Exceptional Control Flow: Exceptions and Processes

Today

- **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>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<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

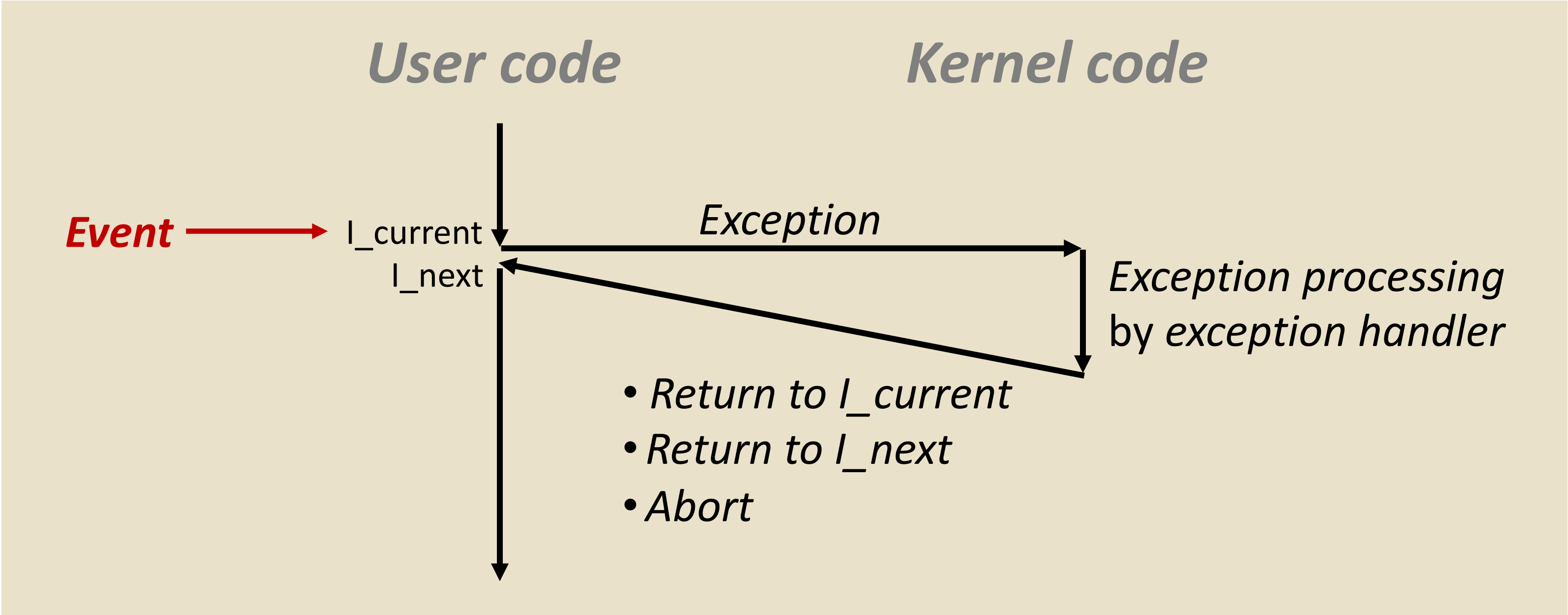
- **Exists at all levels of a computer system**
- **Low level mechanisms**
  - 1. **Exceptions**
    - Change in control flow in response to a system event (i.e., change in system state)
    - Implemented using combination of hardware and OS software
- **Higher level mechanisms**
  - 2. **Process context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

Today

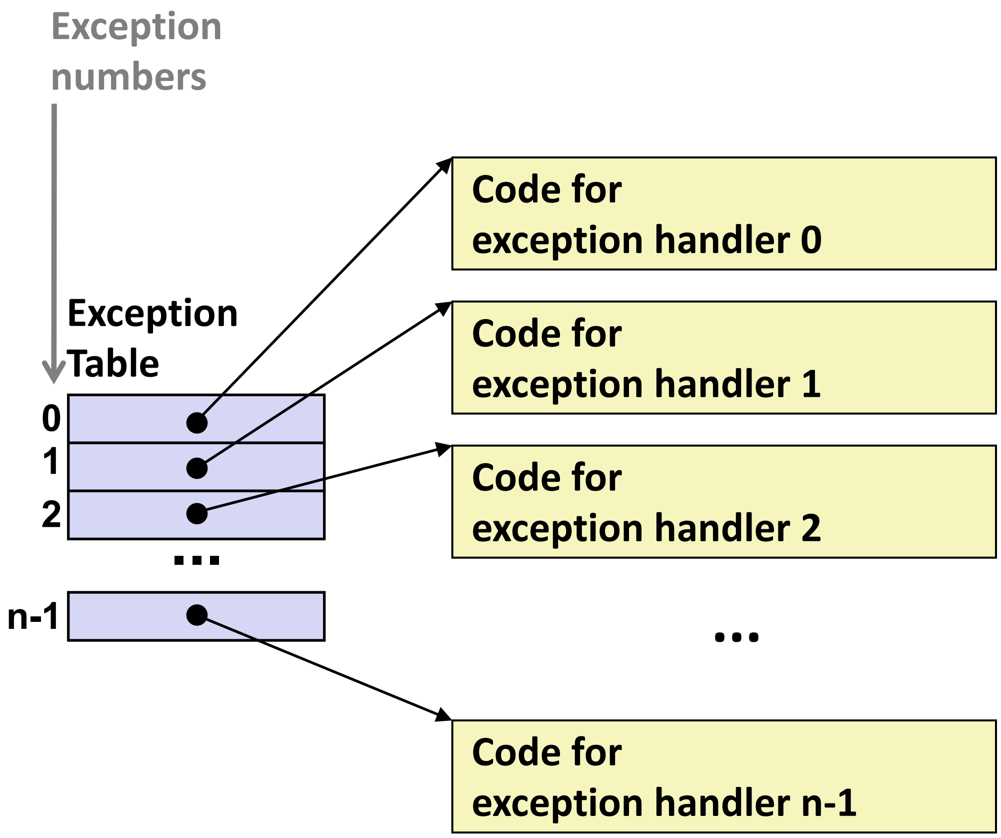
- Exceptional Control Flow
- **Exceptions**
- Processes
- Process Control

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

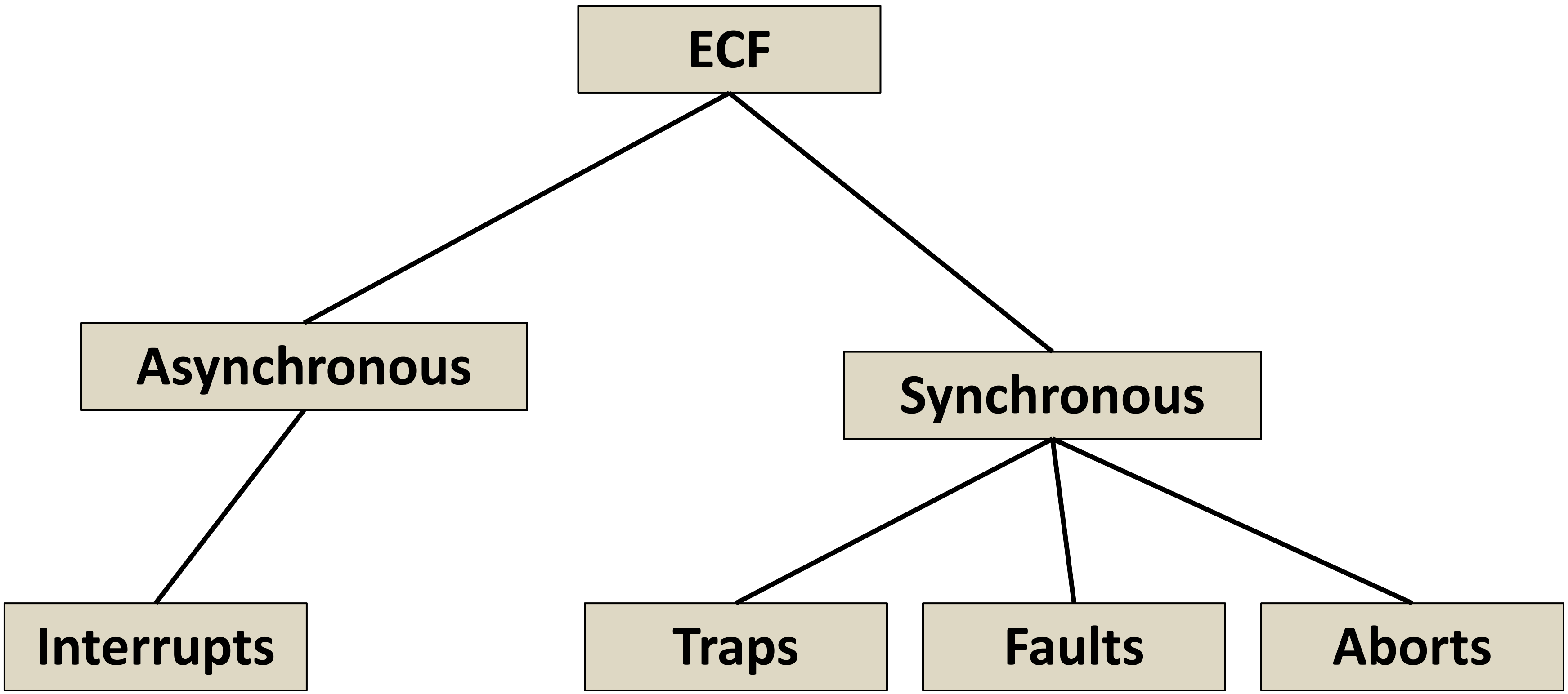


Exception Tables



- 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

(partial) Taxonomy



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

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

System Calls

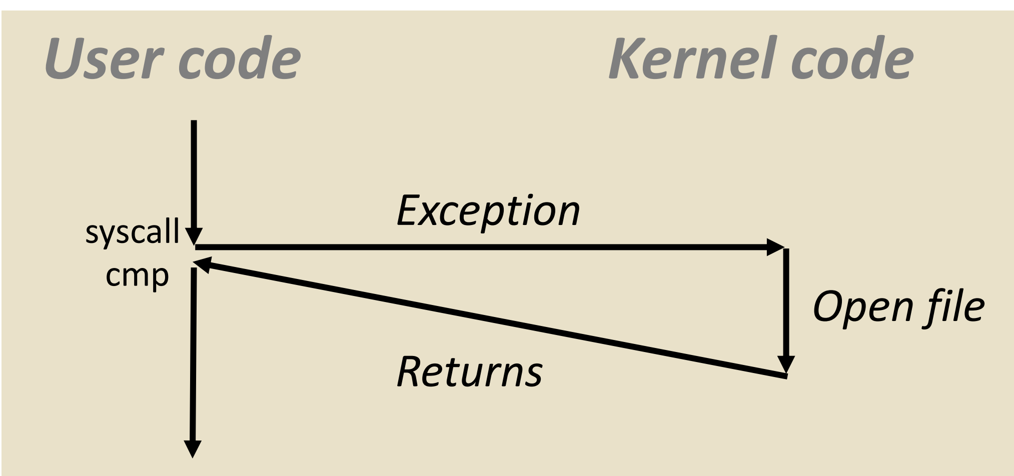
- **Each x86-64 system call has a unique ID number**
- **Examples:**

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

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,%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
```



- `%eax` 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 Example: Opening File

- User calls: **`open(filename, options)`**
- Calls __o .

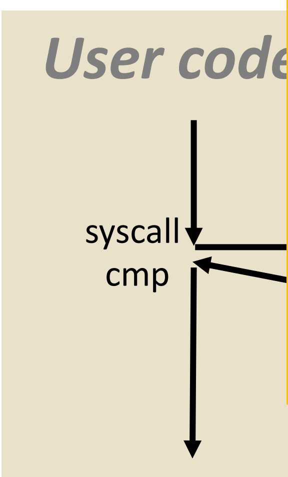
```
000000000
...
e5d79:                                   l #2
e5d7e:                                   n %rax
e5d80:
...
e5dfa:
```

Almost like a function call

- **Transfer of control**
- **On return, executes next instruction**
- **Passes arguments using calling convention**
- **Gets result in `%rax`**

One Important exception!

- **Executed by Kernel**
- **Different set of privileges**
- **And other differences:**
  - **E.g., "address" of "function" is in `%rax`**
  - **Uses `errno`**
  - **Etc.**



l number
rdi,
r8, %r9

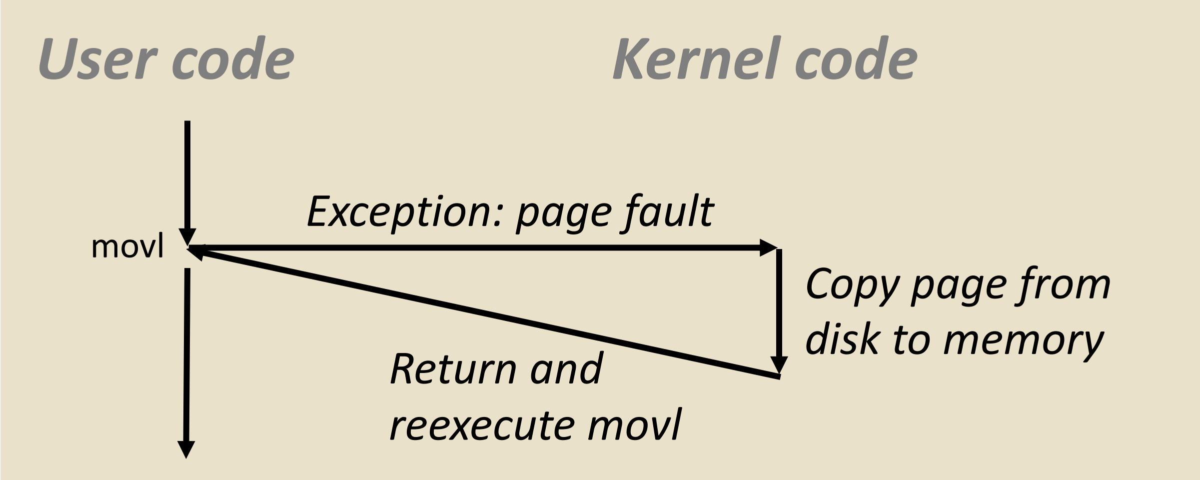
rror
corresponding to negative
`errno`

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;
}
```

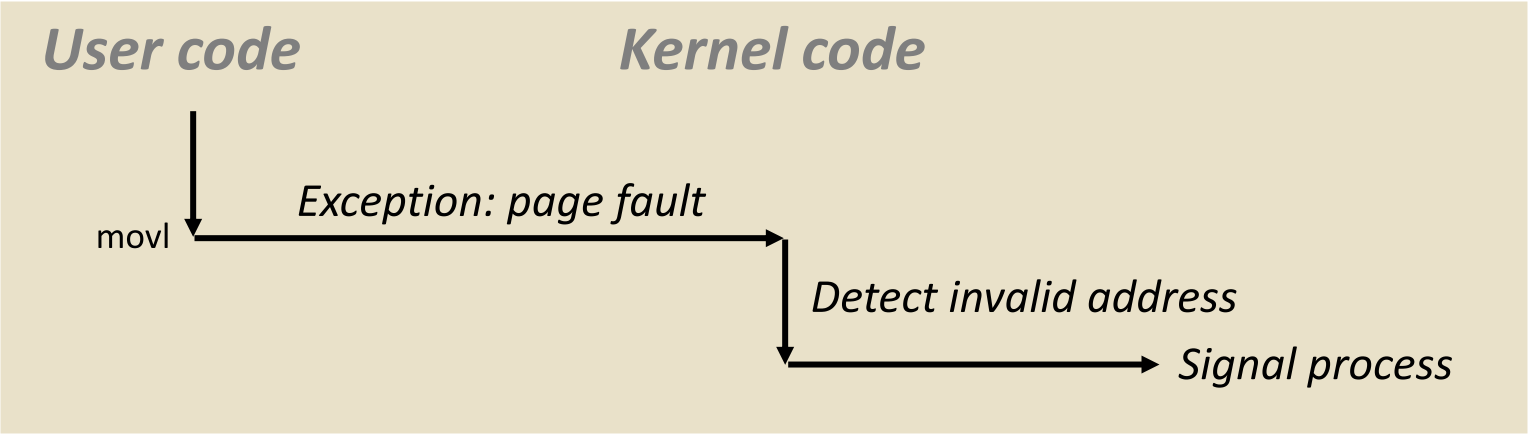
```
80483b7:       c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



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"

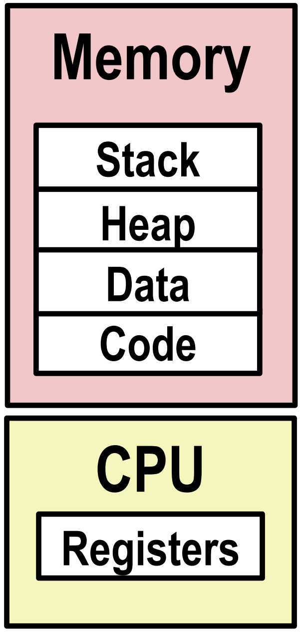
Today

- Exceptional Control Flow
- Exceptions
- **Processes**
- Process Control

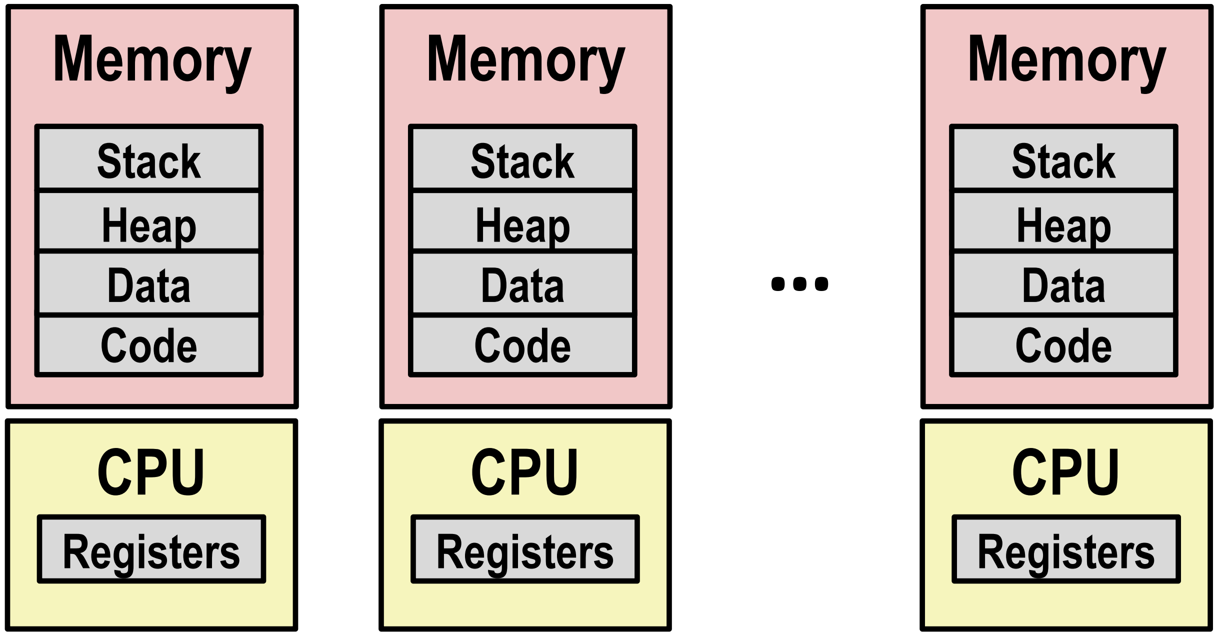
Processes

- **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*



Multiprocessing: The Illusion



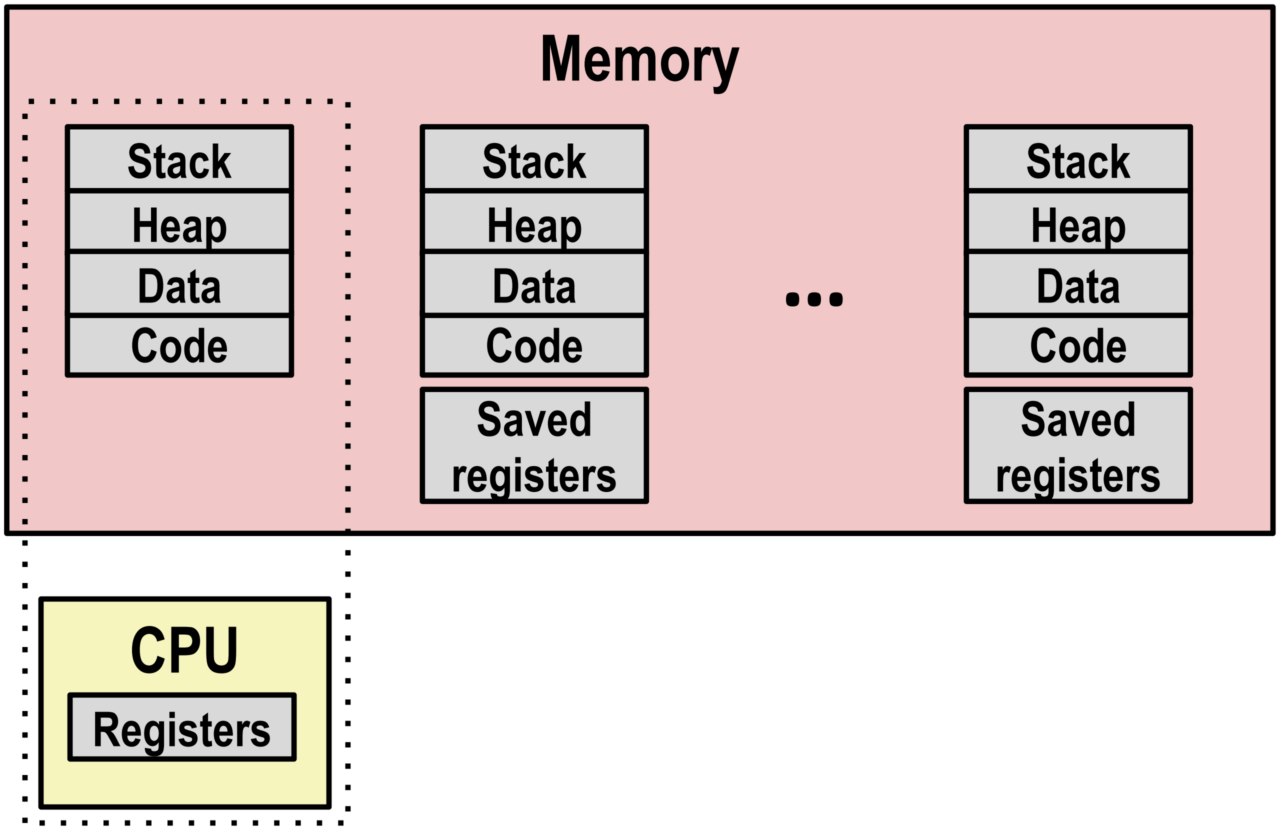
- **Computer runs many processes simultaneously**
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Multiprocessing Example

```
kai — top — 132×56
~ — top
Processes: 389 total, 2 running, 387 sleeping, 1508 threads                                                          18:26:55
Load Avg: 1.75, 1.39, 1.33  CPU usage: 1.56% user, 2.16% sys, 96.27% idle  SharedLibs: 288M resident, 66M data, 69M linkedit.
MemRegions: 61002 total, 3291M resident, 132M private, 2284M shared. PhysMem: 13G used (2730M wired), 3093M unused.
VM: 1899G vsize, 1297M framework vsize, 530885(0) swapins, 634630(0) swapouts.
Networks: packets: 15511710/14G in, 8991616/2034M out. Disks: 1630866/32G read, 1416020/38G written.

PID   COMMAND      %CPU TIME     #TH  #WQ  #PORT MEM    PURG   CMPRS  PGRP PPID STATE    BOOSTS          %CPU_ME %CPU_OTHRS UID
6013  screencaptur 1.5  00:00.42 2    1    54    3264K  636K   0B     369  369  sleeping *0[1]           0.04329 0.00000    501
6012  MTLCompilerS 0.0  00:00.06 2    2    27    5760K  0B     0B     6012 1    sleeping  0[2]           0.00000 0.00000    501
6011  screencaptur 0.2  00:00.37 3    1    200-  13M    132K-  0B     6011 1    sleeping *0[484+]        0.00000 0.04329    501
6009  top          3.6  00:01.65 1/1  0    28    5936K  0B     0B     6009 6003 running  *0[1]           0.00000 0.00000    0
6003  bash         0.0  00:00.01 1    0    21    788K   0B     0B     6003 6002 sleeping *0[1]           0.00000 0.00000    501
6002  login        0.0  00:00.02 2    1    30    1172K  0B     0B     6002 6001 sleeping *0[9]           0.00000 0.00000    0
6001  Terminal     0.9  00:03.46 10   5    280-  118M+  32M-   0B     6001 1    sleeping *0[34]          0.00105 0.00000    501
5999  Google Chrom 0.0  00:00.09 15   1    105   13M    4096B  0B     1593 1593 sleeping *0[3]           0.00000 0.00000    501
5997  Google Chrom 0.0  00:02.11 16   1    126   31M    4096B  0B     1593 1593 sleeping *0[5]           0.00000 0.00000    501
5994  Google Chrom 0.0  00:02.62 16   1    131   75M    8192B  0B     1593 1593 sleeping *0[7]           0.00000 0.00000    501
5981  mdworker_sha 0.0  00:00.08 3    1    54    4400K  0B     0B     5981 1    sleeping *0[1]           0.00000 0.00000    501
5978  Google Chrom 0.0  00:01.85 16   1    129   53M    4096B  0B     1593 1593 sleeping *0[4]           0.00000 0.00000    501
5973  Google Chrom 0.0  00:00.13 9    1    77    12M    4096B  0B     1593 1593 sleeping *0[3]           0.00000 0.00000    501
5969  netbiosd     0.0  00:00.03 2    2    25    2392K  0B     0B     5969 1    sleeping *0[1]           0.00000 0.00000    222
5961  eapolclient  0.0  00:00.07 3    1    52    2812K  0B     0B     5961 62   sleeping *0[1]           0.00000 0.00000    501
5950  com.apple.ac 0.0  00:00.02 2    2    31    936K   0B     0B     5950 1    sleeping  0[64]          0.00000 0.00000    501
5942  CoreServices 0.0  00:00.24 3    1    166   4876K  0B     0B     5942 1    sleeping *0[1]           0.00000 0.00000    501
5930  mdworker_sha 0.0  00:00.22 3    1    56    3980K  0B     0B     5930 1    sleeping *0[1]           0.00000 0.00000    501
5929  mdworker_sha 0.0  00:00.22 3    1    56    3844K  0B     0B     5929 1    sleeping *0[1]           0.00000 0.00000    501
5928  mdworker_sha 0.0  00:00.21 3    1    56    3956K  0B     0B     5928 1    sleeping *0[1]           0.00000 0.00000    501
5927  mdworker_sha 0.0  00:00.23 3    1    56    3912K  0B     0B     5927 1    sleeping *0[1]           0.00000 0.00000    501
5915  Xcode        0.0  00:02.44 3    1    225   71M    0B     0B     5915 1    sleeping *0[115]         0.00000 0.00000    501
5895  com.apple.We 0.0  00:00.17 4    1    74    3544K  0B     0B     5895 1    sleeping *1[11]          0.00000 0.00000    501
5894  com.apple.We 0.0  00:00.87 4    1    100   9048K  0B     0B     5894 1    sleeping *2[11]          0.00000 0.00000    501
5893  com.apple.We 0.0  00:02.82 8    2    126   73M    24M    0B     5893 1    sleeping  0[347]         0.00000 0.00000    501
5890  wpsoffice    1.8  09:13.67 16   1    364   347M   708K   0B     5890 1    sleeping *0[661]         0.00000 0.00000    501
5873  com.apple.We 0.0  00:00.19 4    1    87    3848K  0B     0B     5873 1    sleeping  0[154]         0.00000 0.00000    501
5861  mdworker_sha 0.0  00:01.10 4    1    59    46M    0B     0B     5861 1    sleeping *0[1]           0.00000 0.00000    501
5828  mdworker_sha 0.0  00:01.67 4    1    59    43M    0B     0B     5828 1    sleeping *0[1]           0.00000 0.00000    501
5827  mdworker_sha 0.0  00:02.11 4    1    59    43M    0B     0B     5827 1    sleeping *0[1]           0.00000 0.00000    501
5432  rcd          0.0  00:00.12 2    1    56    1860K  0B     0B     5432 1    sleeping *0[1]           0.00000 0.00000    501
5287  ssh-agent    0.0  00:00.01 1    0    21    824K   0B     0B     5287 1    sleeping *0[1]           0.00000 0.00000    501
5259  mdworker_sha 0.0  00:07.67 4    1    63    48M    0B     0B     5259 1    sleeping *0[1]           0.00000 0.00000    501
5253  syspolicyd   0.0  00:01.16 2    1    64    9008K  64K    0B     5253 1    sleeping  0[139]         0.00000 0.00000    0
5228  mdworker_sha 0.0  00:00.12 3    1    48    3164K  0B     0B     5228 1    sleeping *0[1]           0.00000 0.00000    89
5198  com.apple.Co 0.0  00:00.08 2    2    40    3000K  0B     0B     5198 1    sleeping  0[1]           0.00000 0.00000    501
3432  applessdstat 0.0  00:00.01 2    1    33    788K   0B     0B     3432 1    sleeping *0[1]           0.00000 0.00000    0
3400  cupsd        0.0  00:00.52 3    1    51    3872K  0B     2116K  3400 1    sleeping  0[0]           0.00000 0.00000    0
3379  fpsd         0.0  00:00.02 2    1    24    828K   0B     820K   3379 1    sleeping  0[0]           0.00000 0.00000    265
3353  sandboxd     0.0  00:03.60 5    4    67    41M    0B     0B     3353 1    sleeping *0[1]           0.00000 0.00000    0
3349  PerfPowerSer 0.0  00:05.51 3    2    165   7916K  256K   1056K  3349 1    sleeping  0[160]         0.00000 0.00000    0
3256- JuniperSetup 0.0  00:01.90 4    1    163   5792K  0B     1908K  3256 1    sleeping *0[1052]        0.00000 0.00000    501
3047  Google Chrom 0.0  00:22.58 8    1    72    88M    0B     18M    1593 1593 sleeping *0[5]           0.00000 0.00000    501
3046  Google Chrom 0.0  02:46.45 15   1    122   124M   0B     17M    1593 1593 sleeping *0[8]           0.00000 0.00000    501
3044  Google Chrom 0.0  00:01.50 16   1    119   26M    0B     20M    1593 1593 sleeping *0[6]           0.00000 0.00000    501
2971  MIDIServer   0.0  00:01.99 5    1    129   1968K  0B     656K   2971 1    sleeping *0[1]           0.00000 0.00000    501
2918  com.apple.We 0.0  00:01.39 4    1    86    3972K  0B     2892K  2918 1    sleeping  0[1919]        0.00000 0.00000    501
2901  garcon       0.0  00:02.30 3    1    163   12M    0B     8604K  2901 1    sleeping *0[1813]        0.00000 0.00000    501
```

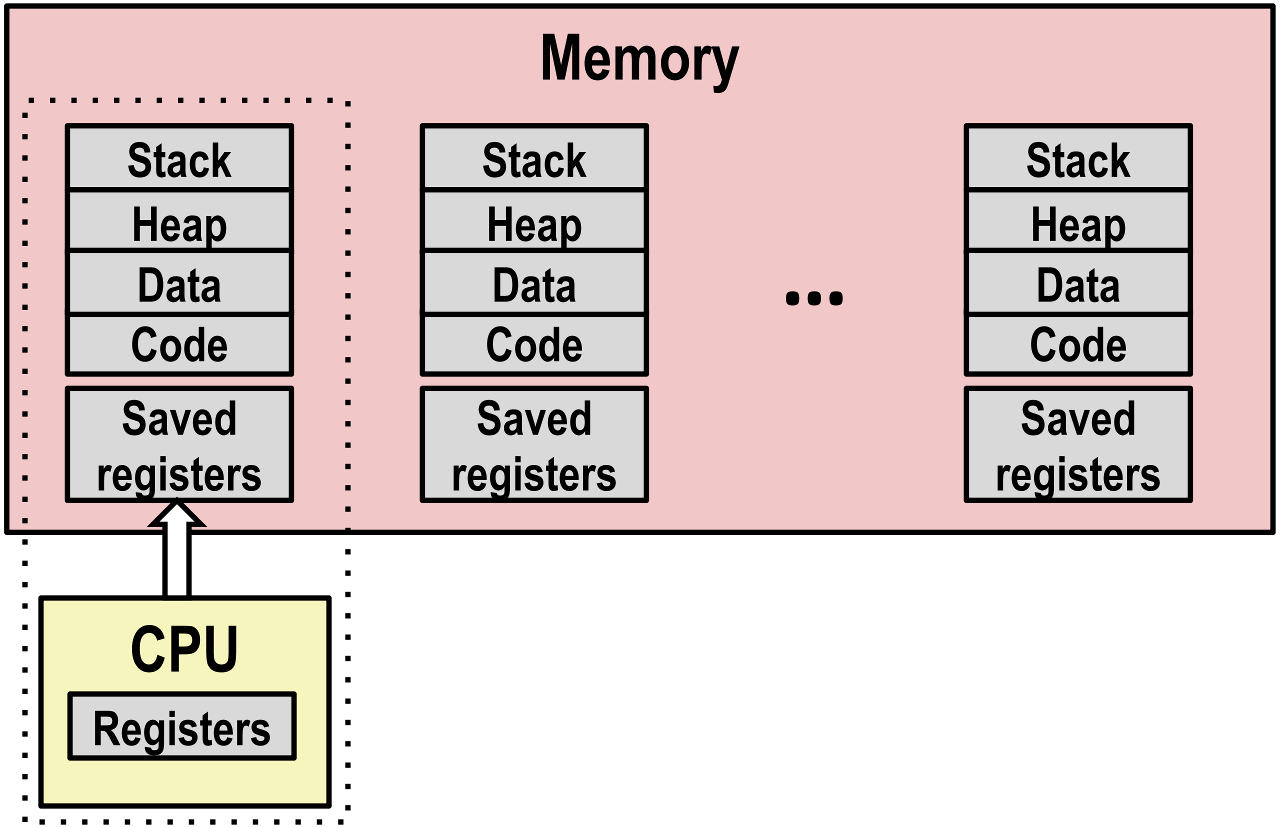
- **Running program “top” on Mac**
  - System has 389 processes, 2 of which are active
  - Identified by Process ID (PID)

Multiprocessing: The (Traditional) Reality



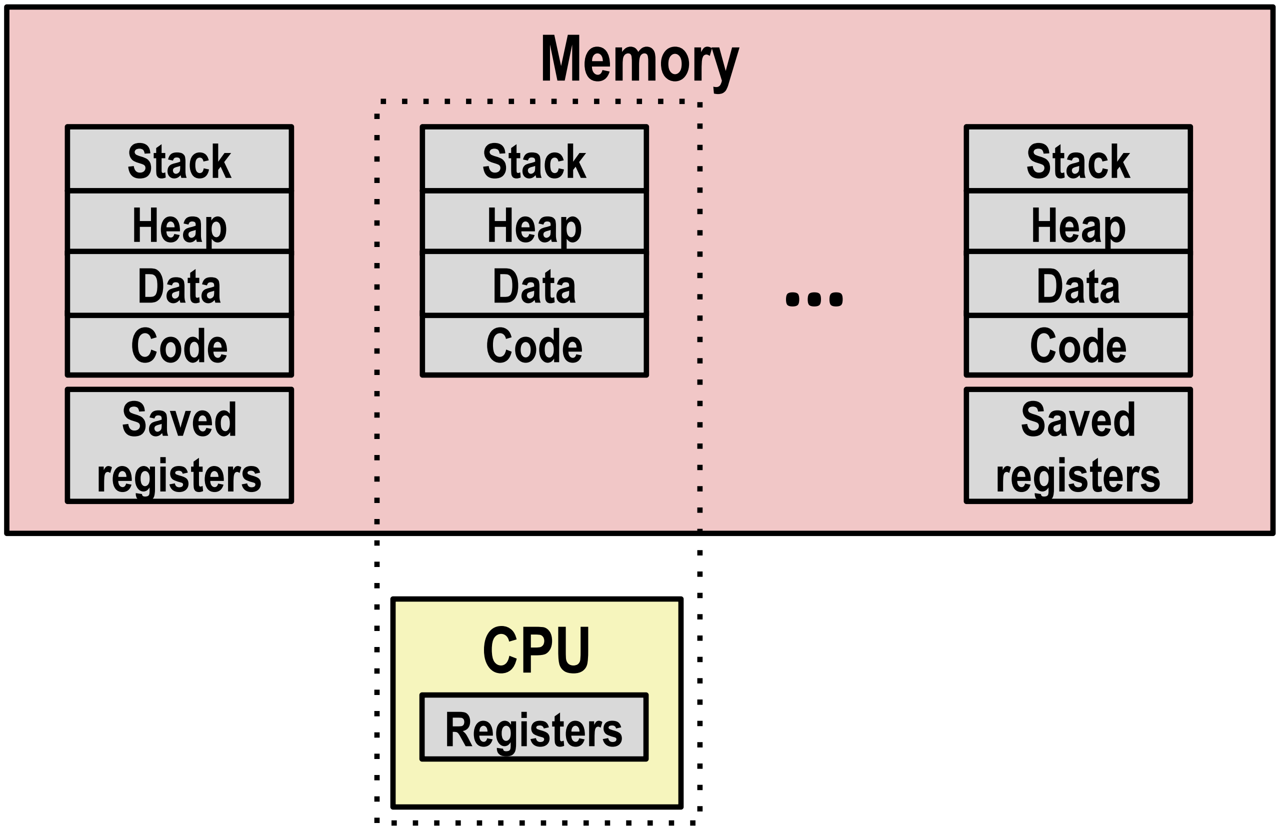
- **Single processor executes multiple processes concurrently**
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



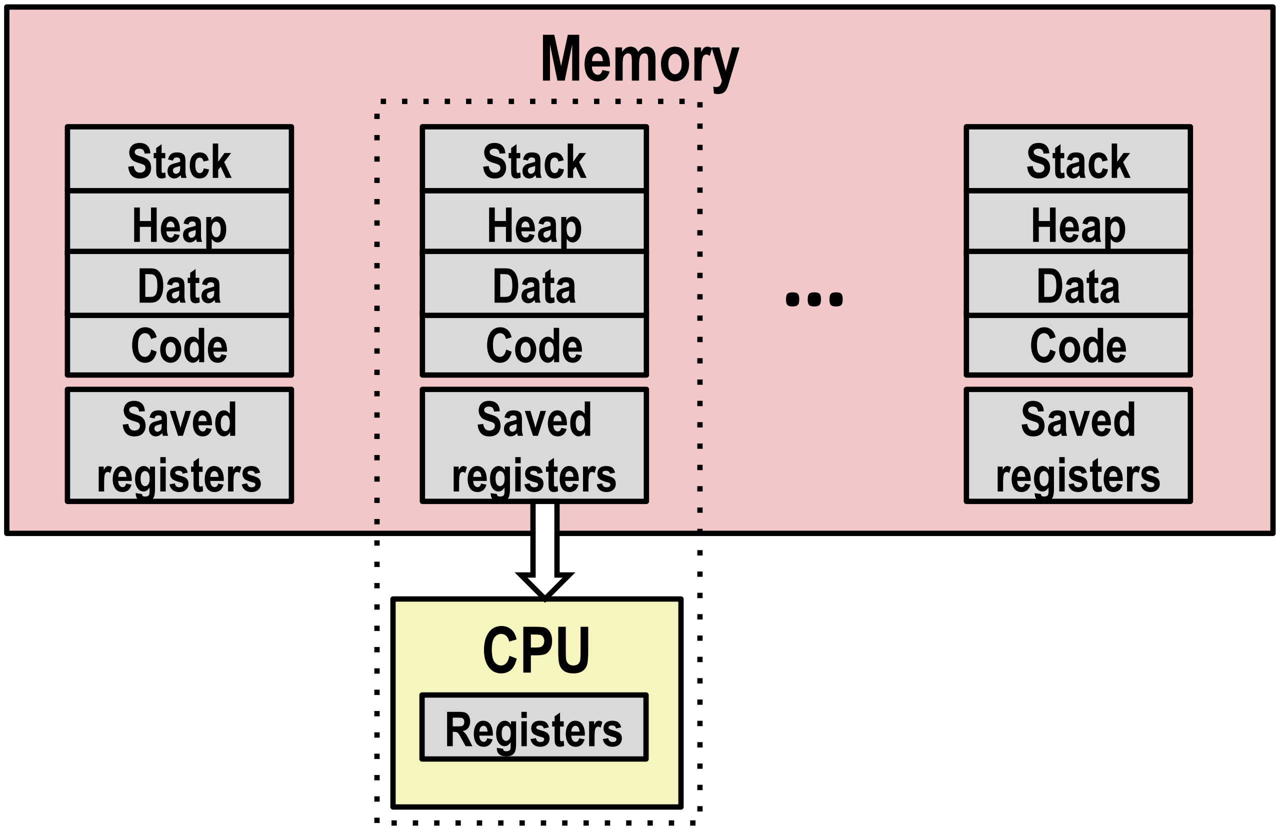
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



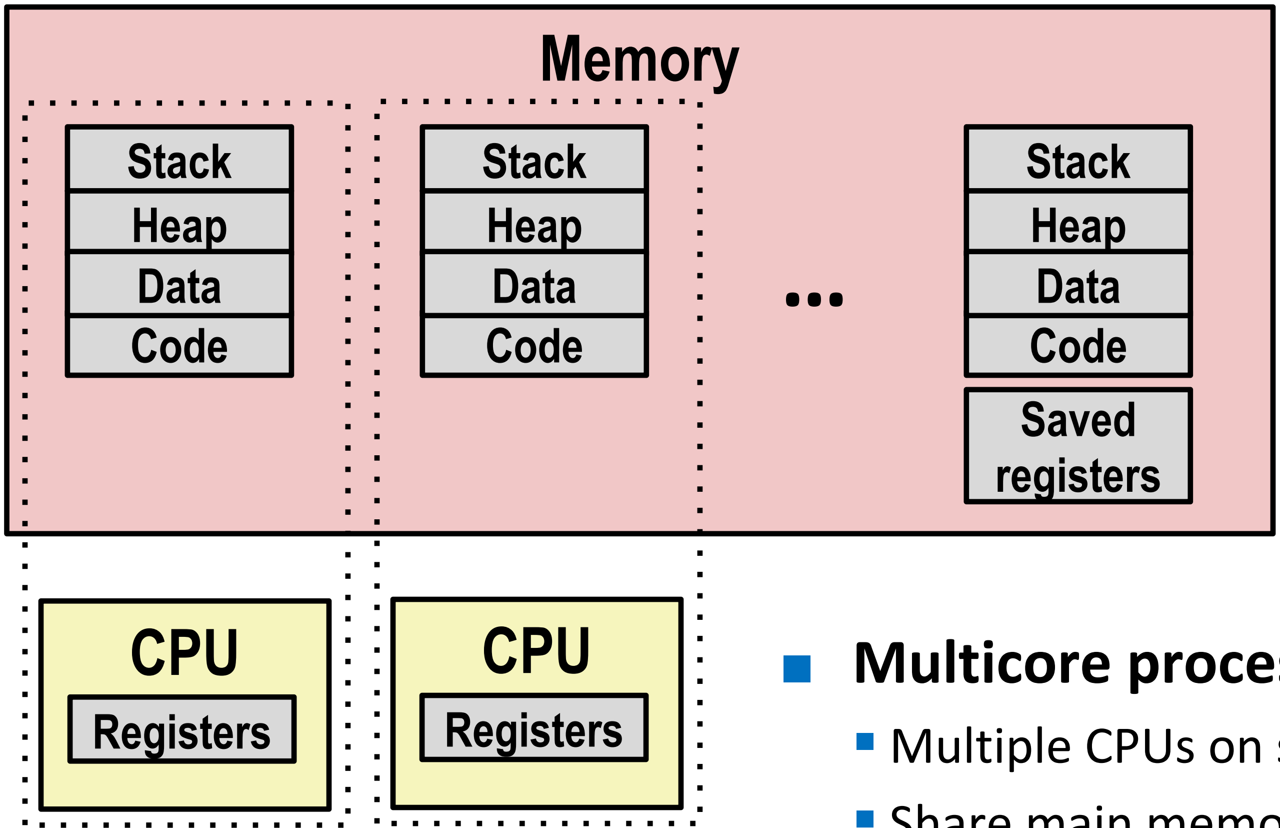
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



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

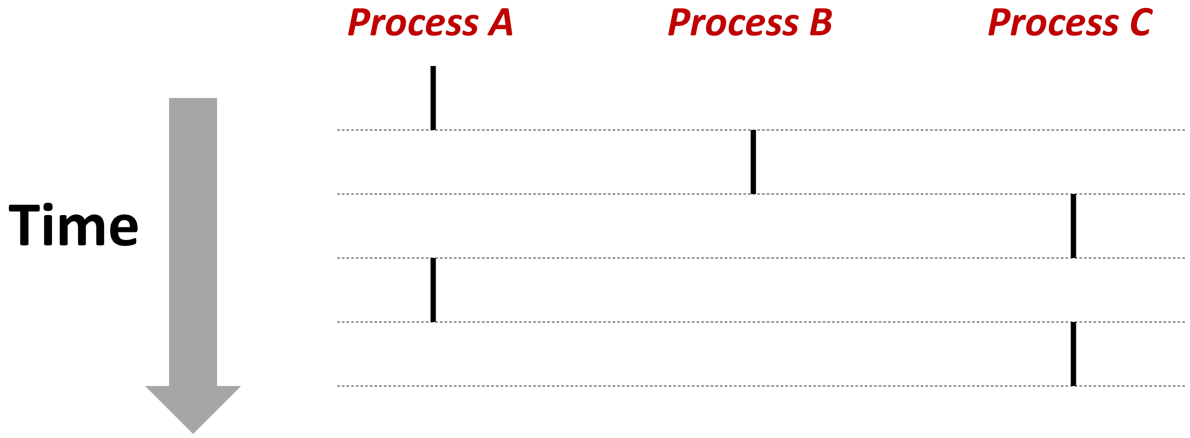
Multiprocessing: The (Traditional) Reality



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

Concurrent Processes

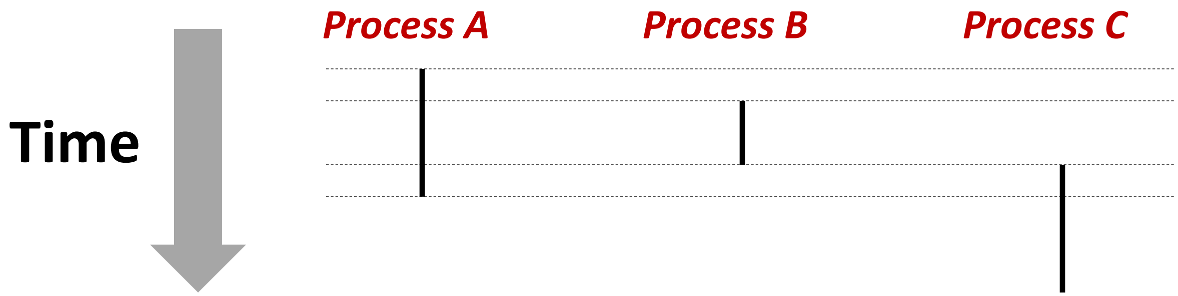
- **Each process is a logical control flow.**
- **Two processes *run concurrently* (*are concurrent)* if their flows overlap in time**
- **Otherwise, they are *sequential***
- **Examples (running on single core):**
  - Concurrent: A & B, A & C
  - Sequential: B & C



User View of Concurrent Processes

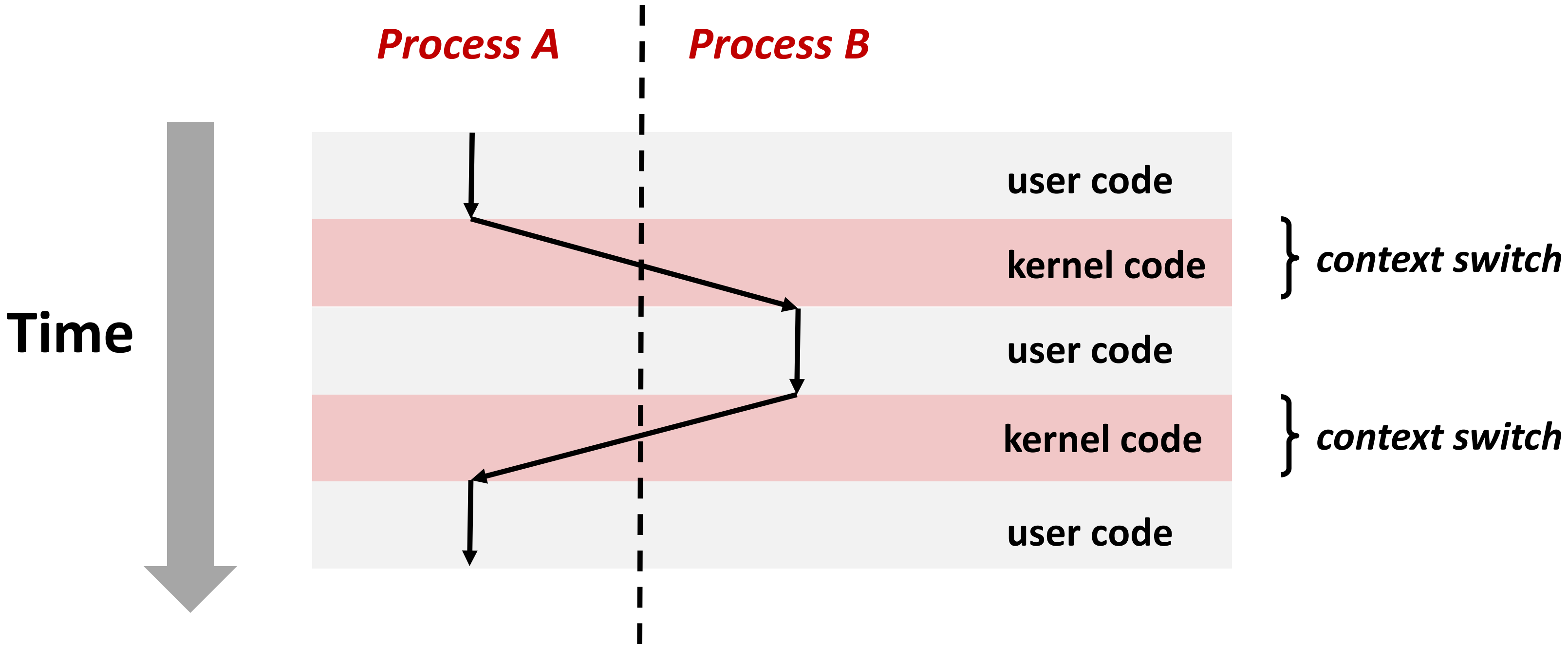
- **Control flows for concurrent processes are physically disjoint in time**

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



Context Switching

- **Processes are managed by a shared chunk of 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***



Execrise

- P509 Textbook (Chinese Version)
- Execrise 8.1

Today

- 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:**
  - You 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(-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);
}
```

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

- But, must think about application. Not always appropriate to exit when something goes wrong.

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, 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.**

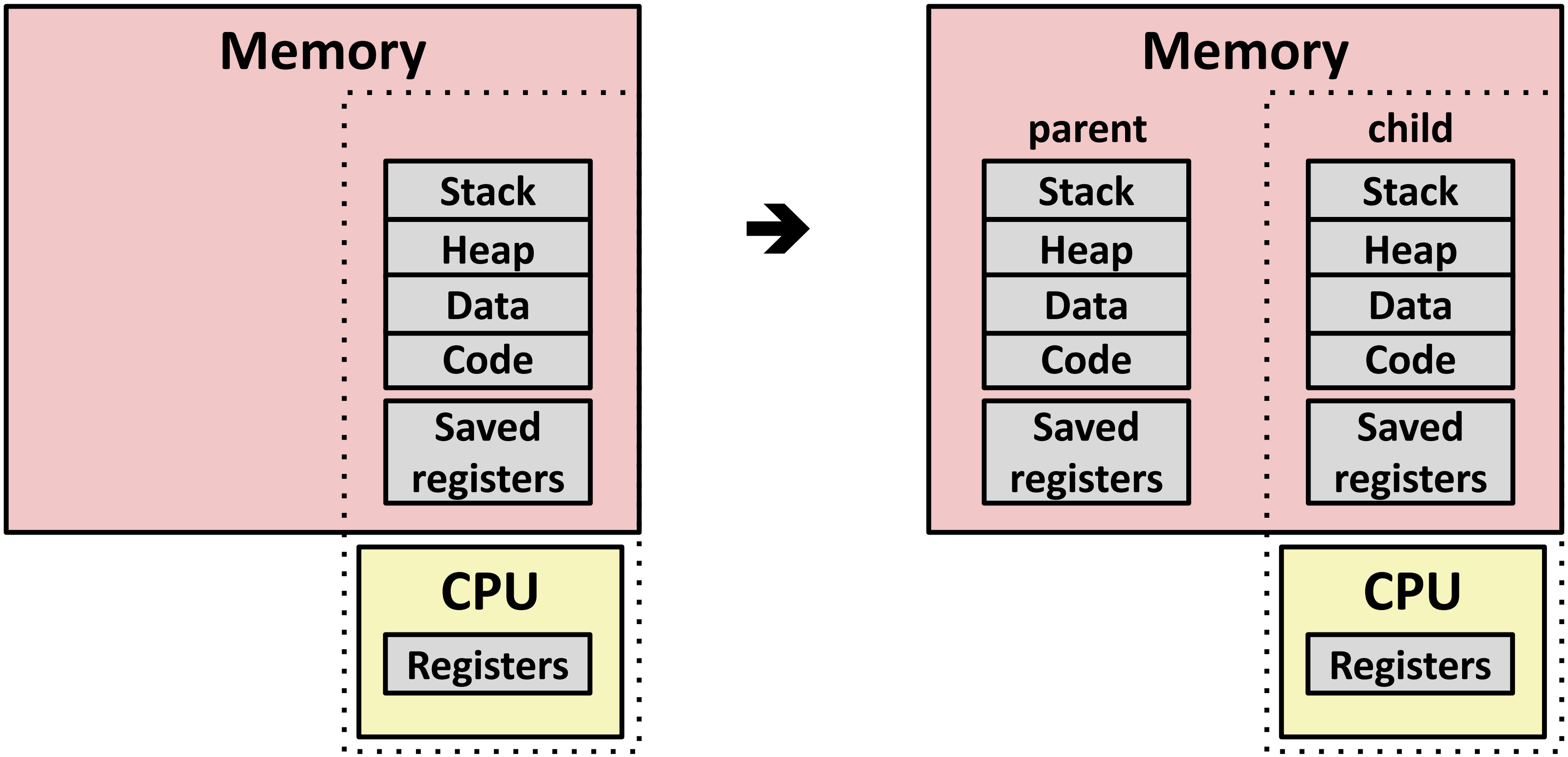
Creating Processes

- ***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

`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
```

Execrise

- P516 Textbook (Chinese Version)
- Execrise 8.2

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`

```
/* 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;
}
```

```
linux> ./fork2
parent: x=0
parent: x=-1
child : x=2
child : x=3
```

- **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

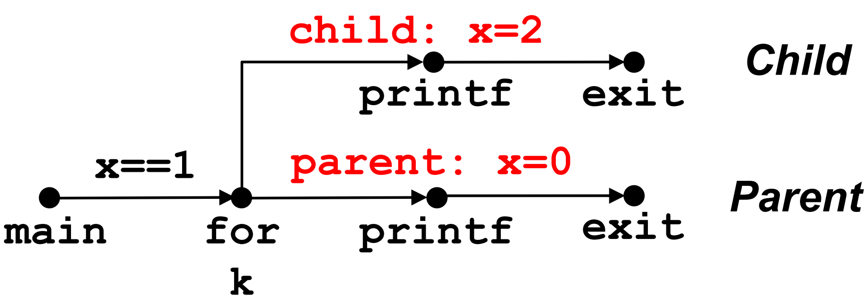
Modeling `fork` with Process Graphs

- **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
  - two events in the graph has a fixed happening order

Process Graph Example

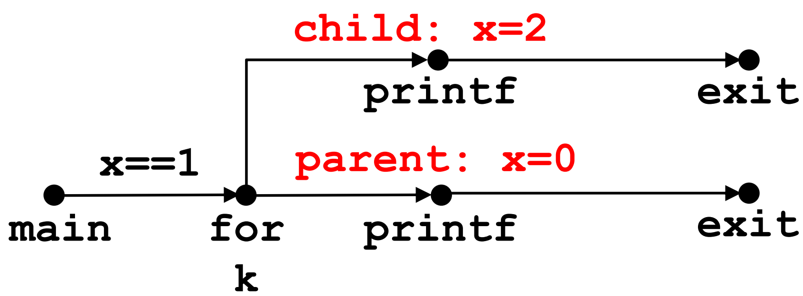
```c
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    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

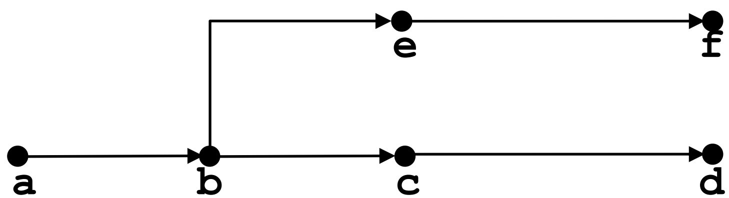


Interpreting Process Graphs

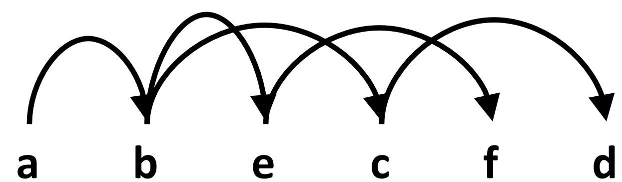
- **Original graph:**



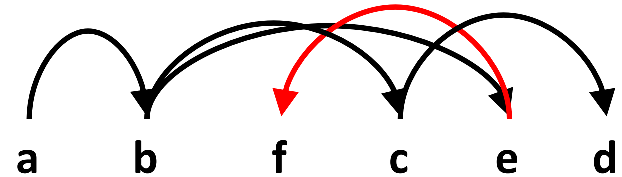
- **Relabled graph:**



Feasible total ordering:

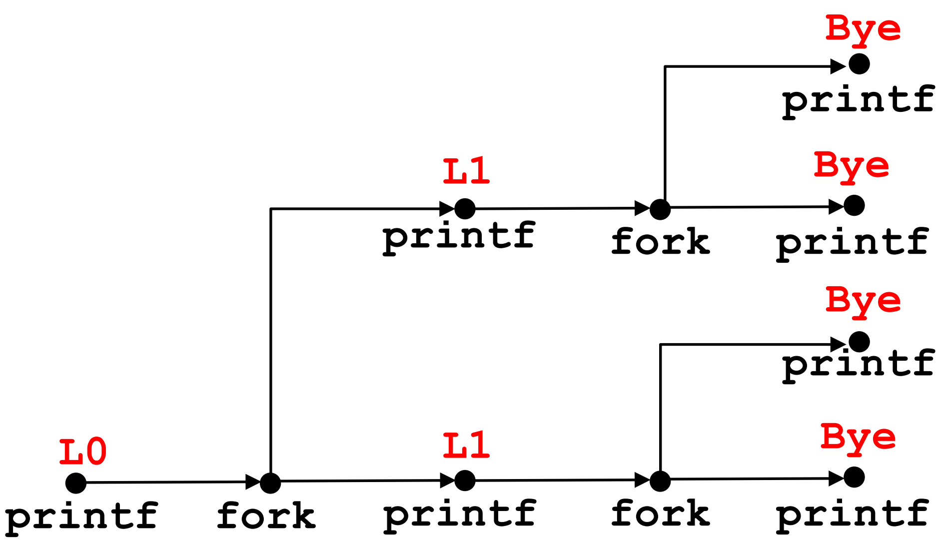


Infeasible total ordering:



fork Example: Two consecutive forks

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

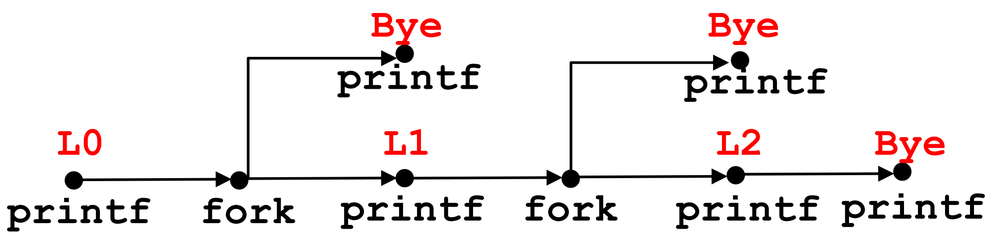


Feasible	Infeasible
L0	L0
L1	Bye
Bye	L1
Bye	Bye
L1	L1
Bye	Bye
Bye	Bye

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

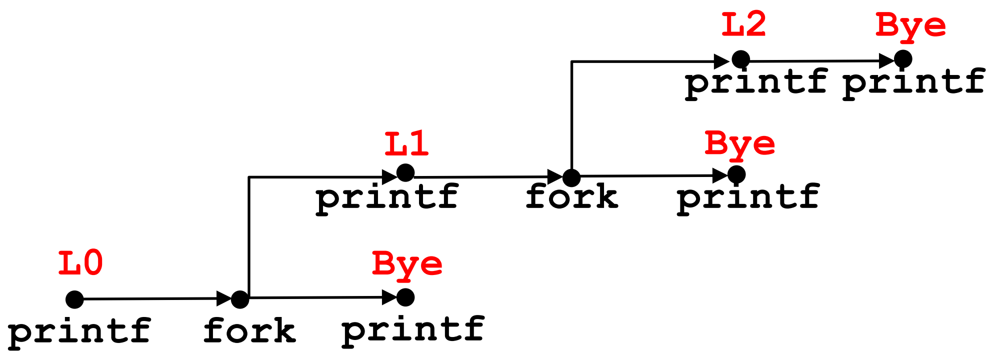


Feasible	Infeasible
L0	L0
L1	Bye
Bye	L1
Bye	Bye
L2	Bye
Bye	L2

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	Infeasible
L0	L0
Bye	Bye
L1	L1
L2	Bye
Bye	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

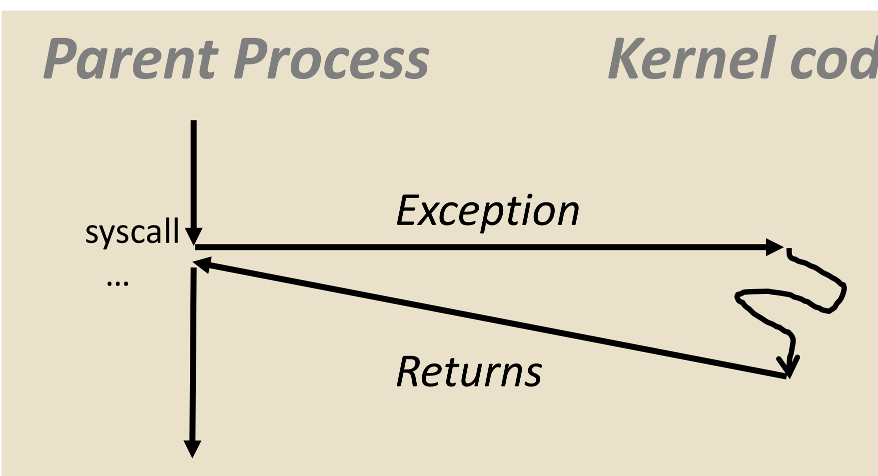
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9   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



And, potentially other user processes, including a child of parent

`wait`: Synchronizing with Children

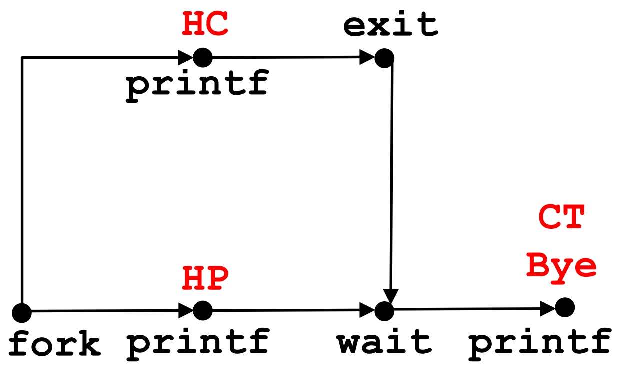
- **Parent reaps a child by calling the `wait` function**

- **`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

```c
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(s):

HC	HP
HP	HC
CT	CT
Bye	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

```c
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

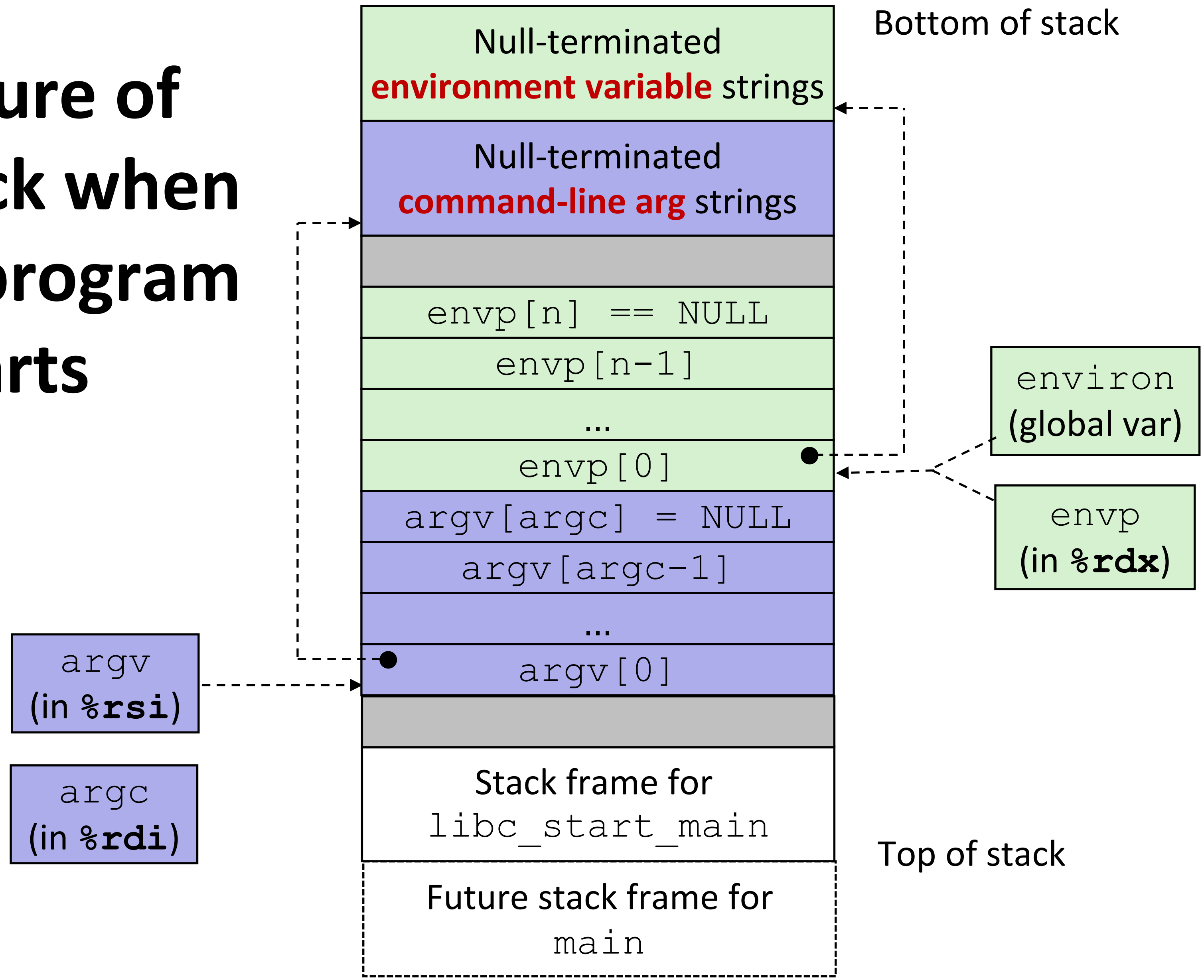
Execrise

- P518 Textbook, Exercise 8.3
- P520 Textbook, Exercise 8.4

`execve`: Loading and Running Programs

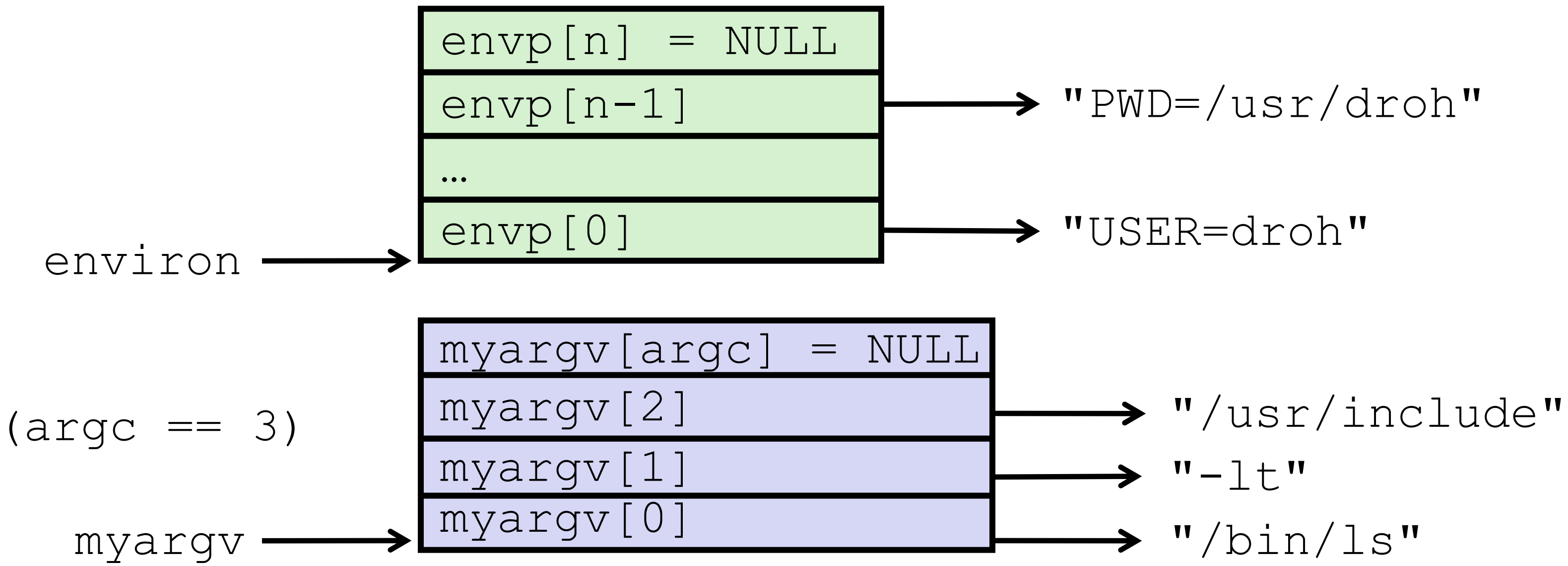
- **`int execve(char *filename, char *argv[], char *envp[])`**
- **Loads and runs in the current process:**
  - Executable file **`filename`**
    - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
  - ...with **argument** list **`argv`**
    - By convention **`argv[0]==filename`**
  - ...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

Structure of the stack when a new program starts



`execve` Example

- **Execute** `"/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 any single core
  - 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

Exception Handling in MIT JOS Lab Based on Intel 80386

```c
// Trap numbers
// These are processor defined:
#define T_DIVIDE     0          // divide error
#define T_DEBUG      1          // debug exception
#define T_NMI        2          // non-maskable interrupt
#define T_BRKPT      3          // breakpoint
#define T_OFLOW      4          // overflow
#define T_BOUND      5          // bounds check
#define T_ILLOP      6          // illegal opcode
#define T_DEVICE     7          // device not available
#define T_DBLFLT     8          // double fault
/* #define T_COPROC  9 */       // reserved (not generated by recent processors)
#define T_TSS       10          // invalid task switch segment
#define T_SEGNP     11          // segment not present
#define T_STACK     12          // stack exception
#define T_GPFLT     13          // general protection fault
#define T_PGFLT     14          // page fault
/* #define T_RES    15 */       // reserved
#define T_FPERR     16          // floating point error
#define T_ALIGN     17          // aligment check
#define T_MCHK      18          // machine check
#define T_SIMDERR   19          // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL   48          // system call
#define T_DEFAULT   500         // catchall

// Hardware IRQ numbers. We receive these as (IRQ_OFFSET+IRQ_WHATEVER)
// IRQ_OFFSET is defined in kern/picirq.h = 32
#define IRQ_TIMER        0
#define IRQ_KBD          1
#define IRQ_SPURIOUS     7
#define IRQ_IDE         14
#define IRQ_ERROR       19
```

```
      IDT                     trapentry.S         trap.c

+----------------+
|   &handler1    |---------> handler1:           trap (struct Trapframe *tf)
|                |             // do stuff        {
|                |             call trap            // handle the exception/interrupt
|                |             // ...             }
+----------------+
|   &handler2    |--------> handler2:
|                |            // do stuff
|                |            call trap
|                |            // ...
+----------------+
       .
       .
       .
+----------------+
|   &handlerX    |--------> handlerX:
|                |             // do stuff
|                |             call trap
|                |             // ...
+----------------+
```

```
/* The TRAPHANDLER macro defines a globally-visible function for handling
 * a trap.  It pushes a trap number onto the stack, then jumps to _alltraps.
 * Use TRAPHANDLER for traps where the CPU automatically pushes an error code.
 */
#define TRAPHANDLER(name, num)                                          \
        .globl name;            /* define global symbol for 'name' */   \
        .type name, @function;  /* symbol type is function */           \
        .align 2;               /* align function definition */         \
        name:                   /* function starts here */              \
        pushl $(num);                                                   \
        jmp _alltraps

_alltraps:
        pushl %ds
        pushl %es
        pushal

        movw $GD_KD, %ax
        movw %ax, %ds
        movw %ax, %es

        pushl %esp /* esp as an argument to trap*/
        call trap
        addl $4, %esp /* jump esp */

        popal
        popl %es
        popl %ds
        addl $8, %esp /* jump trap number and error c
        iret
```

```
.data
.global vectors
vectors:
        .long vector0
        .long vector1
        .long vector2
        .long vector3
        .long vector4
        .long vector5
```

```
        TRAPHANDLER_NOEC(vector0,       0)
        TRAPHANDLER_NOEC(vector1,       1)
        TRAPHANDLER_NOEC(vector2,       2)
        TRAPHANDLER_NOEC(vector3,       3)
        TRAPHANDLER_NOEC(vector4,       4)
        TRAPHANDLER_NOEC(vector5,       5)
        TRAPHANDLER_NOEC(vector6,       6)
        TRAPHANDLER_NOEC(vector7,       7)
        TRAPHANDLER     (vector8,       8)
        TRAPHANDLER_NOEC(vector9,       9)
        TRAPHANDLER     (vector10,      10)
```

```
void
idt_init(void)
{
        extern struct Segdesc gdt[];

        // LAB 3: Your code here.
        int i;

        for (i = 0; i < 256; i++)
                SETGATE(idt[i], 0, GD_KT, vectors[i], 0)

        // Difference between interrupt gate and trap gate:
        // After transfering the control, an interrupt gate clears IF to disable interrupts
        // however, trap gate does not change the IF flag
        // There will be errors in handling interrput when what's currently running is an exception
        // JOS only allows interrupt happens in user space, and forbid it in kernel space, when a
        // timer interrupt arrives, it will run sched_yield and eax is currently the syscall number
        // which is >0, so user panic at lib/syscall.c
        SETGATE(idt[T_SYSCALL], 0, GD_KT, vectors[T_SYSCALL], 3)
        SETGATE(idt[T_BRKPT], 0, GD_KT, vectors[T_BRKPT], 3)
        SETGATE(idt[T_OFLOW], 0, GD_KT, vectors[T_OFLOW], 3);
        SETGATE(idt[T_BOUND], 0, GD_KT, vectors[T_BOUND], 3);

        // Setup a TSS so that we get the right stack
        // when we trap to the kernel.
        ts.ts_esp0 = KSTACKTOP;
        ts.ts_ss0 = GD_KD;

        // Initialize the TSS field of the gdt.
        gdt[GD_TSS >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
                                        sizeof(struct Taskstate), 0);
        gdt[GD_TSS >> 3].sd_s = 0;

        // Load the TSS
        ltr(GD_TSS);

        // Load the IDT
        asm volatile("lidt idt_pd");
```

```c
void
trap(struct Trapframe *tf)
{
	if ((tf->tf_cs & 3) == 3) {
		// Trapped from user mode.
		// Copy trap frame (which is currently on the stack)
		// into 'curenv->env_tf', so that running the environment
		// will restart at the trap point.
		assert(curenv);
		curenv->env_tf = *tf;
		// The trapframe on the stack should be ignored from here on.
		tf = &curenv->env_tf;
	}

	// Dispatch based on what type of trap occurred
	trap_dispatch(tf);

	// If we made it to this point, then no other environment was
	// scheduled, so we should return to the current environment
	// if doing so makes sense.
	if (curenv && curenv->env_status == ENV_RUNNABLE)
		env_run(curenv);
	else
		sched_yield();
}
```

```c
static void
trap_dispatch(struct Trapframe *tf)
{
	// Handle processor exceptions.
	// LAB 3: Your code here.
	struct PushRegs *regs;
	switch (tf->tf_trapno) {
	case T_SYSCALL:
		regs = &(tf->tf_regs);
		regs->reg_eax = syscall(regs->reg_eax, regs->reg_edx,
			regs->reg_ecx, regs->reg_ebx, regs->reg_edi, regs->reg_esi);
		return;
	case T_PGFLT:
		page_fault_handler(tf);
		return;
	case T_BRKPT:
		monitor(tf);
		return;
	case T_DEBUG:
		monitor(tf);
		return;

	// Handle clock interrupts.
	// LAB 4: Your code here.
	case IRQ_OFFSET + IRQ_TIMER:
	// Add time tick increment to clock interrupts.
	// LAB 6: Your code here.
		time_tick();
		sched_yield();
		return;
	}
```

```c
// Dispatches to the correct kernel function, passing the arguments.
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t a5)
{
	// Call the function corresponding to the 'syscallno' parameter.
	// Return any appropriate return value.
	// LAB 3: Your code here.

	int32_t ret = 0;

	switch (syscallno) {
	case SYS_cputs:
		sys_cputs((char *)a1, (size_t)a2);
		break;
	case SYS_cgetc:
		ret = sys_cgetc();
		break;
	case SYS_getenvid:
		ret = sys_getenvid();
		break;
	case SYS_env_destroy:
		ret = sys_env_destroy((envid_t)a1);
		break;
	case SYS_yield:
		sys_yield();
		break;
	case SYS_exofork:
		ret = sys_exofork();
		break;
	case SYS_env_set_status:
		ret = sys_env_set_status((envid_t)a1, a2);
		break;
```