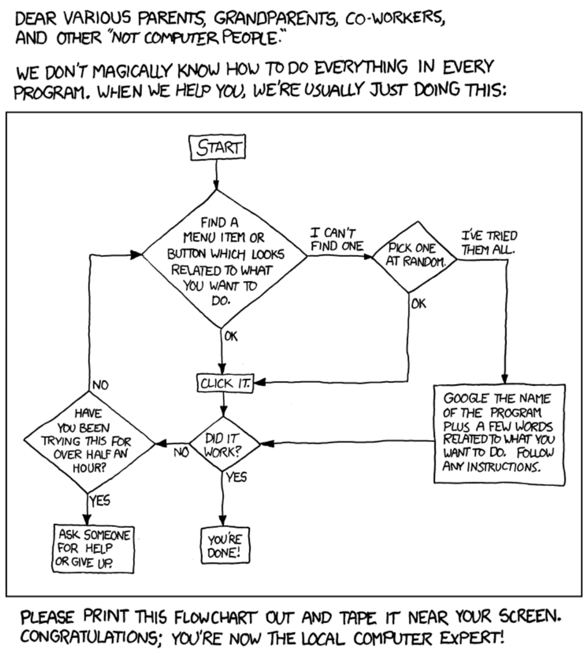
Processes

CSE 351 Winter 2017

hi!! :)



https://xkcd.com/627/

Administrivia

- Lab 4 released!

∅

∅ + cache size

cache empty

Addr access	H/M
0	M
1	H
2	H
3	H
4	Miss

Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - **Asynchronous exceptions (interrupts)**
  - **Synchronous exceptions (traps & faults)**

Asynchronous Exceptions (Interrupts)

IRQ CPU

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to "next" instruction
- <u>Examples</u>:
  - I/O interrupts
    - Hitting Ctrl-C on the keyboard
    - Clicking a mouse button or tapping a touchscreen
    - Arrival of a packet from a network
    - Arrival of data from a disk
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - **Intentional**: transfer control to OS to perform some function
    - Examples: *system calls*, breakpoint traps, special instructions
    - Returns control to "next" instruction ("current" instr did what it was supposed to)
  - ***Faults***
    - **Unintentional** but possibly recoverable
    - Examples: *page faults*, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting ("current") instruction or aborts
      ↑ if recoverable ↑ if not recoverable
  - ***Aborts***
    - **Unintentional** and unrecoverable
    - Examples: parity error, machine check (hardware failure detected)
    - Aborts current program

System Calls

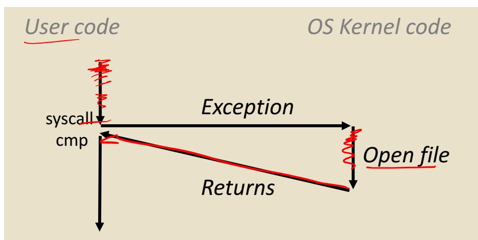
- Each system call has a unique ID number
- Examples for Linux on x86-64:

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

Traps 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 value is an error corresponding to negative `errno`

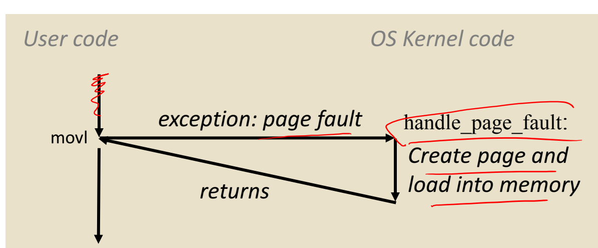
UNIVERSITY of WASHINGTON L19: Processes CSE351, Winter 2017

Fault Example: Page Fault

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

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

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

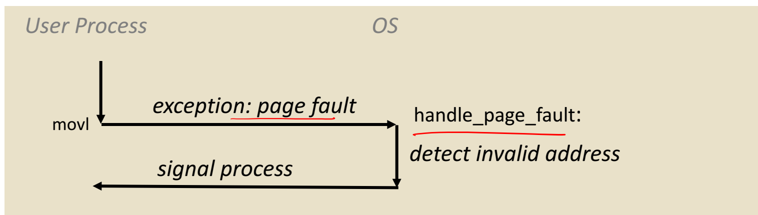


- Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try

Fault Example: Invalid Memory Reference

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

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



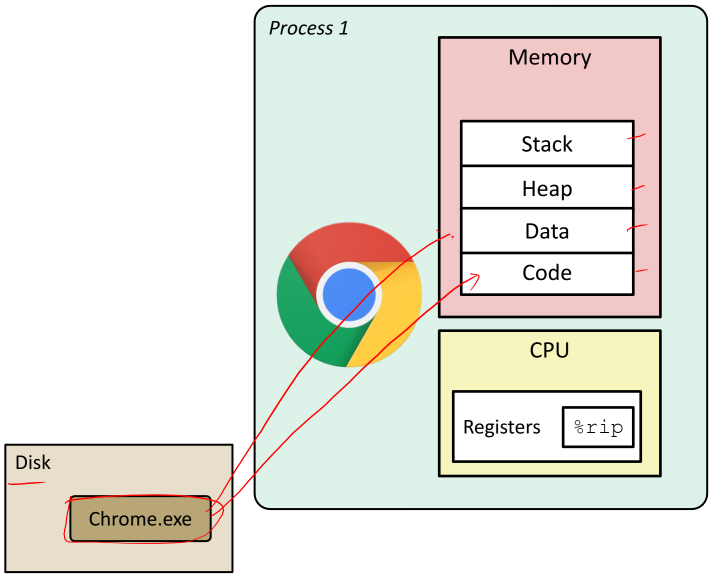
- Page fault handler detects invalid address
- Sends `SIGSEGV` signal to user process
- User process exits with "segmentation fault"

Processes

- **Processes and context switching**
- Creating new processes
  - `fork()` and `wait()`
- Zombies

What is a process?

It's an *illusion*!

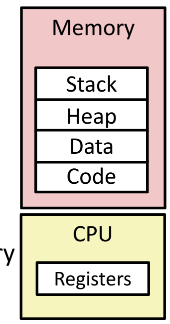


What is a process?

- Another *abstraction* in our computer system
  - Provided by the OS
  - OS uses a data structure to represent each process
  - Maintains the ***interface*** between the program and the underlying hardware (CPU + memory)
- What do *processes* have to do with *exceptional control flow*?
  - Exceptional control flow is the *mechanism* the OS uses to enable **multiple processes** to run on the same system
- What is the difference between:
  - A processor? A program? A process?

Processes

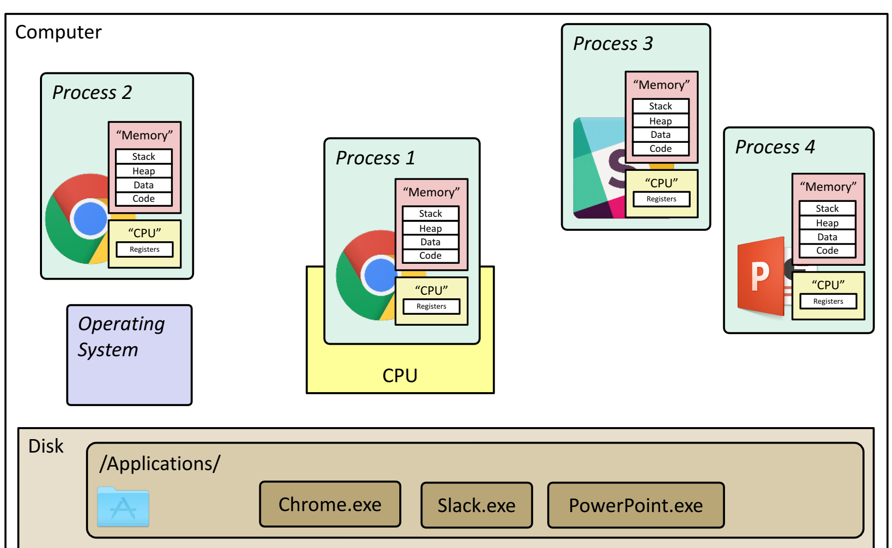
- A ***process*** is an instance of a running program
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"
- Process provides each program with two key abstractions:
  - *Logical control flow*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called ***context switching***
  - *Private address space*
    - Each program seems to have exclusive use of main memory
    - Provided by kernel mechanism called ***virtual memory*** addresses



13

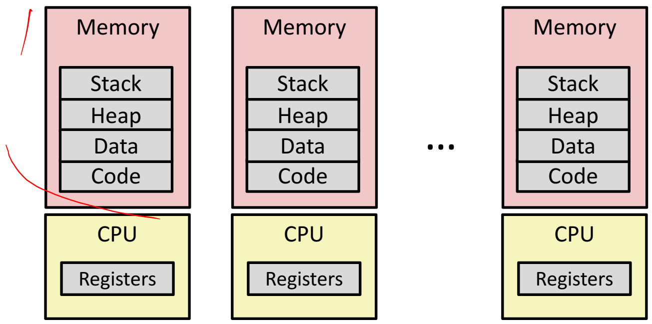
What is a process?

It's an *illusion*!



14

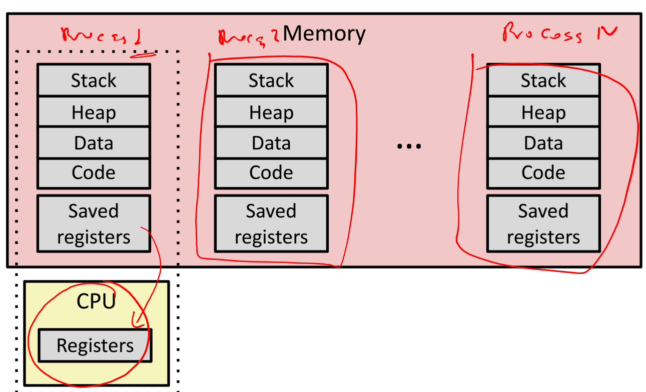
Multiprocessing: The Illusion



- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

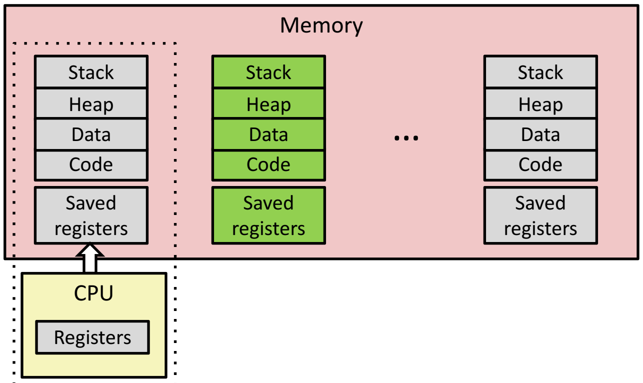
UNIVERSITY of WASHINGTON L19: Processes CSE351, Winter 2017

Multiprocessing: The Reality



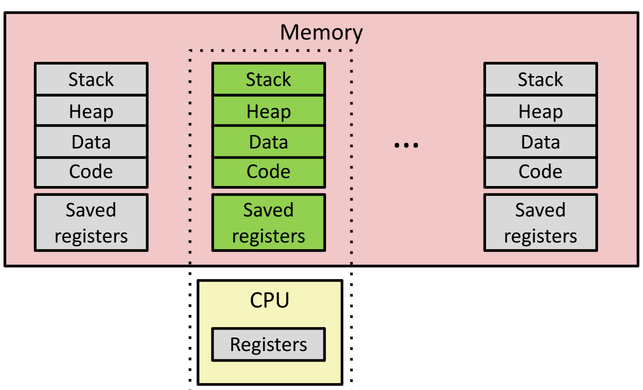
- Single processor executes multiple processes *concurrently*
  - Process executions interleaved, CPU runs *one at a time*
  - Address spaces managed by virtual memory system (later in course)
  - *Execution context* (register values, stack, …) for other processes saved in memory

Multiprocessing



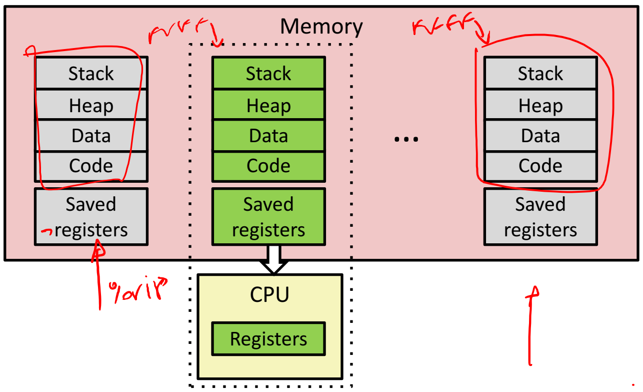
- Context switch
  1) **Save current registers in memory**

Multiprocessing



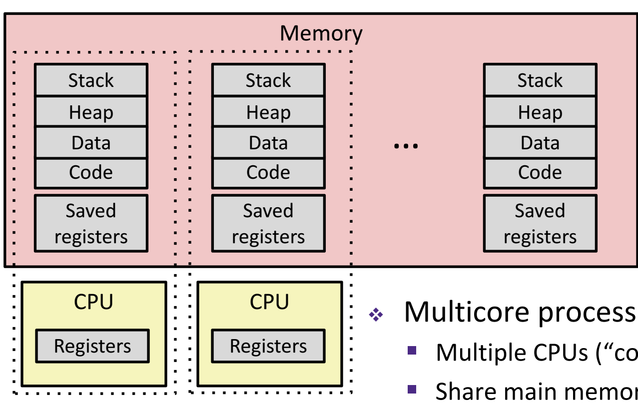
- Context switch
  1) Save current registers in memory
  2) **Schedule next process for execution**

Multiprocessing



- Context switch
  1) Save current registers in memory
  2) Schedule next process for execution
  3) **Load saved registers and switch address space**

Multiprocessing: The (Modern) Reality

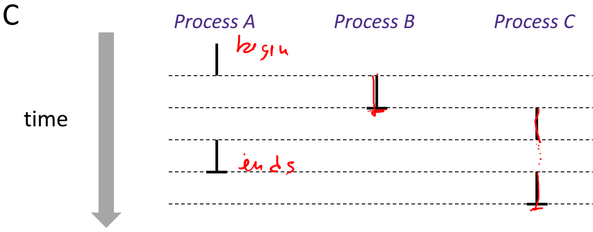


- Multicore processors
  - Multiple CPUs ("cores") on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Kernel schedules processes to cores
    - ***Still* constantly swapping processes**

Concurrent Processes

Assume only one CPU

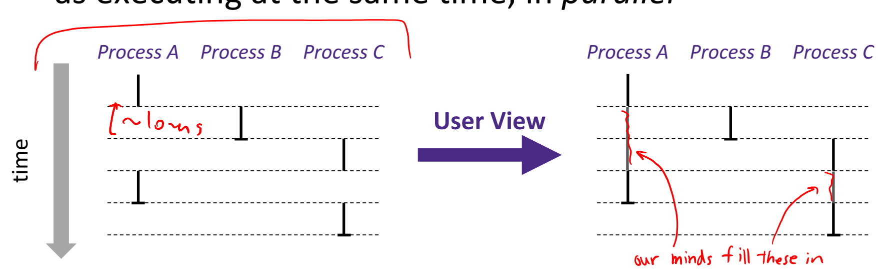
- Each process is a logical control flow
- Two processes *run concurrently* (are concurrent) if their instruction executions (flows) overlap in time
  - Otherwise, they are *sequential*
- Example: (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C



User's View of Concurrency

Assume only one CPU

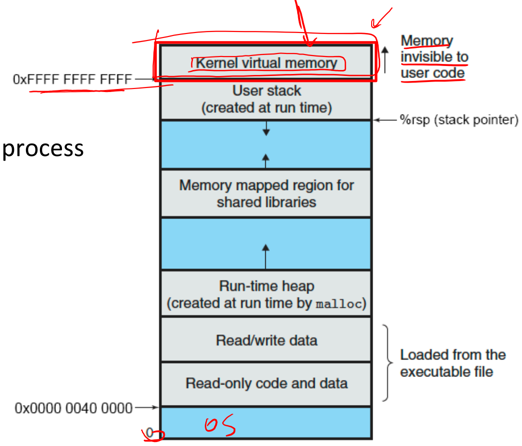
- Control flows for concurrent processes are physically disjoint in time
  - CPU only executes instructions for one process at a time
- However, the user can *think of* concurrent processes as executing at the same time, in *parallel*



Context Switching

Assume only one CPU

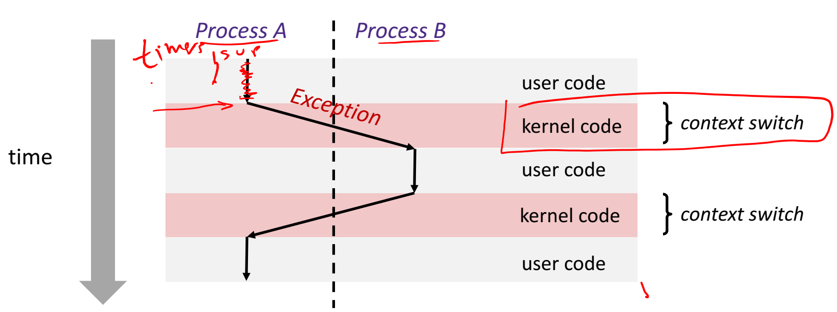
- Processes are managed by a *shared* chunk of OS code called the kernel
  - The kernel is not a separate process, but rather runs as part of a user process
- In x86-64 Linux:
  - Same address in each process refers to same shared memory location



Context Switching

Assume only one CPU

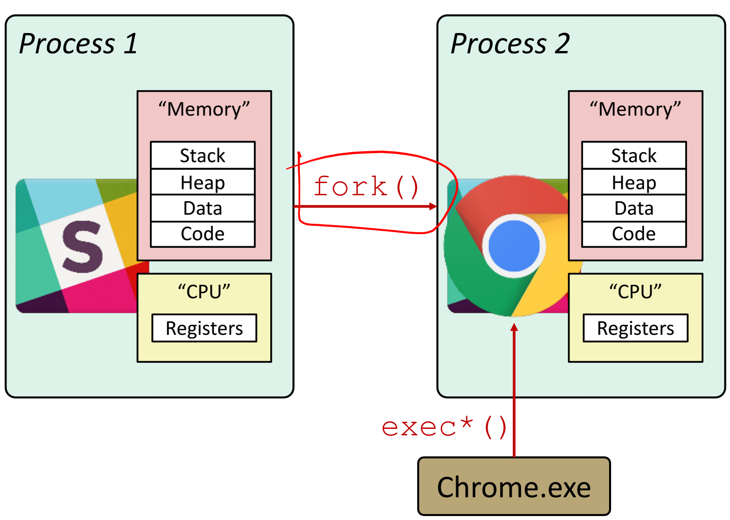
- Processes are managed by a *shared* chunk of OS code called the kernel
  - The kernel is not a separate process, but rather runs as part of a user process
- Context switch passes control flow from one process to another and is performed using kernel code



Processes

- Processes and context switching
- **Creating new processes**
  - **`fork()` and `wait()`**
- Zombies

Creating New Processes & Programs



Creating New Processes & Programs

- fork-exec model (Linux):
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process' code and address space with the code for a different program
    - Family: `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`
  - `fork()` and `execve()` are *system calls*
- Other system calls for process management:
  - `getpid()`
  - `exit()`
  - `wait()`, `waitpid()`

`fork`: Creating New Processes

- `pid_t fork(void)`
  - Creates a new "child" process that is *identical* to the calling "parent" process, including all state (memory, registers, etc.)
  - Returns 0 to the child process
  - Returns child's process ID (PID) to the parent process
- Child is *almost* identical to parent:
  - Child gets an identical (but separate) copy of the parent's virtual address space
  - Child has a different PID than the parent

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

- `fork` is unique (and often confusing) because it is called once but returns "twice"

Understanding fork

Process X (parent)

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

Process Y (child)

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


Understanding fork

Process X (parent)

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

Process Y (child)

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

pid = Y

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

pid = 0

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

Understanding fork

Copy

Process X (parent)

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

Process Y (child)

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

pid = Y

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

pid = 0

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

`hello from parent`

`hello from child`

Which one appears first?

Fork Example

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

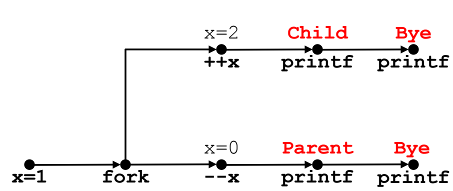
- Both processes continue/start execution after `fork`
  - Child starts at instruction after the call to `fork` (storing into `pid`)
- Can't predict execution order of parent and child
- Both processes start with `x=1`
  - Subsequent changes to `x` are independent
- Shared open files: stdout is the same in both parent and child

Modeling `fork` with Process Graphs

- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program
  - Each vertex is the execution of a statement
  - `a` → `b` means `a` happens before `b`
  - Edges can be labeled with current value of variables
  - `printf` vertices can be labeled with output
  - Each graph begins with a vertex with no inedges
- Any *topological sort* of the graph corresponds to a feasible total ordering
  - Total ordering of vertices where all edges point from left to right

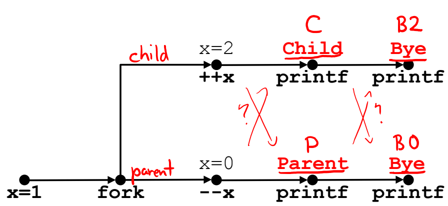
Fork Example: Possible Output

```
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);
}
```



Fork Example: Possible Output

```
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 possibilities:

C	P	P
B2	B0	C
P	C	B0
B0	B2	B2

etc...

as long as C comes before B2
and P comes before B0.

Fork-Exec

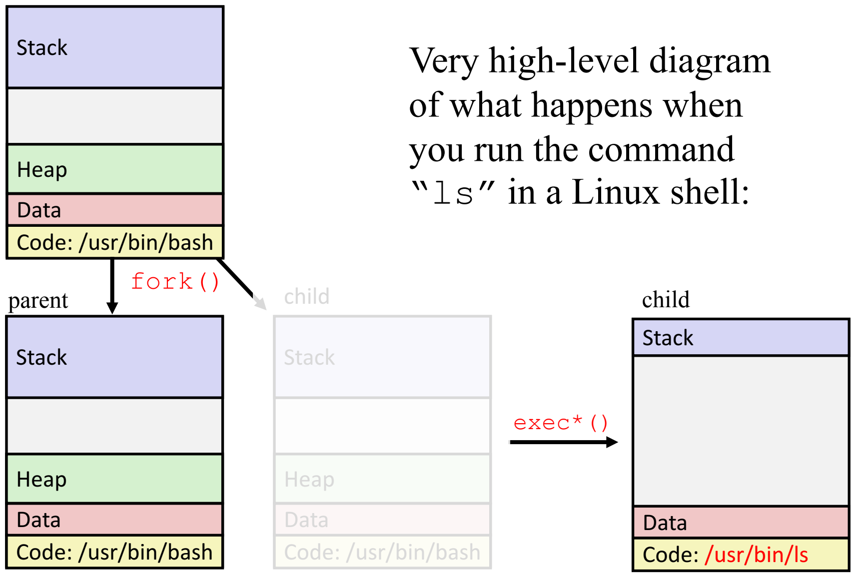
Note: the return values of `fork` and `exec*` should be checked for errors

- fork-exec model:
  - `fork()` creates a copy of the current process
  - `exec*()` replaces the current process' code and address space with the code for a different program
    - Whole family of `exec` calls – see **`exec(3)`** and **`execve(2)`**

```
// Example arguments: path="/usr/bin/ls",
//      argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[]) {
   pid_t pid = fork();
   if (pid != 0) {
      printf("Parent: created a child %d\n", pid);
   } else {
      printf("Child: about to exec a new program\n");
      execv(path, argv);
   }
   printf("This line printed by parent only!\n");
}
```

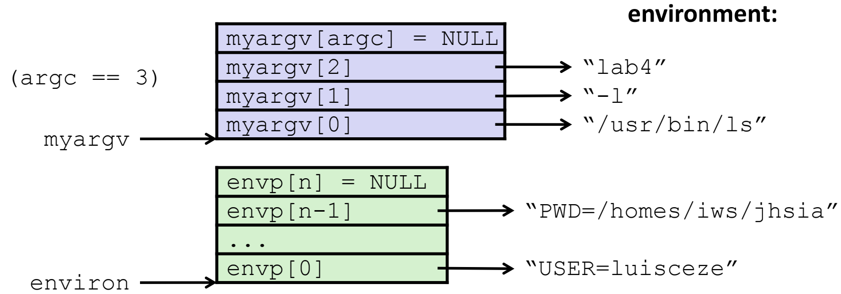
Exec-ing a new program

Very high-level diagram of what happens when you run the command "`ls`" in a Linux shell:



execve Example

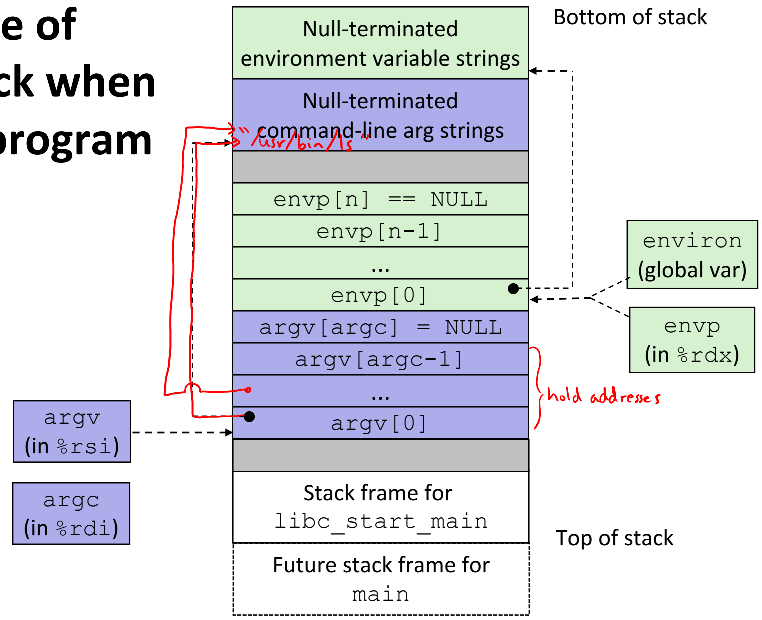
Execute "`/usr/bin/ls –l lab4`" in child process using current environment:




```
if ((pid = fork()) == 0) {    /* Child runs program */
   if (execve(myargv[0], myargv, environ) < 0) {
      printf("%s: Command not found.\n", myargv[0]);
      exit(1);
   }
}
```

Run the `printenv` command in a Linux shell to see your own environment variables

Structure of the Stack when a new program starts



`exit`: Ending a process

- `void exit(int status)`
  - Exits a process
    - Status code: 0 is used for a normal exit, nonzero for abnormal exit
  - `atexit()` registers functions to be executed upon exit

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

void fork2() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

"cleanup" is a function pointer

Processes

- Processes and context switching
- Creating new processes
  - `fork()` and `wait()`
- **Zombies**

Zombies

- When a process terminates, it still consumes system resources
  - Various tables maintained by OS
  - Called a "zombie" (a living corpse, half alive and half dead)
- *Reaping* is performed by parent on terminated child
  - Parent is given exit status information and kernel then deletes zombie child process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (pid == 1)
    - **Note:** on more recent Linux systems, `init` has been renamed `systemd`
  - In long-running processes (e.g. shells, servers) we need *explicit* reaping

`wait`: Synchronizing with Children

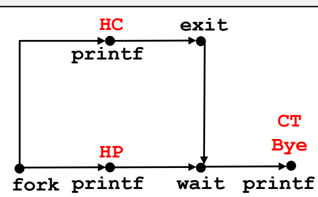
- `int wait(int *child_status)`
  - Suspends current process (i.e. the parent) until one of its children terminates
  - Return value is the PID of the child process that terminated
    - *On successful return, the child process is reaped*
  - If `child_status != NULL`, then the `*child_status` value indicates why the child process terminated
    - Special macros for interpreting this status – see **`man wait(2)`**
- **Note:** If parent process has multiple children, `wait` will return when *any* of the children terminates
  - `waitpid` can be used to wait on a specific child process

`wait`: Synchronizing with Children

```
void fork_wait() {
   int child_status;

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

forks.c



Feasible output:	Infeasible output:
HC	HP
HP	CT
CT	Bye
Bye	HC

Process Management Summary

- `fork` makes two copies of the same process (parent & child)
  - Returns different values to the two processes
- `exec*` replaces current process from file (new program)
  - Two-process program:
    - First `fork()`
    - **if** (pid == 0) { */* child code */* } **else** { */* parent code */* }
  - Two different programs:
    - First `fork()`
    - **if** (pid == 0) { execv(…) } **else** { */* parent code */* }
- `wait` or `waitpid` used to synchronize parent/child execution and to reap child process

Summary

- Processes
  - At any given time, system has multiple active processes
  - On a one-CPU system, only one can execute at a time, but each process appears to have total control of the processor
  - OS periodically "context switches" between active processes
    - Implemented using *exceptional control flow*
- Process management
  - `fork`: one call, two returns
  - `execve`: one call, usually no return
  - `wait` or `waitpid`: synchronization
  - `exit`: one call, no return

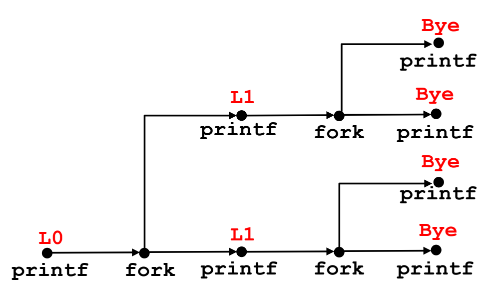
BONUS SLIDES

Detailed examples:

- Consecutive forks
- Nested forks
- Zombie example
- `wait()` example
- `waitpid()` example

Example: Two consecutive `forks`

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



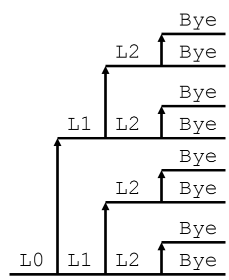
Feasible output:
L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:
L0
Bye
L1
Bye
L1
Bye
Bye

Example: Three consecutive `fork`s

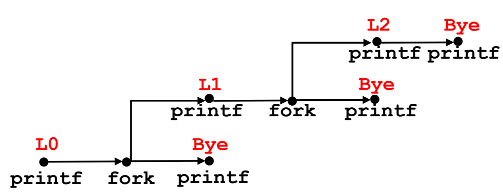
- Both parent and child can continue forking

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



Example: Nested `fork`s in children

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



Feasible output:	Infeasible output:
L0	L0
Bye	Bye
L1	L1
L2	Bye
Bye	Bye
Bye	L2

Example: Zombie

```
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 */
   }
}
```

parent persists

forks.c

```
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
```

parent

child

- `ps` shows child process as "defunct"
- Killing parent allows child to be reaped by `init`
  ↳ only because child terminated first

Example: Non-terminating Child

```
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);
   }
}
```

child persists

forks.c

```
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()` 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 terminated abnormally\n", wpid);
   }
}
```


`waitpid()`: Waiting for a Specific Process

`pid_t waitpid(pid_t pid, int &status, int options)`

- suspends current process until specific process terminates
- various options (that we won't talk about)

```
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);
   }
}
```