Altering the Control Flow

Up to Now: two mechanisms for changing control flow:

- Jumps and branches
- Call and return using the stack discipline.
- Both react to changes in program state.

Insufficient for a useful system

- Difficult for the CPU to react to changes in system state.
  - data arrives from a disk or a network adapter.
  - Instruction divides by zero
  - User hits ctl-c at the keyboard
  - System timer expires

System needs mechanisms for “exceptional control flow”

Exceptional Control Flow

- Mechanisms for exceptional control flow exists at all levels of a computer system.

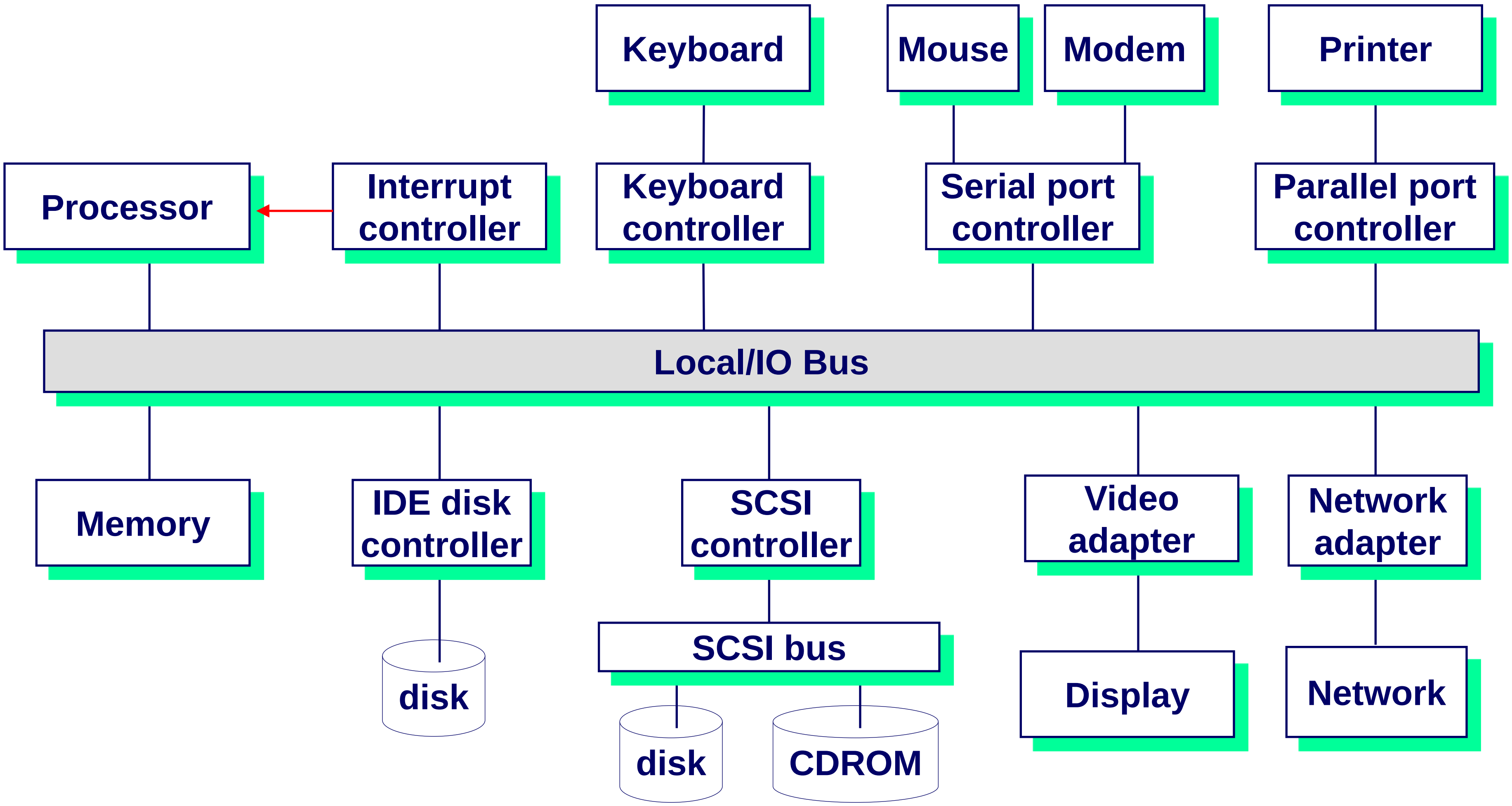
Low level Mechanism

- exceptions
    - change in control flow in response to a system event (i.e., change in system state)
- Combination of hardware and OS software

Higher Level Mechanisms

- Process context switch
- Signals
- Nonlocal jumps (setjmp/longjmp)
- Implemented by either:
    - OS software (context switch and signals).
    - C language runtime library: nonlocal jumps.

System context for exceptions



Keyboard

Mouse

Modem

Printer

Processor

Interrupt controller

Keyboard controller

Serial port controller

Parallel port controller

Local/IO Bus

Memory

IDE disk controller

SCSI controller

Video adapter

Network adapter

disk

SCSI bus

disk

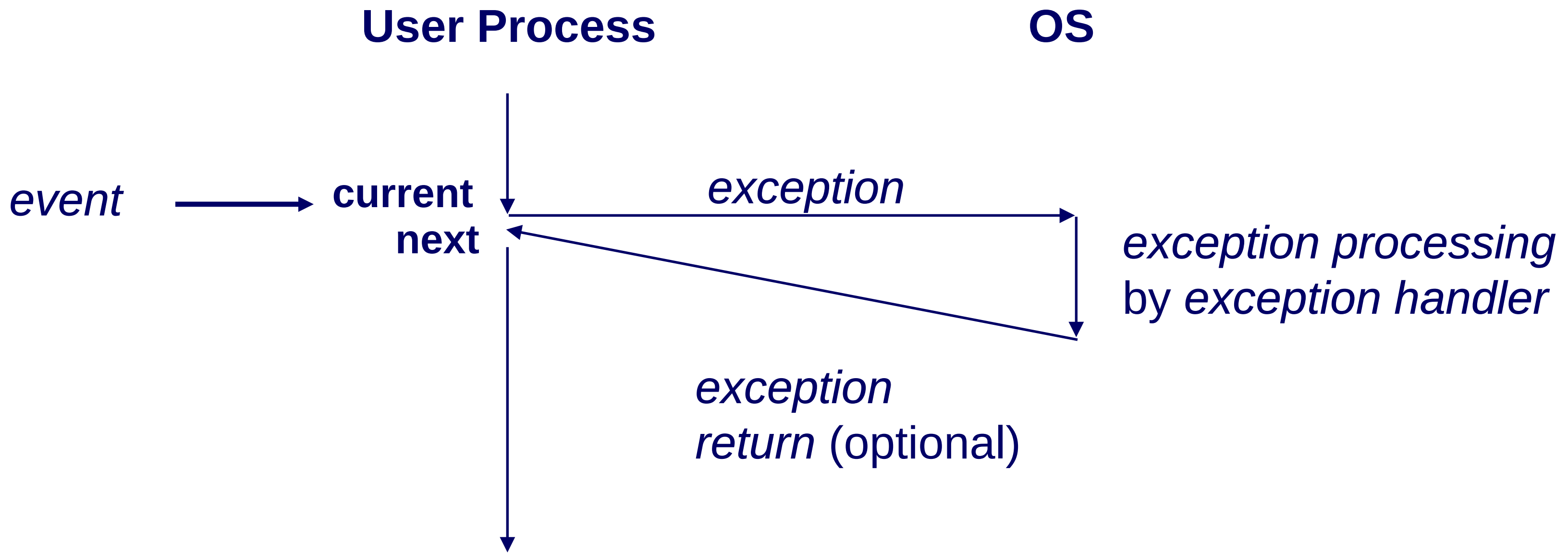
CDROM

Display

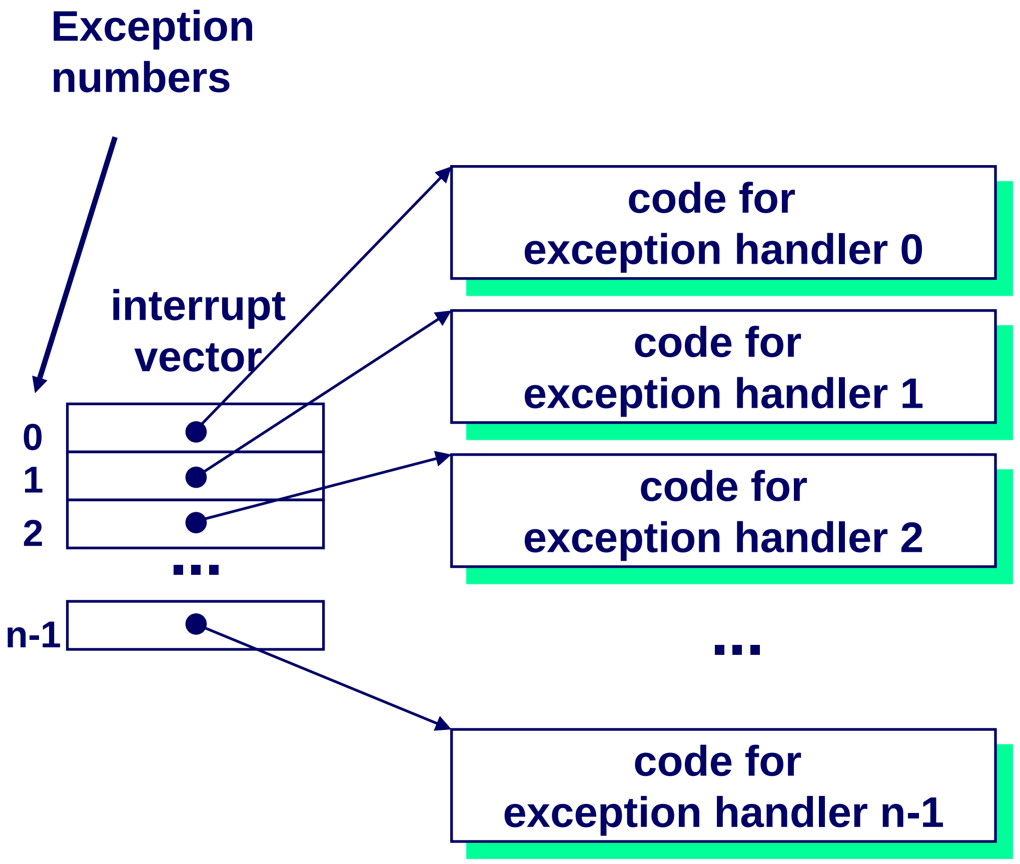
Network

Exceptions

An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)

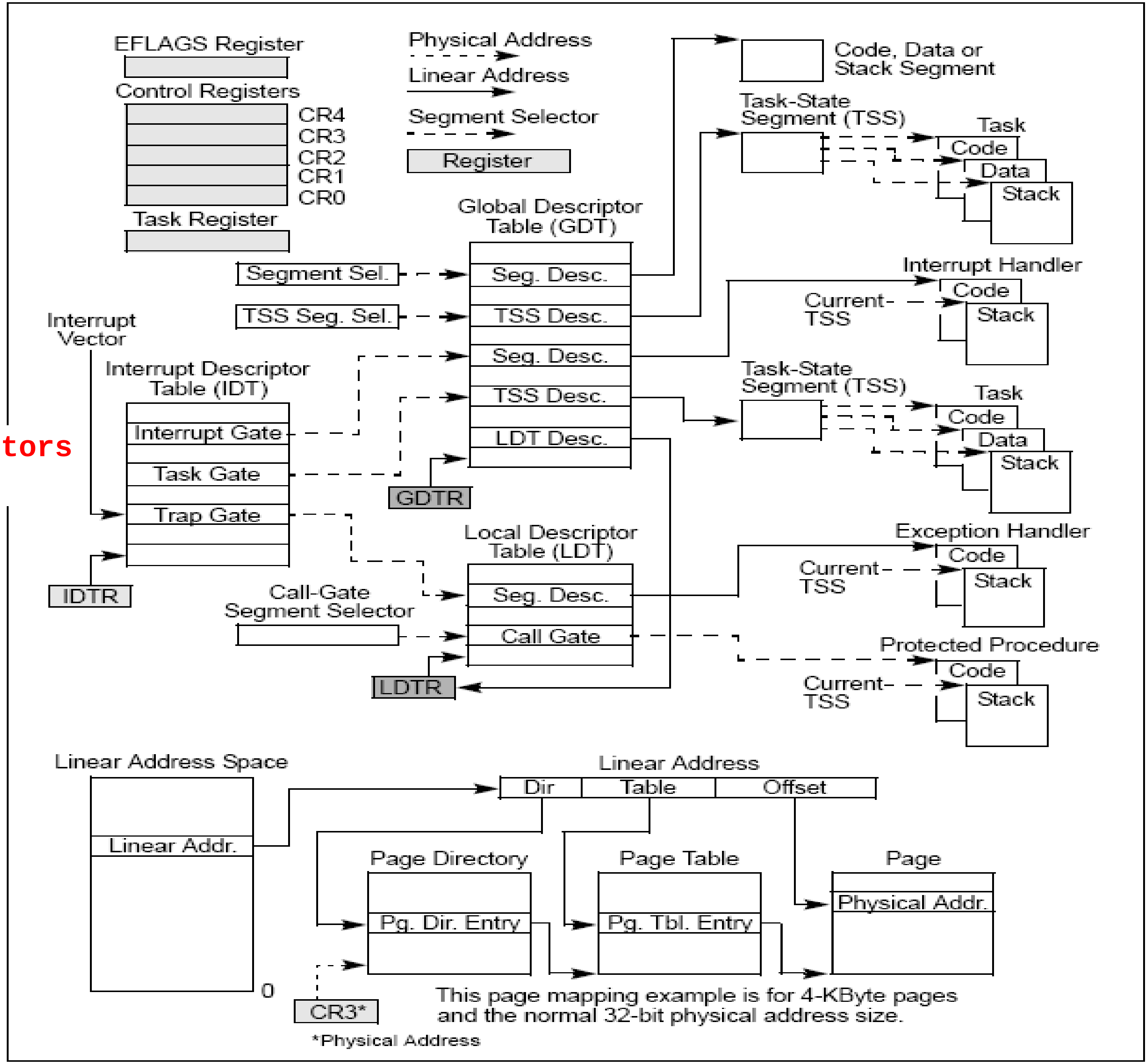


Interrupt Vectors



- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

80x86 System Level Registers



256 vectors max

Asynchronous Exceptions (Interrupts)

Caused by events external to the processor

- Indicated by setting the processor’s interrupt pin
- handler returns to “next” instruction.

Examples:

- I/O interrupts
  - hitting ctl-c at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector from a disk
- Hard reset interrupt
  - hitting the reset button
- Soft reset interrupt
  - hitting ctl-alt-delete on a PC

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).
  - Either re-executes faulting (“current”) instruction or aborts.
- Aborts
  - unintentional and unrecoverable
  - Examples: parity error, machine check.
  - Aborts current program

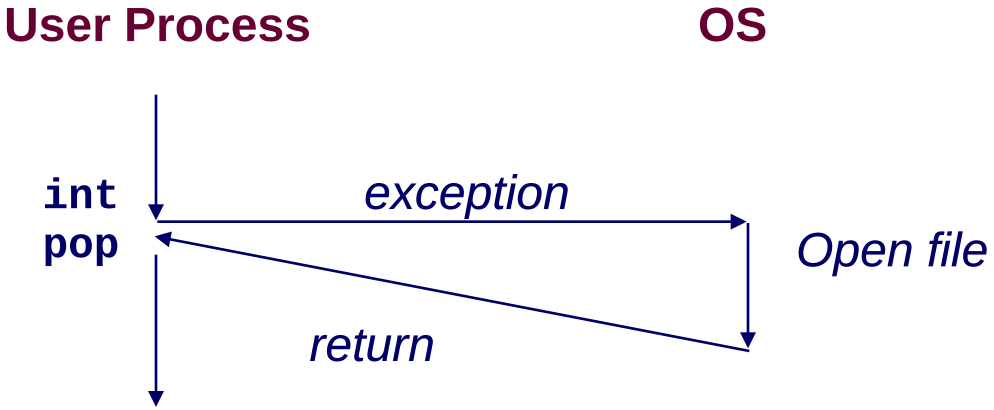
Trap Example

Opening a File

- User calls `open(filename, options)`

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                          int     $0x80
 804d084:       5b                             pop     %ebx
 . . .
```

  - Function open executes system call instruction `int`
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor



Fault Example #1

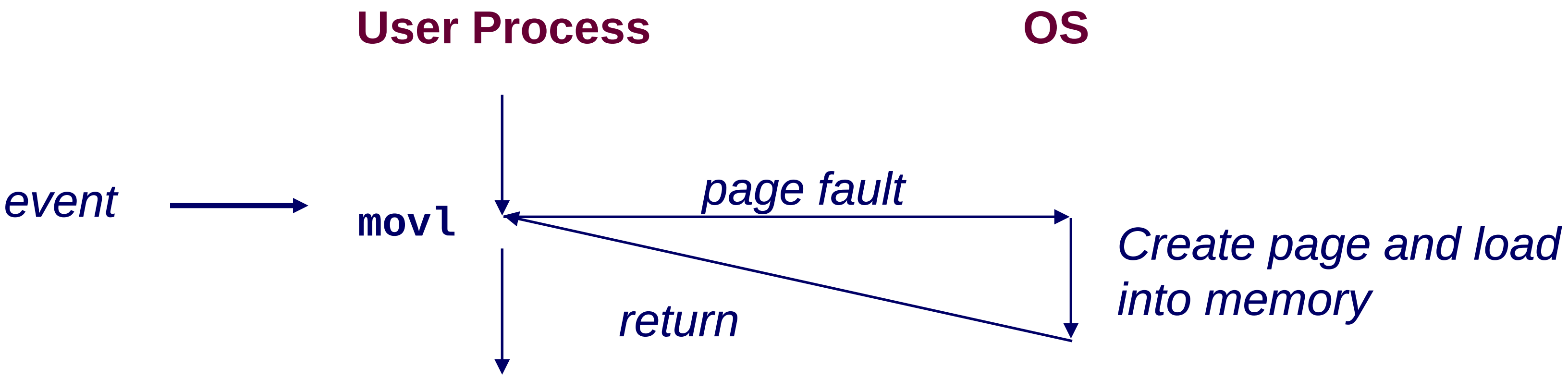
```
int a[1000];
main ()
{
    a[500] = 13;
}
```

Memory Reference

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

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

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



Fault Example #2

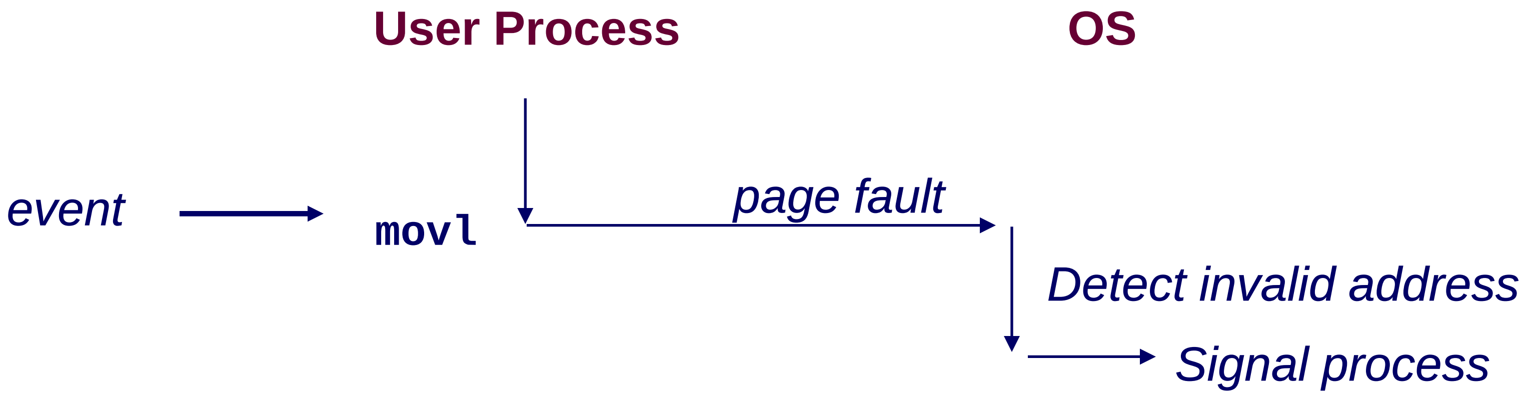
```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

Memory Reference

- User writes to memory location
- Address is not valid

```
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"



User Process

OS

event

`movl`

page fault

Detect invalid address

Signal process

Processes

Def: 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”

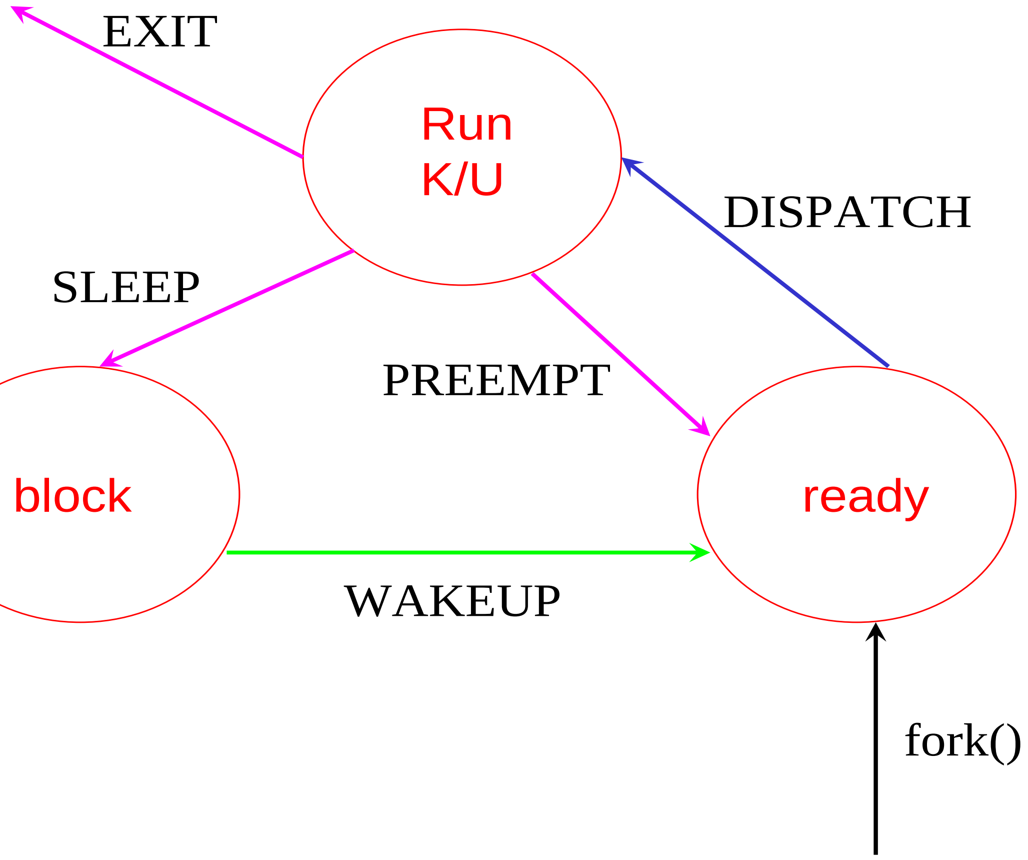
A process provides each program with two key abstractions:

- Logical control flow
  - Each thread of a process seems to have exclusive use of a CPU.
- Private address space
  - Each process seems to have exclusive use of main memory.

How are these Illusions maintained?

- Process thread executions are interleaved (multitasking)
- Address spaces are managed by a virtual memory system

Thread States and Transitions



Threads

The executable (schedulable) elements in a Linux system

Each thread in the system is uniquely contained by some process

- Each user thread is contained by some user PID
- Each kernel thread is contained in PID 0

When a new process is created, it is populated by exactly one executable thread, known as the *Initial Thread* (IT) of the new process

The IT of a process can create new threads only within its own process

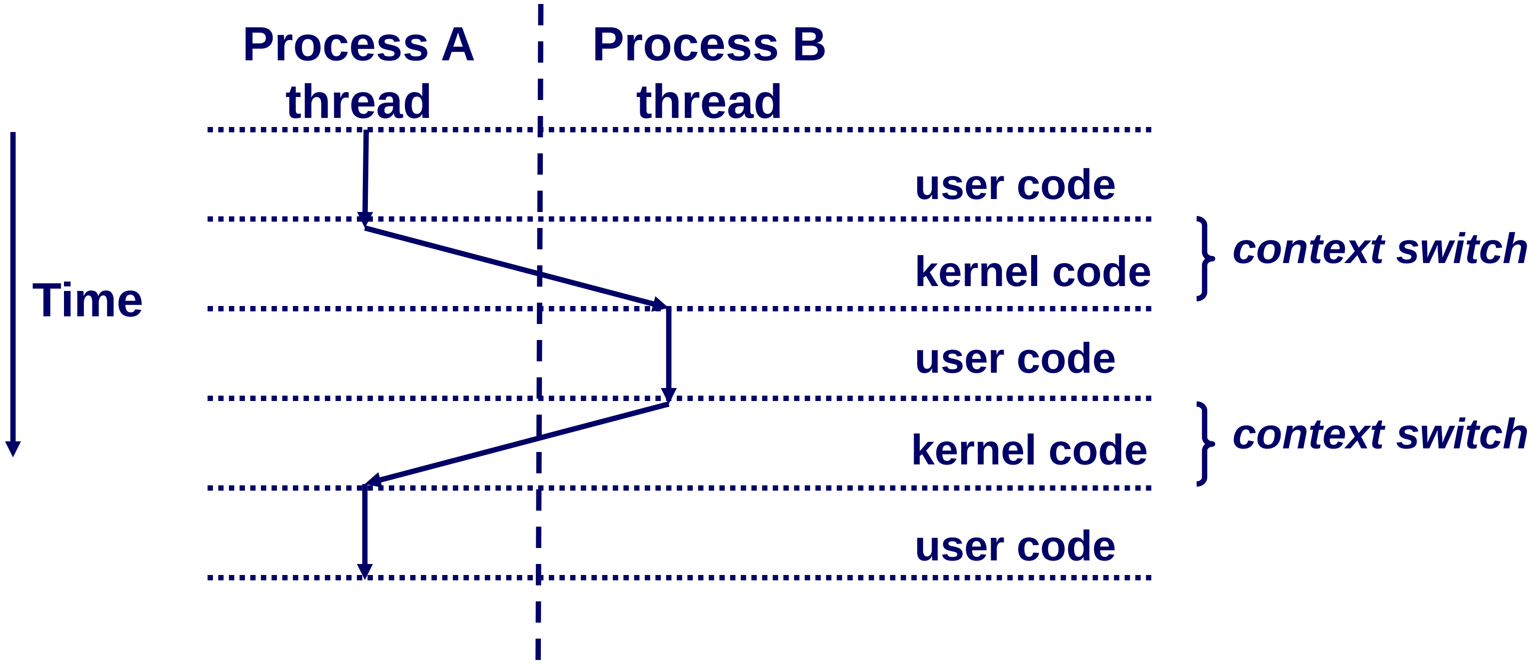
While the IT must create the *second thread* in a process, any subsequent threads can then create new threads, but only within their own process

Context Switching

Processes are managed by a shared chunk of OS code called the *kernel*

- Important: the kernel is not a separate process, but rather runs as part of some thread in some user process

Control flow passes from one thread in a process to another thread in the same or a different process via a *context switch.*



fork: Creating new processes

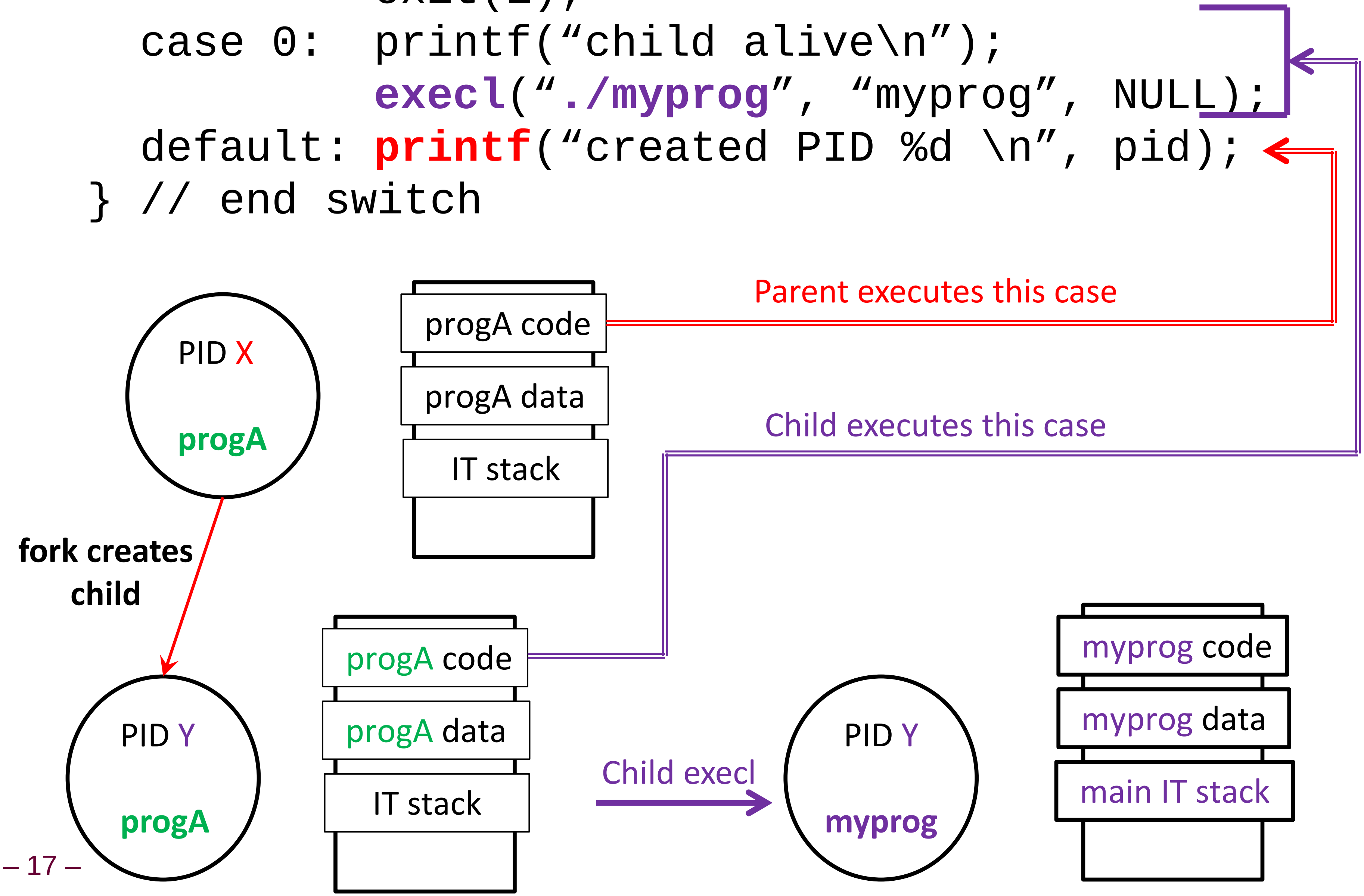
`int fork(void)`

- creates a new process (child process) that is identical to the calling process (parent process)
- returns 0 to the child process
- returns child's pid to the parent process

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

```
switch (int pid = fork()){
  case -1: perror(“fork failed “);
           exit(1);
  case 0:  printf(“child alive\n”);
           execl(“./myprog”, “myprog”, NULL);
  default: printf(“created PID %d \n”, pid);
} // end switch
```



Parent executes this case

Child executes this case

fork creates child

Child execl

Fork Example #1

Key Points

- Parent and child both run same code
  - Distinguish parent from child by return value from `fork`
- Child inherits a copy-on-write (COW) version of parent
  - Including all parent open file descriptors (stdin, stdout, etc.)
  - Relative ordering of parent/child print statements undefined

```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Some fork_test runs

cs

```
bill@cs3:~/cs305demo$ ./fork_test
Parent has x = 0
Bye from process 24697 with x = 0
Child has x = 2
Bye from process 24698 with x = 2
```

cs

```
bill@cs3:~/cs305demo$ ./fork_test
Child has x = 2
Parent has x = 0
Bye from process 24700 with x = 2
Bye from process 24699 with x = 0
```

mercury

```
-bash-4.1$ ./fork_test
Parent has x = 0
Bye from process 10279 with x = 0
Child has x = 2
Bye from process 10280 with x = 2
```

cs

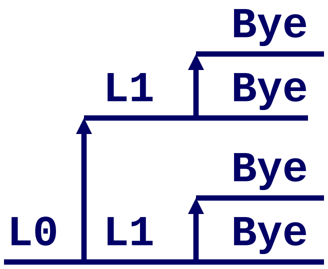
```
bill@cs3:~/cs305demo$ ./fork_test
Parent has x = 0
Child has x = 2
Bye from process 24350 with x = 0
Bye from process 24351 with x = 2
```

Fork Example #2

Key Points

- **Both parent and child can continue forking**

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



`exit`: Destroying Process

`void exit(int status)`

- exits a process
  - Normally return with status 0
- `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Zombies

Idea

- When process terminates, still consumes system resources
  - Various tables maintained by OS
- Called a “zombie”
  - Living corpse, half alive and half dead

Reaping

- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards process

What if Parent Doesn’t Reap?

- If any parent terminates without reaping a child, then child will be reaped by `init` process
- Only need explicit reaping for long-running processes
  - E.g., shells and servers

`wait`: Synchronizing with children

`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 object it points to will be set to a status indicating why the child process terminated

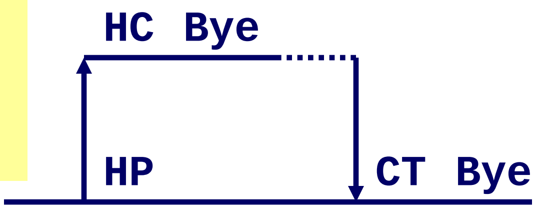
Declare a typedef for the exit status information returned from the `wait()` call (`pid = wait(int *status)`)

```
typedef union{
        int exit_status;
        struct{
                unsigned  sig_ num:7;
                unsigned  core_dmp:1;
                unsigned  exit_num:8;
        }parts;
}LE_Wait_Status
```

`wait`: Synchronizing with children

```
void fork9() {
   int child_status;

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



exec: Running new programs

`int execl(char *path, char *arg0, char *arg1, …, (char *)NULL)`

- loads and runs executable at `path` with args `arg0`, `arg1`, …
  - `path` is the complete path of an executable
  - `arg0` becomes the name of the process
    - typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`
  - “real” arguments to the executable start with `arg1`, etc.
  - list of args is terminated by a `(char *)NULL` argument
- returns `-1` if error, otherwise doesn’t return!
  - “Toto, we’re not in Kansas anymore”

```
main() {
   if (fork() == 0) {
    execl("/usr/bin/cp", "cp", "foo", "bar", (char *)NULL);
   }
   wait(NULL);
   printf("copy completed\n");
   exit();
}
```

Summarizing

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
- Each process must have at least one execution thread
- Only one thread can execute on a processor (core) at a time, but the address space used on a core is that of the process whose thread is currently running there
- All threads of a given process share a common address space
- Each running thread appears to have total control of its core and its process's private address space
- The address space of a process can be in simultaneous use on multiple cores if the process has multiple running threads deployed across these multiple cores

Summarizing (cont.)

Spawning Processes

- Call to `fork()`
  - One call, two returns; one to parent, one to child in new process

Terminating Processes

- Call `exit(int exit_code)`
  - One call, no return
  - If called by any thread of a process, then all threads in the process will terminate, as will the process itself

Reaping Processes

- Call `wait (int * exit_status);`

Replacing Program Executed by Process

- Call `execl(char* path, char* argv0, … (char *)NULL);`
  - Actually can use any of 6 exec variants (execl, execlp, execv, etc.)
  - One call, new program starts at main()   (no return to caller)