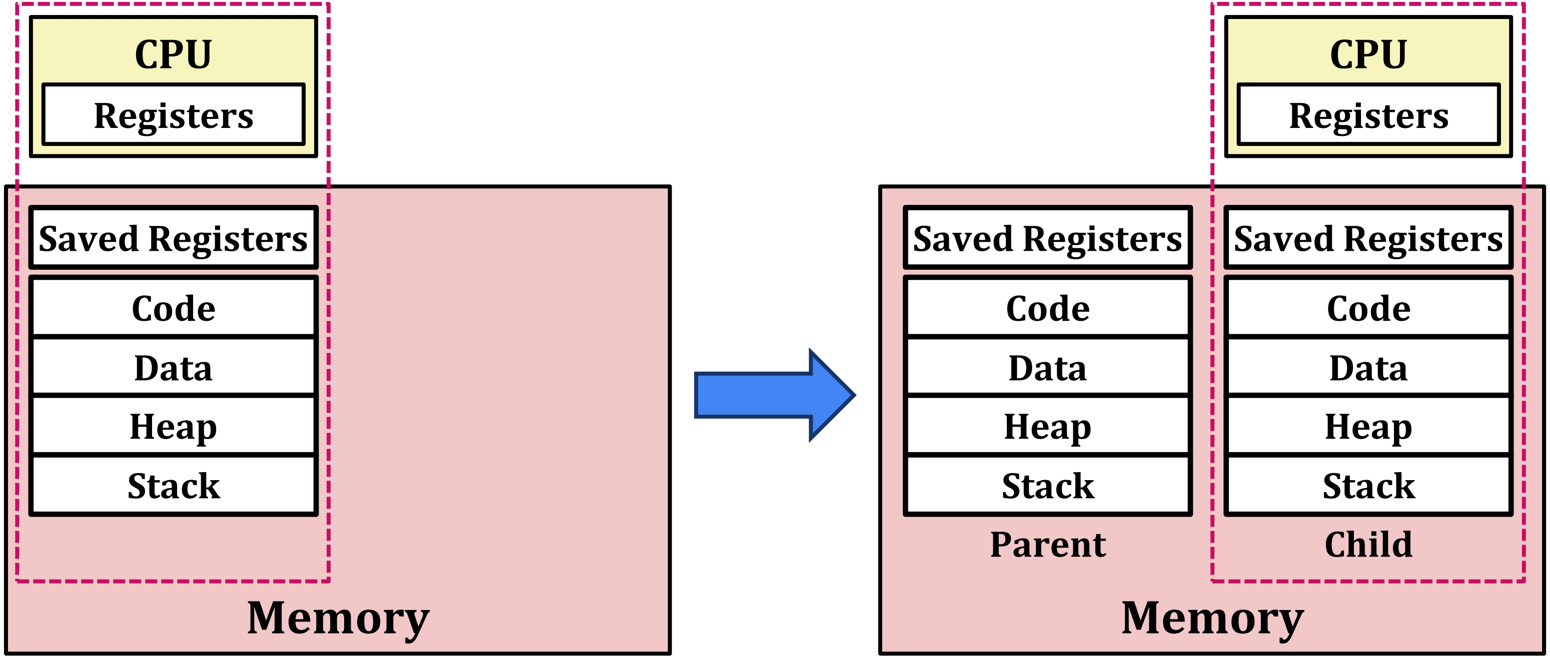
- Parent process creates a new running child process by calling `fork`

- `int fork(void)`
  - Returns `0` to the child process and 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): 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



# fork Example

```c
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: cannot predict execution order of parent and child

```
$ ./fork
parent: x=0
child : x=2
```

```
$ ./fork
child : x=2
parent: x=0
```

```
$ ./fork
parent: x=0
child : x=2
```

```
$ ./fork
parent: x=0
child : x=2
```

```
$ ./fork
child : x=2
parent: x=0
```

…

# Making `fork` More Nondeterministic

- Problem
  - Linux scheduler does not create much run-to-run variance
  - Hides potential race conditions in nondeterministic programs
    - e.g., does fork return to child first or to parent?

- Solution
  - Create custom version of library routine that inserts random delays along different branches
    - e.g., for parent and child in fork
  - Use runtime interpositioning to have program use special version of library code

# Variable Delay `fork`

```c
/* fork wrapper function */
pid_t fork(void){
  initialize();
  int parent_delay = choose_delay();
  int child_delay = choose_delay();
  pid_t parent_pid = getpid();
  pid_t child_pid_or_zero = real_fork();
  if(child_pid_or_zero > 0){ /* Parent */
    if(verbose){
      printf("Fork. Child pid=%d, delay=%dms. Parent pid=%d, delay=%dms\n",
             child_pid_or_zero, child_delay,parent_pid, parent_delay);
      fflush(stdout);
    }
    ms_sleep(parent_delay);
  } else { /* Child */
    ms_sleep(child_delay);
  }
  return child_pid_or_zero;
}
```

myfork.c

# forkx2 Example

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

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

forkx2.c

```
$ ./fork
parent: x=0
parent: x=-1
child : x=2
child : x=3
```

- Call once, return twice
- Concurrent execution: cannot 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

# Making `fork` with Process Graphs

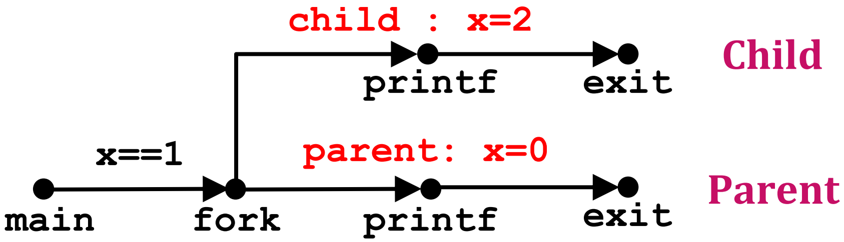
- Process graph: a useful tool to capture the partial ordering of statements in a concurrent program
  - Each vertex is the execution of a statement
  - `a` → `b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inward edge

- Any topological sort of the graph corresponds to a feasible total ordering
  - Total ordering of vertices where all edges point from left to right

# Making `fork` with Process Graphs

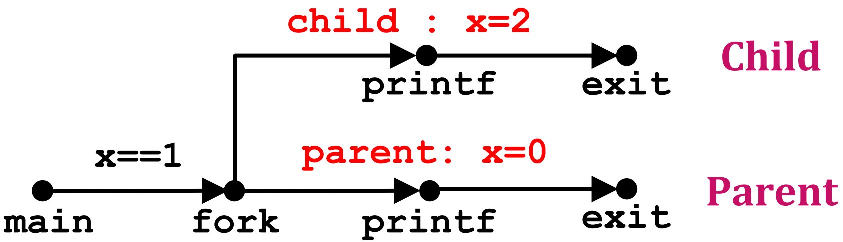
```c
int main(int argc, char** argv){
    pid_t pid;
    int x = 1;

    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

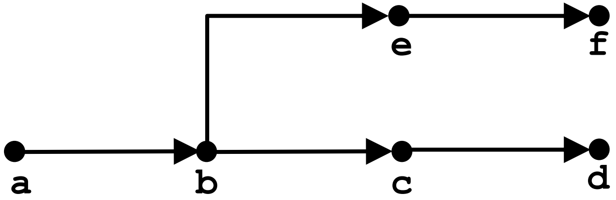


# Interpreting Process Graphs

- Original graph




- Relabeled graph

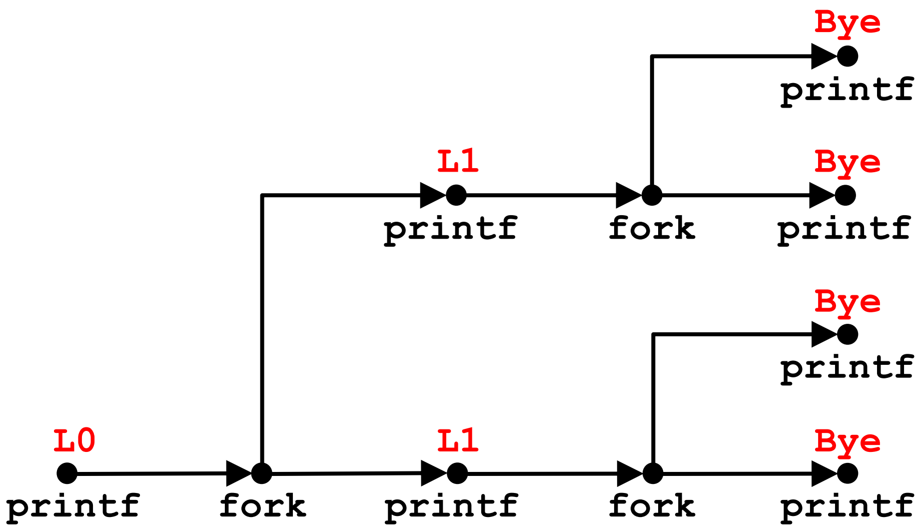



**Feasible total ordering:**


**Infeasible total ordering:**

# `fork` Example: Two Consecutive `fork`s

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

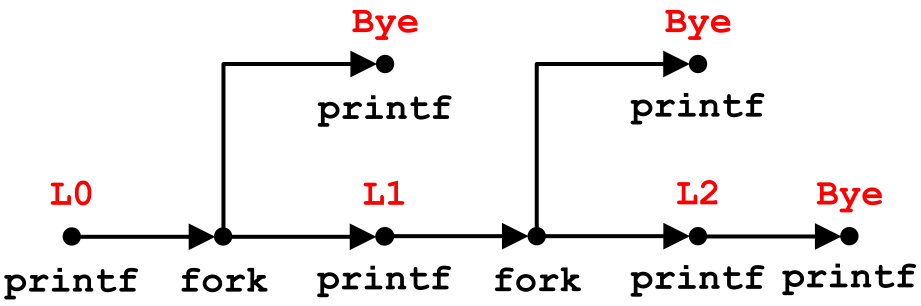



- Feasible output: **L0 → L1 → Bye → Bye → L1 → Bye**
- Infeasible output: **L0 → Bye → L1 → Bye → Bye → L1**

# fork Example: Nested forks in Parent

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

forks.c

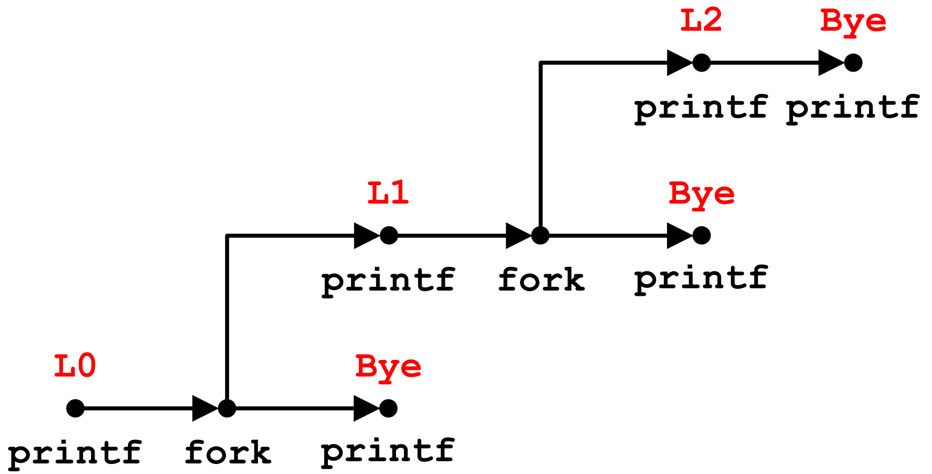



- Feasible output: **L0 → L1 → Bye → Bye → L2 → Bye**
- Infeasible output: **L0 → Bye → L1 → Bye → Bye → L2**

# `fork` Example: Nested `fork`s in Children

```c
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 → 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
    - e.g., 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

# 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 does not 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(){                                    forks.c
  if(fork() == 0){
    /* Child */
    printf("Terminating Child, PID = %d\n",
           getpid());
    exit(0);
  } else {
    /* Parent */
    printf("Running Parent, PID = %d\n",
           getpid());
    while(1); /* Infinite loop */
  }
}
```

```
$ ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
$ 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
$ kill 6639
[1]    Terminated
$ 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

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

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

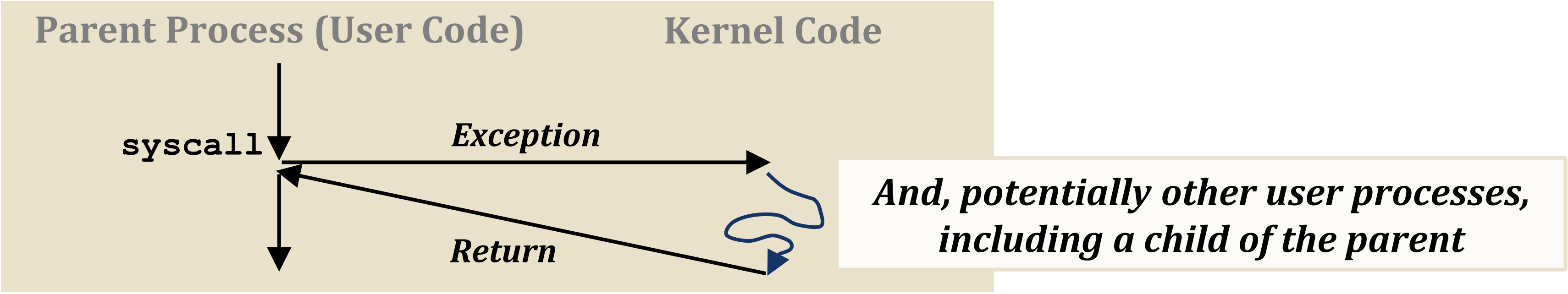
**Child process still active even though parent has terminated**

**Must kill child explicitly; otherwise it 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
  - Implemented as syscall



# `wait`: Synchronizing with Children

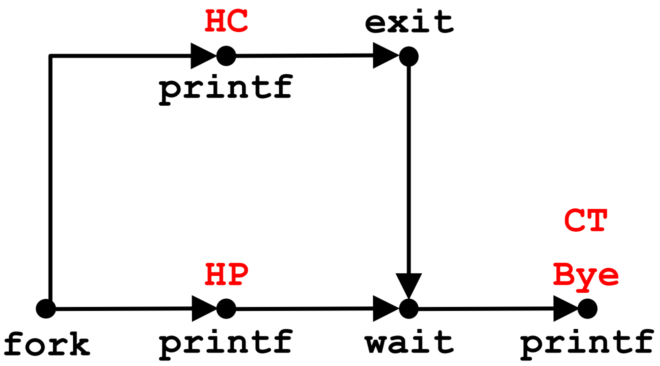
- Parent reaps a child by calling the **`wait`** function

- **`int wait(int *child_status)`**
  - Suspends current process until one of its children terminates
  - Implemented as syscall
  - Return value is the PID of the terminated child process
  - 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 more 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
- **`WIFEXITED`** and **`WEXITSTATUS`** macros to get information about exit status

```
void fork10() {                                                          forks.c
  pid_t pid[N];
  int i, child_status;
  for(i = 0; i < N; i++)
    if((pid[i] = fork()) == 0) /* Child */
      exit(100+i);
  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);
  }
}
```

# waitpid: Waiting for a Specific Process

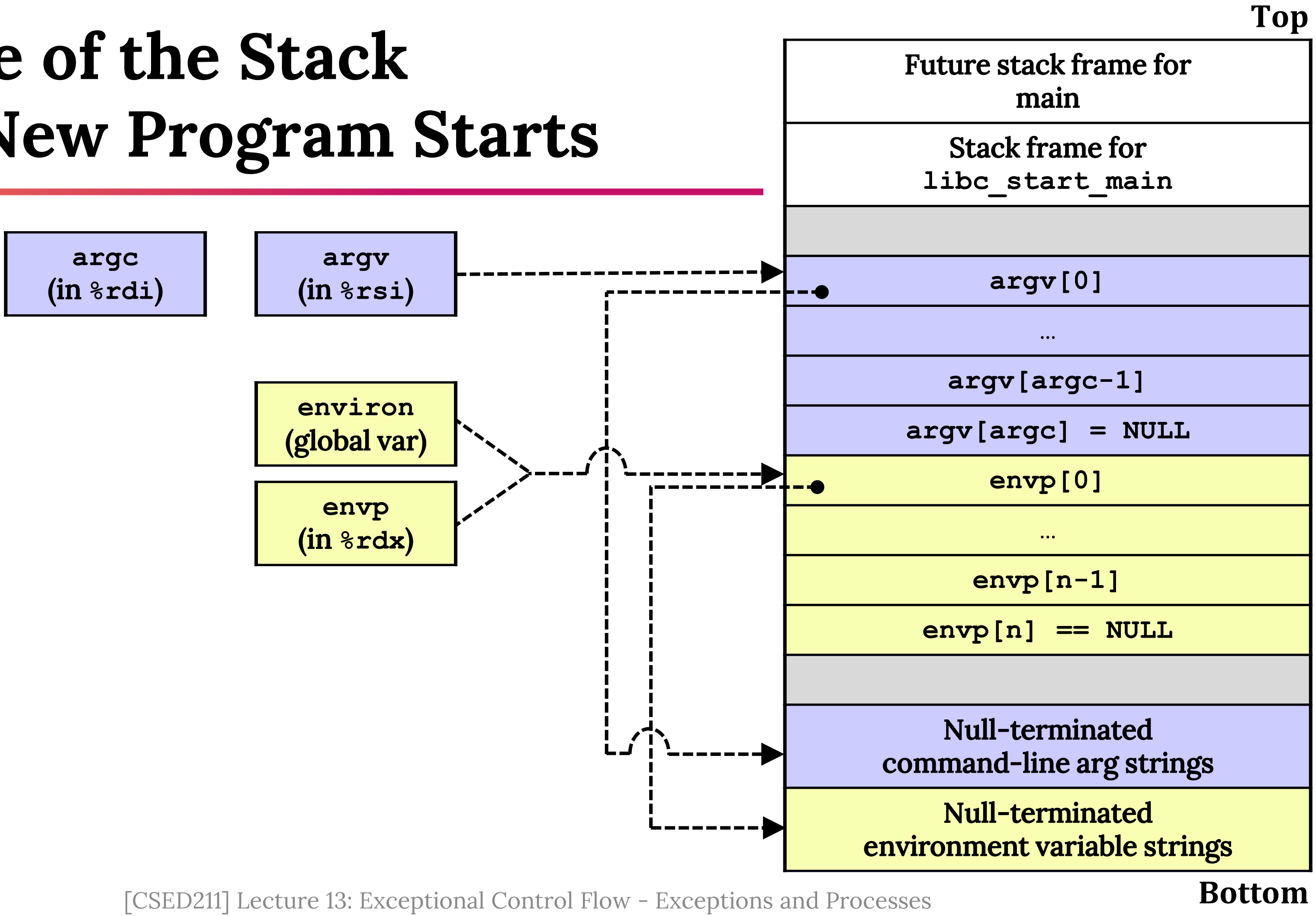
- `pid_t waitpid(pid_t pid, int &status, int options)`
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```
void fork10() {                                                          forks.c
  pid_t pid[N];
  int i, child_status;
  for(i = 0; i < N; i++)
    if((pid[i] = fork()) == 0) /* Child */
      exit(100+i);
  for(i = N-1; i >= 0; i--){ /* Parent */
    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);
  }
}
```

# `execve`: Loading and Running Programs

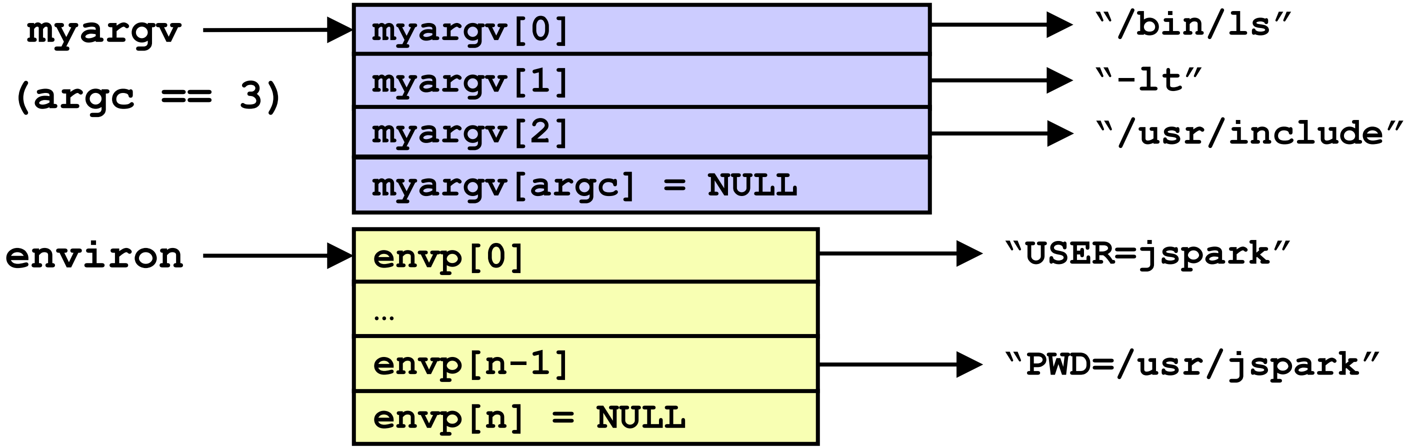
- `int execve(char *filename, char *argv[], char *envp[])`
- Loads and runs in the current process
  - Executable file **`filename`**
    - Object file or script file beginning with #!interpreter (e.g., **`#!/bin/bash`**)
  - Argument list **`argv`**
    - By convention **`argv[0]==filename`**
  - Environment variable list **`envp`**
    - *name=value* strings (e.g., **`USER=jspark`**)
    - **`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

# Structure of the Stack When a New Program Starts



# execve Example

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




```
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

- 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

# [CSED211] Introduction to Computer Software Systems

## Lecture 13: Exceptional Control Flow - Exceptions and Processes

Prof. Jisung Park

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.11.22