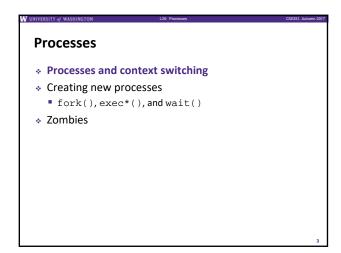
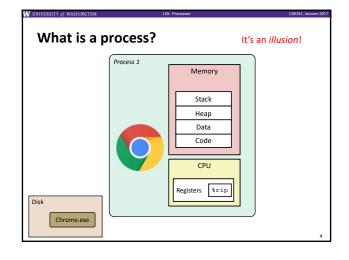
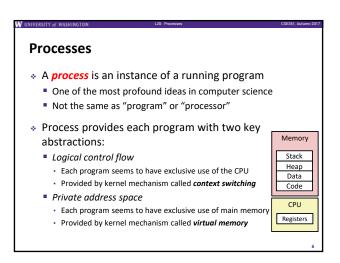


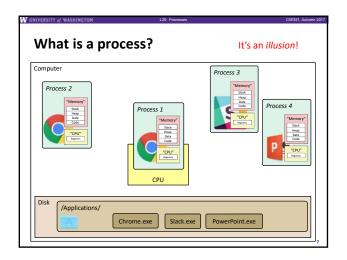
Administrivia Homework 4 due Friday (11/17) Lab 4 due after Thanksgiving (11/27) Parts of this lab are new, so don't hesitate to ask if anything is unclear or seem buggy!

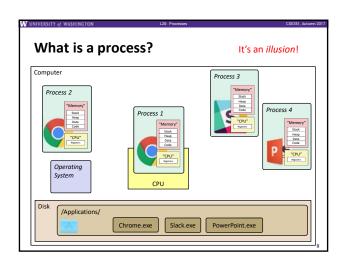


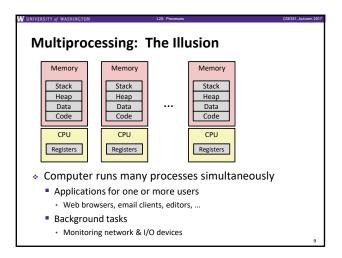


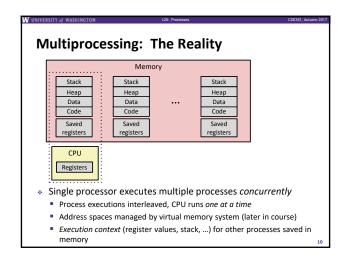
What is a process? Another abstraction in our computer system Provided by the OS OS uses a data structure to represent each process Maintains the interface between the program and the underlying hardware (CPU + memory) What do processes have to do with exceptional control flow? Exceptional control flow 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?

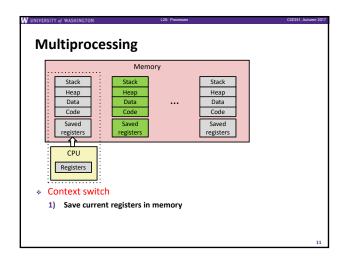


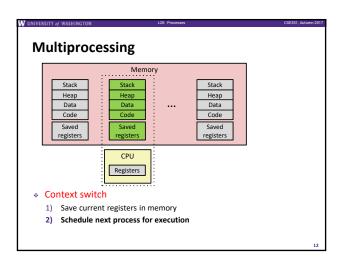


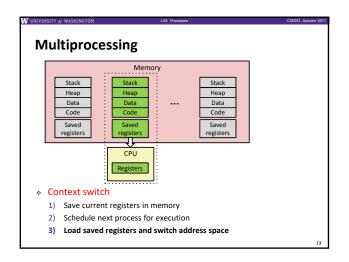


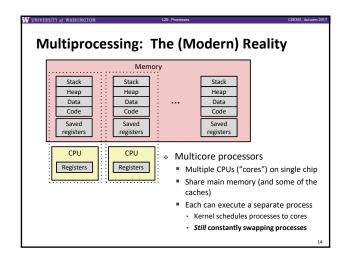


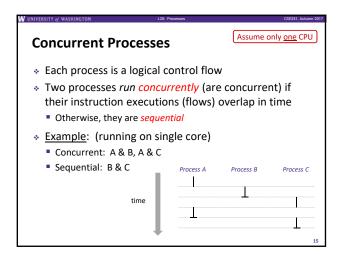


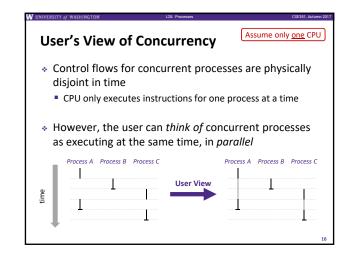


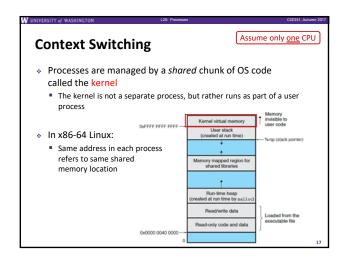


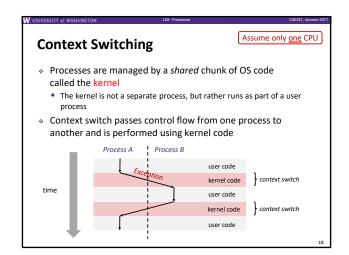


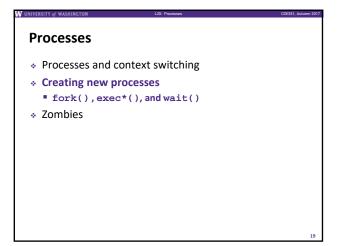


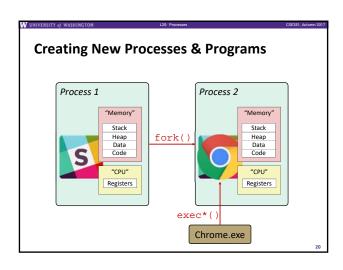












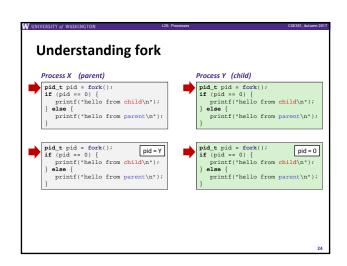
fork: Creating New Processes * pid_t fork(void) Creates a new "child" process that is identical to the calling "parent" process, including all state (memory, registers, etc.) Returns 0 to the child process Returns child's process ID (PID) to the parent process Child is almost identical to parent: Child gets an identical pid_t pid = fork(); (but separate) copy of the if (pid == 0) { parent's virtual address printf("hello from child\n"); space else { printf("hello from parent\n"); Child has a different PID than the parent * fork is unique (and often confusing) because it is called once but returns "twice"

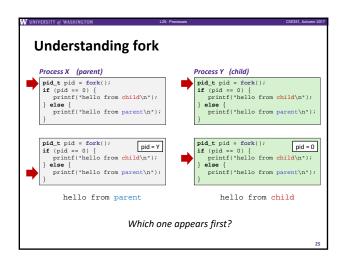
Understanding fork

Process X (parent)

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

printf(*hello from parent\n");
}





```
Fork Example

void forkl() {
   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);
}

* 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

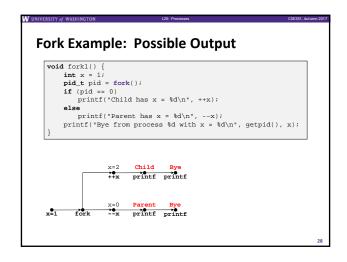
Edges can be labeled with current value of variables

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
```



```
Peer Instruction Question
Are the following sequences of outputs possible?
    Vote at <a href="http://PollEv.com/justinh">http://PollEv.com/justinh</a>
                                                  Seq 1:
                                                               Seq 2:
    void nestedfork()
printf("L0\n")
                                                  L0
                                                               L0
                                                  L1
                                                               Bye
        if (fork() == 0) {
    printf("L1\n");
                                                               L1
                                                  Bye
            if (fork() == 0) {
    printf("L2\n");
                                                  Вуе
                                                               L2
                                                  Вуе
                                                               Вуе
                                                  L2
                                                               Bye
        printf("Bye\n");
                                             A.
                                                 No
                                                               No
                                            В.
                                                  No
                                                               Yes
                                             C.
                                                  Yes
                                                               No
                                                  Yes
                                            D.
                                                               Yes
                                            E. We're lost...
```

```
Fork-Exec

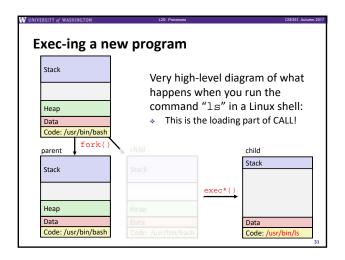
* fork-exec model:

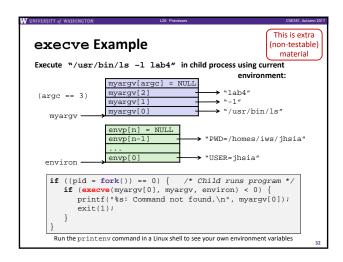
* fork() creates a copy of the current process

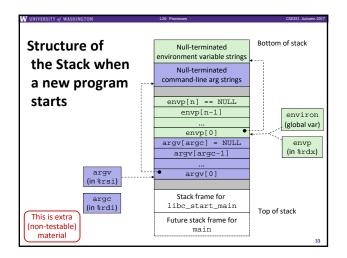
* exec*() replaces the current process' code and address space with the code for a different program

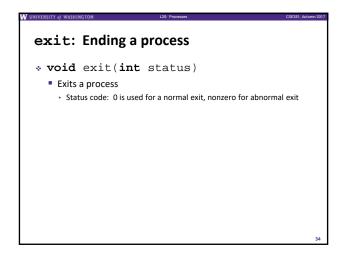
* Whole family of exec calls – see exec(3) and execve(2)

// Example arguments: path="/usr/bin/ls", argv(1)="-ah1", argv(2)=NULL void fork_exec(char *path, char *argv[1) {
    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");
}
```



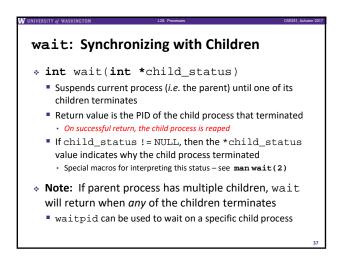






Processes Processes Processes and context switching Creating new processes Fork(), exec*(), and wait() Zombies

WINNTESTIT of WASHINGTON 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

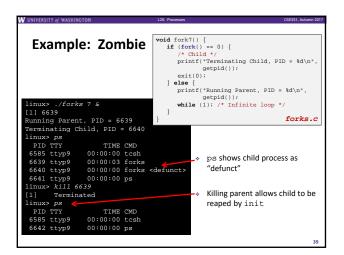


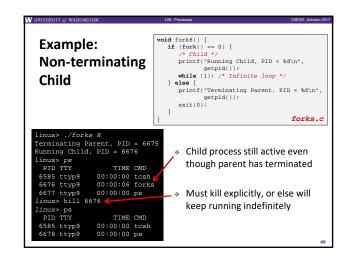
```
wait: Synchronizing with Children

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");
  }

HC exit

Printf Feasible output: HC
HP
CT
HP
Bye
Fork printf wait printf
Bye
HC
38
```





```
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()

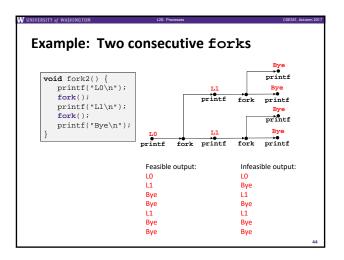
* if (pid == 0) { execv(...) } else {/* parent code */}

* 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 * wait() example * waitpid() example



Example: Three consecutive forks

. Both parent and child can continue forking

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



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)