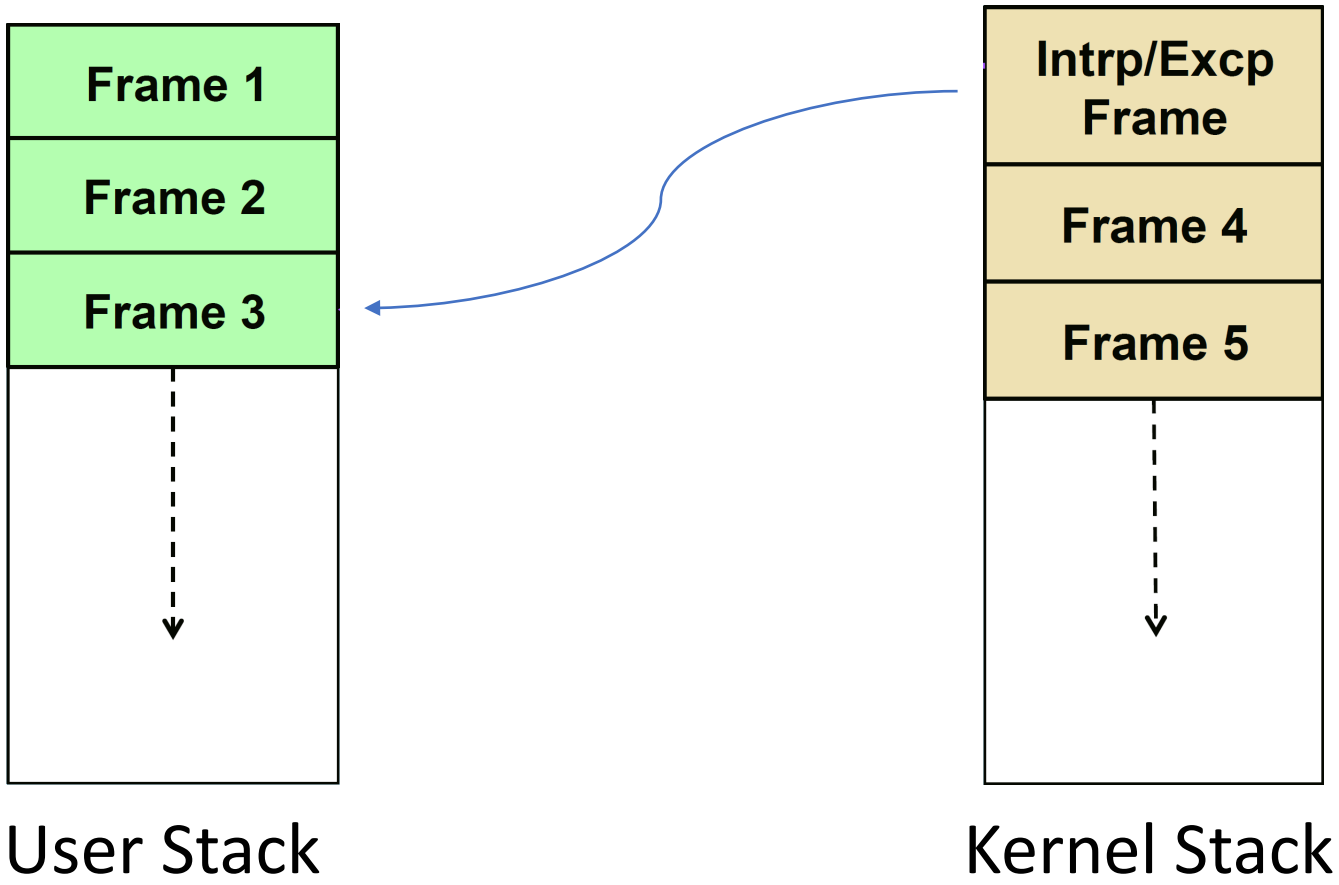
Architecture and OS, Process

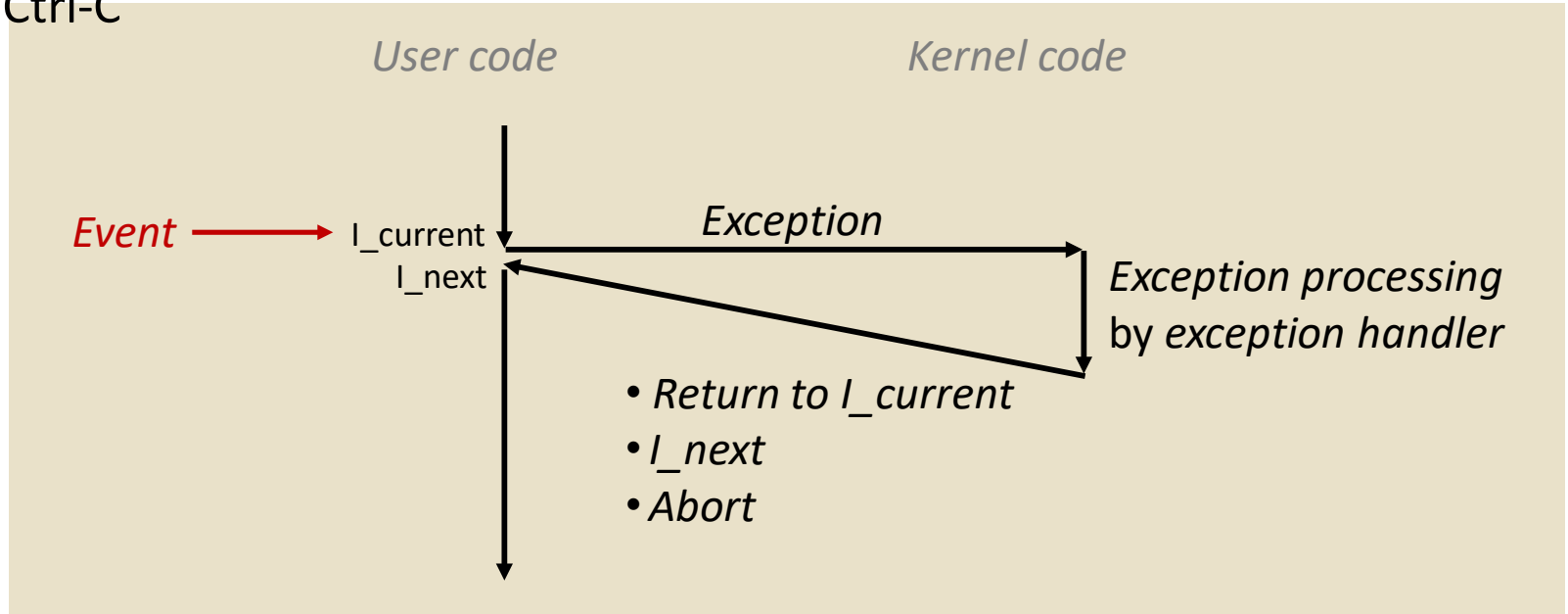
* Some material adapted from CS:APP, Prof. Bryant, O'Hallaron, Doeppner, Galvin, etc.

One Stack Per Mode



Exceptions

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



System Calls

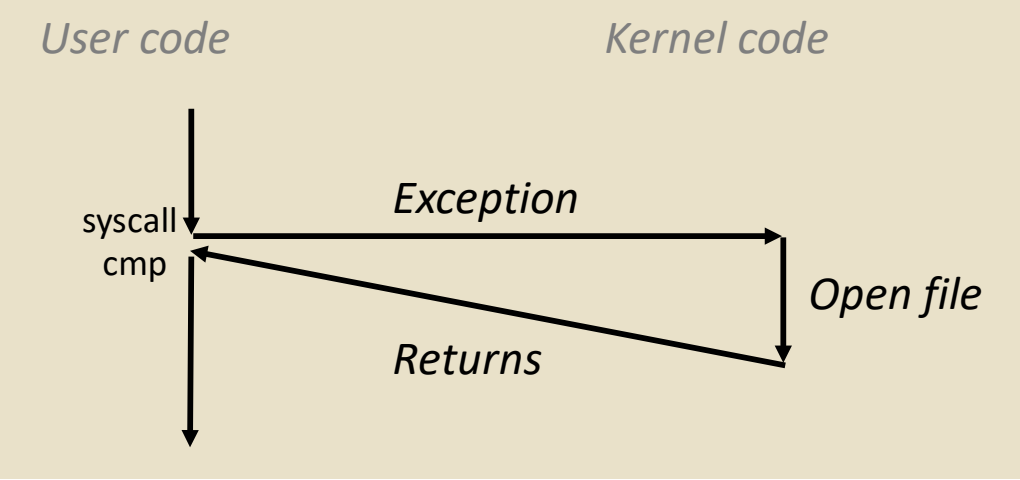
- Each x86-64 system call has a unique ID number
- Examples:

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
```



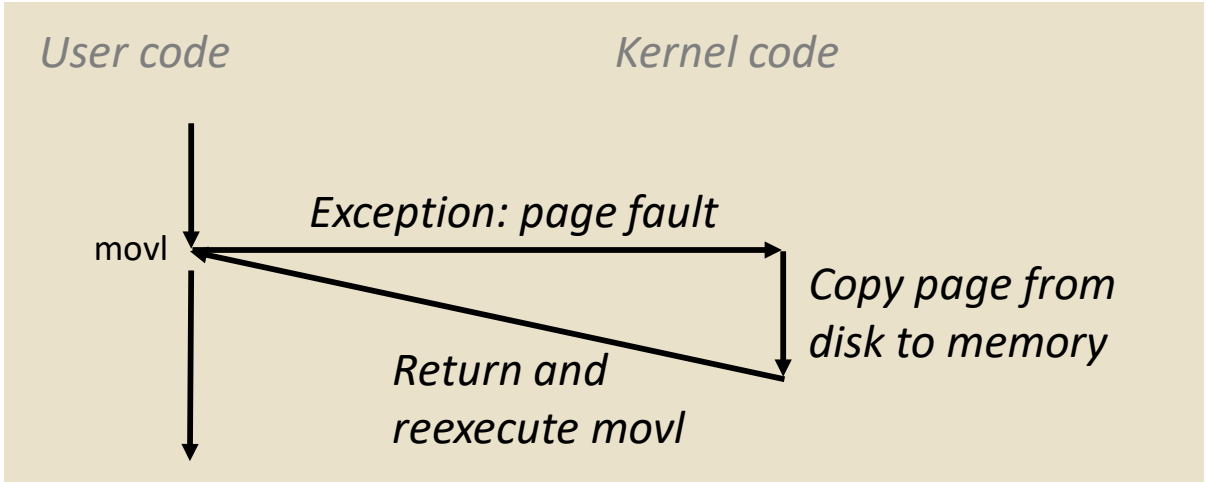
- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9`
- Return value in `%rax`
- Negative return value is an error : `errno`

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;
}
```

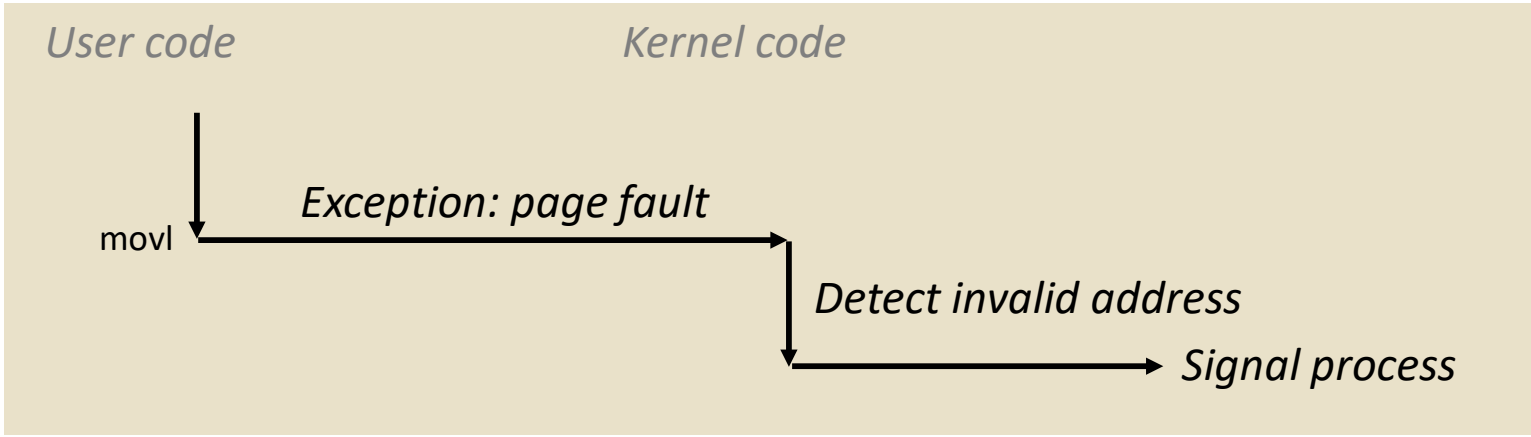
```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



Fault Example: Invalid Memory Reference

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

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

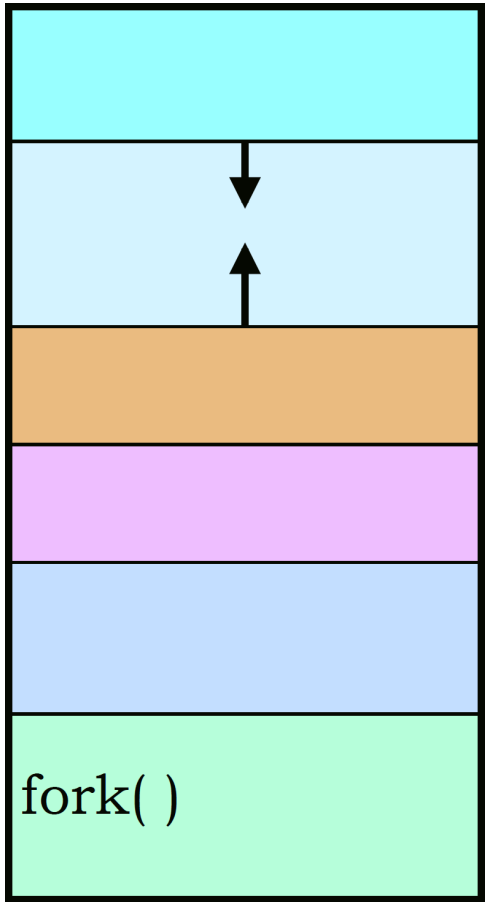


- User process exits with “segmentation fault”

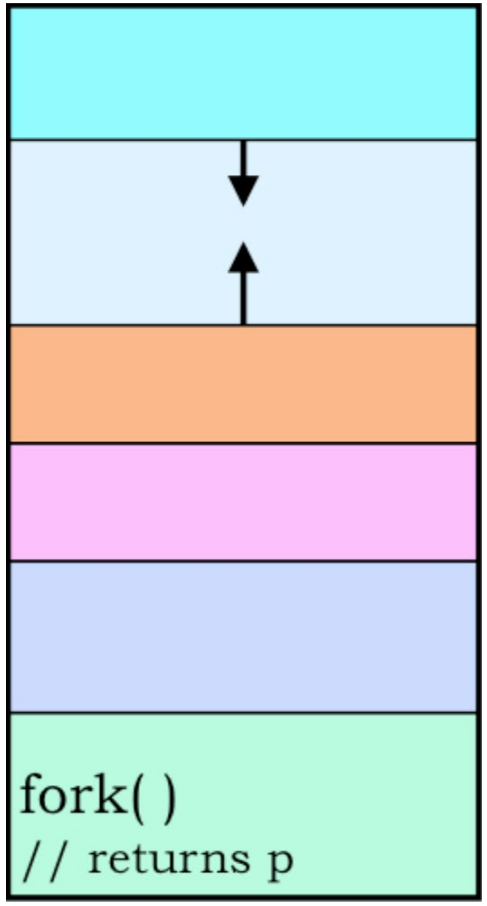
Creating Your Own Processes

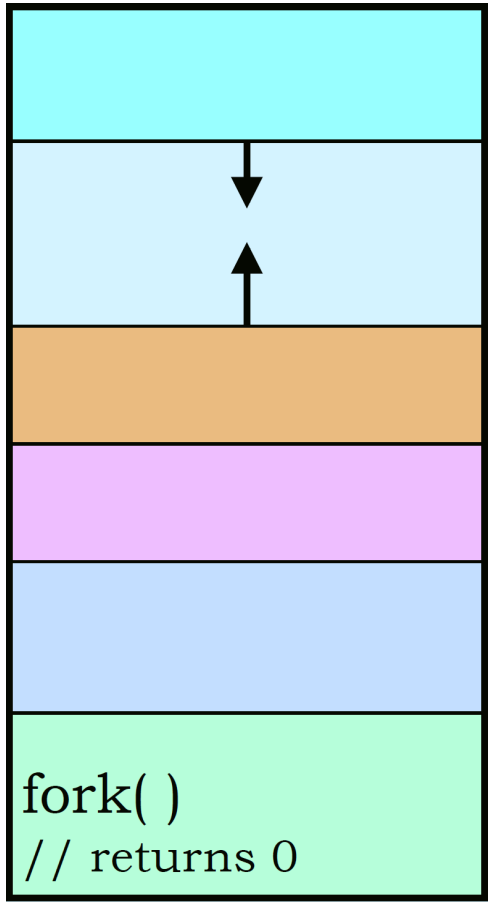
```
#include <unistd.h>
int main( ) {
    pid_t pid;
    if ((pid = fork()) == 0) {
        /* new process starts running here */
    }
    /* old process continues here */
}
```

Creating a Process: Before



Creating a Process: After

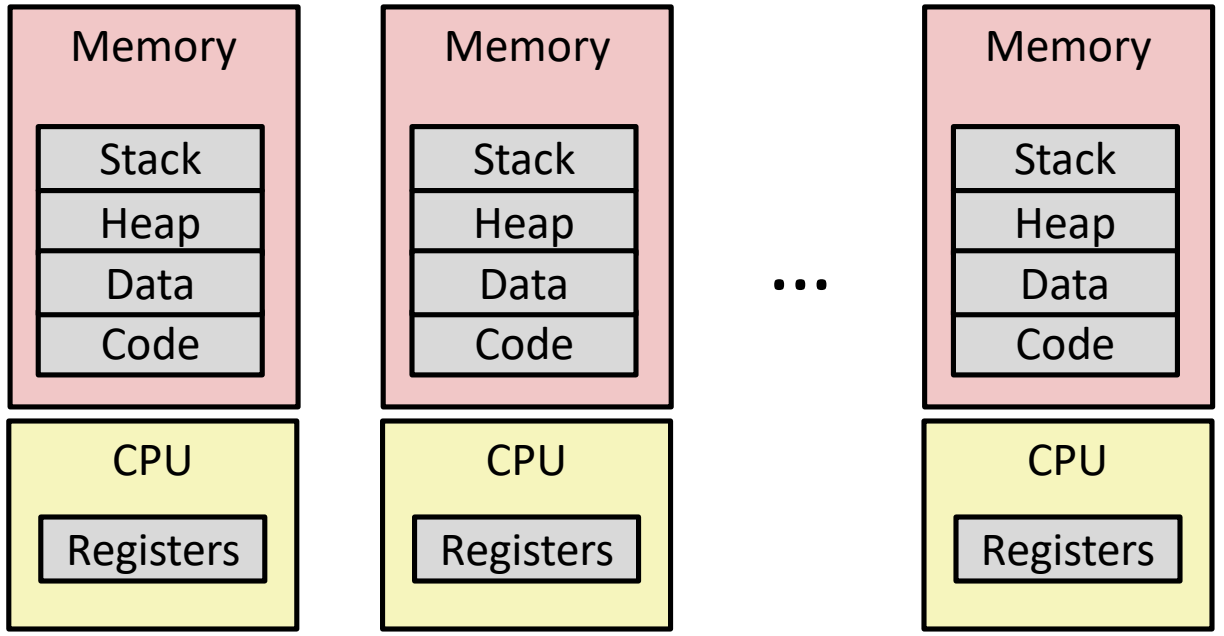




```
int main()
{
int value = 5;
pid_t pid;
if ((pid = fork()) == 0) { /* child process */
        value += 15;
        printf("CHILD: value = %d\n",value);
        return 0;
}
else if (pid > 0) { /* parent process */
        printf("PARENT: value = %d\n",value);
        return 0;
}
}
```

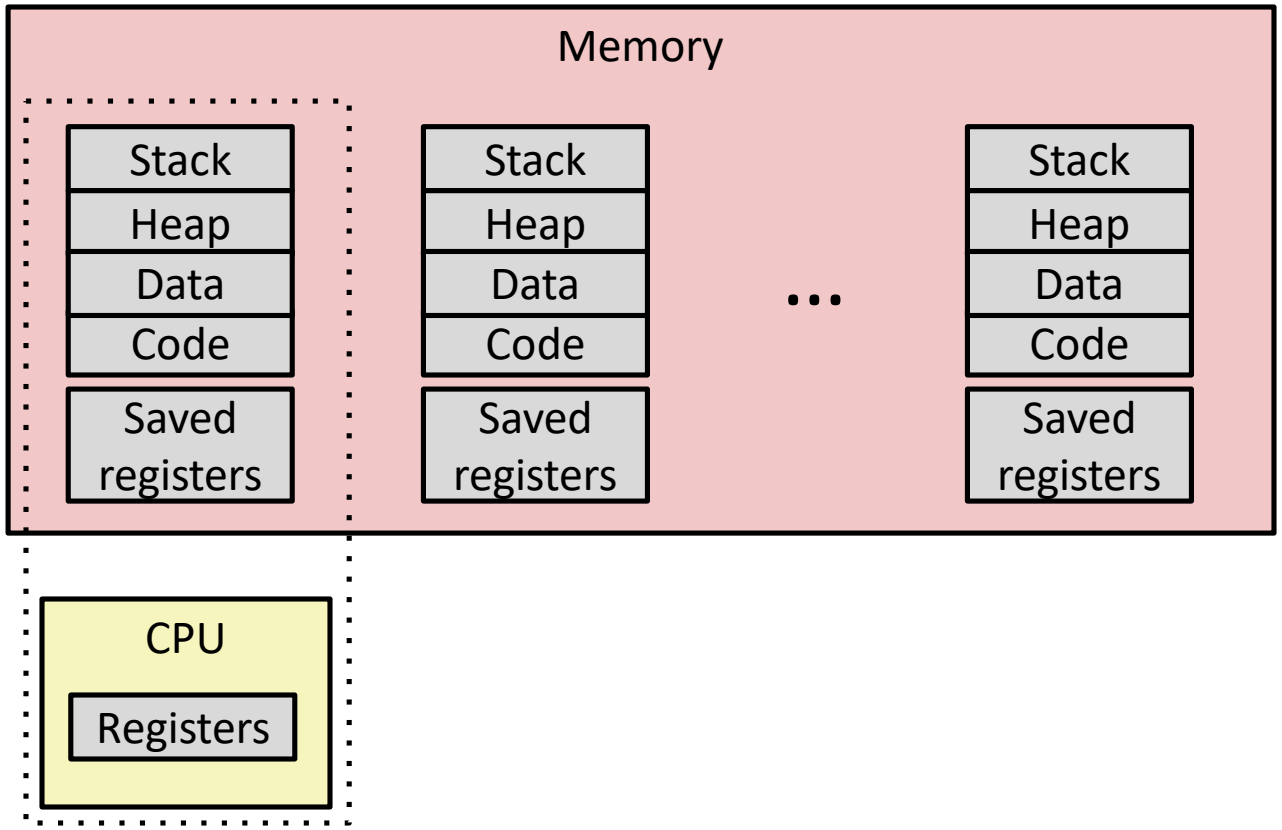
```
int main()
{
int value = 5;
pid_t pid;
if ((pid = fork()) == 0) { /* child process */
        value += 15;
        printf("CHILD: value = %d\n",value);
        return 0;
}
else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d\n",value);
        return 0;
}
}
```

Multiprocessing: The Illusion



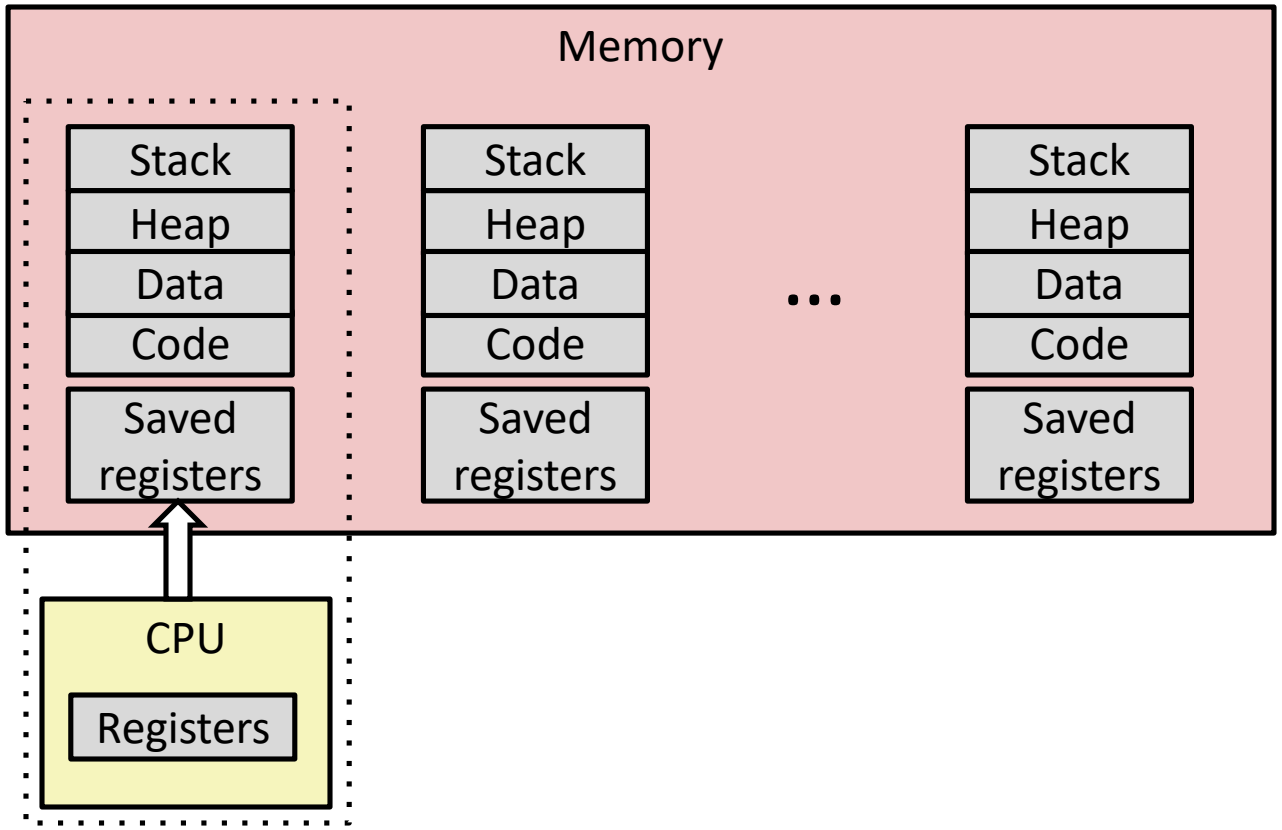
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …

Multiprocessing: The (Traditional) Reality



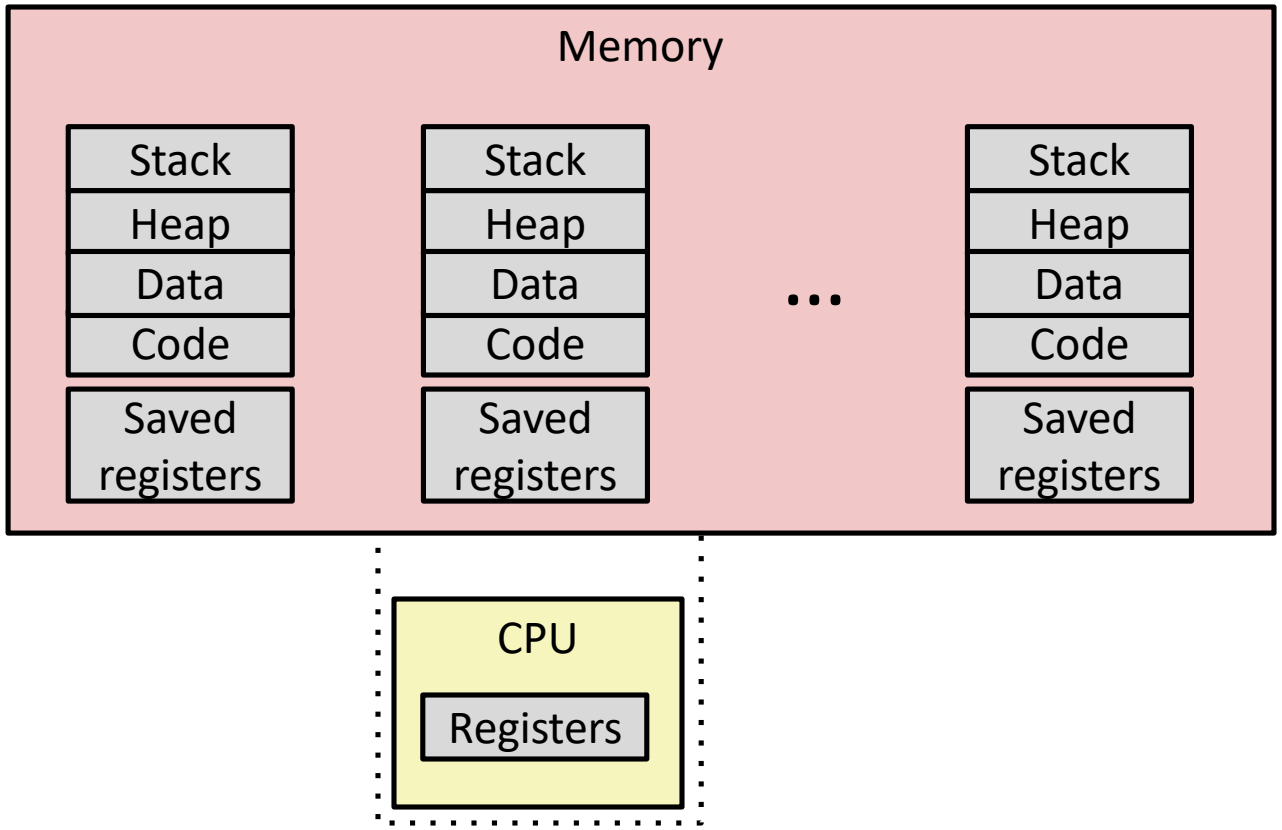
- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system (later in course)
  - Register values for nonexecuting processes saved in memory

Multiprocessing: The (Traditional) Reality



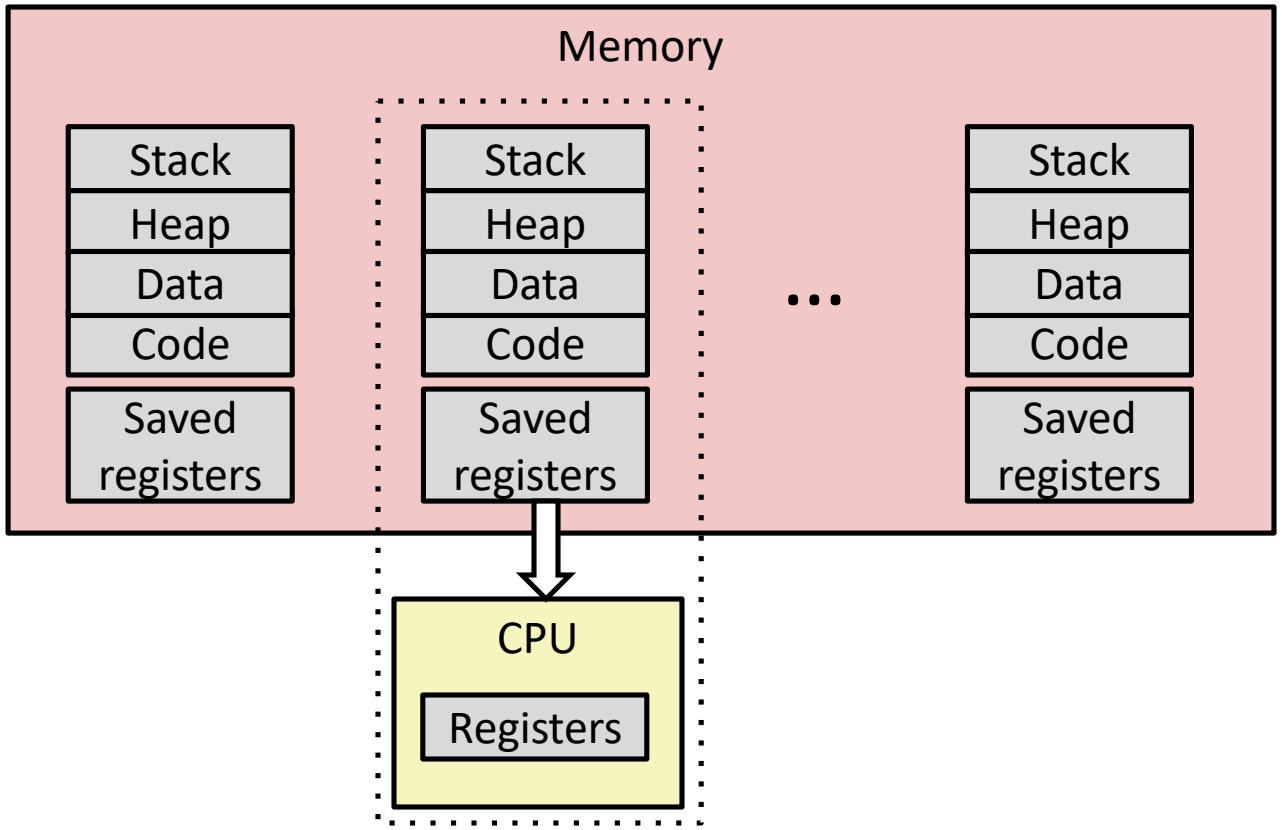
- Save current registers in memory

Multiprocessing: The (Traditional) Reality



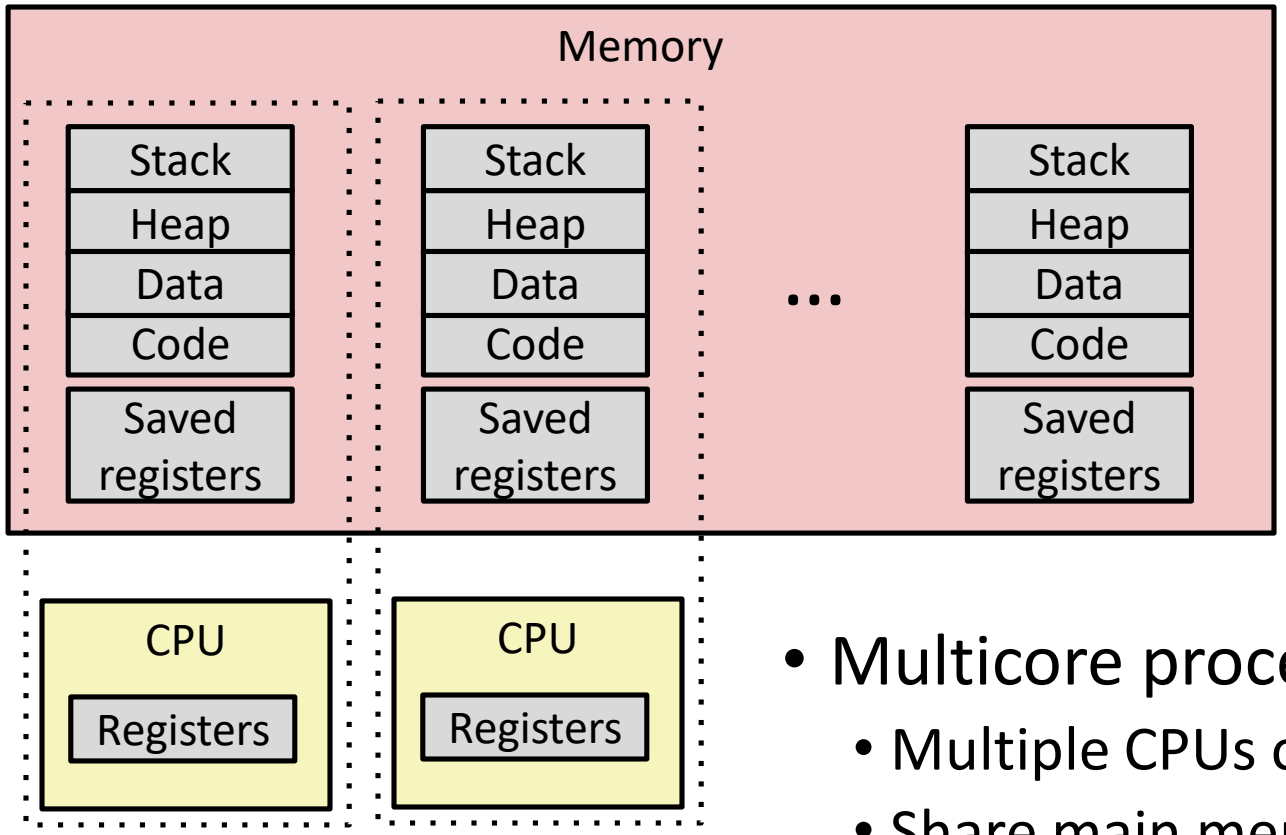
- Schedule next process for execution

Multiprocessing: The (Traditional) Reality



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

Multiprocessing: The (Modern) 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