Exceptional Control Flow Part I

Today

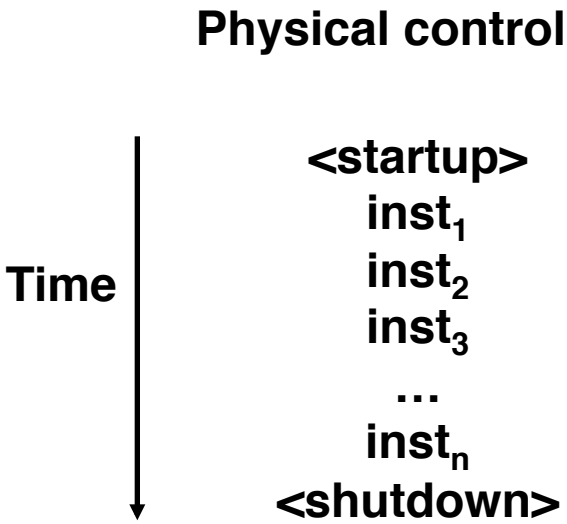
- Exceptions
- Process context switches
- Creating and destroying processes

Next time

- Signals, non-local jumps, …

Control flow

- Computers 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 system's physical *control flow* (or *flow of control*).



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” (ECF)

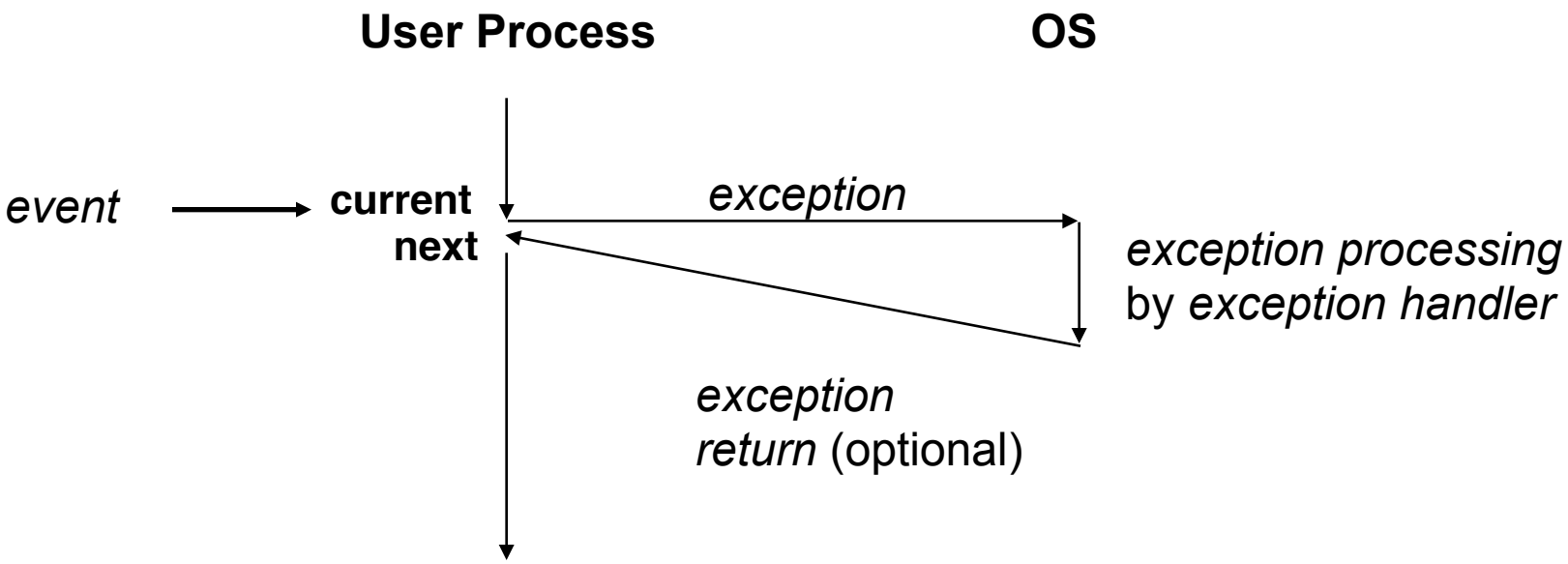
Exceptional control flow

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

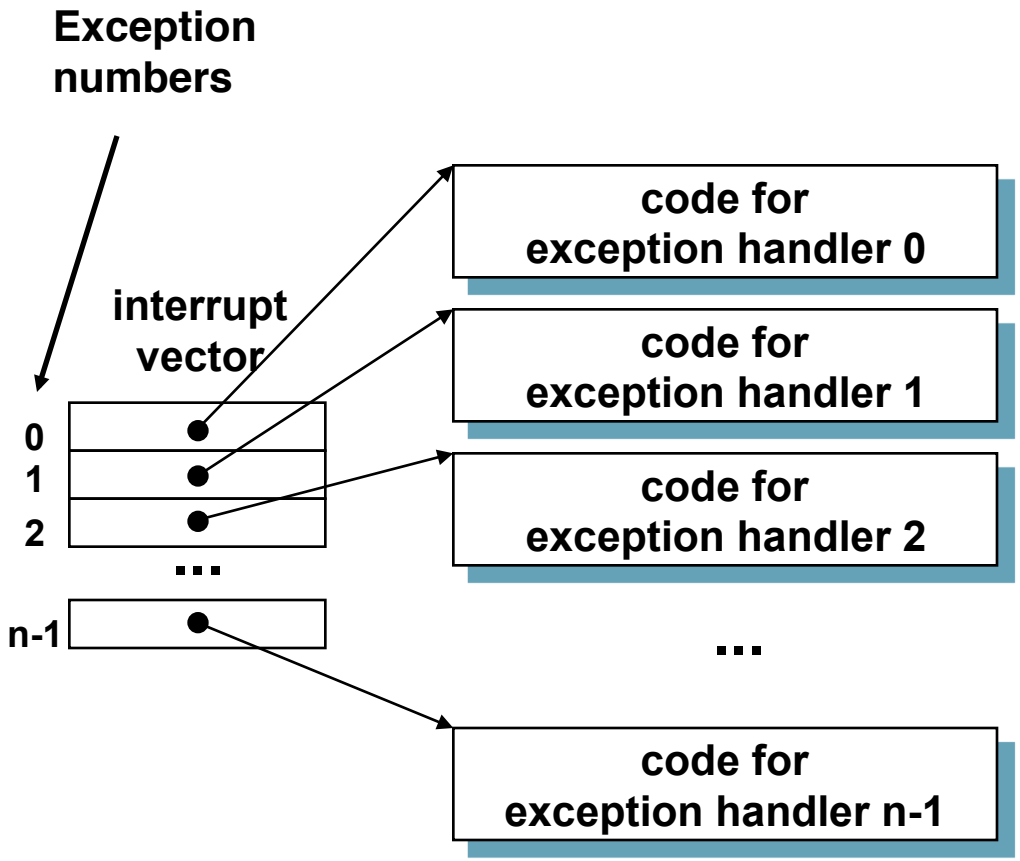
Exceptions

- Exception – a transfer of control to the OS in response to some event (i.e., change in processor state)



- E.g. page fault, arithmetic overflow, I/O done, …

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

Classes of exceptions

Class	Cause	(A)Sync	Return behavior
Interrupt	Signal from I/O device	Async	*Always* return to *next* instruction
Trap	Intentional exception	Sync	*Always* return to *next* instruction
Fault	Potentially recoverable error	Sync	*Might* return to *current* instruction
Abort	Non-recoverable error	Sync	*Never* returns

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:

Class	Cause	Examples	Return behavior
Trap	Intentional exception	System calls, breakpoint traps	*Always* return to *next* instruction
Fault	Potentially recoverable error	Page fault (recoverable), protection faults (unrecoverable)	*Might* return to *current* instruction
Abort	Non-recoverable error	Parity error, machine check	*Never* returns

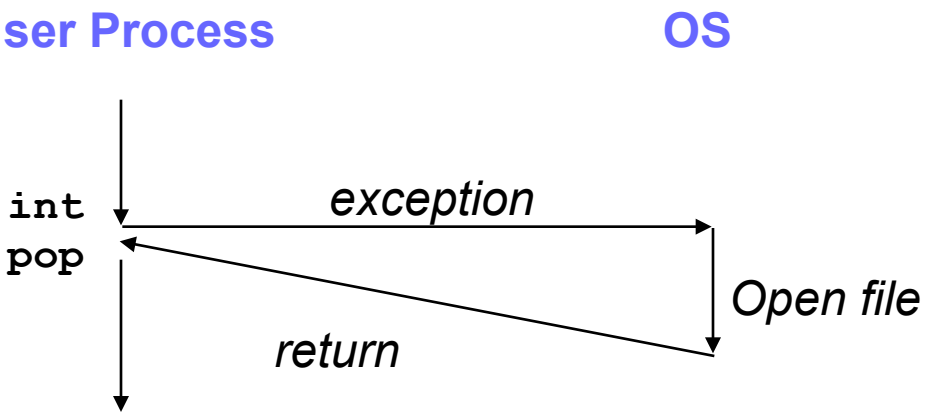
Trap example

- Opening a File
  - User calls `open(filename, options)`

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

System call code

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



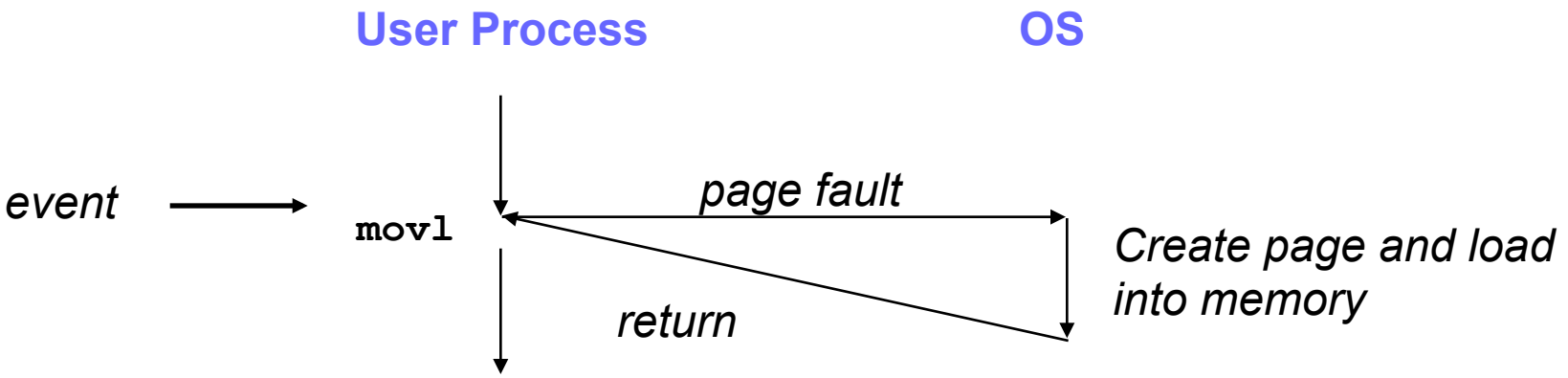
Fault example #1

- Memory reference
  - 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
```

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



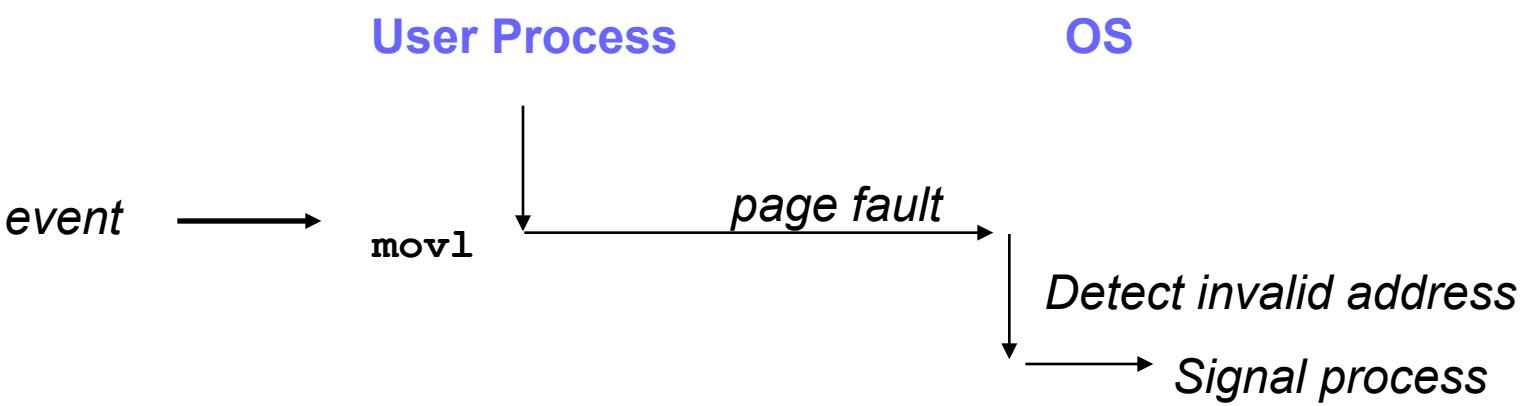
Fault example #2

- Memory reference
  - User writes to memory location
  - Address is not valid

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

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

  - Page handler detects invalid address
  - Sends SIGSEG signal to user process
  - User process exits with “segmentation fault”

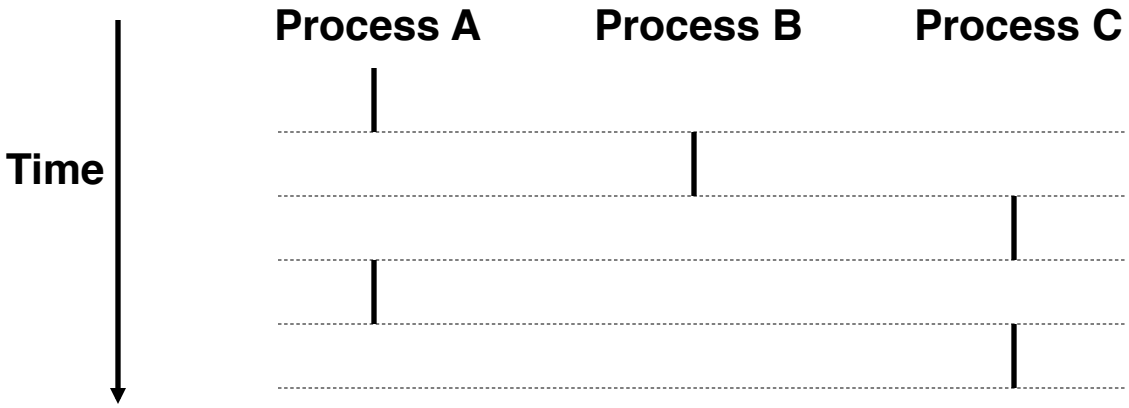


Processes

- 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”
  - Exceptions are key to implementing processes
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private address space
    - Each program seems to have exclusive use of main memory
- How are these Illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

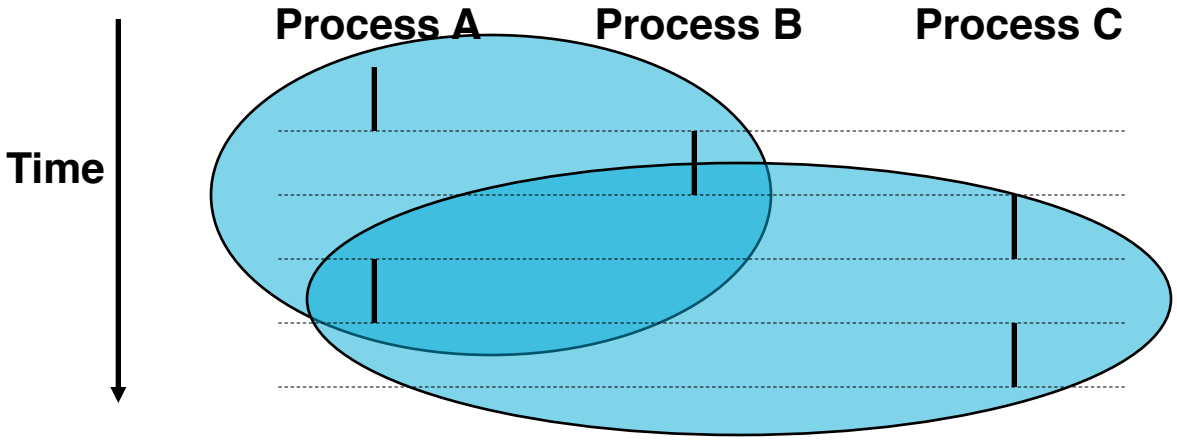
Logical control flows

Each process has its own logical control flow



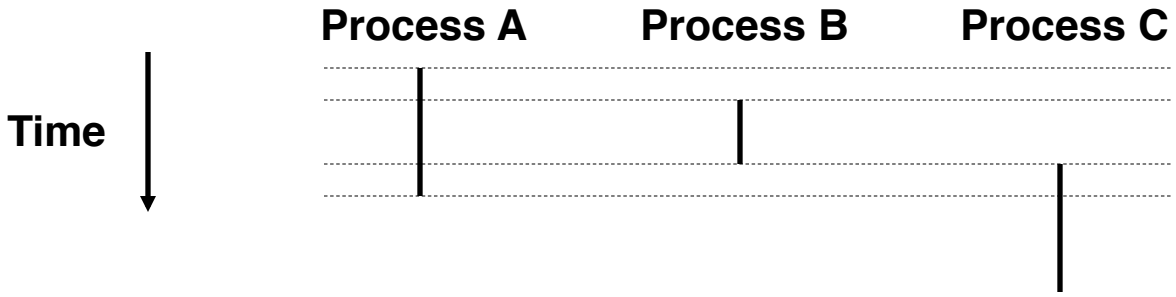
Concurrent processes

- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples:
  - 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 (this is concurrency)



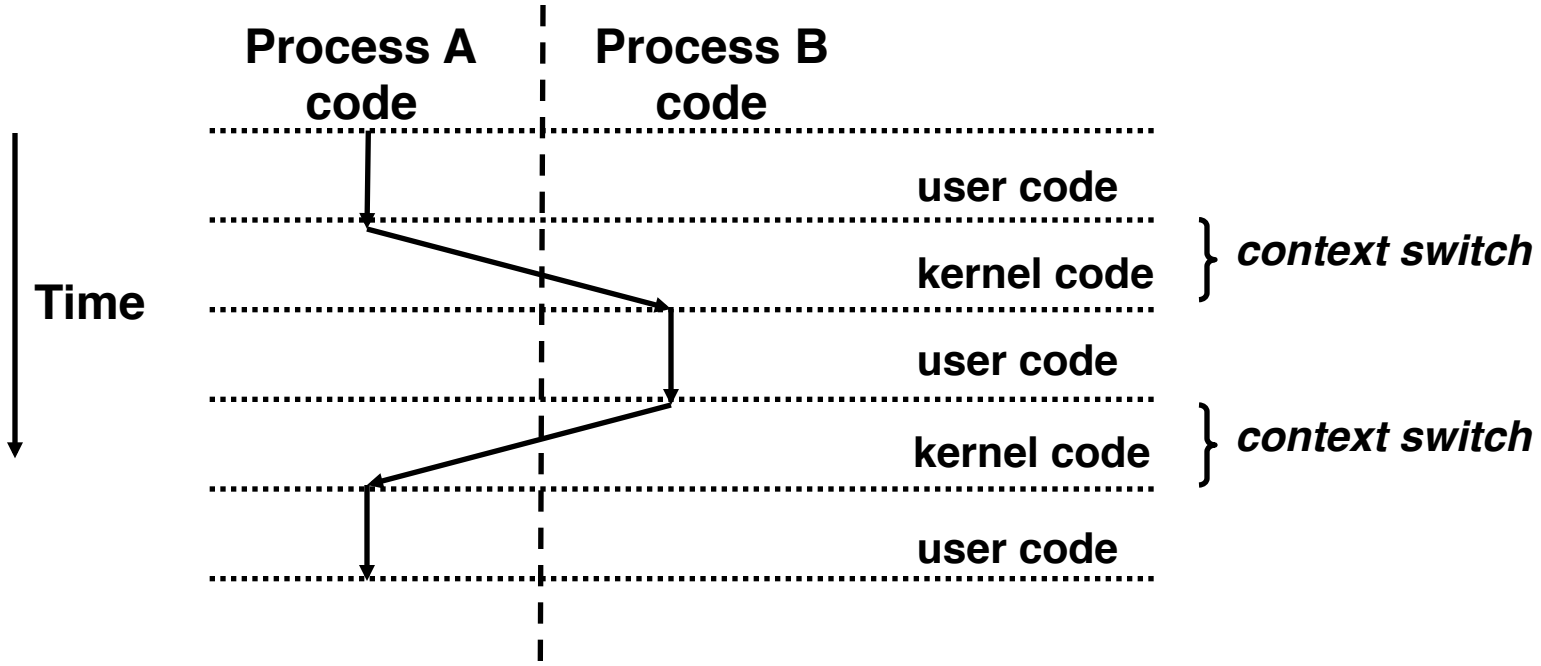
- Independently of how many processors/cores we have

Context switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Not a separate process, but runs as part of user process
- Kernel maintains a *context* for each process – state the kernel needs to restart a process
  - Value of general purpose and floating point registers
  - Program counter
  - User's stack
  - Status registers
  - Page table
  - ….

Context switching

- Control flow passes from one process to another via a *context switch* - a higher-level form of exceptional control flow
  - While the kernel is executing a system call
  - As a result of an interrupt (e.g. timer)

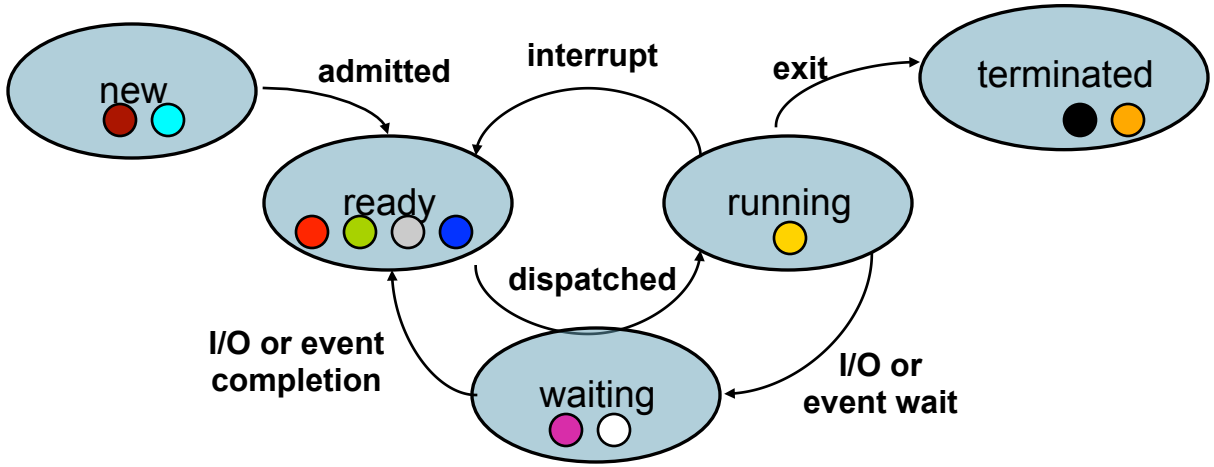


Process control

- Every processes has a unique nonzero process ID

```
pid_t getpid(void);  /* process PID */
pid_t getppid(void); /* parent PID */
```

- Through its life, it moves among various states (in Unix run `ps`)
  - New – being created
  - Ready – waiting to get the processor
  - Running – being executed (*how many at once?*)
  - Waiting – waiting for some event to occur
  - Terminated – finished executing



Process control

- A process terminates because
  - It received a signal whose default action is to terminate it
  - It returned from the main routine or
  - It called `exit`
    - `atexit()` registers functions to be executed upon exit

```
void exit(int status);
int atexit(void (*function)(void));
```

```
#include <stdlib.h>
#include <stdio.h>

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

int main()
{
   atexit(cleanup);
   printf("Making a mess and ...\n")
   exit(0);
}
```

```
unix$ ./atExit
Making a mess and ...
cleaning up
```

Process control

- A process (parent) can create another one (child) by calling `fork`

```
pid_t fork(void);
```

  - Child is nearly identical to parent process
    - User-level virtual address space, copies of open file descriptors
    - Different PID
  - Fork *returns twice*, for parent and child
    - Returns 0 to the child process
    - Returns child's `pid` to the parent process

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

   pid = fork();
   if (pid == 0) { /* child */
      ...
   }
   /* parent */
   ...
}
```

Fork example

- Some key points
  - Parent and child both run same code, concurrently
    - Distinguish parent from child by return value from `fork`
  - Start with same state, but each has private copy
    - Including shared output file descriptor
    - Relative ordering of their print statements undefined

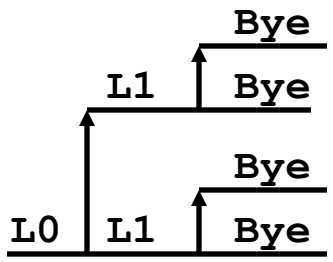
```
int main()
{
   pid_t pid;
   int x = 1;

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

```
usenix$ ./fork1
Parent x =  0
Child x = 2
```

Fork example #2

- Key points
  - Both parent and child can continue forking

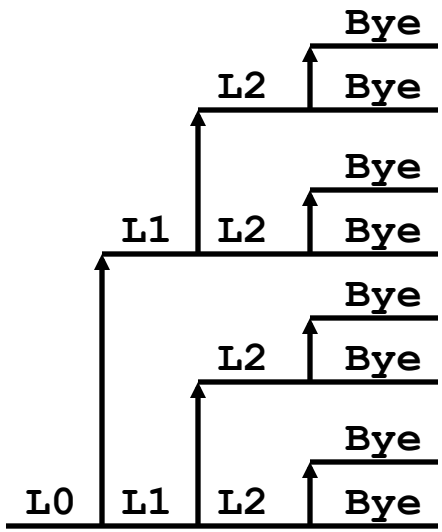
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork example #3

- Key points
  - Both parent and child can continue forking

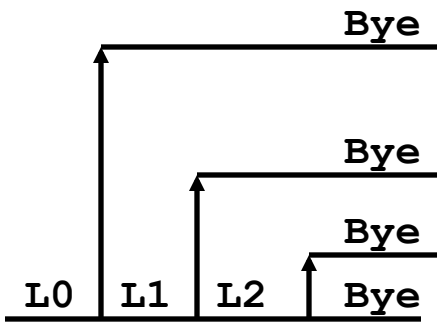
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork example #4

- Key points
  - Both parent and child can continue forking

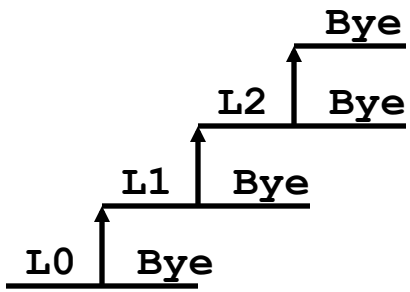
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Fork example #5

- Key points
  - Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



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

Zombie - Example

- `ps` shows child process as “defunct”
- Killing parent allows child to be reaped

```
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 */
    }
}
```

```
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
```

Nonterminating 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);
    }
}
```

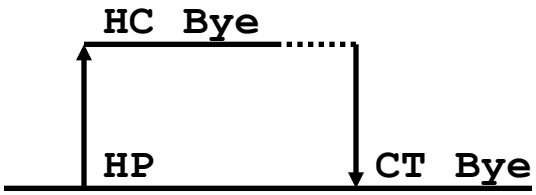
```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9   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

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

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

```
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();
}
```



wait Example

- If multiple children completed, pick in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    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 = 0; i < N; i++) { /*FIXME - whichever ends first*/
        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

- `waitpid(pid, &status, options)`
  - Can wait for specific process
  - Various options

```
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 = 0; i < N; 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 terminated abnormally\n", wpid);
    }
```

`wait/waitpid` example outputs

Using `wait` (`fork10`)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using `waitpid` (`fork11`)

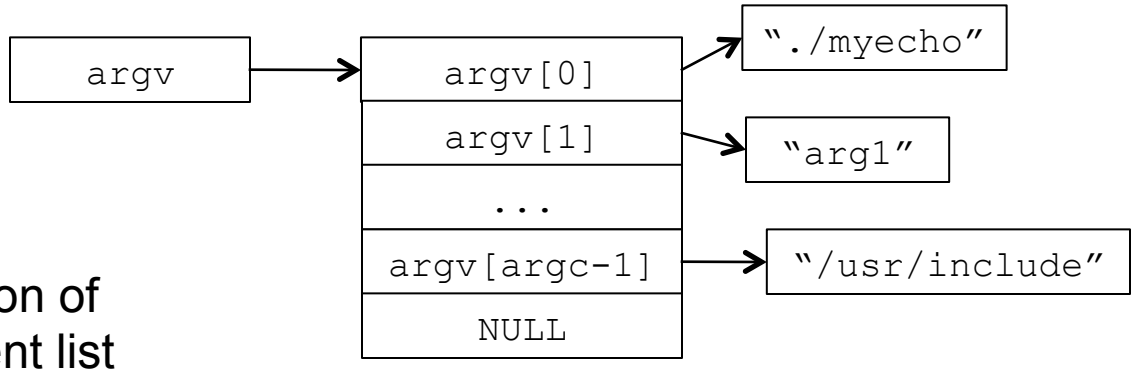
```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

Running new programs

- Load & run a new program in current process context

```
#include <unistd.h>

int execve(const char *filename, const char *argv[],
           const char *envp[]);
```

  - Runs executable object file `filename`
  - With argument list `argv`
  - And environment variable list `envp`



Organization of an argument list

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
  - Only one can execute at a time, though
  - Each process appears to have total control of processor + private memory space
- Spawning processes
  - Call to `fork`: one call, two returns
- Terminating processes
  - Call `exit`: one call, no return
- Reaping processes
  - Call `wait` or `waitpid`
- Replacing program executed by process
  - Call `execl` (or variant): one call, (normally) no return