Control Flow

From startup to shutdown, a CPU reads and executes a sequence of instructions

This sequence is normal ***control flow***

Jumps and calls/returns determine control flow based on ***program state***

```
movq %rax, %rbx
addq %rcx, %rbx
movl (%rbx), %eax
cmpl $0x5, %eax
jne  0x864c22
addq $1, %rax
jmp  0x864a06
```

System State

Changes in ***system state***:

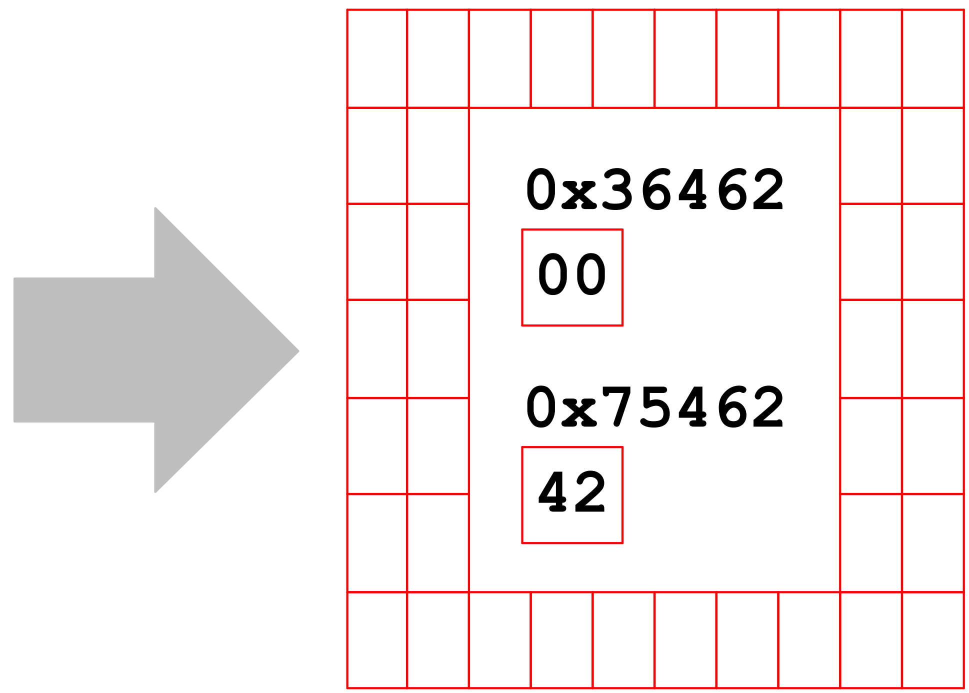
- Data arrives from the network
- The user hits Ctrl-C
- A timer expires
- An instruction divides by zero

Need a mechanism for ***exceptional control flow***

Kernel vs. User Code

When you turn on a processor, instructions can do anything: the processor starts in ***privileged mode***

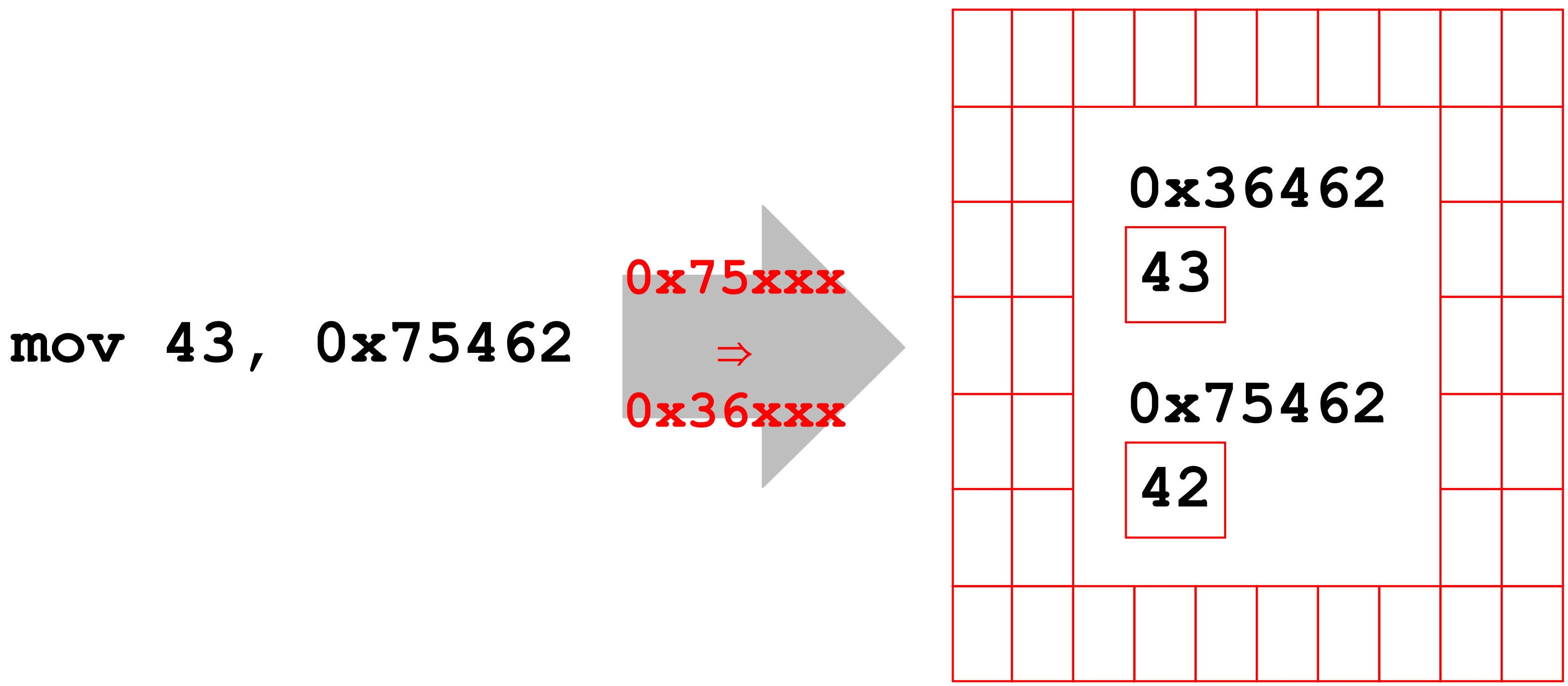
`mov 42, 0x75462`



The operating system ***kernel*** runs in privileged mode

Kernel vs. User Code

In privileged mode, the kernel can change the way that ***virtual addresses*** are mapped to physical memory

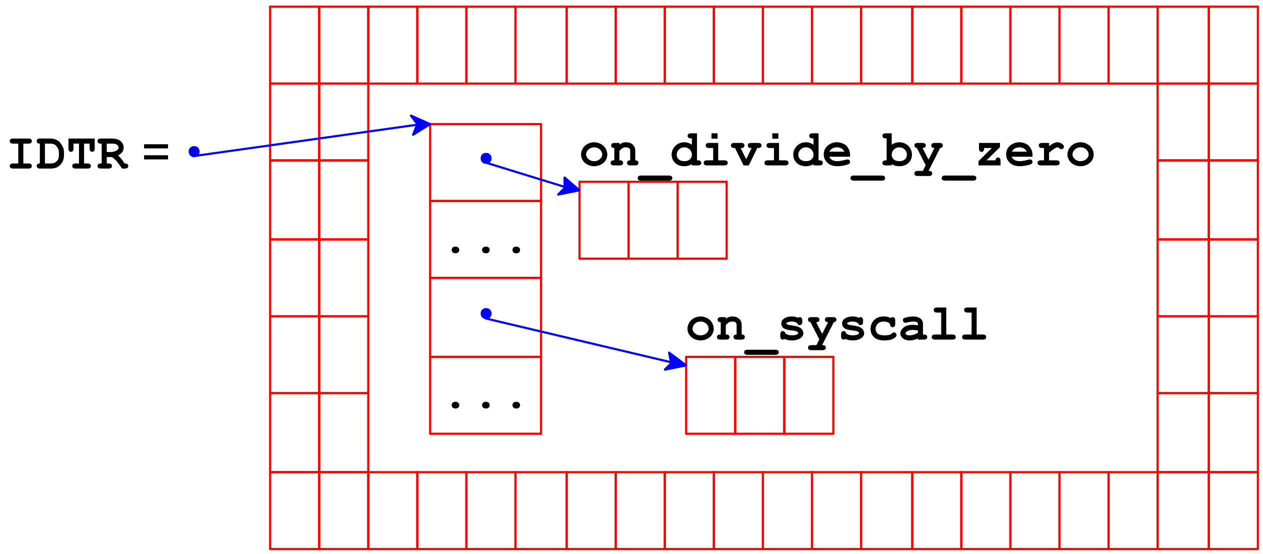


So, the kernel can hide memory from unprivileged user code

but, before doing that...

Kernel vs. User Code

Special register **`IDTR`** points to memory (not accessible to user code) for a table of functions to handle ***exceptions***:

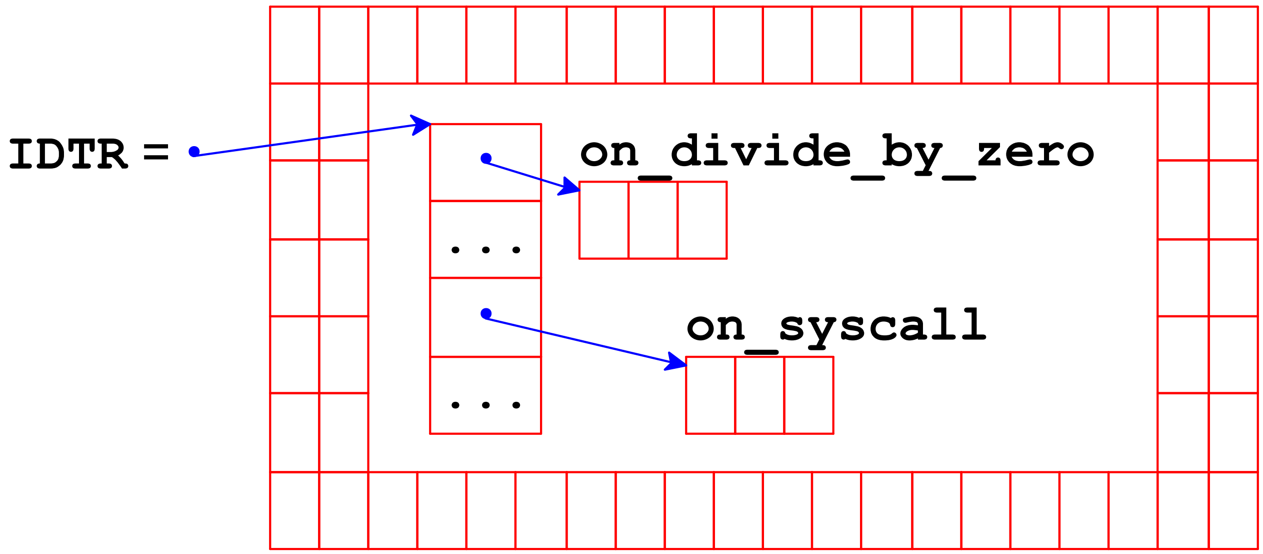


This is the ***exception table***

a.k.a. the ***interrupt vector***

Kernel vs. User Code

Special register **`IDTR`** points to memory (not accessible to user code) for a table of functions to handle ***exceptions***:

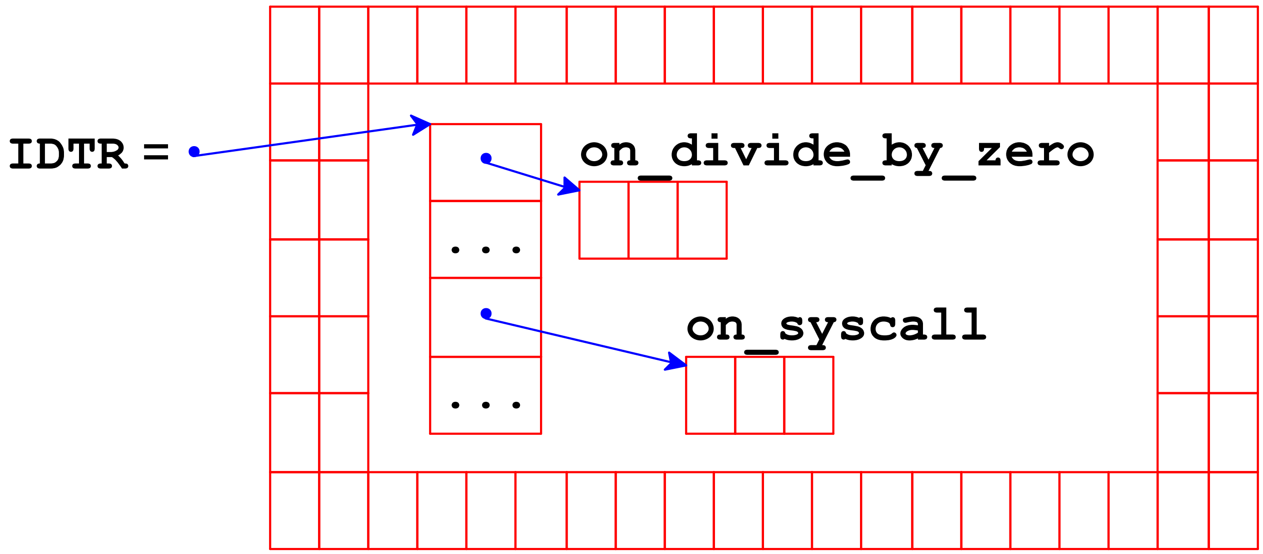


Call exception handler: ignore address remappings and switch back to privileged mode

Control the table ⇒ control the way back to privileged mode

Kernel vs. User Code

Special register **`IDTR`** points to memory (not accessible to user code) for a table of functions to handle ***exceptions***:



`int` k

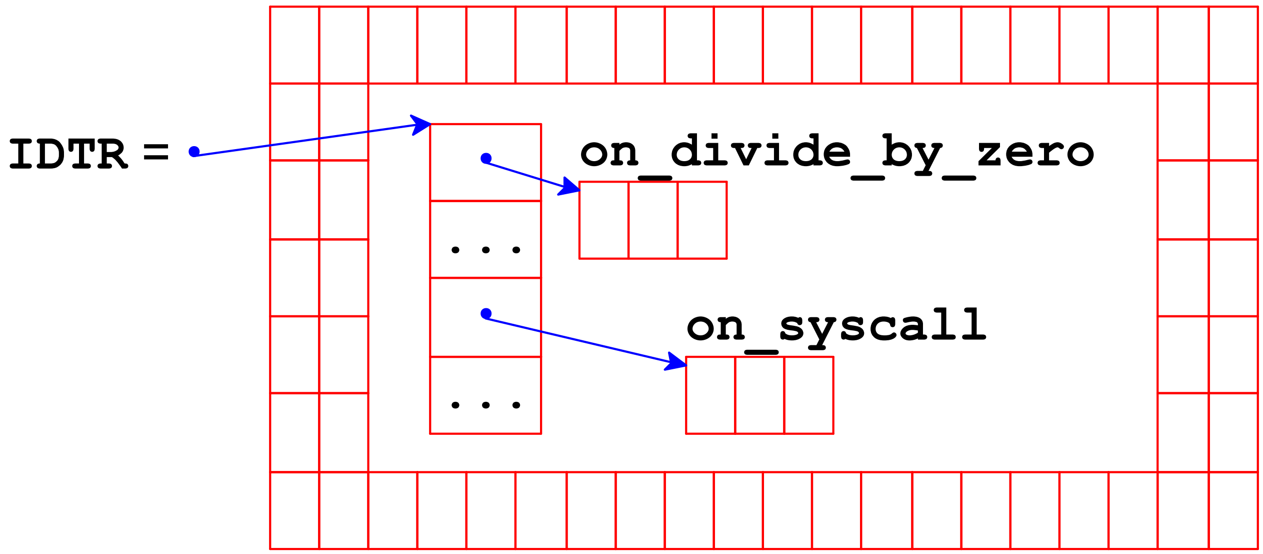
Trigger k exception

k = **`0x80`** means "system call"

```
mov $0x2,%eax
int $0x80
```

Kernel vs. User Code

Special register **`IDTR`** points to memory (not accessible to user code) for a table of functions to handle ***exceptions***:



`syscall`

Same idea as **`int $0x80`**, but faster

```
mov $0x2,%eax
syscall
```

Exception Handling

User code

...

$I_{n\text{-}1}$

I_n

$I_{current}$

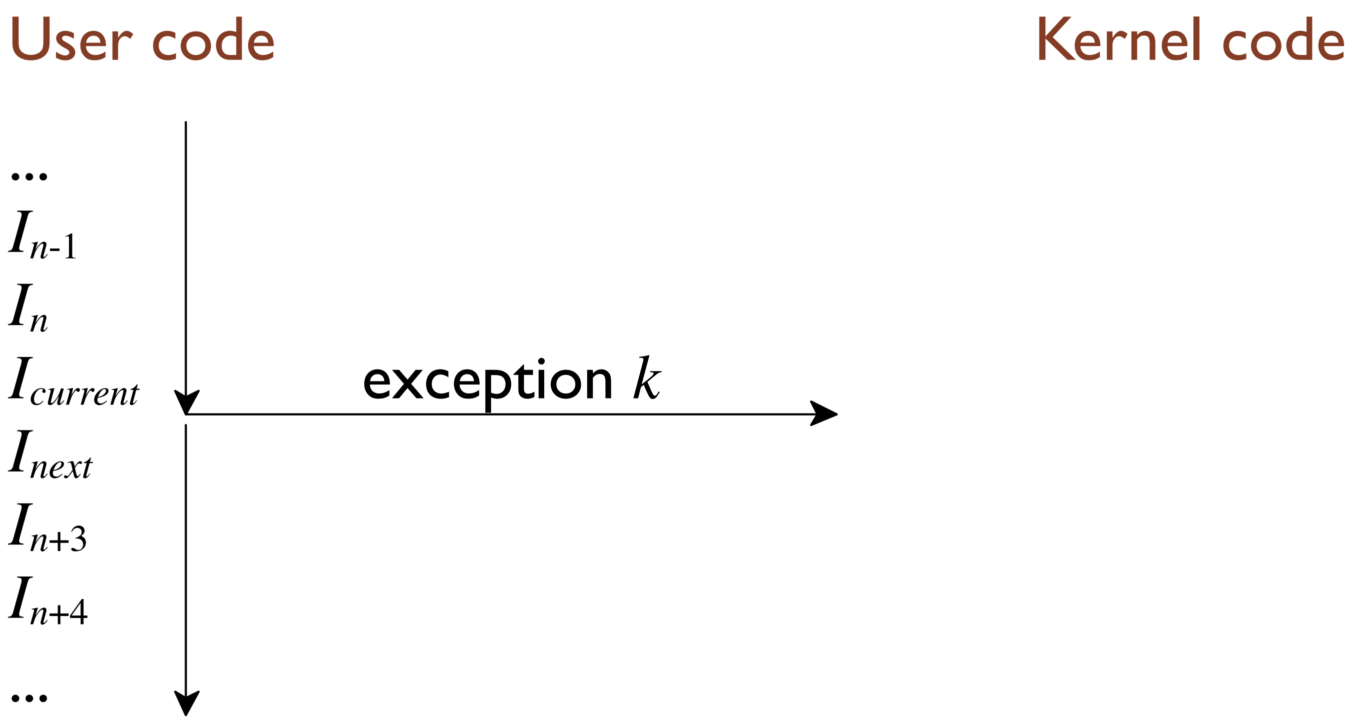
I_{next}

I_{n+3}

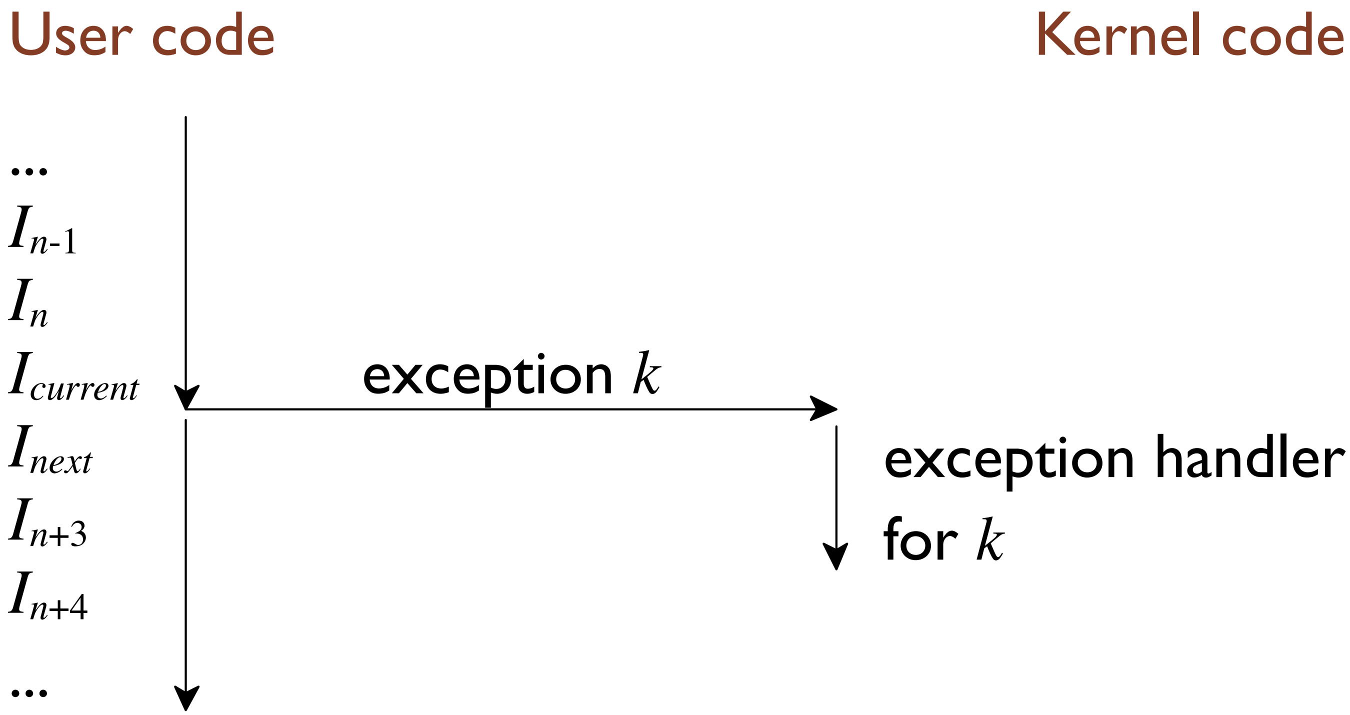
I_{n+4}

...

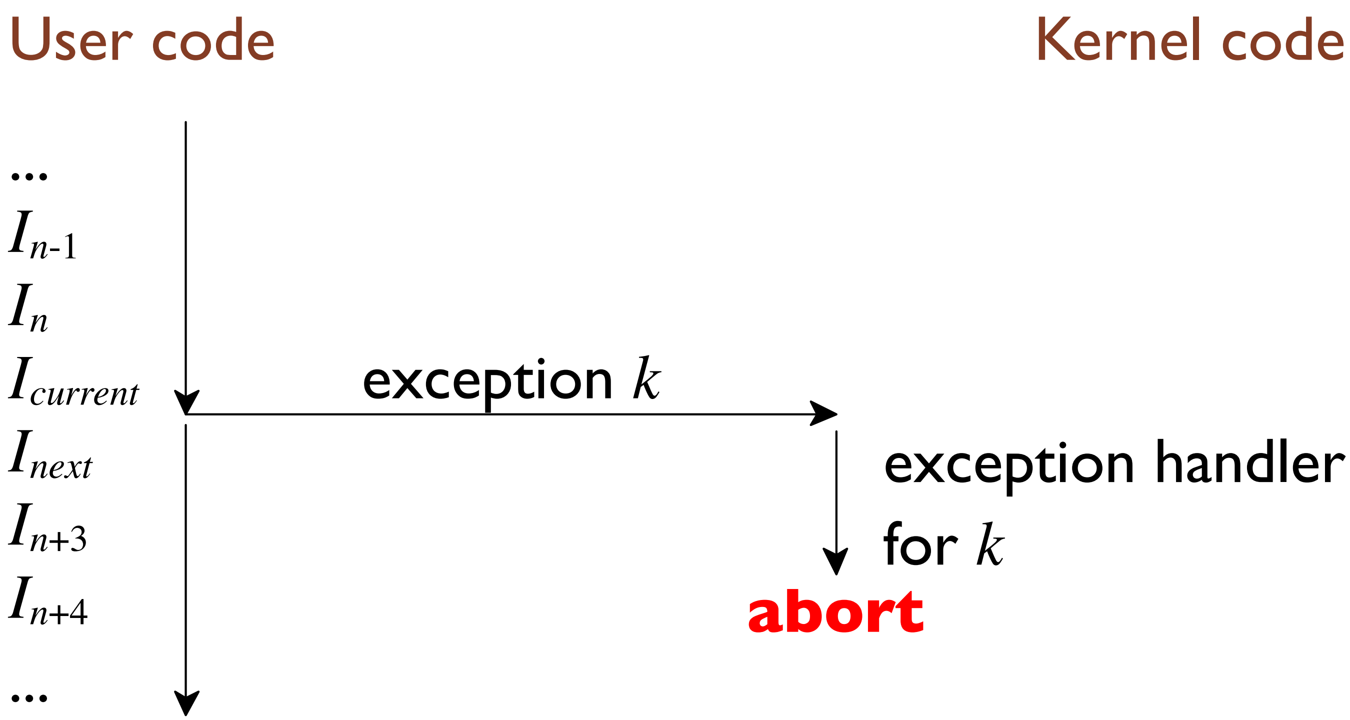
Exception Handling



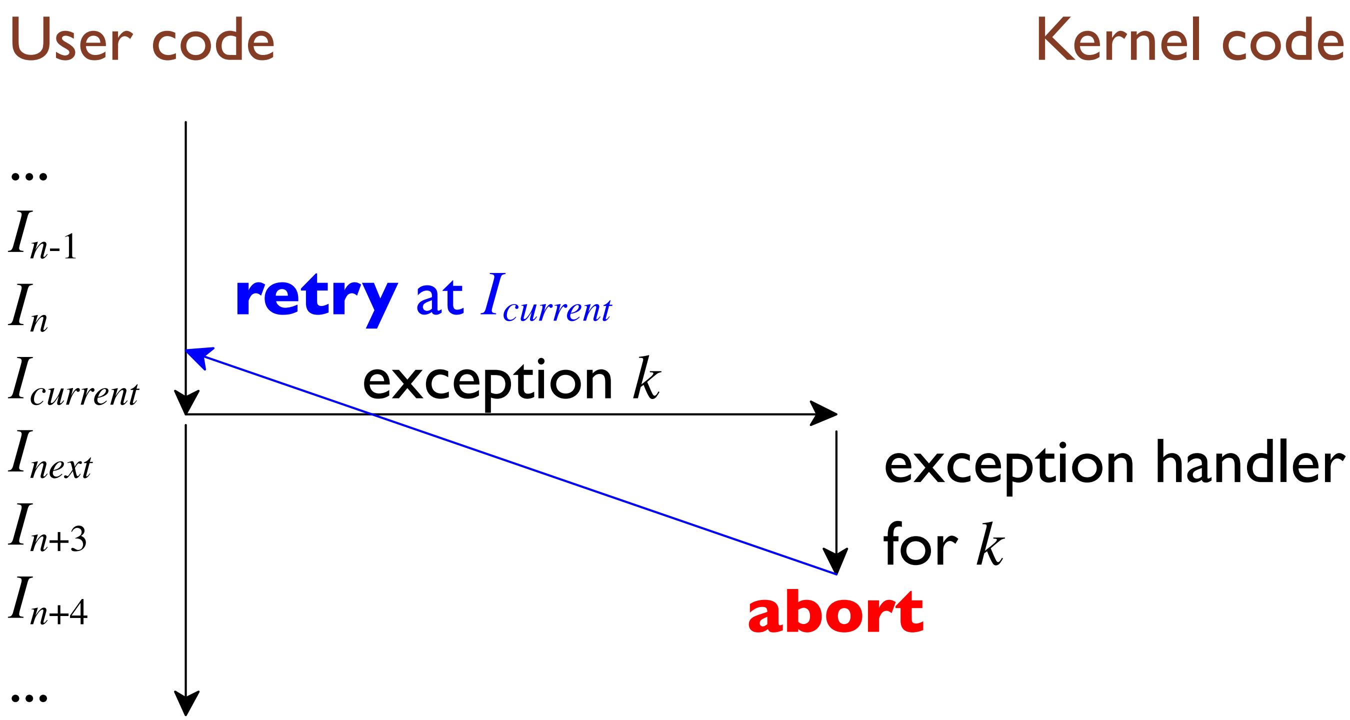
Exception Handling



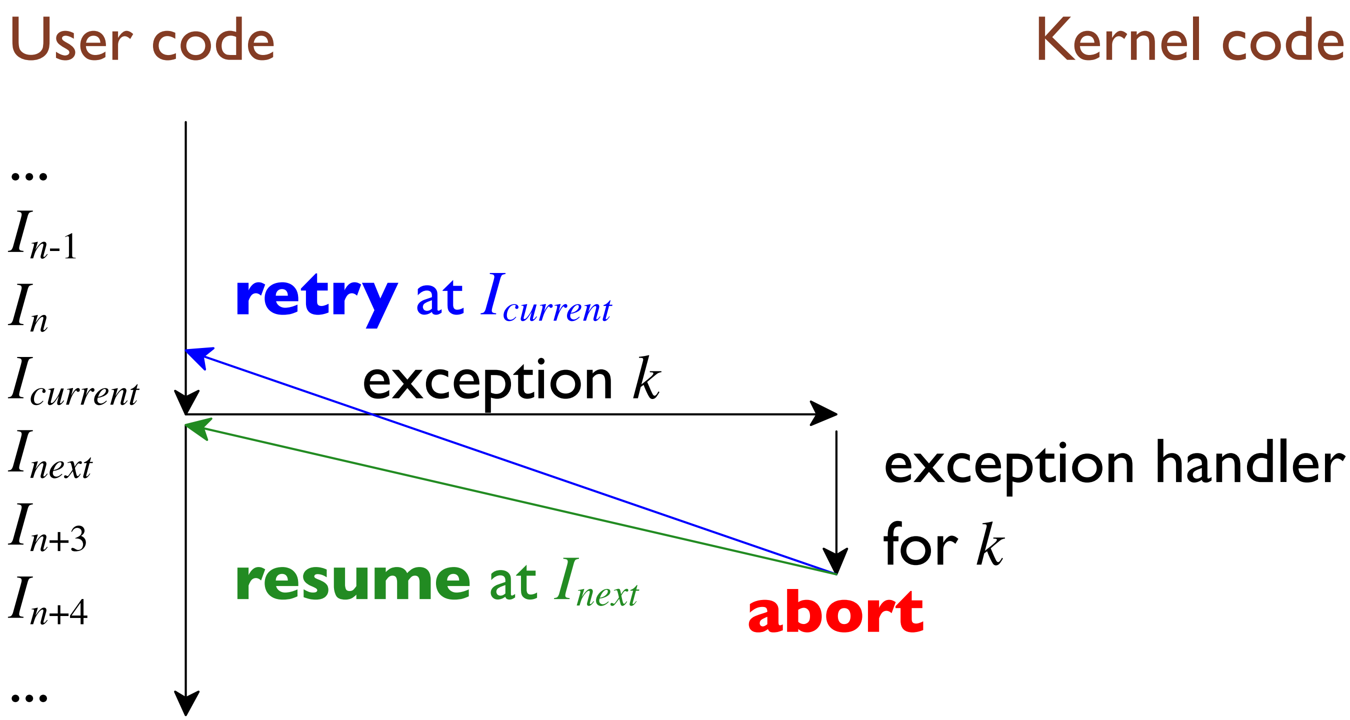
Exception Handling



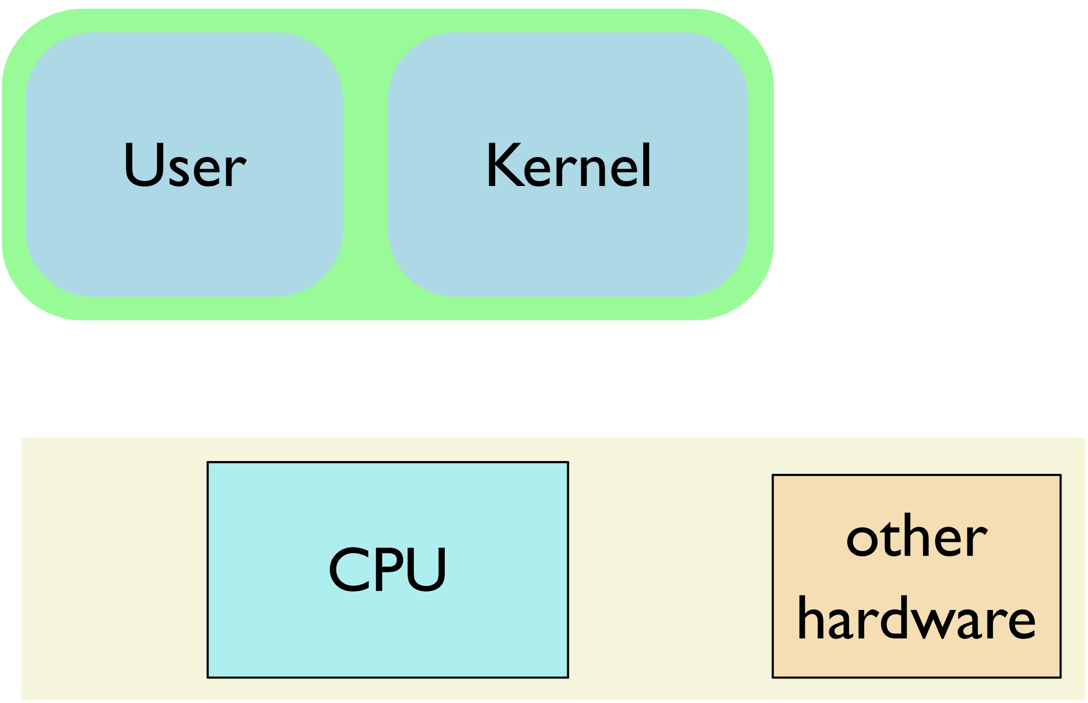
Exception Handling



Exception Handling



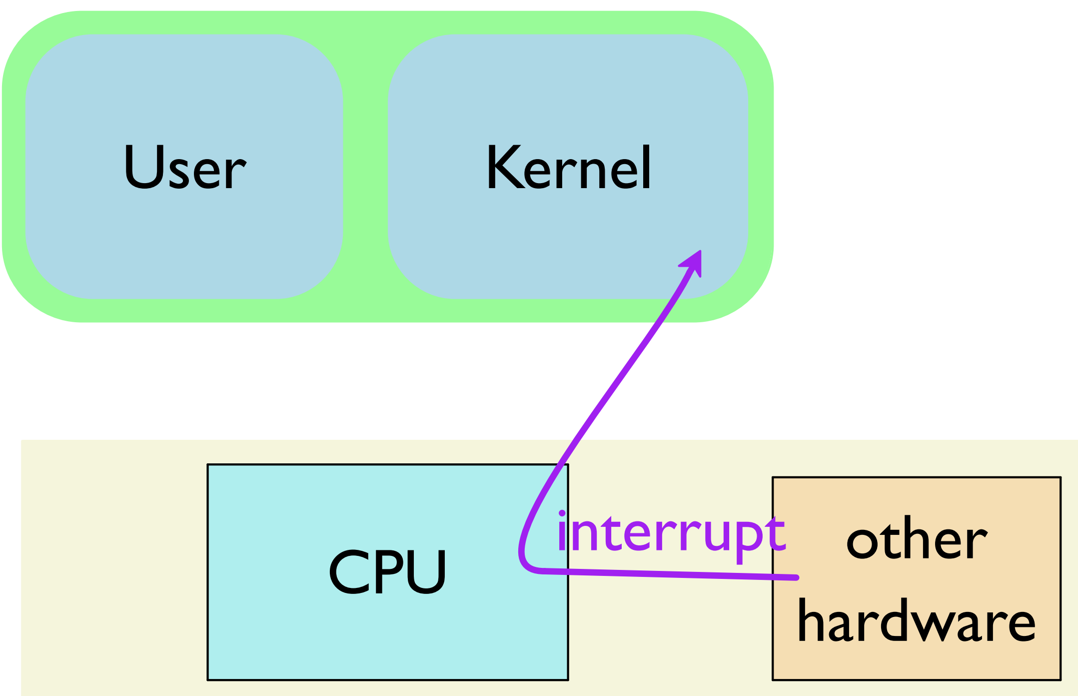
Four Kinds of Exceptions



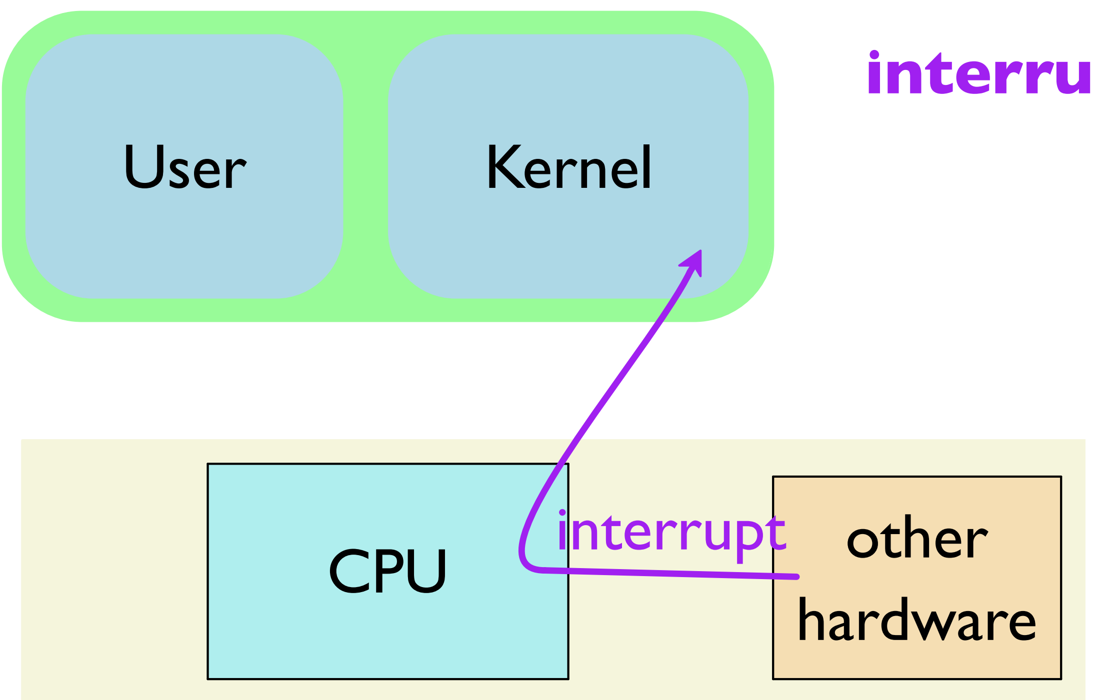
Four Kinds of Exceptions

interrupt — from hardware: keyboard, network packet, ...

- *asynchronous* with respect to the program
- handled by kernel, which then resumes program



Four Kinds of Exceptions

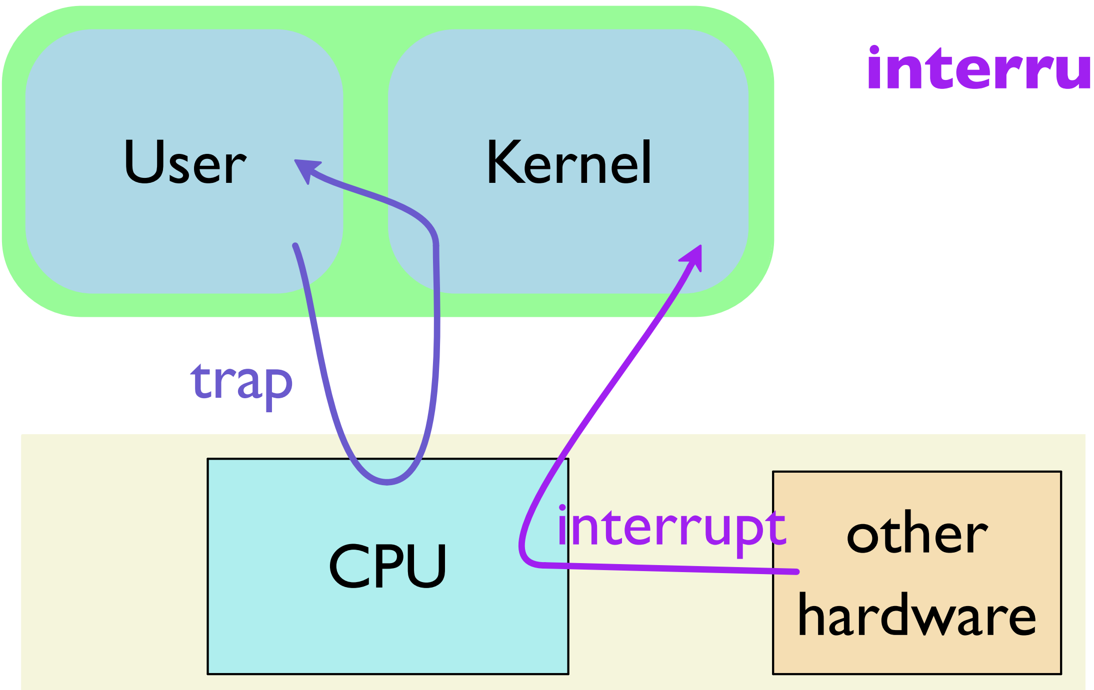


interrupt asynch resume

Four Kinds of Exceptions

trap — from program: system call, breakpoint, ...

- *synchronous* and *intentional*
- handled by kernel, which then resumes program



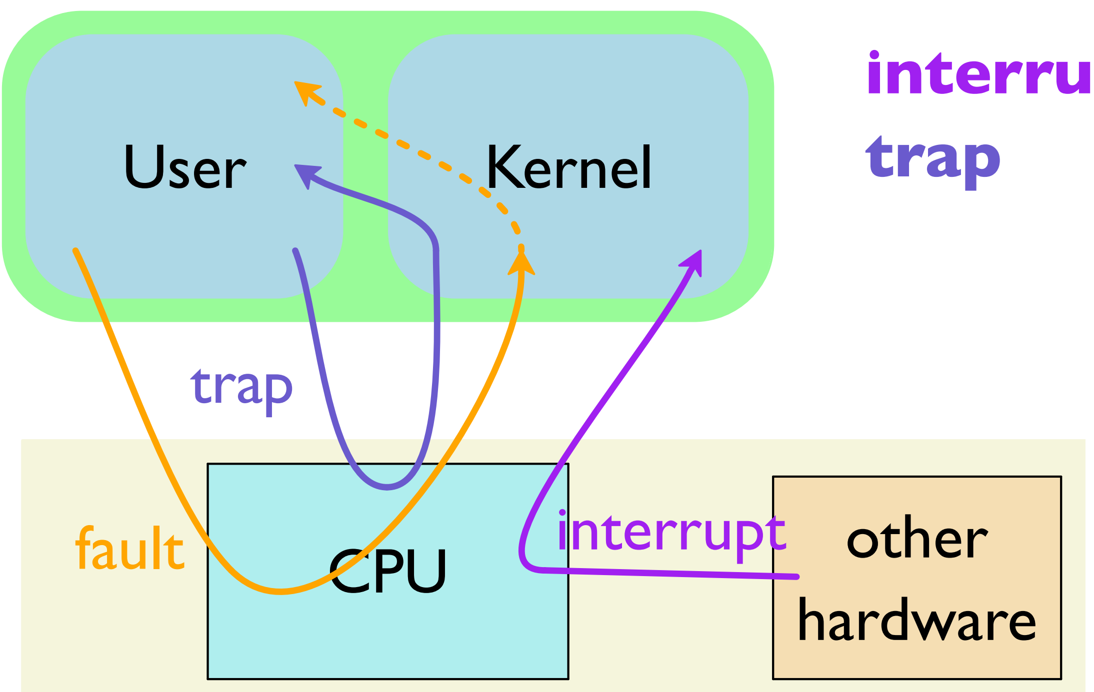
interrupt asynch resume

Four Kinds of Exceptions

fault — by program: bad memory reference, ...

- *synchronous* and usually *unintentional*
- handled by kernel, which may retry or abort

...maybe with program help

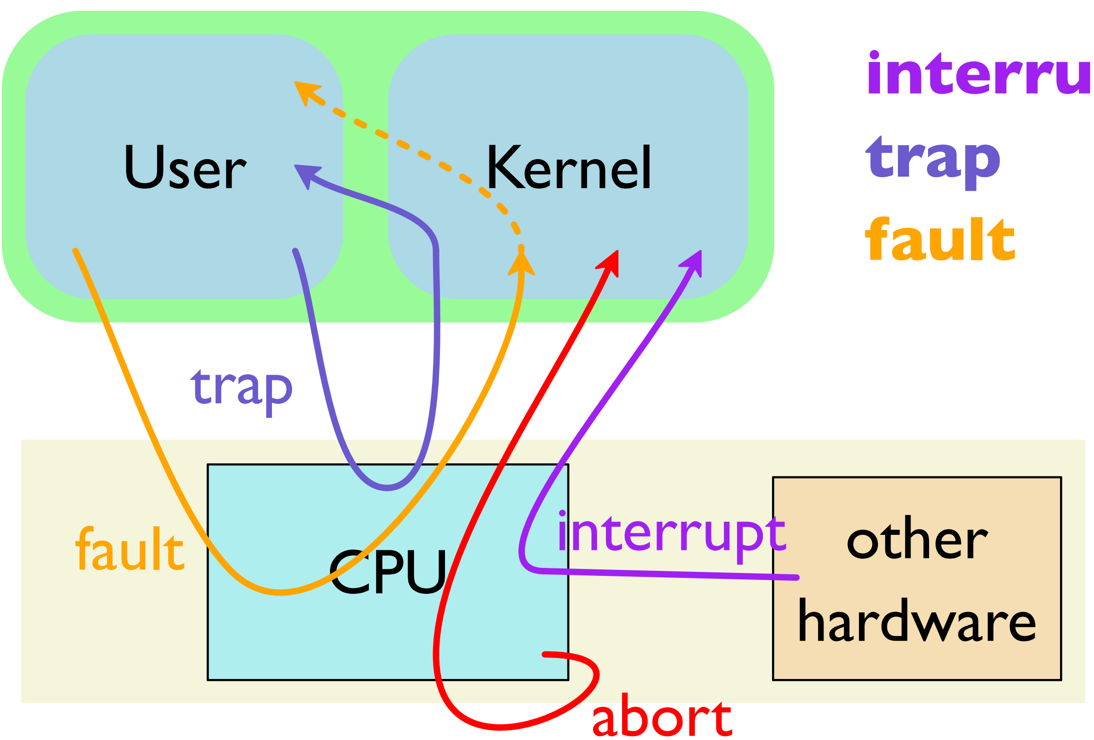


interrupt	asynch	resume
trap	intentional	resume

Four Kinds of Exceptions

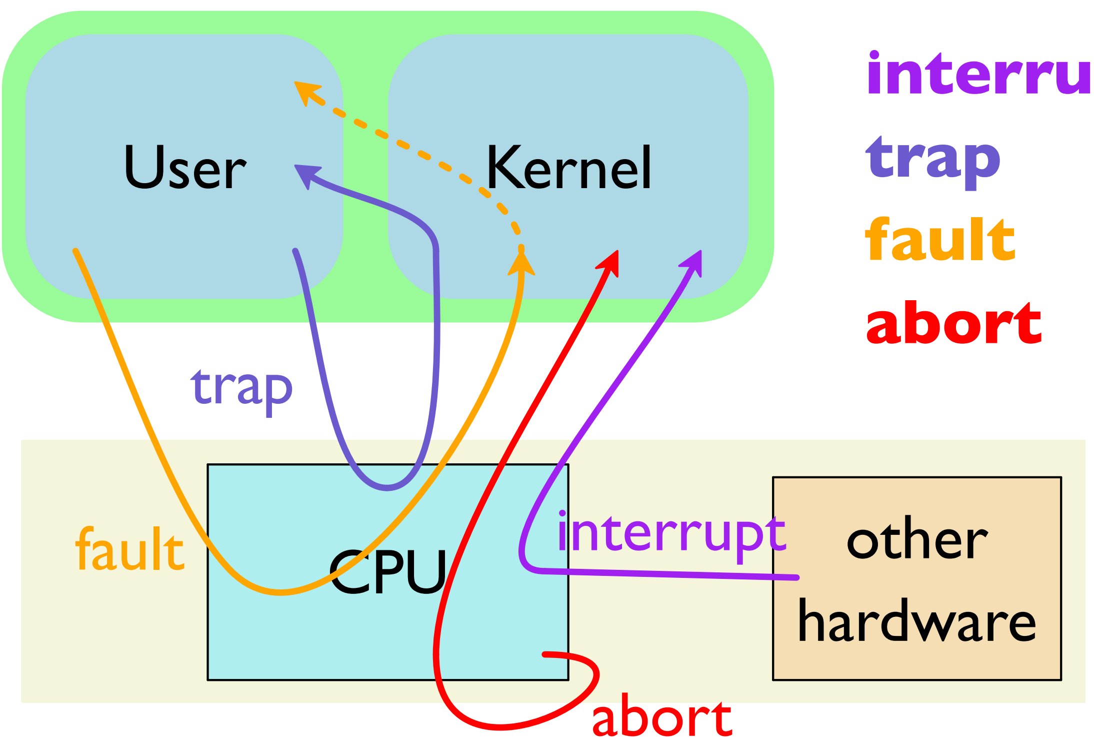
abort — hardware errors and such

- *synchronous* and *unintentional*
- kernel takes emergency measures to abort



interrupt	asynch	resume
trap	intentional	resume
fault	unintentional	retry/abort

Four Kinds of Exceptions



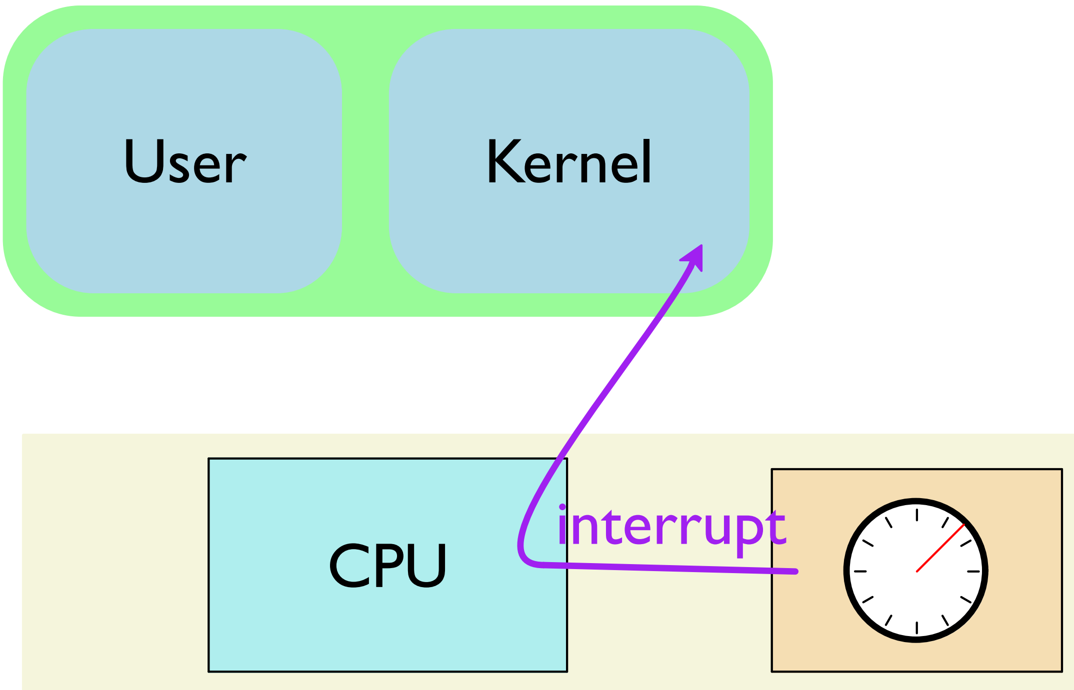
interrupt	asynch	resume
trap	intentional	resume
fault	unintentional	retry/abort
abort	unintentional	abort

Controlling User Code

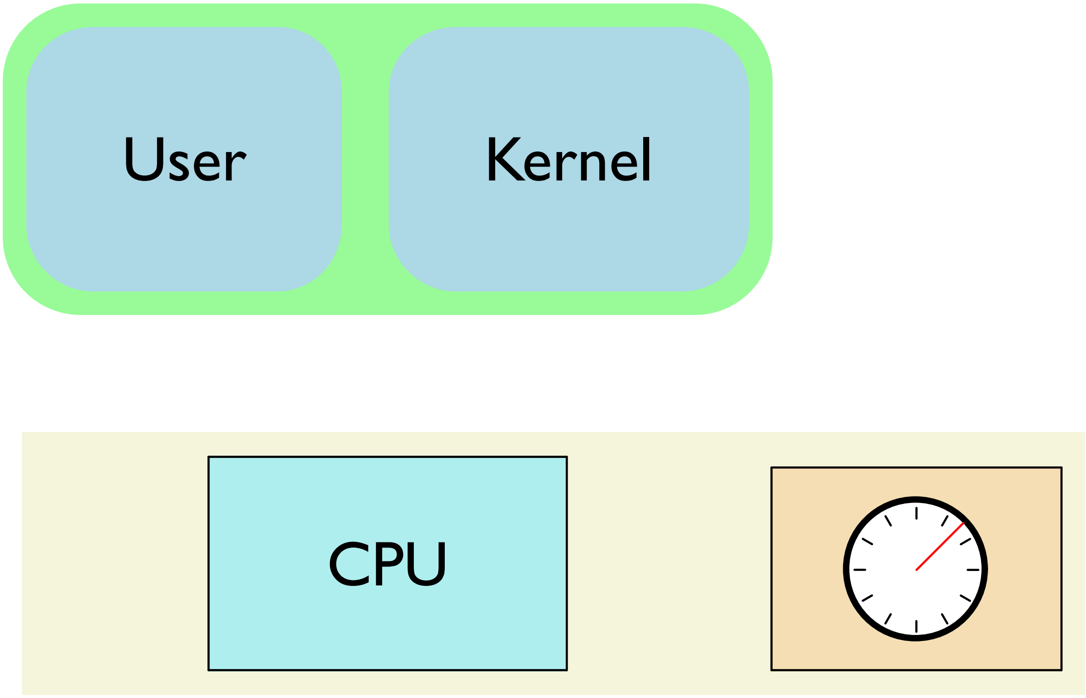
Exceptions explain how an OS can control your code:

- External interrupts ⇒ kernel can handle network, etc.
- Timer interrupt ⇒ kernel gets control often enough
- System calls via trap ⇒ kernel as more privileged
- Errors as faults ⇒ kernel can take over

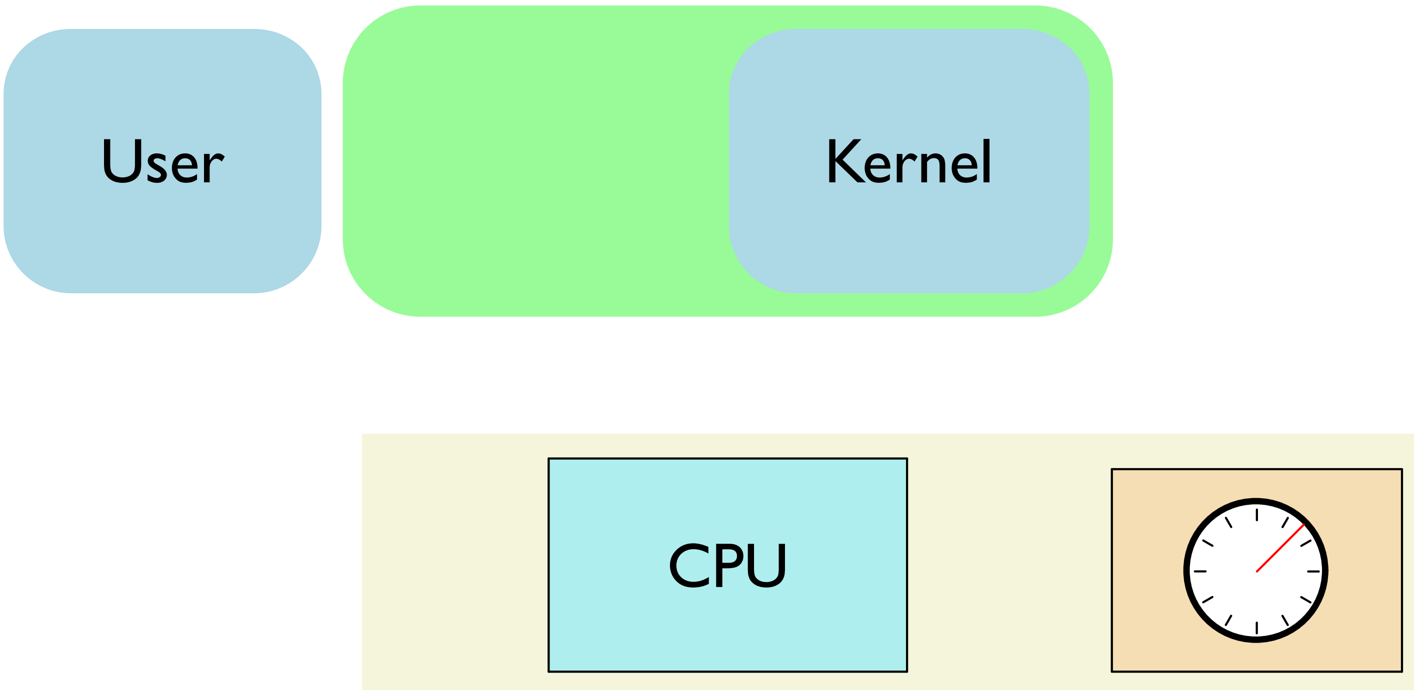
Switching User Code



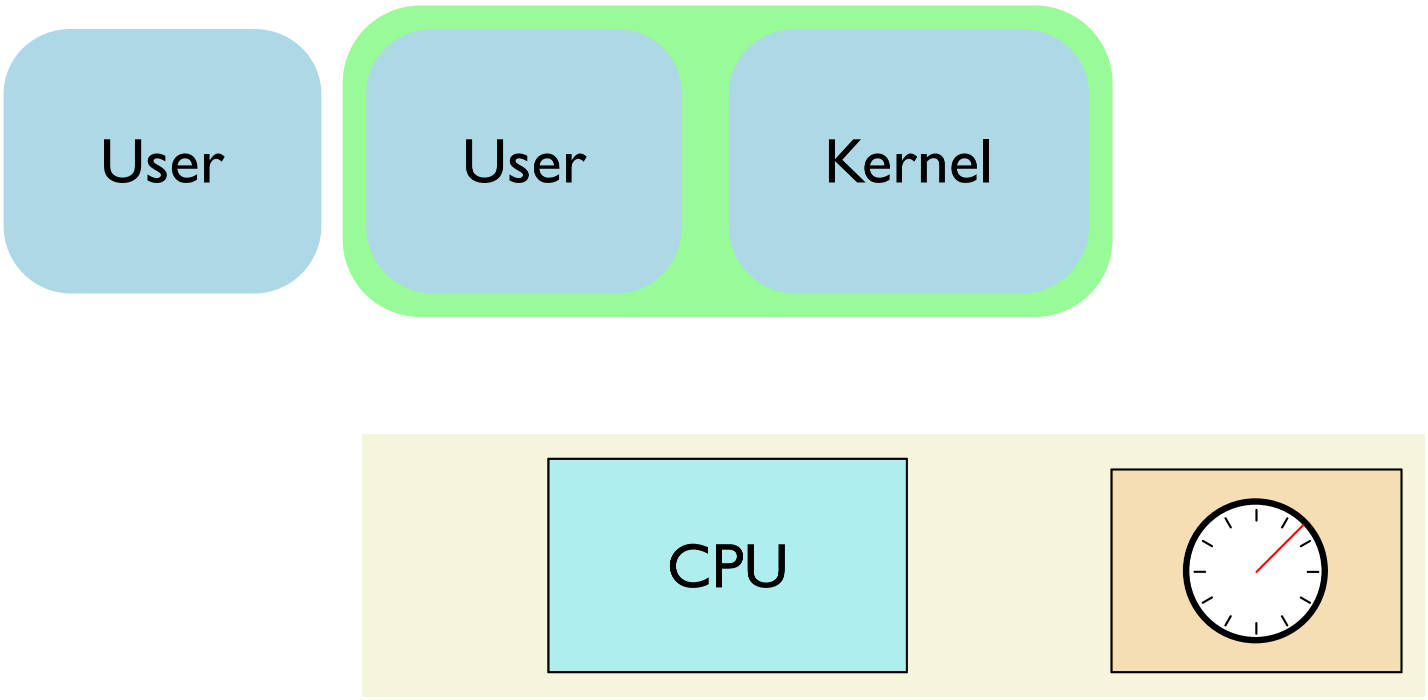
Switching User Code



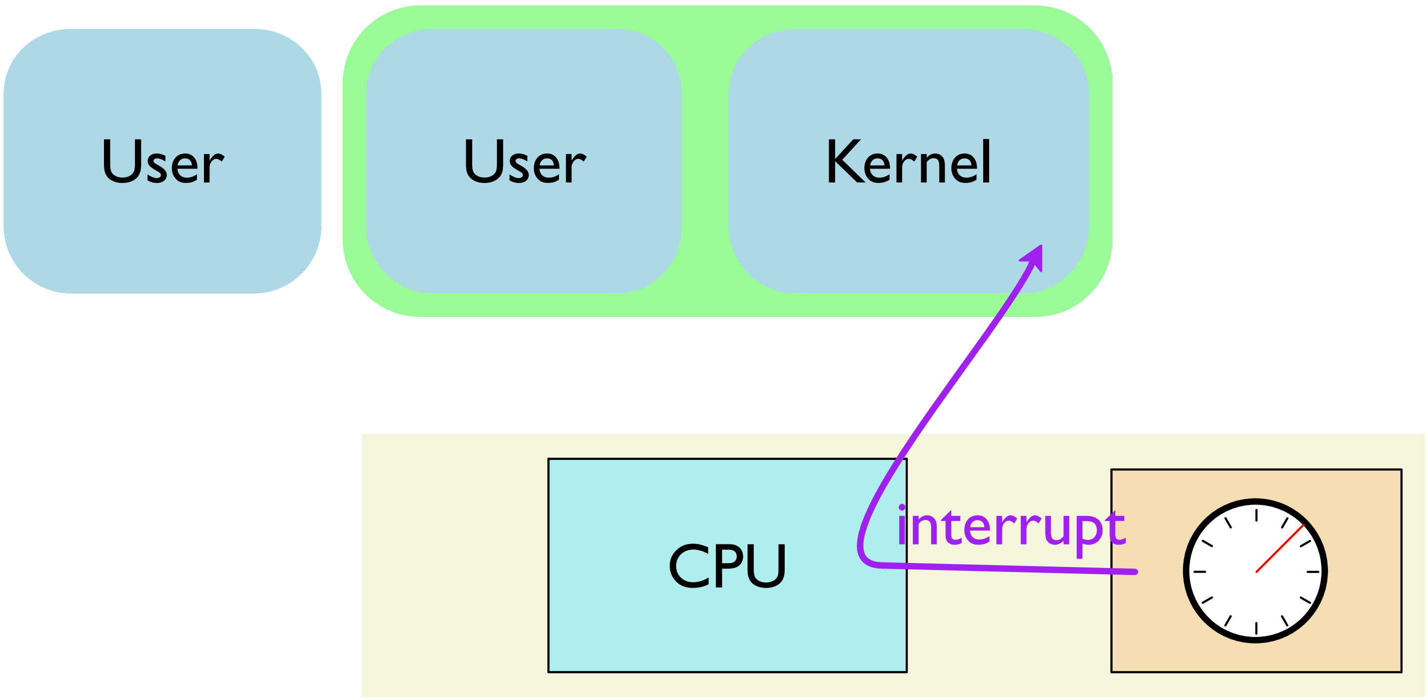
Switching User Code



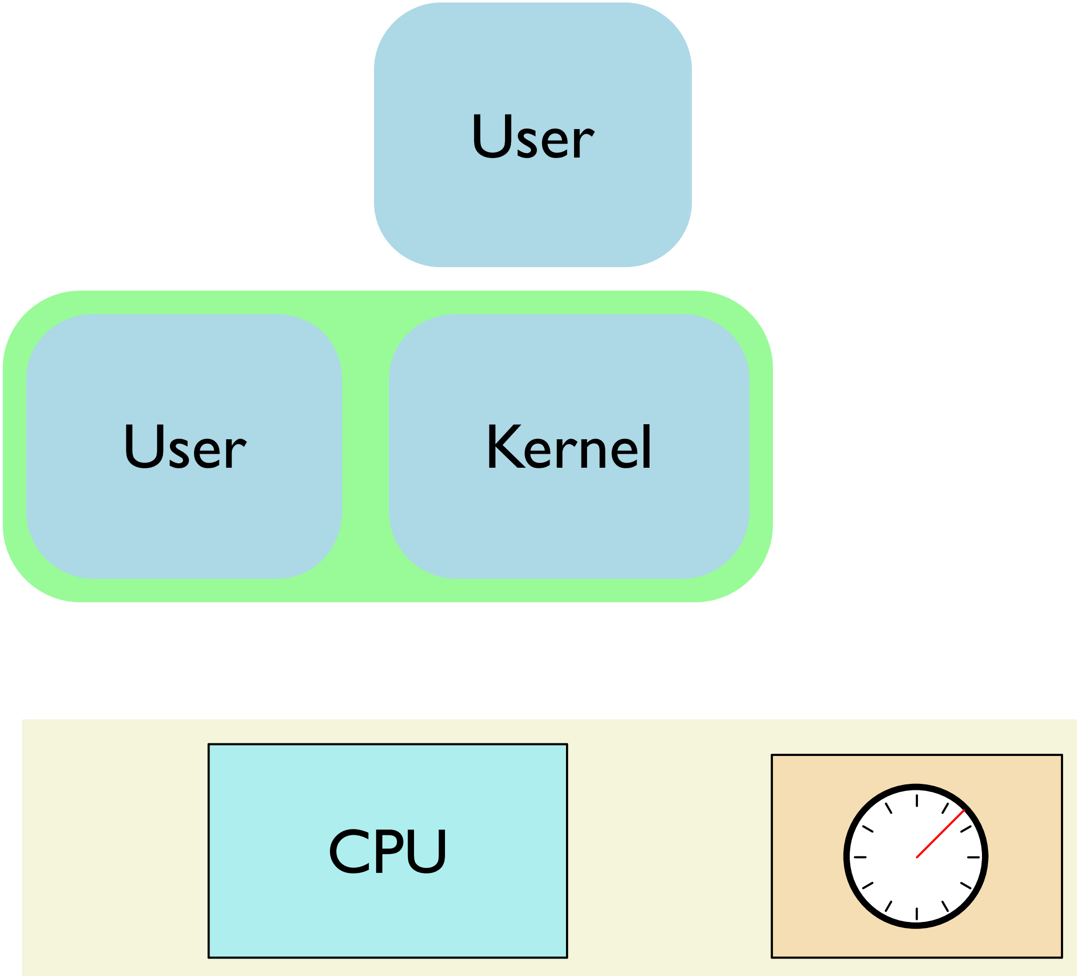
Switching User Code



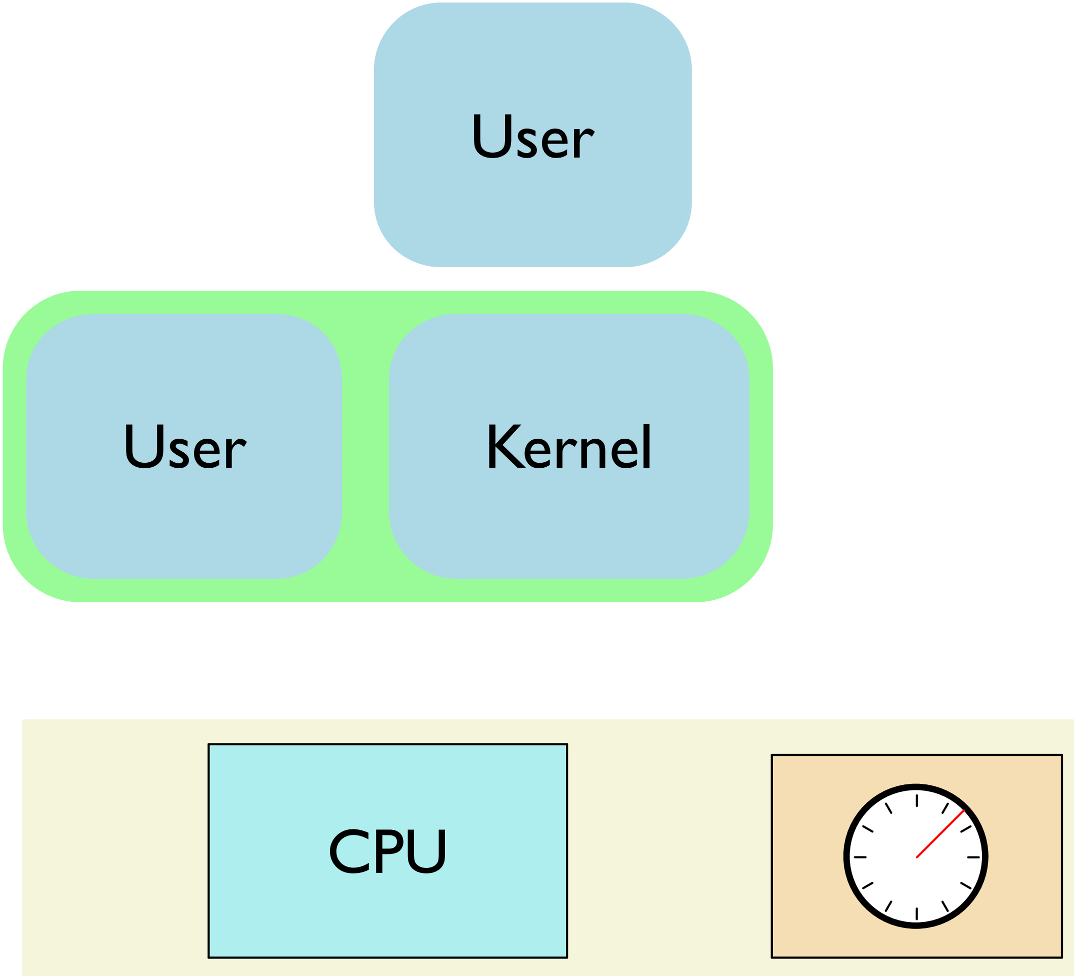
Switching User Code



Switching User Code

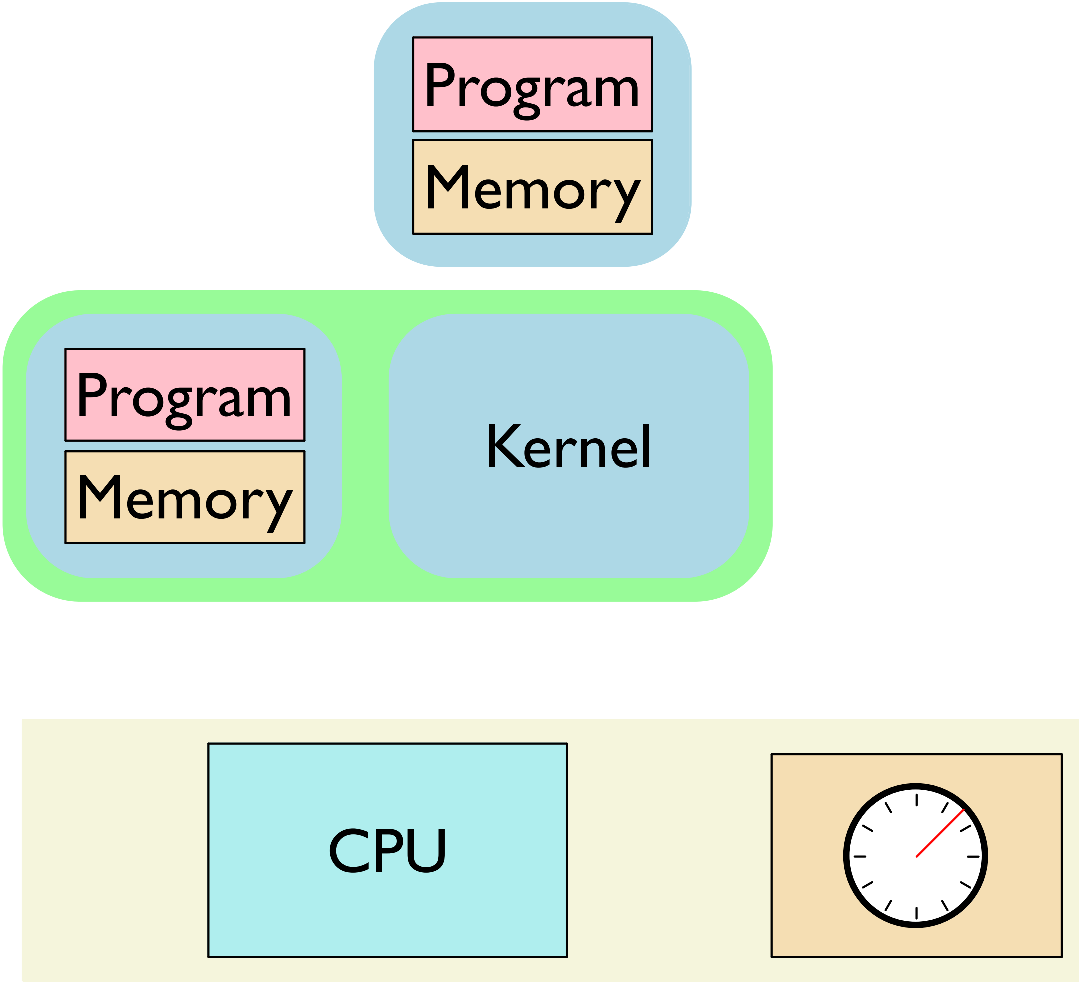


Switching User Code

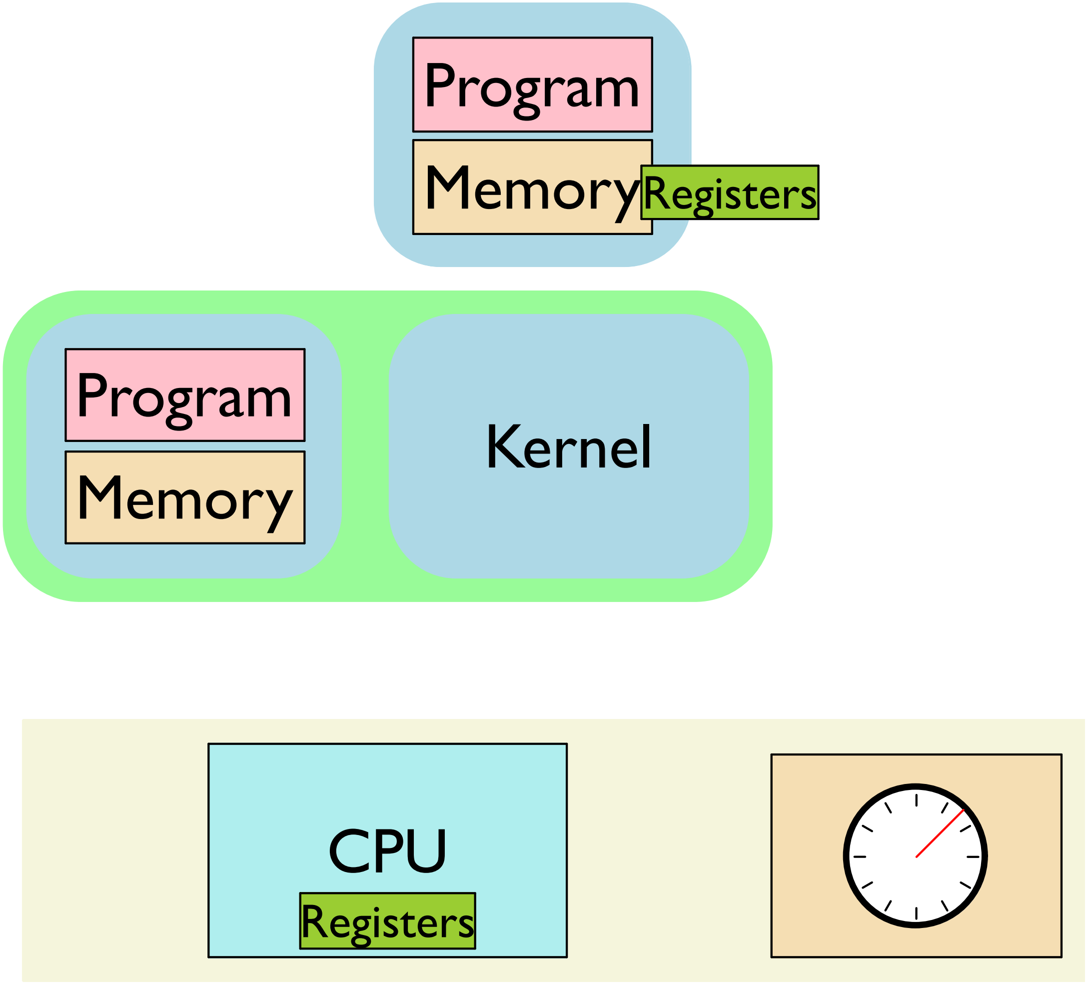


Switching user code is a ***context switch***

Switching User Code

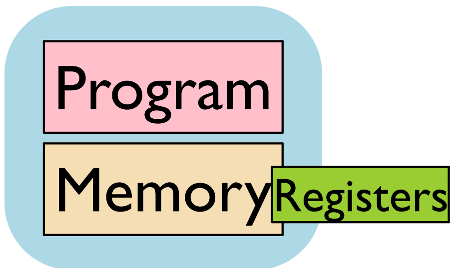


Switching User Code



Process

A ***process*** is a running *instance* of a program



Each process gets:

- local control flow

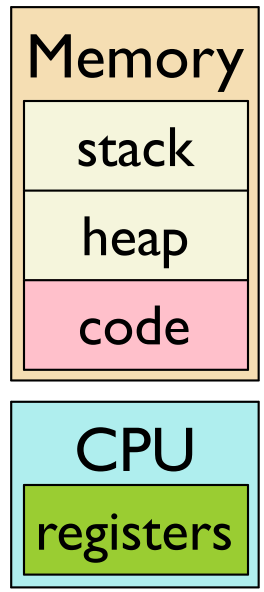
  a program seems to have the whole CPU

- private address space

  a program seems to have all of memory

Process

A ***process*** is a running *instance* of a program



Each process gets:

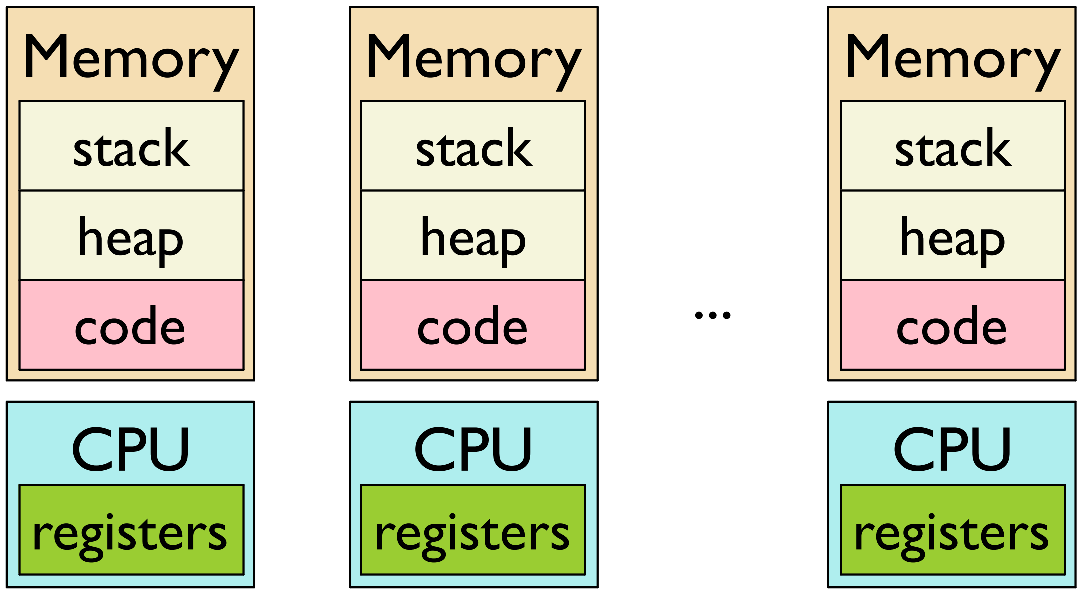
- local control flow

  a program seems to have the whole CPU

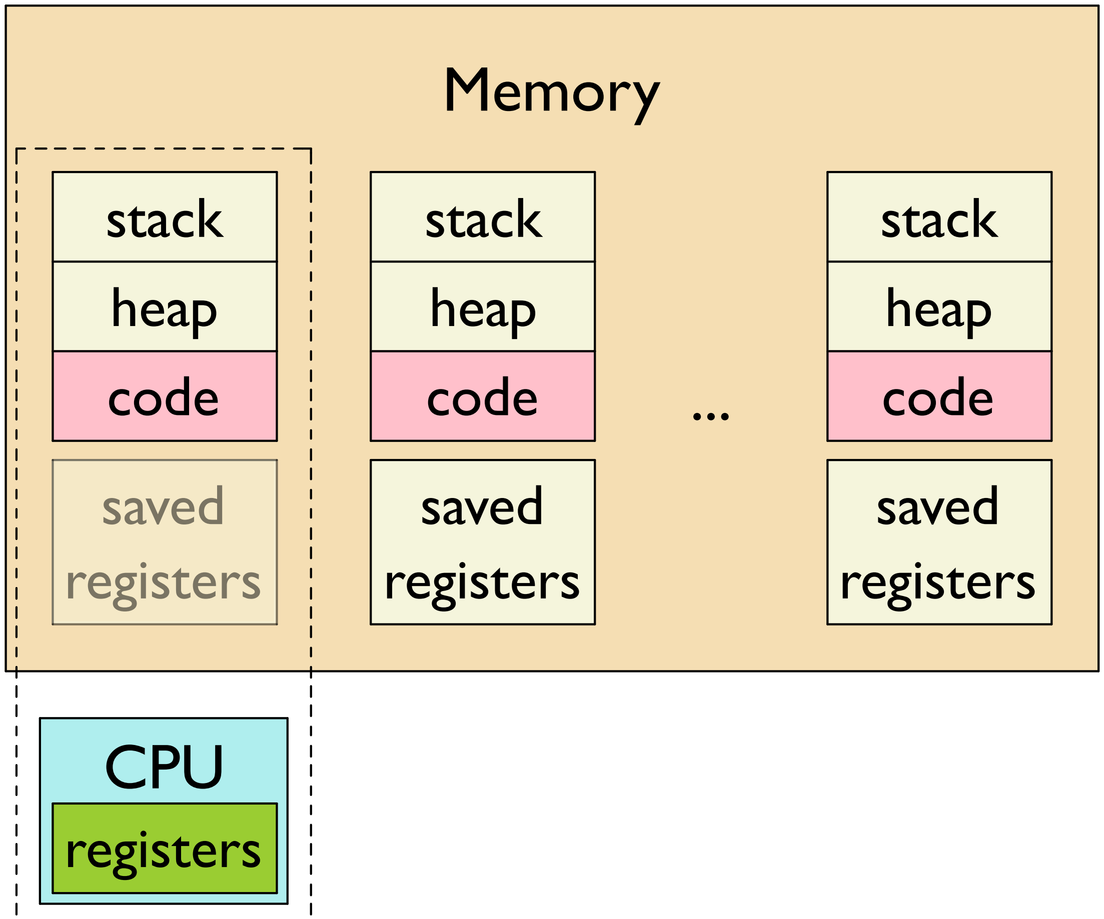
- private address space

  a program seems to have all of memory

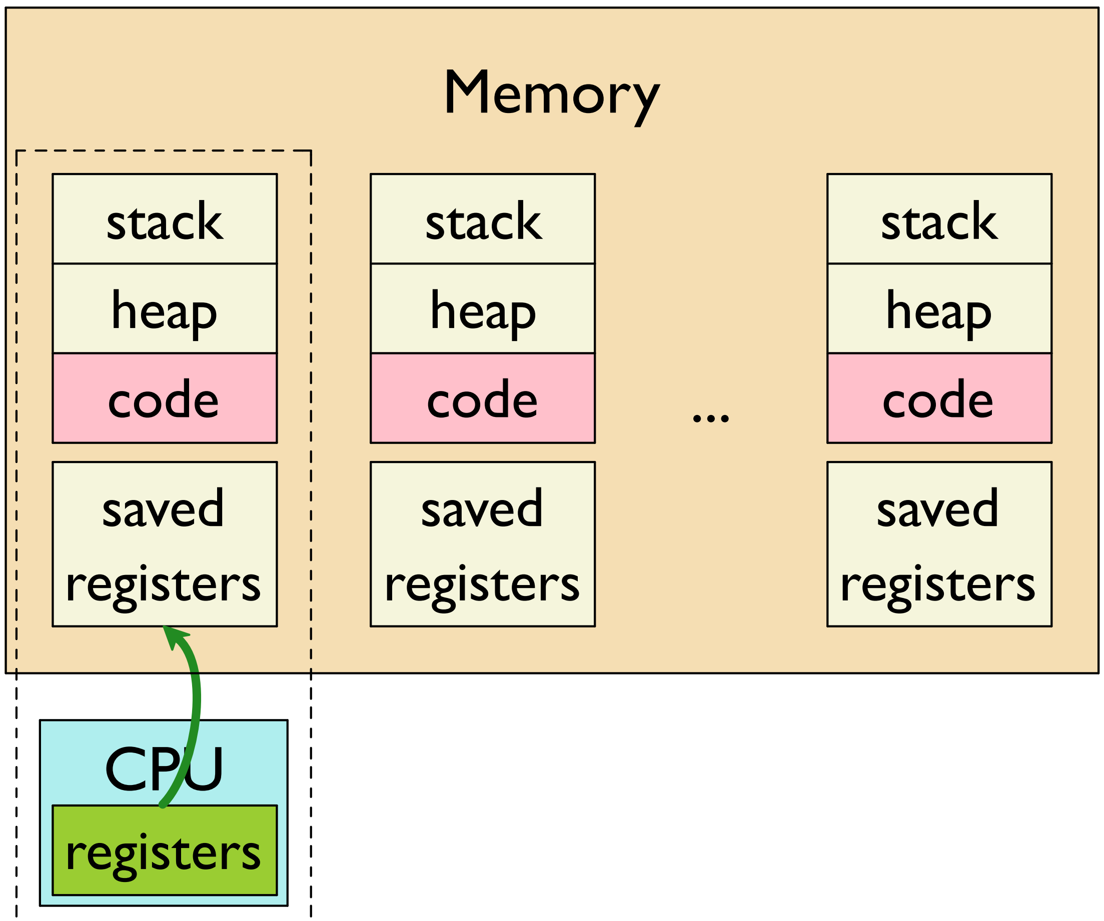
Multiprocessing: The Illusion



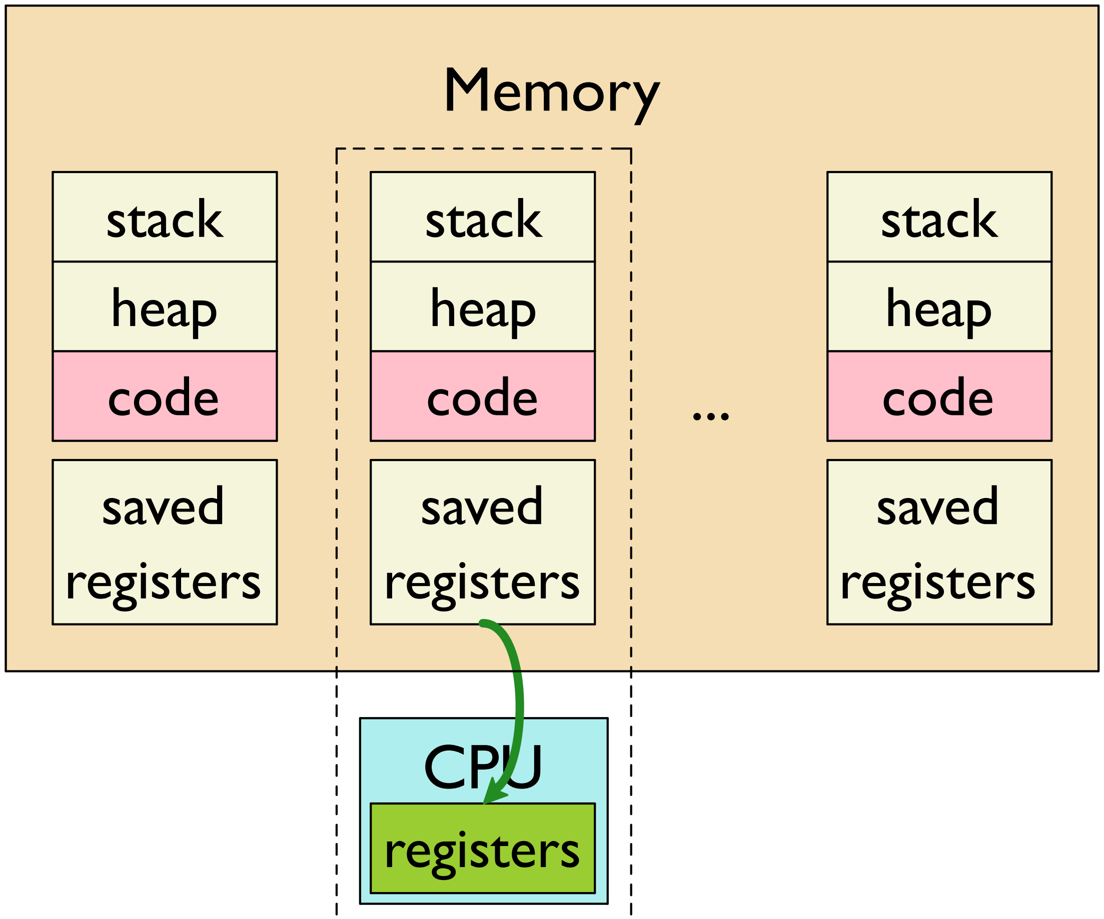
Multiprocessing: The Reality (Single Core)



Multiprocessing: The Reality (Single Core)

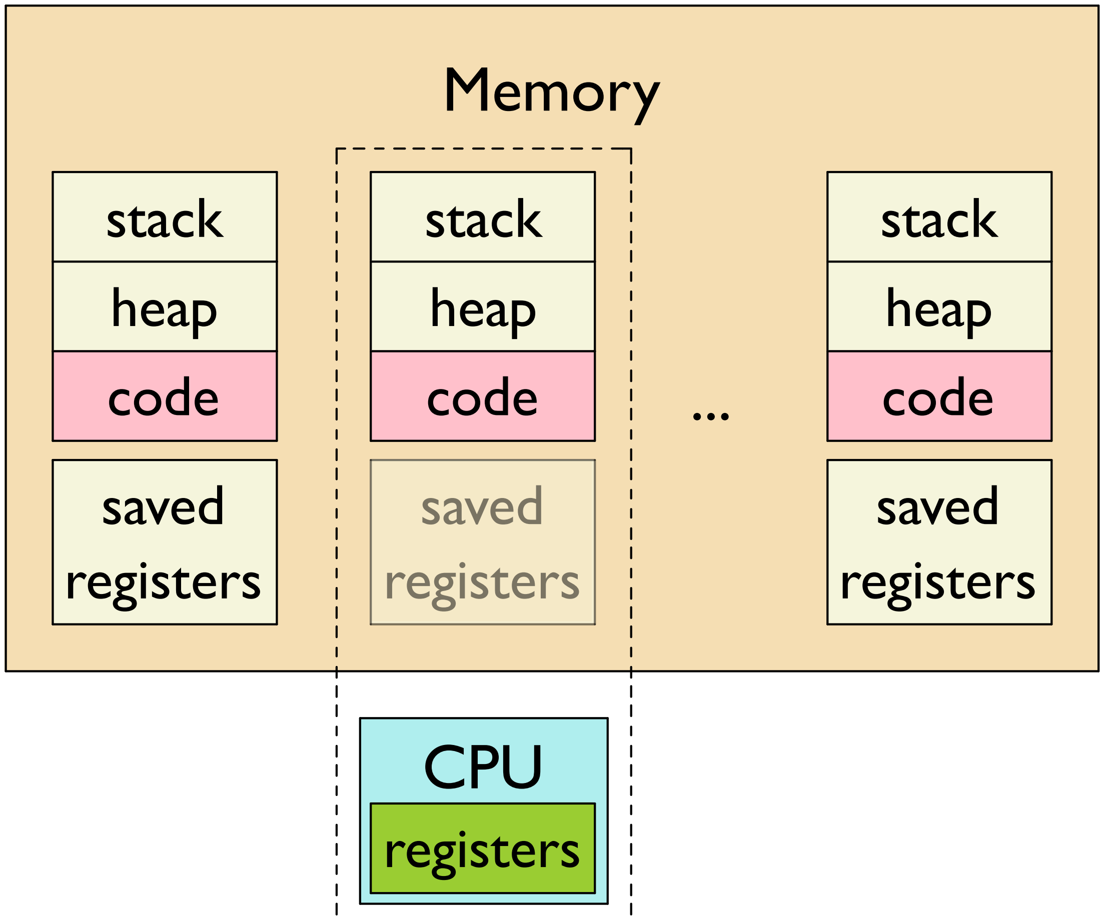


Multiprocessing: The Reality (Single Core)

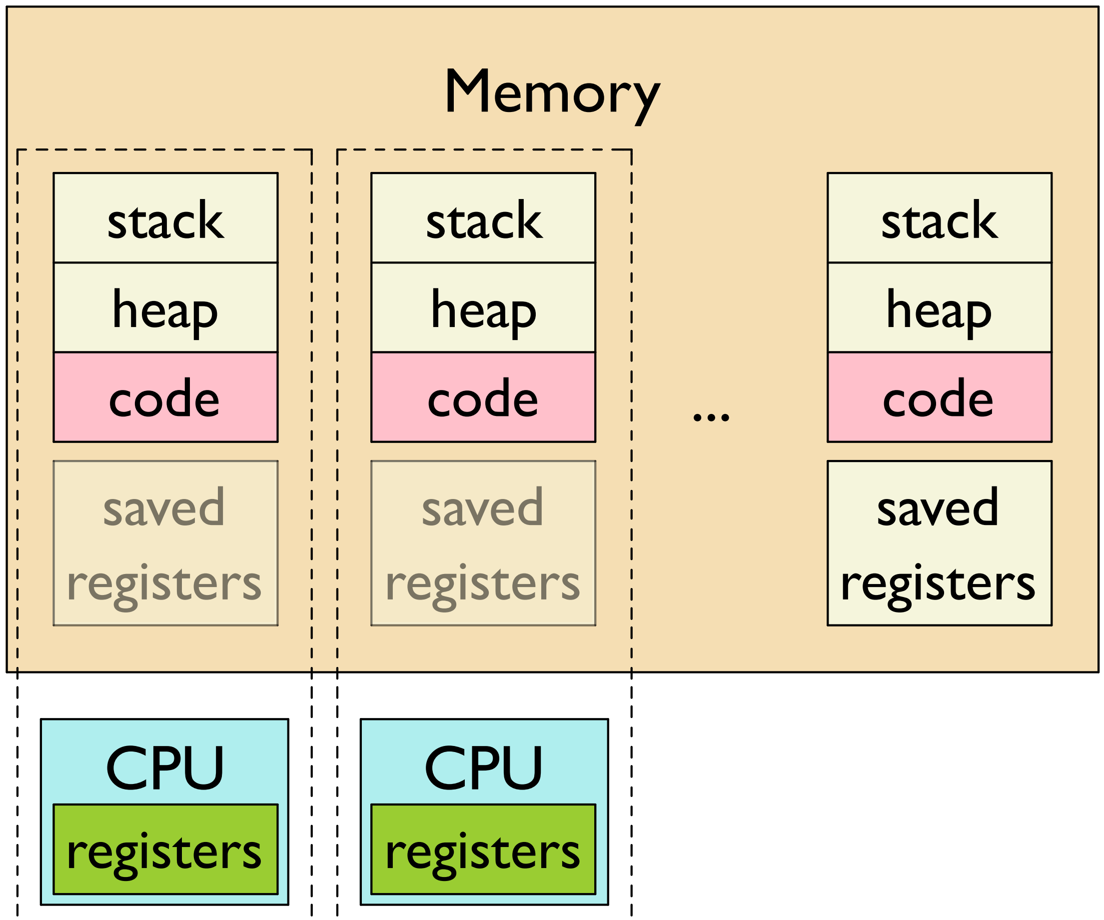


Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

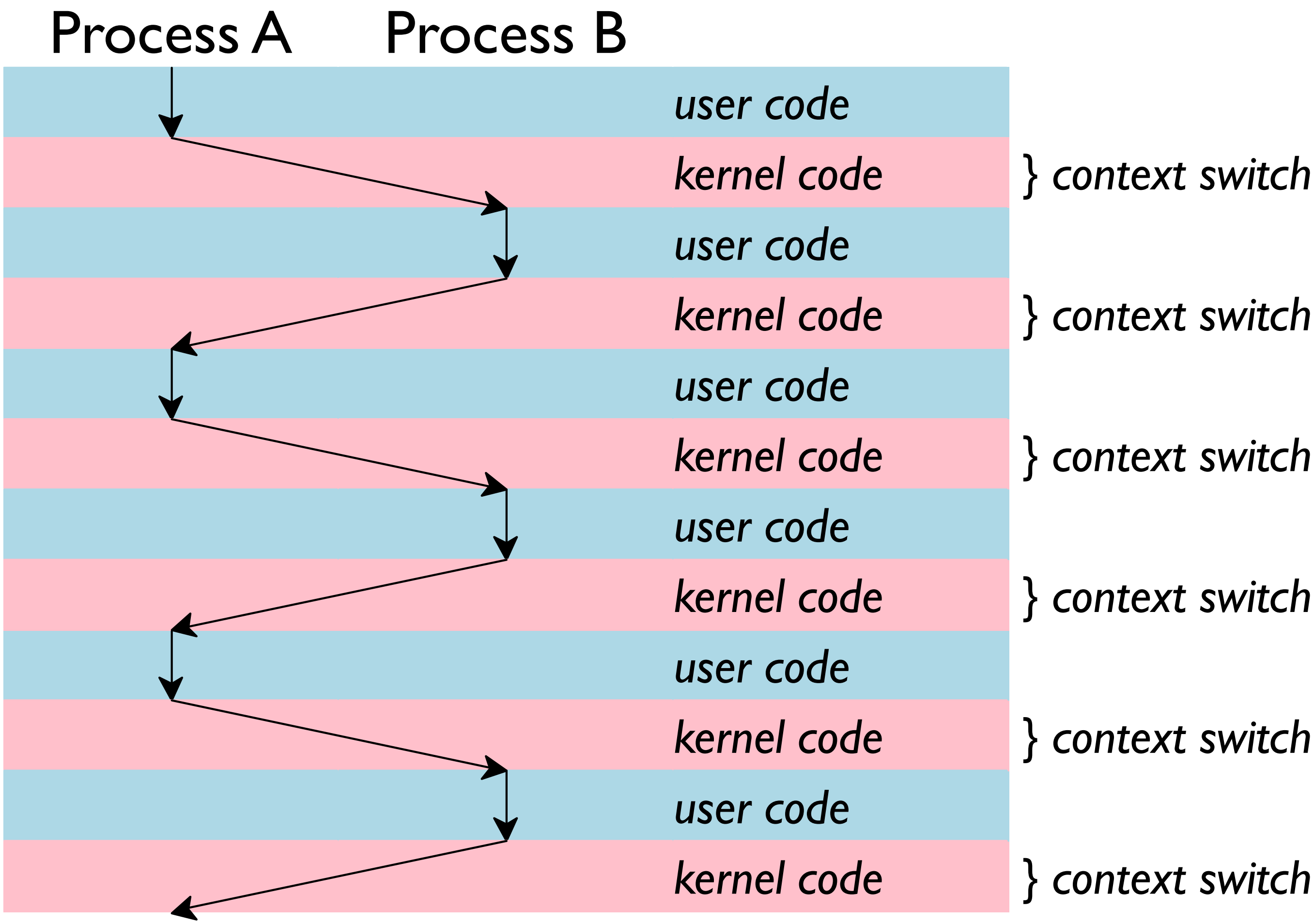
Multiprocessing: The Reality (Single Core)



Multiprocessing: The Reality (Multicore)

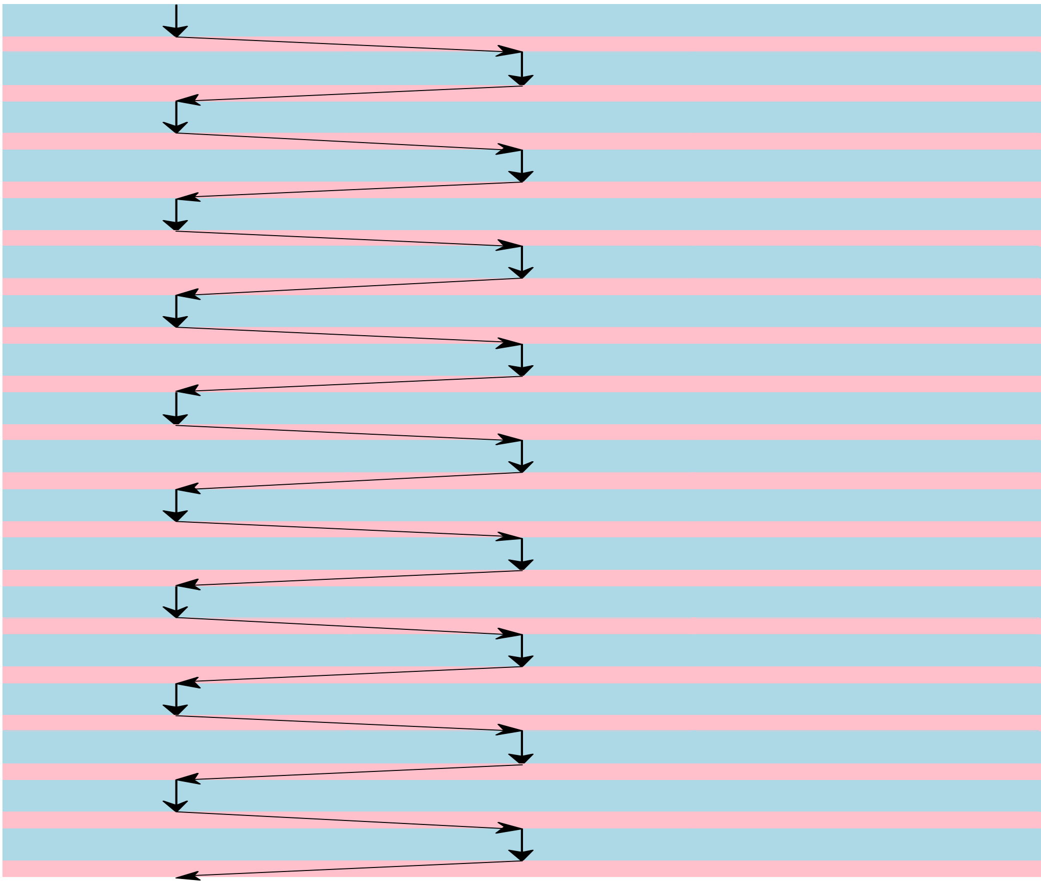


Multiprocessing Concurrency



Multiprocessing Concurrency

Process A Process B



top

mflatt@localhost:~/cs4400

File Edit View Search Terminal Help

```
top - 06:54:47 up 8 days,  8:04,  2 users,  load average: 0.29, 0.09, 0.08
Tasks: 177 total,   2 running, 175 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.6 us,  0.7 sy,  0.0 ni, 87.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.
KiB Mem :  1560592 total,   150756 free,   703452 used,   706384 buff/cache
KiB Swap:  1257468 total,   992736 free,   264732 used.   670244 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
15945 mflatt    20   0 1549092 259756  14532 S 12.3 16.6 201:29.76 gnome-s+
 7841 root      20   0  255548  49036   3508 S  3.7  3.1 160:10.31 X
  457 mflatt    20   0 1003284 170764  51552 S  3.3 10.9   0:05.58 firefox
  384 root      20   0       0      0      0 S  0.3  0.0   2:54.44 xfsaild+
 9026 mflatt    20   0  473224   9960    952 S  0.3  0.6  12:45.92 ibus-da+
 9039 mflatt    20   0  558012  13308   5628 S  0.3  0.9   2:37.71 gnome-t+
 9044 mflatt    20   0  452392   2020   1048 S  0.3  0.1  10:38.84 ibus-x11
 9172 mflatt    20   0  581396  10624   1428 S  0.3  0.7   1:43.52 caribou
20291 mflatt    20   0  585084  43416  17460 S  0.3  2.8   0:42.26 emacs
    1 root      20   0  126516   4908   2404 S  0.0  0.3   0:38.40 systemd
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.49 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:03.07 ksoftir+
    7 root      rt   0       0      0      0 S  0.0  0.0   0:00.00 migrati+
    8 root      20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S  0.0  0.0   0:00.00 rcuob/0
```

A CPU-Wasting Program

spin.c

```c
int main() {
  while (1) { }
}
```

Copy

`ps` and `kill`

List some processes:

```
$ ps
```

List all processes started by you:

```
$ ps x
```

List all processes:

```
$ ps ax
```

`kill`

Interrupt a process:

> **`$ kill`** *id*

An Uncooperative CPU-Wasting Program

spin.c

```
#include <signal.h>

int main() {
  signal(SIGINT, SIG_IGN);
  signal(SIGTERM, SIG_IGN);
  while (1) { }
}
```

Copy

`kill -9`

Interrupt an uncooperative program:

`$ kill -SIGKILL` *pid*

or

`$ kill -9` *pid*

getpid

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

Copy

Gets the current process's ID as an integer

getppid

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid(void);
```

Copy

Gets the ID of the process that started the current process

Getting A Process ID

In **`/usr/lib64/libc.so.6`**:

```
<getppid>:
  mov    $0x6e,%eax
  syscall
  retq
```

The C Library vs. System Calls

Opening a file in **portable** C:

```
FILE *f = fopen("data.txt", "r");
```

Opening a file in **Unix**:

```
int f = open("data.txt", O_RDONLY);
```

man fopen ⇒ FOPEN(3)

(3) means "C library"

man open ⇒ OPEN(2)

(2) means "system call"

System Calls and Error

```
int f = open("nosuchfile.txt", O_RDONLY);
```

No exception... just a **`-1`** value for **`f`**

System Calls and Error

Most system calls return an integer result

Most report an error as a **`-1`** result

The **`errno`** variable provides details

```
int f = open("nosuchfile.txt", O_RDONLY);

if (f == -1) {
  /* Handle error */
  fprintf(stderr, "open failed (%s)",
                  strerror(errno));
  exit(1);
}
```

This is a pain...

Syscall Wrapper for Errors

Slightly simplified **`open`** implementation:

```
<open>:
  e82a9:  mov    $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp    $0xfffffffffffff001,%rax
  e82b6:  jae    e82e9   # jump if in error range
  e82b8:  retq
  e82e9:  mov    0x2d2b78(%rip),%rcx     # &errno
  e82f0:  neg    %eax
  e82f2:  mov    %eax,(%rcx)            # set errno
  e82f5:  or     $0xffffffffffffffff,%rax
  e82f9:  retq
```

Syscall Wrapper for Errors

Slightly simplified **open** implementation:

0x2 means **open**

```
<open>:
  e82a9:  mov     $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp     $0xfffffffffffff001,%rax
  e82b6:  jae     e82e9   # jump if in error range
  e82b8:  retq
  e82e9:  mov     0x2d2b78(%rip),%rcx       # &errno
  e82f0:  neg     %eax
  e82f2:  mov     %eax,(%rcx)            # set errno
  e82f5:  or      $0xffffffffffffffff,%rax
  e82f9:  retq
```

Syscall Wrapper for Errors

Slightly simplified **`open`** implementation:

```
<open>:
  e82a9:  mov     $0x2,%ea
  e82ae:  syscall
  e82b0:  cmp     $0xfffffffffffff001,%rax
  e82b6:  jae     e82e9   # jump if in error range
  e82b8:  retq
  e82e9:  mov     0x2d2b78(%rip),%rcx     # &errno
  e82f0:  neg     %eax
  e82f2:  mov     %eax,(%rcx)             # set errno
  e82f5:  or      $0xffffffffffffffff,%rax
  e82f9:  retq
```

`-1` to **`-4096`** means an error

Syscall Wrapper for Errors

Slightly simplified **`open`** implementation:

```
<open>:
  e82a9:  mov    $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp    $0xfffffffffffff001,%rax
  e82b6:  jae    e82e9                              nge
  e82b8:  retq
  e82e9:  mov    0x2d2b78(%rip),%rcx       # &errno
  e82f0:  neg    %eax
  e82f2:  mov    %eax,(%rcx)             # set errno
  e82f5:  or     $0xffffffffffffffff,%rax
  e82f9:  retq
```

address of shared **`errno`**

Syscall Wrapper for Errors

Slightly simplified **open** implementation:

```
<open>:
  e82a9:  mov     $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp     $0xfffffffffffff001,%rax
  e82b6:  jae     e82e9   # jump if in error range
  e82b8:  retq
  e82e9:  mov     0x2[illegible]2b70(%rip),%rcx     # &errno
  e82f0:  neg     %eax
  e82f2:  mov     %eax,(%rcx)             # set errno
  e82f5:  or      $0xffffffffffffffff,%rax
  e82f9:  retq
```

negate result as **errno**

Syscall Wrapper for Errors

Slightly simplified **open** implementation:

```
<open>:
  e82a9:  mov     $0x2,%eax
  e82ae:  syscall
  e82b0:  cmp     $0xfffffffffffff001,%rax
  e82b6:  jae     e82e9    # jump if in error range
  e82b8:  retq
  e82e9:  mov     0x2d2b78(%rip),%rcx      # &errno
  e82f0:  neg     %eax
  e82f2:  mov     %eax,(%rcx)            # set errno
  e82f5:  or      $0xffffffffffffffff,%rax
  e82f9:  retq
```

return -1

Textbook Wrapper for Errors

More help from **`csapp.h`** and **`csapp.c`**:

`csapp.c`

```
....

void unix_error(char *msg) {
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(0);
}
....

int Open(const char *pathname, int flags, mode_t mode) {
  int rc;

  if ((rc = open(pathname, flags, mode))  < 0)
    unix_error("Open error");
  return rc;
}
....
```

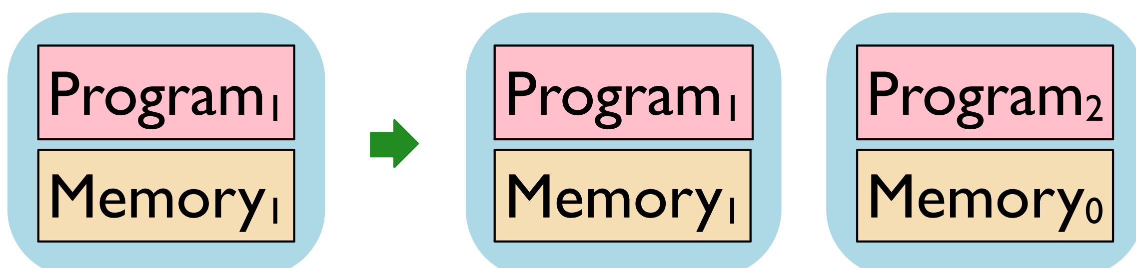
Creating a New Process

The system call that you'd expect:

```
int newprocess(char *prog, int argc, char **argv);
```

Create a new process with a given program

If Program$_1$ starts Program$_2$:



Creating a New Process

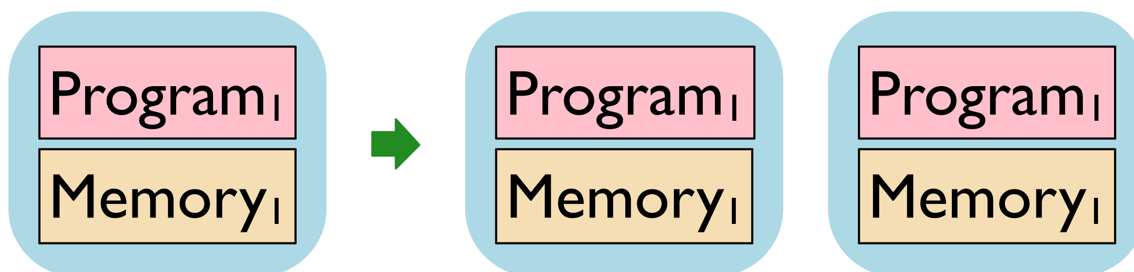
The system calls provided by Unix:

```
int fork();
int execve(char *prog, char **argv, char **env);
```

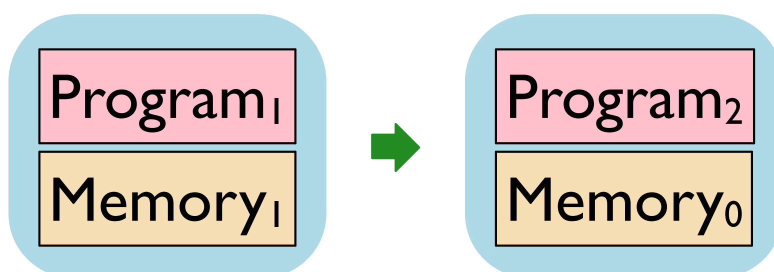
`fork` creates a *copy* of the current process

`execve` *replaces* the current process

`fork`:



`execve`:



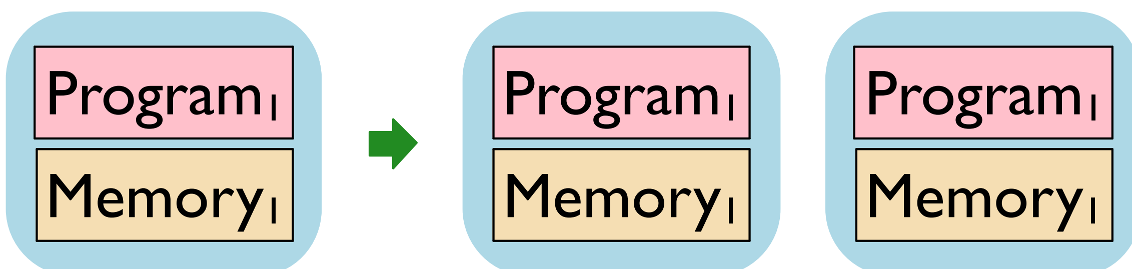
Creating a New Process

The system calls provided by Unix:

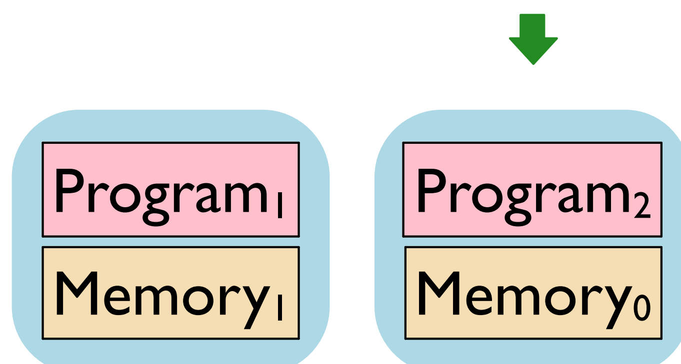
```
int fork();
int execve(char *prog, char **argv, char **env);
```

`newprocess` = **`fork`** + **`execve`**

`fork`



+ **`execve`** in copy, only



Fork

```
#include <unistd.h>

pid_t fork(void);
```

Creates a new process as a copy of the current one, but:

- Copy has a different PID
- Returns that PID to the original, *parent* process
- Returns 0 to the new, *child* process

Called once, returns twice!

Fork Example

```
#include "csapp.h"

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

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

  return 0;
}
```

Copy

- Separate copies of **`x`**
- Order of **`printf`**s unspecified

Process Graphs

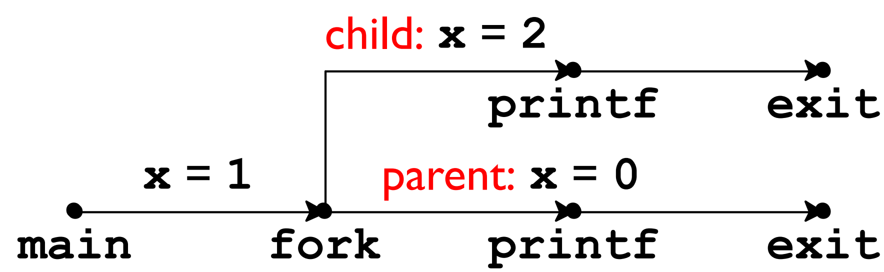
We can reason about concurency with a ***process graph***

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

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

  return 0;
}
```

Copy

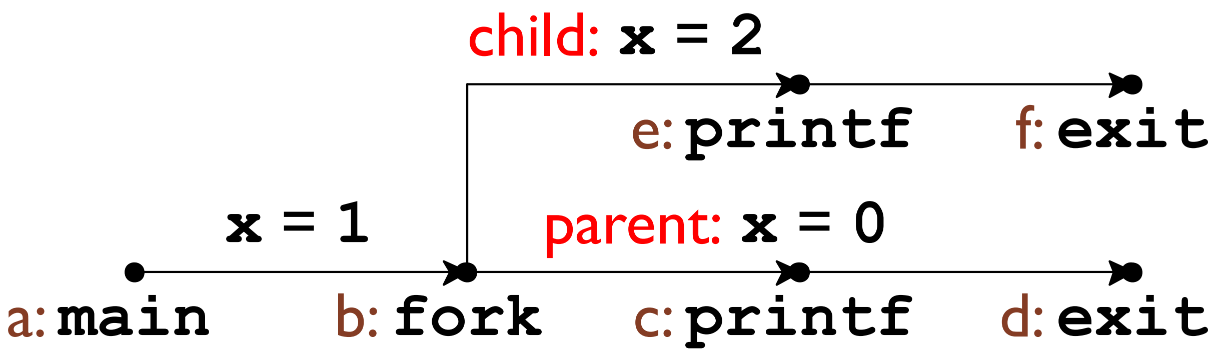


Each node • is an externally visible action

Edges can be annotated with internal state changes

A *topological sort* of the graph is a possible ordering of events

Ordering by Process Graph



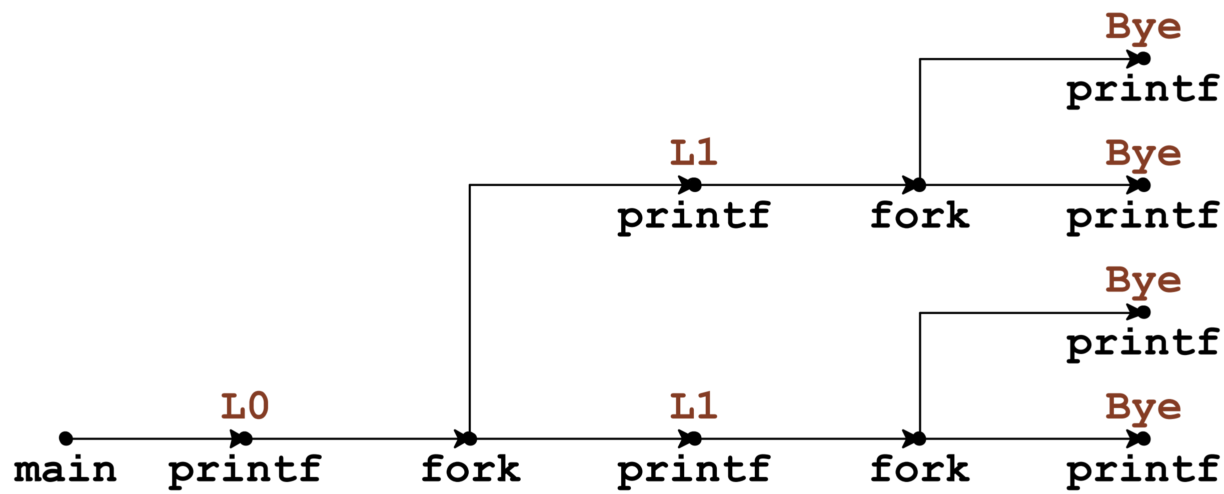
possible order: a b e c f d

impossible order: a b f c e d

Consecutive Forks

```
int main() {
  printf("L0\n");
  Fork();
  printf("L1\n");
  Fork();
  printf("Bye\n");
  return 0;
}
```

Copy



Possible output:	Impossible output:
L0	L0
L1	Bye
Bye	L1
Bye	Bye
L1	L1
Bye	Bye
Bye	Bye

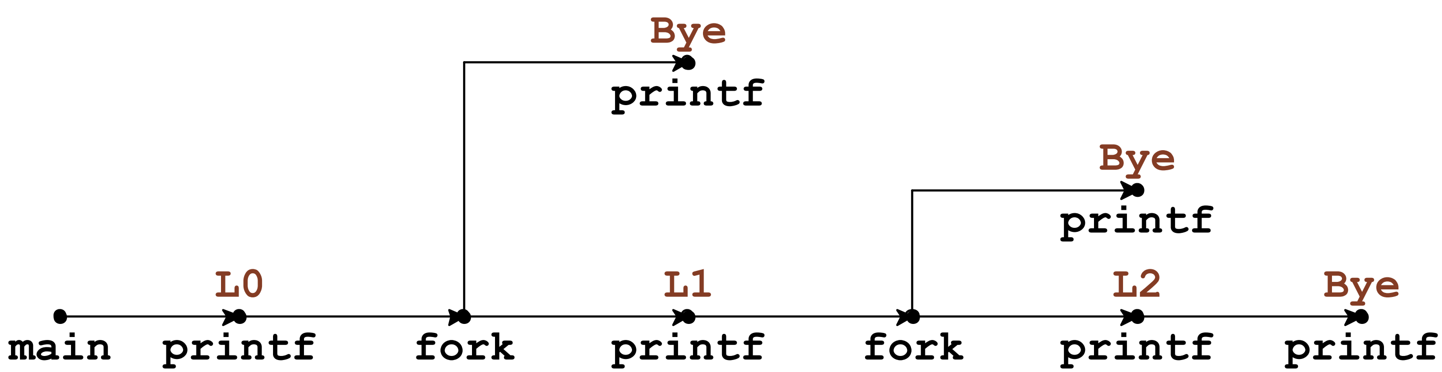
Nested Forks in Parent

```
int main() {
  printf("L0\n");
  if (Fork() != 0) {
    printf("L1\n");
    if (Fork() != 0)
      printf("L2\n");
  }
  printf("Bye\n");
}
```

Copy

Possible output:
`L0`
`L1`
`Bye`
`Bye`
`L2`
`Bye`

Impossible output:
`L0`
`Bye`
`L1`
`Bye`
`Bye`
`L2`



Nested Forks in Children

```
int main() {
  printf("L0\n");
  if (Fork() == 0) {
    printf("L1\n");
    if (Fork() == 0)
      printf("L2\n");
  }
  printf("Bye\n");
}
```

Copy

Possible output:	Impossible output:
L0	L0
Bye	Bye
L1	L1
L2	Bye
Bye	Bye
Bye	L2