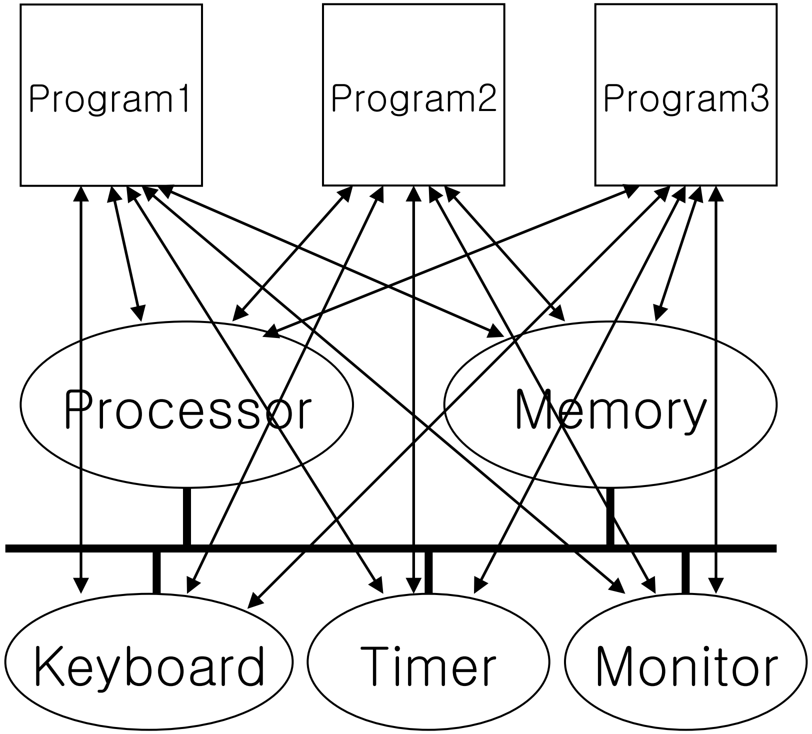
Multitasking

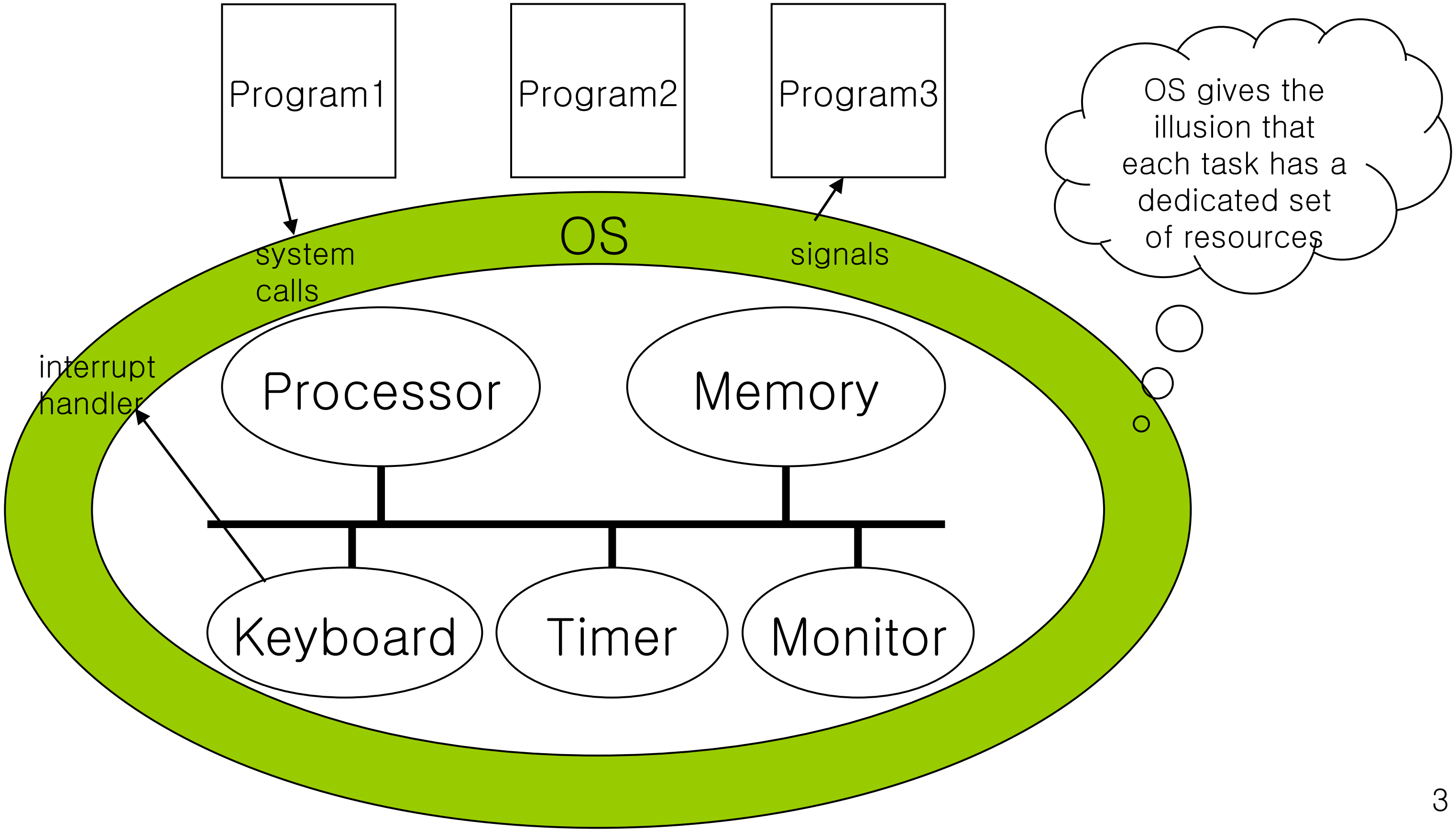
Until now....

- We understand how a program runs on a computer system.
- In reality, multiple programs (tasks) run concurrently on multiple resources.

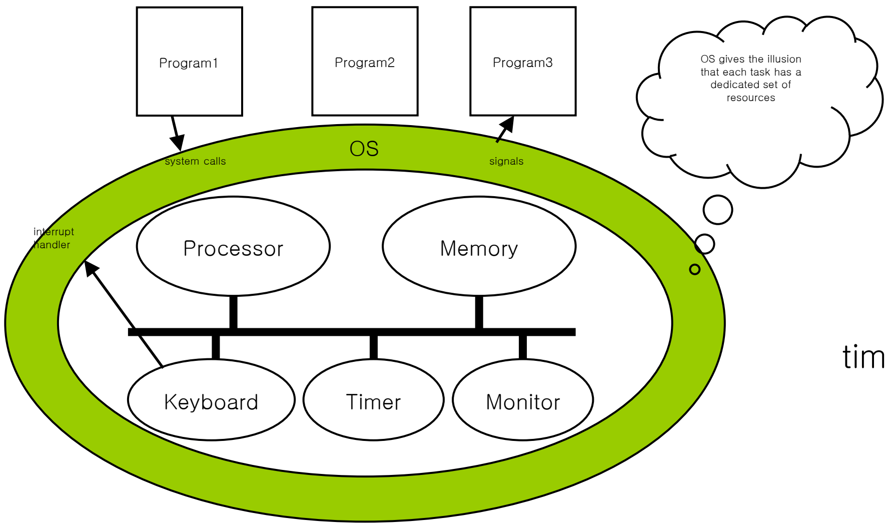


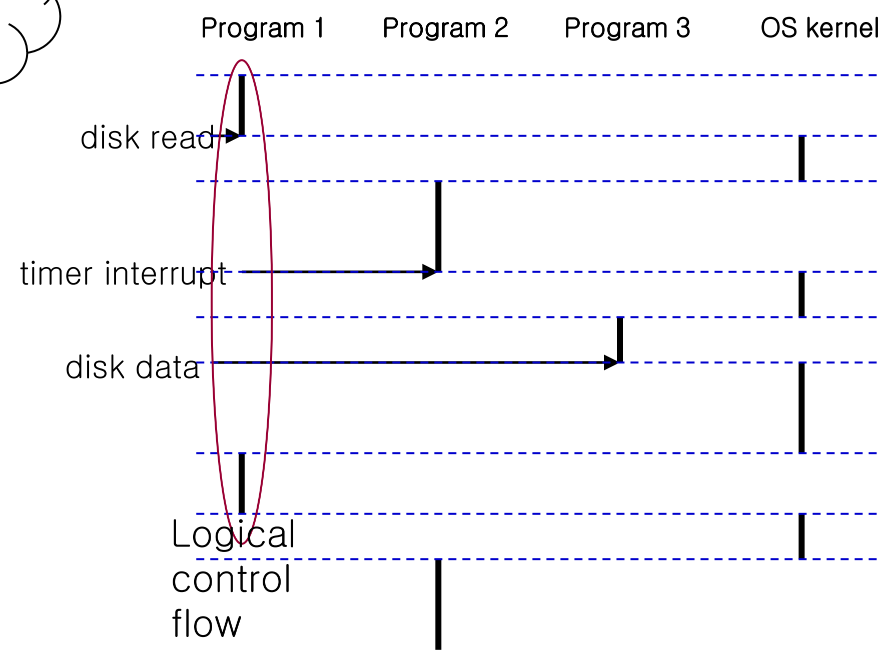
Until now....

- We understand how a program runs on a computer system.
- In reality, multiple programs (tasks) run concurrently on multiple resources.



Interleaving in time





- Control abruptly changes by events (not by normal jumps and calls): <u>Exceptional Control Flow</u>

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*).

Physical control flow

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

How the Control Flow Changes

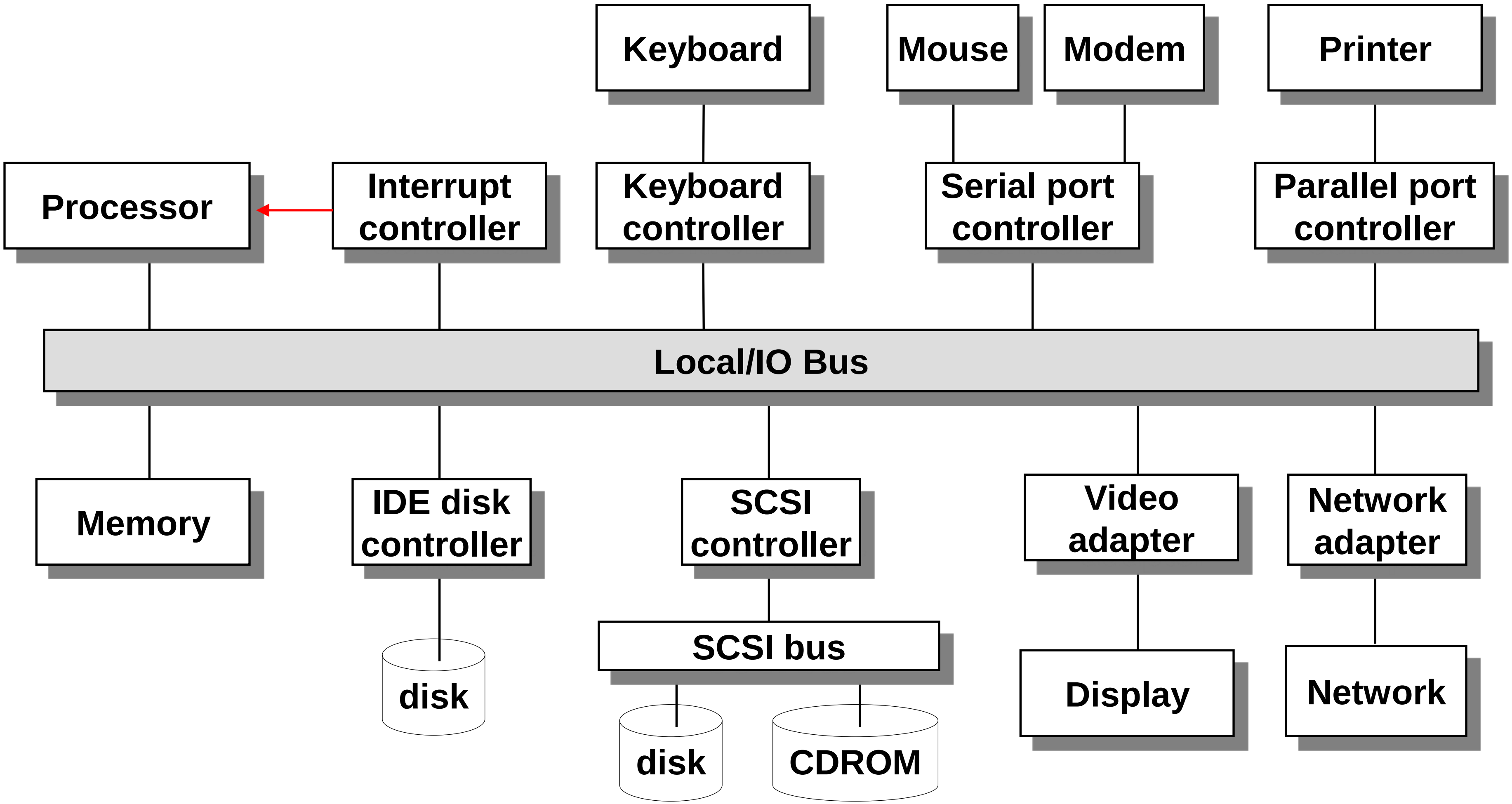
- Up to Now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return using the stack discipline.
  - Both react to changes in program state.
- Insufficient for a useful system
  - Difficult for the CPU to react to changes in system state.
    - data arrives from a disk or a network adapter.
    - Instruction divides by zero
    - User hits ctl-c at the keyboard
    - System timer expires
- System needs mechanisms for "exceptional control flow"
- Supporting "Exceptional control flow" is the basic mechanism with which OS serve multiple concurrent tasks controlling multiple resources.

Exceptional Control Flow

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

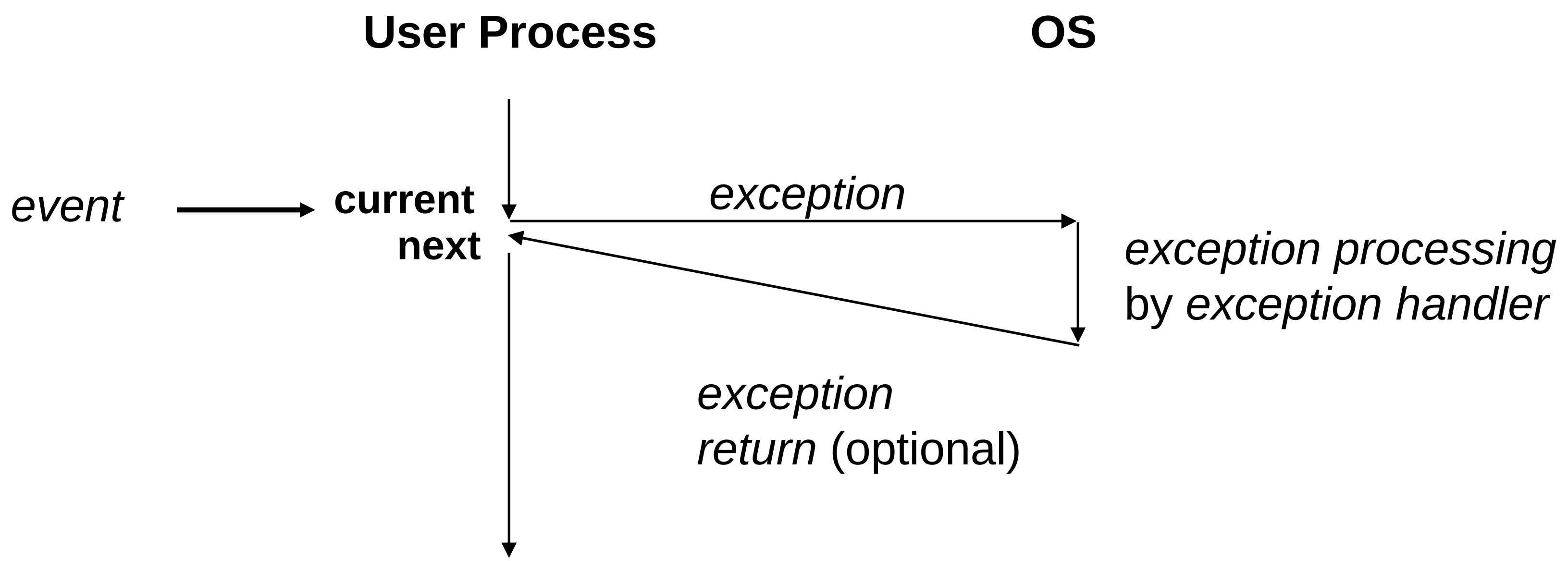
- Low level Mechanism
  - exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher Level Mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps (setjmp/longjmp)
  - Implemented by either:
    - OS software (context switch and signals).
    - C language runtime library: nonlocal jumps.

System context for exceptions

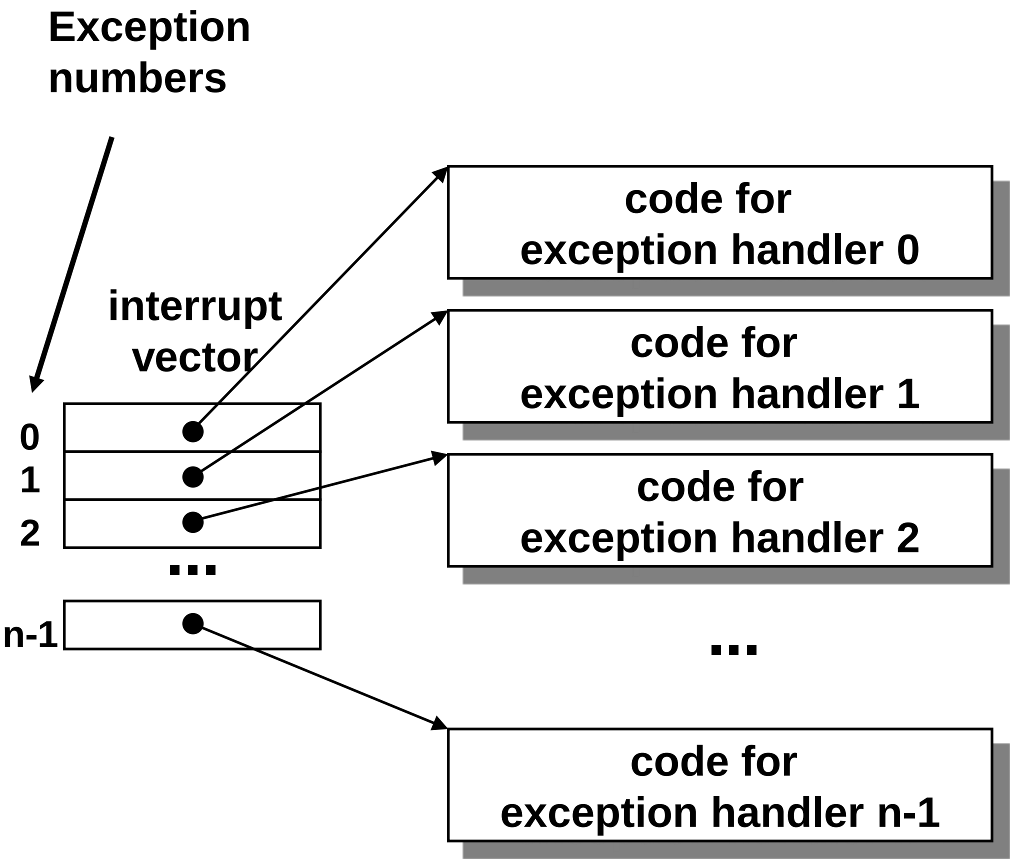


Exceptions

- An *exception* is a transfer of control to the OS in response to some *event* (i.e., Page Fault, Timer expires)



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.

Exception Types

- Asynchronous Exceptions (Interrupts)
- Synchronous Exceptions
  - trap (e.g., system call)
  - fault (e.g., page fault)
  - abort (e.g., parity error)

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

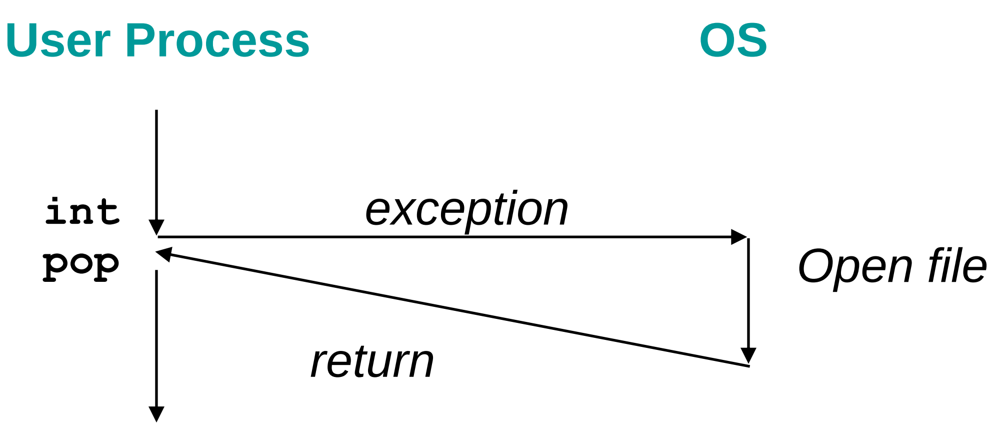
- Caused by events that occur as a result of executing an instruction:
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), 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



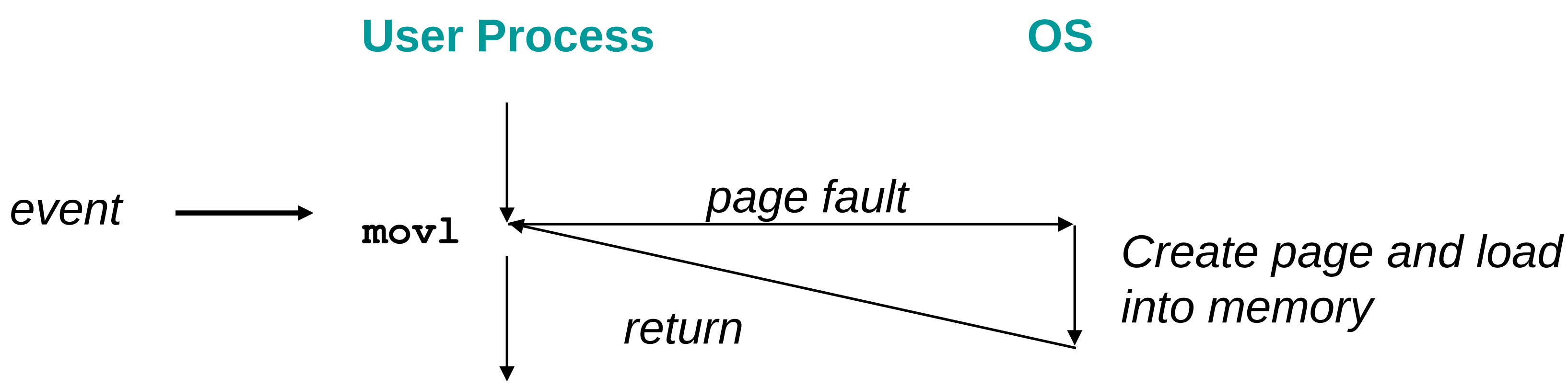
Fault Example #1

```
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 (This gives the task the illusion of exclusive use of memory)
  - Returns to faulting instruction
  - Successful on second try



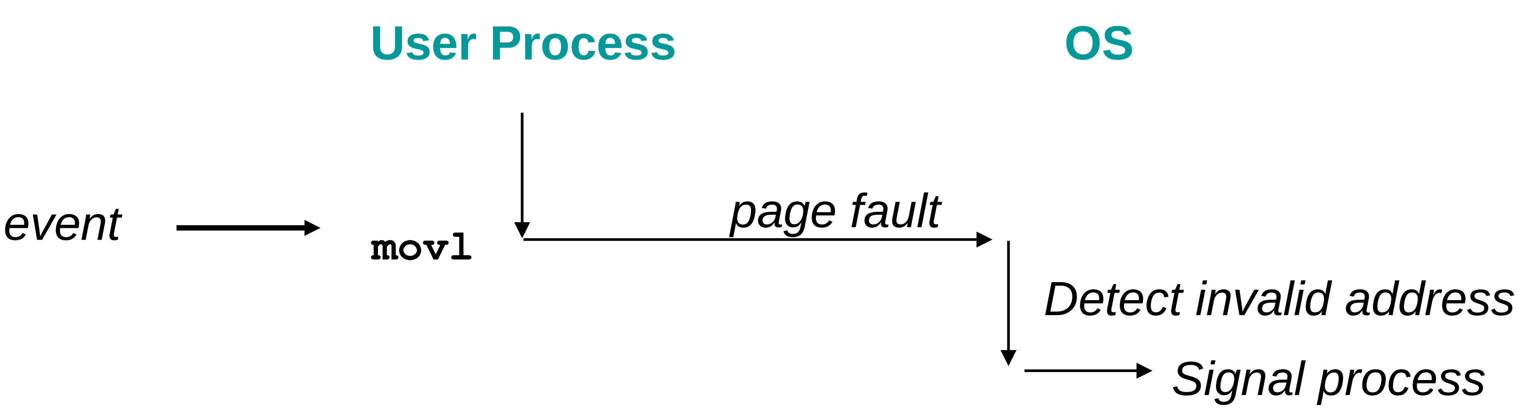
Fault Example #2

```
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 `SIGSEG` signal to user process
  - User process exits with “segmentation fault”

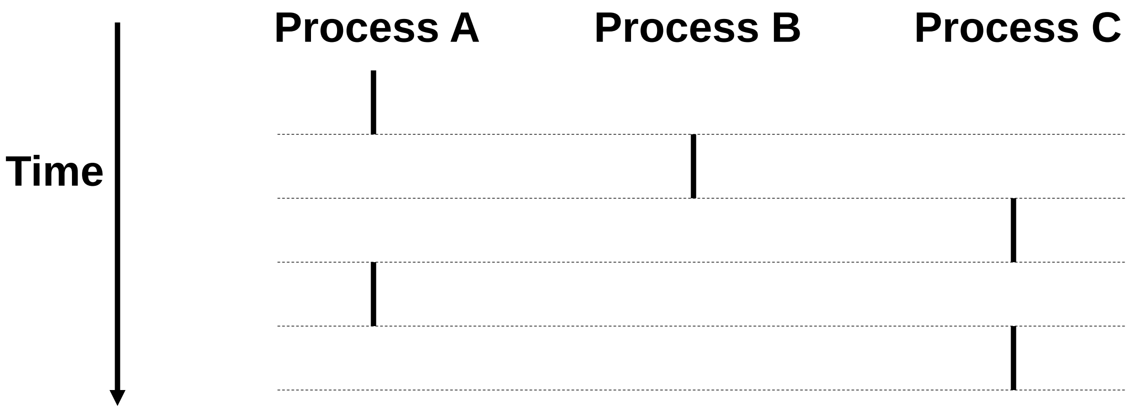


Multitasking with the Concept of Processes

- 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”
- 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

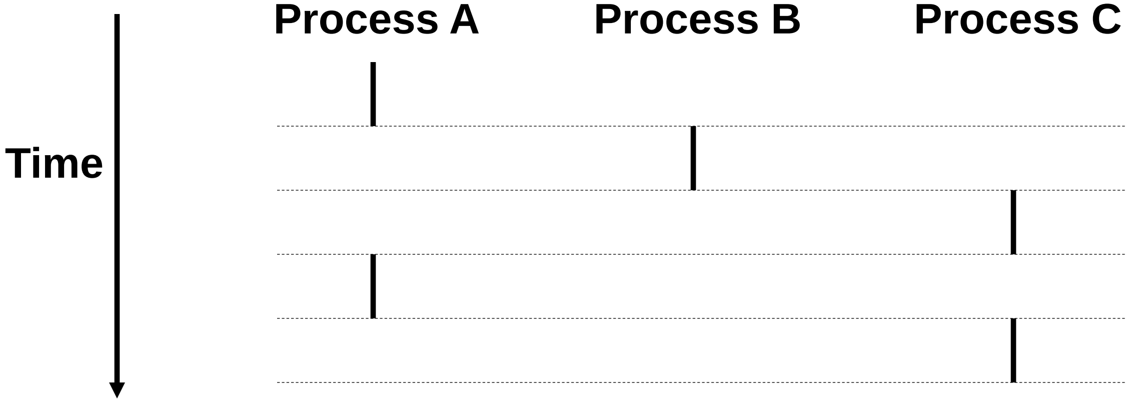
Logical Control Flows

Each process has its own logical control flow



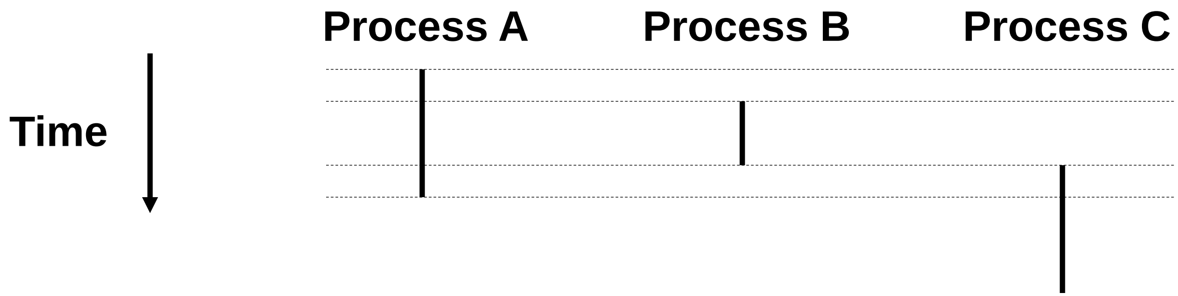
Concurrent Processes

- 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



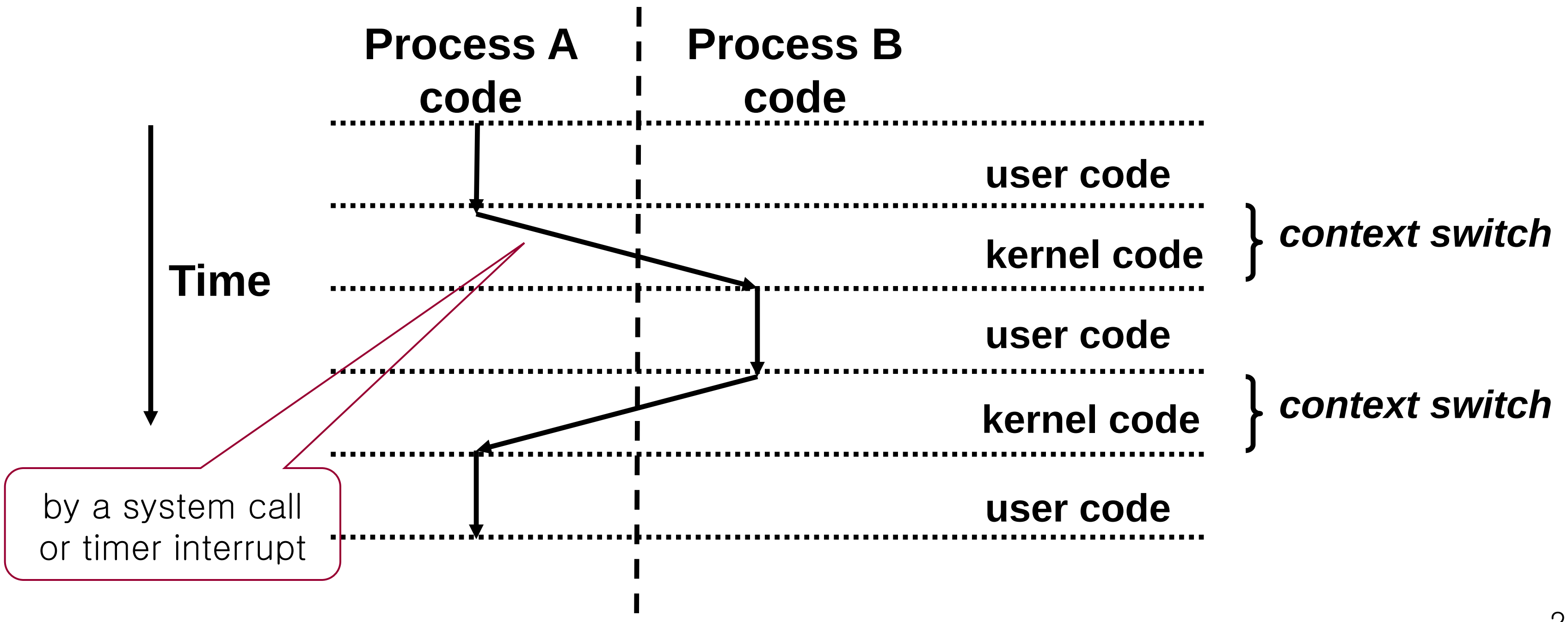
User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time. (Because CPU can run only a single instruction at a time)
- However, we can think of concurrent processes are running in parallel with each other.



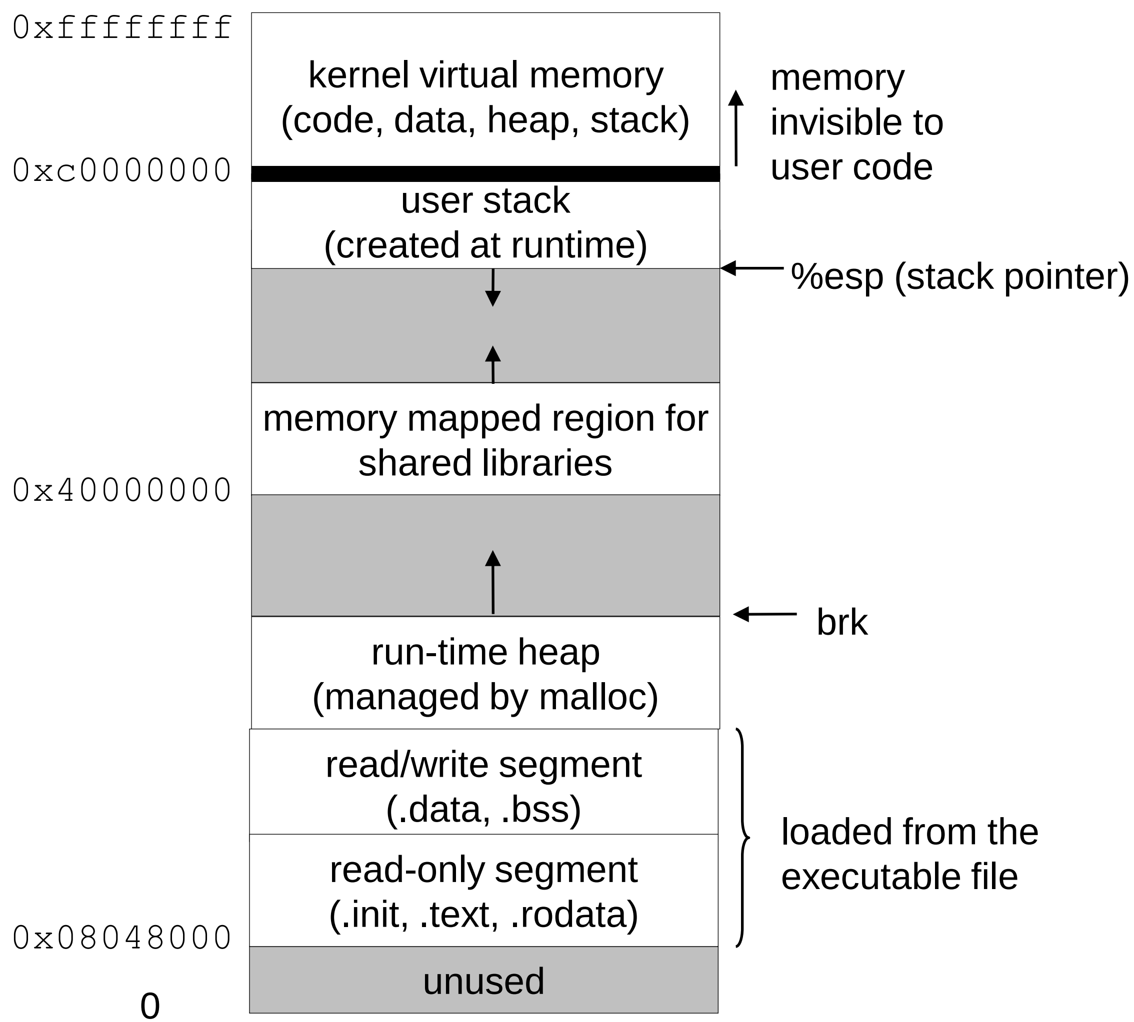
Context Switching

- 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 user process
- Control flow passes from one process to another via a *context switch.*



Private Address Spaces

- Each process has its own private address space.



Process Related System Calls

- Now, we understand how multiple processes run concurrently
- How multiple processes can be created?
- How existing processes can be removed from the system?
- OS provides system calls to do this
  - fork
  - exit

`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*

Kernel Data Structure for Processes

- Process Table (Array of PCB)
  - Save information of active processes
  - Display process information using a PCB(Process Control Block) as a table entry

- Process Control Block (PCB)
  - Save all information related to process
  - Process and kernel thread have independent PCB.

PCB for each process

PCB

Process state Exit code Process ID, Parent process ID	Process classification and state information
Scheduling priority Time slice Resister set	Scheduling information
User ID, Group ID	User and group information of executable file
Virtual address space (text, data, heap, stack etc.) Physical memory mapping (page table)	Virtual address space and physical memory mapping information
Open file table File system information	Open file and file system information
Signal handling Terminal	Signal handling and terminal information

Fork Example #1

- Key Points
  - Parent and child both run same code
    - Distinguish parent from child by return value from `fork`
  - Start with same state (e.g., stack, registers, program counter, environment variables, and open file descriptors)
  - But, each has private copy and thus can evolve separately

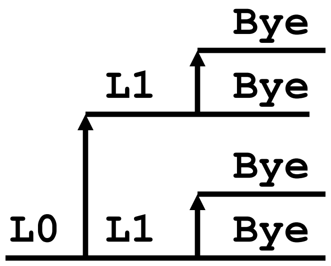
```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

Relative ordering of their print statements undefined

Fork Example #2

- Key Points
  - Both parent and child can continue forking

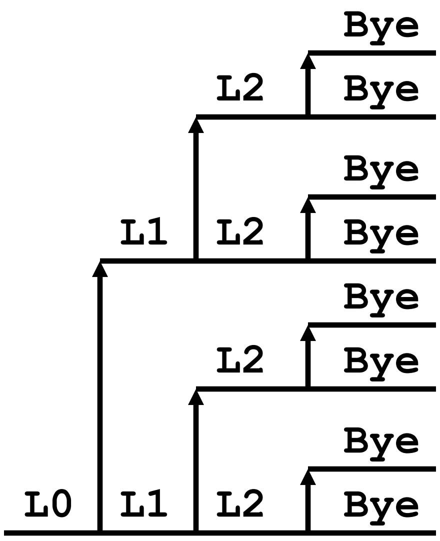
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

- Key Points
  - Both parent and child can continue forking

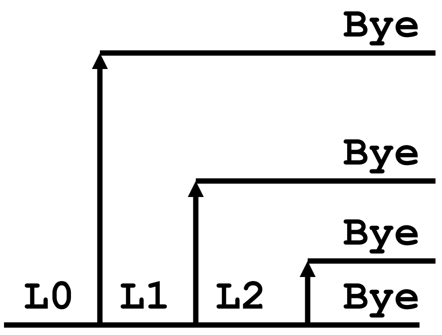
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

- Key Points
  - Both parent and child can continue forking

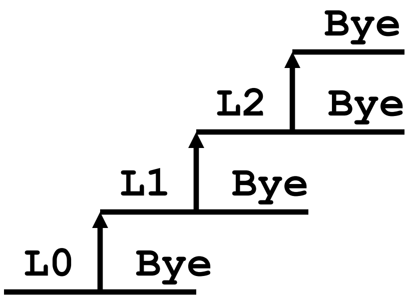
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



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");
}
```



`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

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 */
    }
}
```

```
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”
- Killing parent allows child to be reaped

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

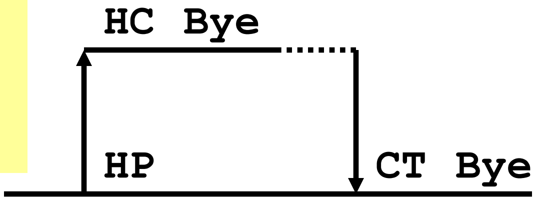
`wait`: Synchronizing with children and Reaping zombies

- `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

`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();
}
```



Wait Example

- If multiple children completed, will take 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++) {
       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
```

`exec`: Running new programs

- `int execl(char *path, char *arg0, char *arg1, …, 0)`
  - 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 *)0` argument
  - returns `-1` if error, otherwise doesn’t return!

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

The World of Multitasking

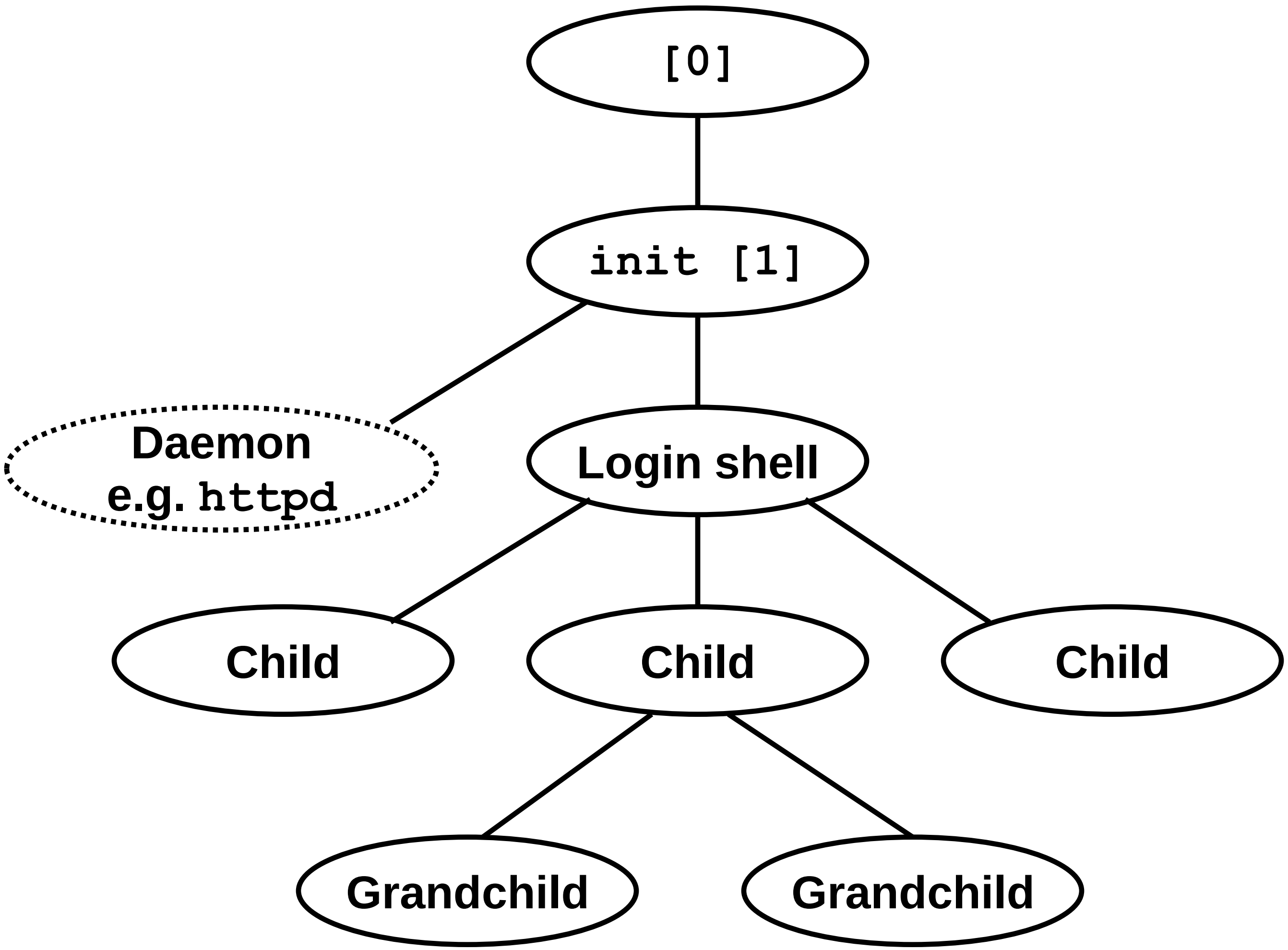
- System Runs Many Processes Concurrently
  - Process: executing program
    - State consists of memory image + register values + program counter
  - Continually switches from one process to another
    - Suspend process when it needs I/O resource or timer event occurs
    - Resume process when I/O available or given scheduling priority
  - Appears to user(s) as if all processes executing simultaneously
    - Even though most systems can only execute one process at a time
    - Except possibly with lower performance than if running alone

Programmer's Model of Multitasking

- Basic Functions
  - `fork()` spawns new process
    - Called once, returns twice
  - `exit()` terminates own process
    - Called once, never returns
    - Puts it into "zombie" status
  - `wait()` and `waitpid()` wait for and reap terminated children
  - `execl()` and `execve()` run a new program in an existing process
    •can give env variables
    - Called once, (normally) never returns
- Programming Challenge
  - Understanding the nonstandard semantics of the functions
  - Avoiding improper use of system resources
    - E.g. "Fork bombs" can disable a system.

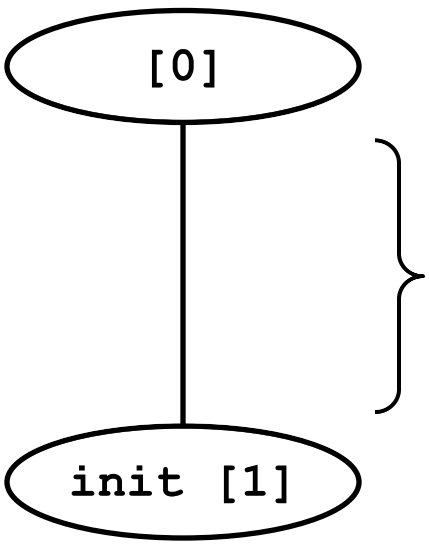
Unix Process Hierarchy

- Now, we are ready to understand how UNIX starts up and run many user application programs



Unix Startup: Step 1

1. **Pushing reset button loads the `PC` with the address of a small bootstrap program.**
2. **Bootstrap program loads the boot block (disk block 0).**
3. **Boot block program loads kernel binary (e.g., `/boot/vmlinux`)**
4. **Boot block program passes control to kernel.**
5. **Kernel handcrafts the data structures for process 0.**

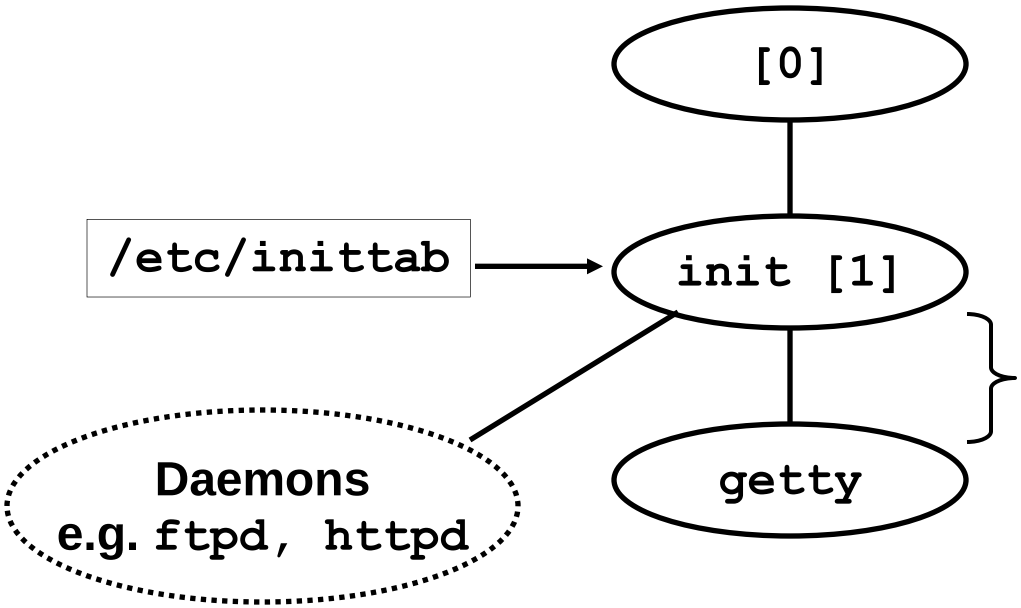


Process 0: handcrafted kernel process

Process 0 forks child process 1

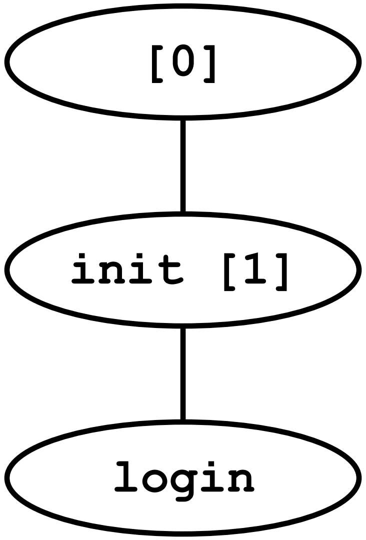
Child process 1 execs `/sbin/init`

Unix Startup: Step 2



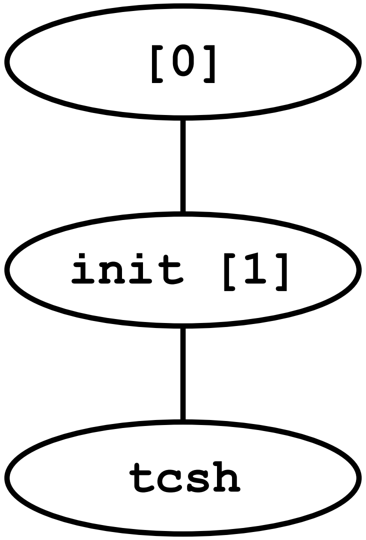
`init` **forks and execs daemons per** `/etc/inittab`**, and forks and execs a** `getty` **program for the console**

Unix Startup: Step 3



The `getty` process execs a `login` program

Unix Startup: Step 4



`login` **reads login and passwd.**
if OK, it execs a *shell.*
if not OK, it execs another `getty`

Shell Programs

- A *shell* is an application program that runs programs on behalf of the user.
  - `sh` – Original Unix Bourne Shell
  - `csh` – BSD Unix C Shell, `tcsh` – Enhanced C Shell
  - `bash` –Bourne-Again Shell

```
int main()
{
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

- Execution is a sequence of read/evaluate steps

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) {   /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else         /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

Summarizing

- Exceptions are the basic for multitasking
  - 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
- Programmer's perspective
  - fork(): creating a process (one call, two returns)
  - exit(): terminating a process (one call, no return)
  - wait(), waitpid(): reaping a zombie
  - execl(), execve(): replacing the program (one call, no return)
- UNIX start-up sequence until shell runs
- Shell forks processes and run user programs