



Processes and Multitasking

15-213/15-513/14-513: Introduction to Computer Systems
16th Lecture, October 28, 2025

Today

- **Processes**
- System Calls
- Process Control
- Shells

Earliest days: One batch job at a time



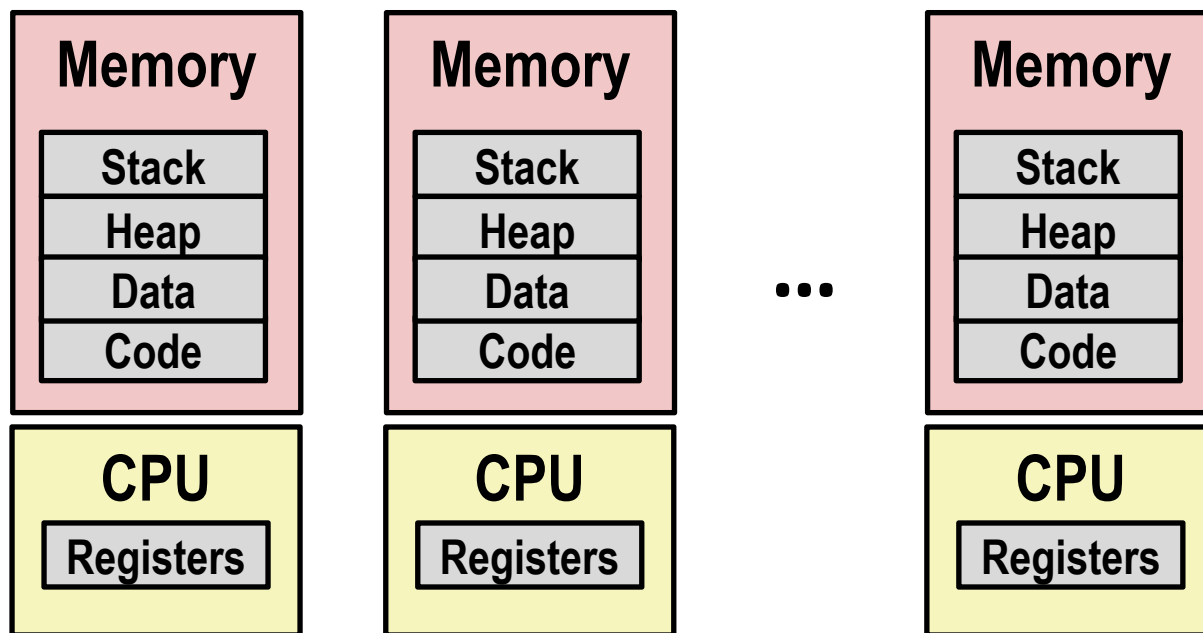
IBM 704 at Langley Research Center (NASA), 1957

<https://commons.wikimedia.org/w/index.php?curid=6455009>

How can many people share one computer efficiently?



Multiprocessing



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

Multiprocessing Example

```

shark.ics.cmu.edu - PuTTY
top - 12:52:25 up 7:50, 12 users, load average: 4.94, 4.06, 2.72
Tasks: 425 total, 7 running, 418 sleeping, 0 stopped, 0 zombie
%Cpu(s): 11.2 us, 21.9 sy, 0.0 ni, 66.0 id, 0.0 wa, 0.0 hi, 0.9 si, 0.0 st
KiB Mem : 24508768 total, 19088248 free, 3228068 used, 2192452 buff/cache
KiB Swap: 1048572 total, 1048572 free, 0 used. 20822672 avail Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
30569 zilongz   20   0   20.0t  25896  1324  R  100.0   0.1   0:05.89 mdriver-dbg
26365 zilongz   20   0 2566560 231684  8428  S   92.4   0.9   6:20.52 cpptools
17759 julietf   20   0  164876   3864  1284  R   84.4   0.0  15:31.82 sshd
  1673 root      20   0      0      0      0  R   58.3   0.0   5:55.84 afs_rxlst+
20161 julietf   20   0   20.0t 112840  1348  R   57.6   0.5  10:36.80 mdriver-dbg
30624 jjli2     20   0  130708  16896  1692  R   36.4   0.1   0:01.10 ld
24896 root      20   0      0      0      0  S   11.6   0.0   0:17.94 kworker/5:1
29234 root      20   0      0      0      0  R    8.9   0.0   0:02.95 kworker/1:0
29616 root      20   0      0      0      0  S    6.6   0.0   0:02.54 kworker/13+
26141 root      20   0      0      0      0  S    4.3   0.0   0:13.43 kworker/3:1
29254 root      20   0      0      0      0  S    4.3   0.0   0:03.02 kworker/9:0
26787 root      20   0      0      0      0  S    4.0   0.0   0:08.78 kworker/11+
26785 root      20   0      0      0      0  S    2.0   0.0   0:09.53 kworker/13+
25644 zilongz   20   0 1051004 158028  19260 S    1.3   0.6   0:19.99 node
27858 bbendou   20   0  898344  64932  18832 S    1.3   0.3   0:03.01 node
15130 yixuey    20   0  903052  70108  18976 S    1.0   0.3   0:12.12 node
30194 zweinber  20   0  164268   2552   1568 R    1.0   0.0   0:00.27 top

```

- Running program “top” on hammerheadshark
 - System has 425 “tasks”, 7 of which are active
 - Identified by Process ID (PID), user account, command name

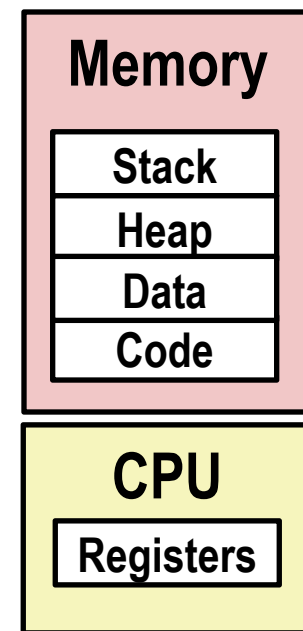
Processes

- **Definition:** 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:**

- *Private address space*
 - Each program seems to have exclusive use of main memory.
 - Provided by kernel mechanism called *virtual memory*
- *Logical control flow*
 - Each program seems to have exclusive use of the CPU
 - Provided by kernel mechanism called *context switching*

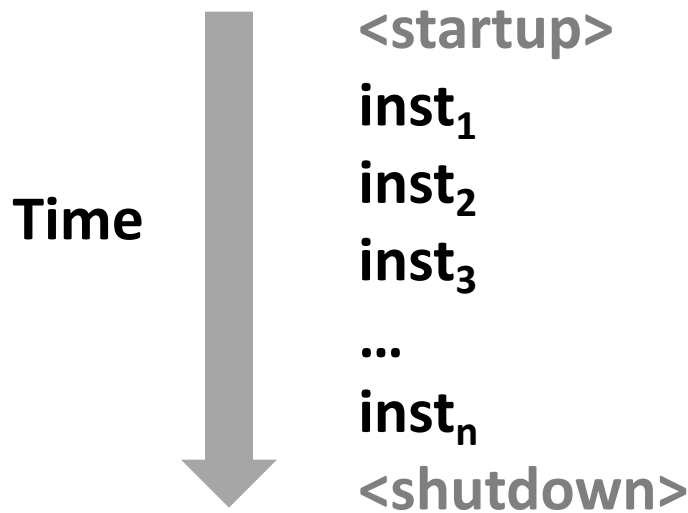


Control Flow

■ Processors do only one thing:

- From startup to shutdown, each CPU core simply reads and executes a sequence of machine instructions, one at a time *
- This sequence is the CPU's *control flow* (or *flow of control*)

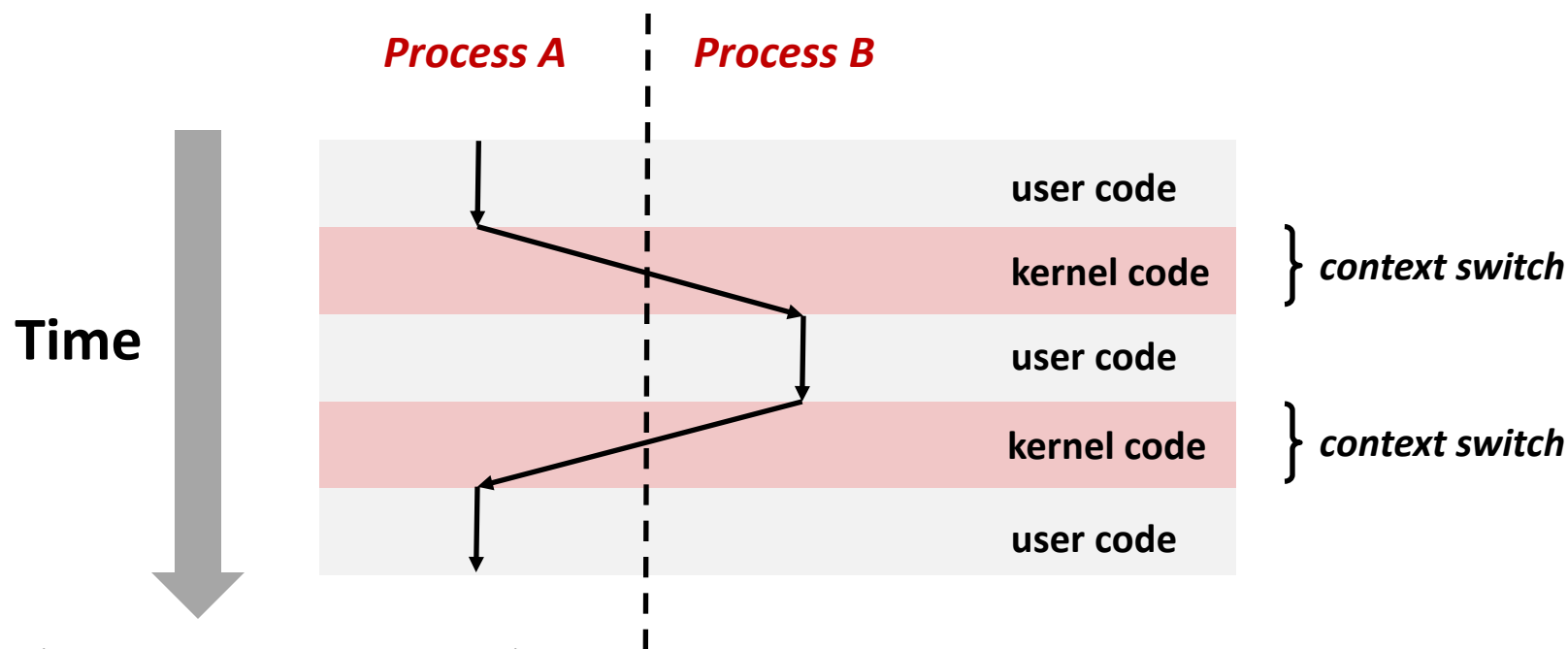
Physical control flow



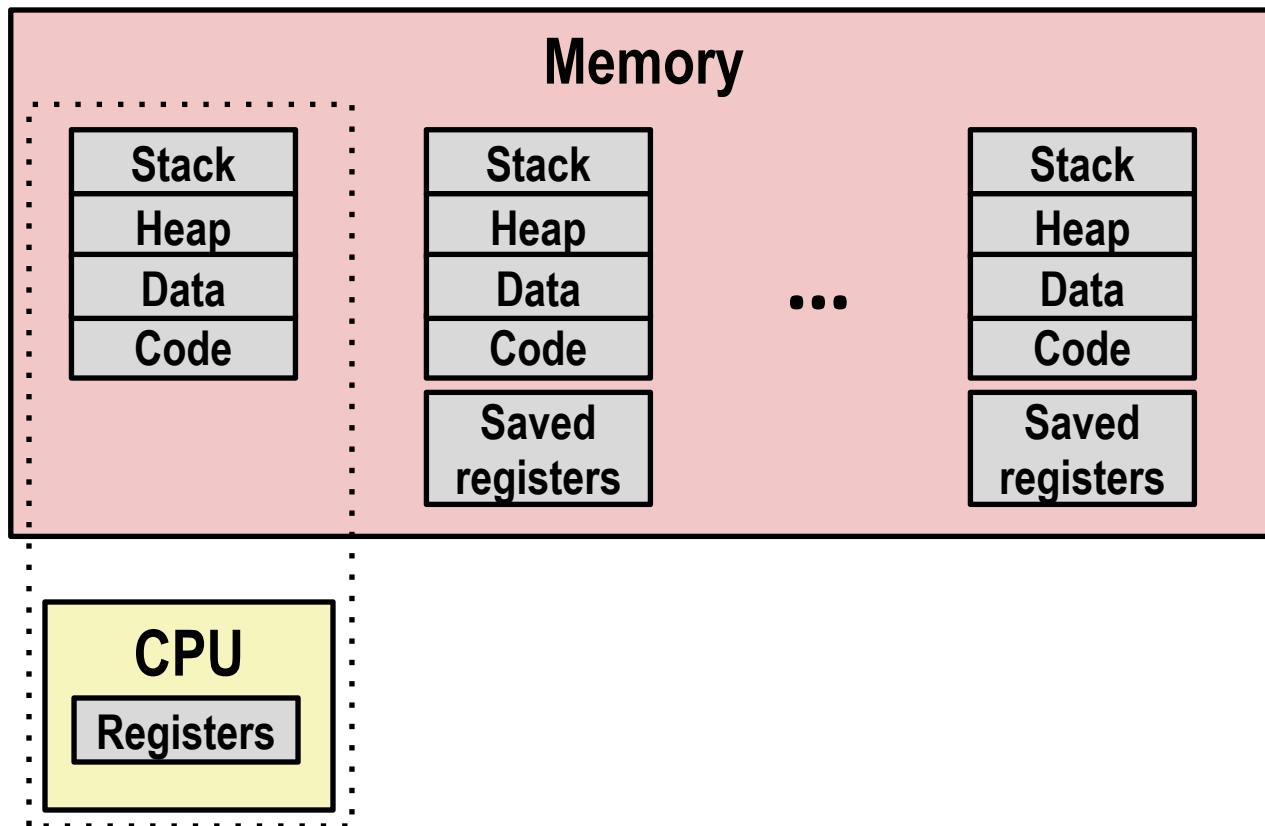
- * many modern CPUs execute several instructions at once and/or out of program order, but this is invisible to the programmer

Context Switching

- Processes are managed by a shared chunk of memory-resident OS code called the *kernel*
 - Important: the kernel is not a separate process, but rather runs as part of some existing process.
- Control flow passes from one process to another via a *context switch*

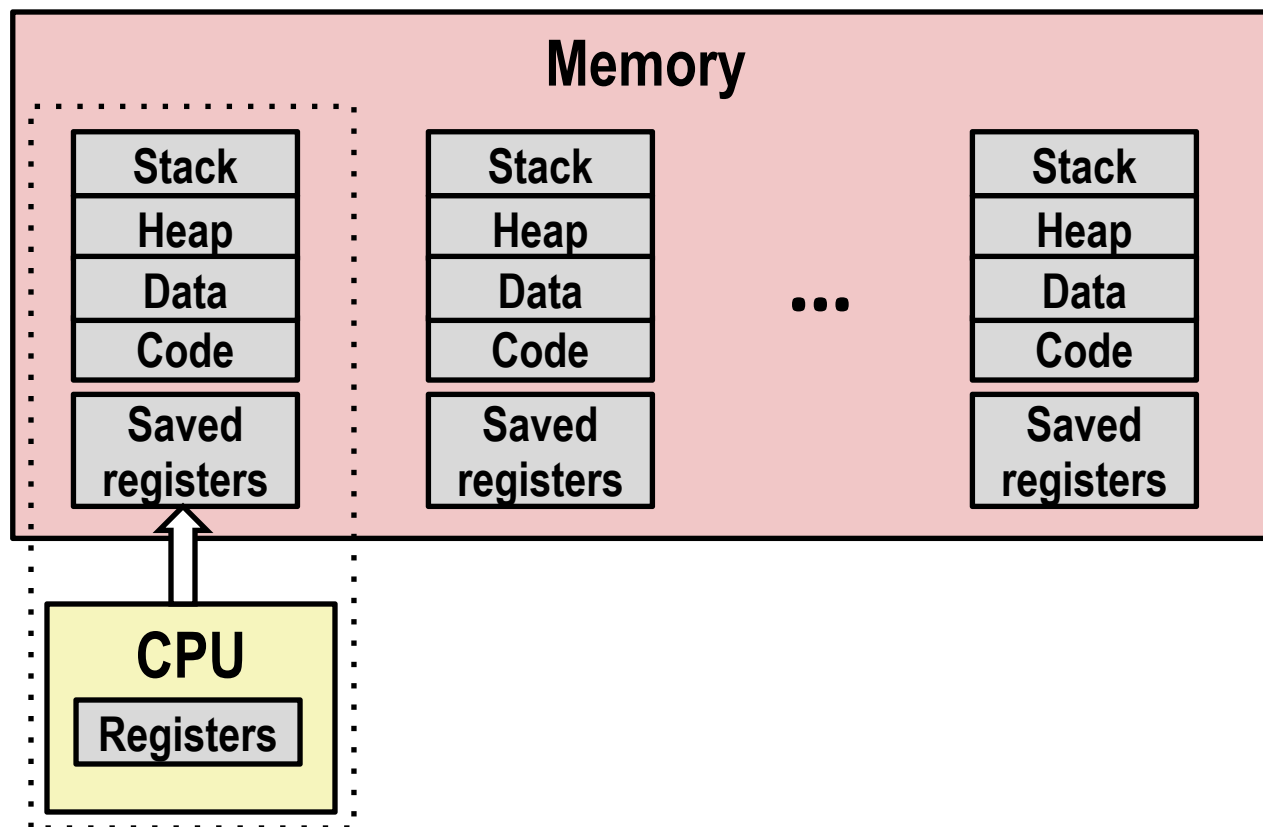


Context Switching (Uniprocessor)



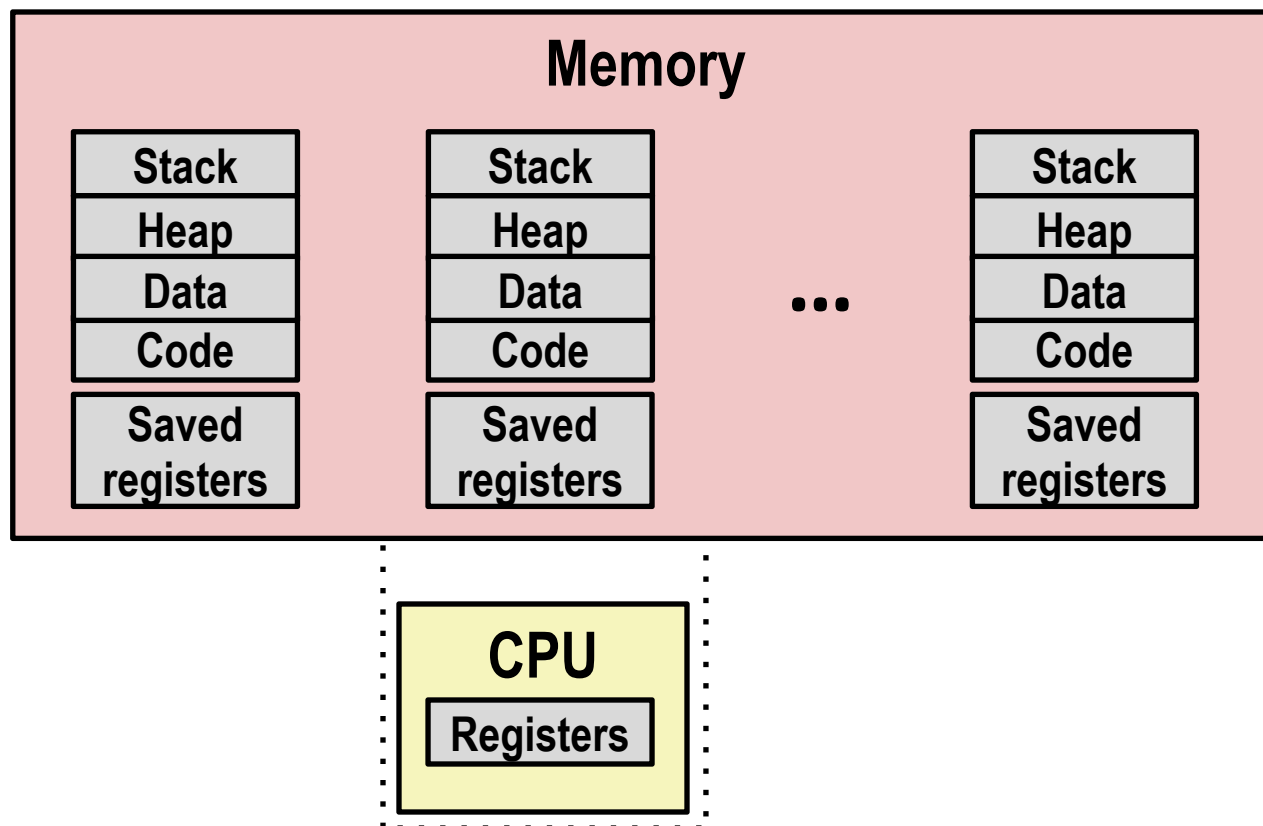
- **Single processor executes multiple processes concurrently**
 - Process executions interleaved (multitasking)
 - Address spaces managed by virtual memory system (like last week)
 - Register values for nonexecuting processes saved in memory

Context Switching (Uniprocessor)



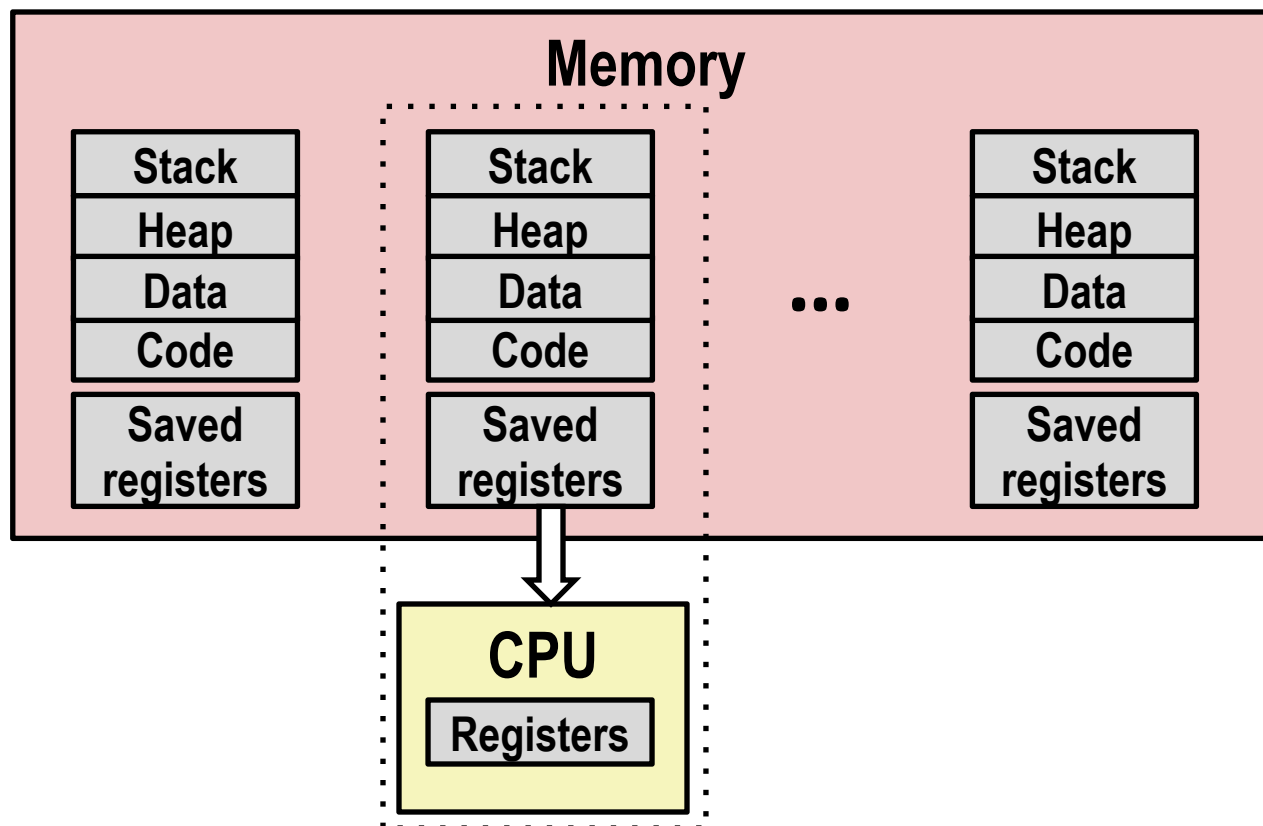
- **Save current registers in memory**

Context Switching (Uniprocessor)



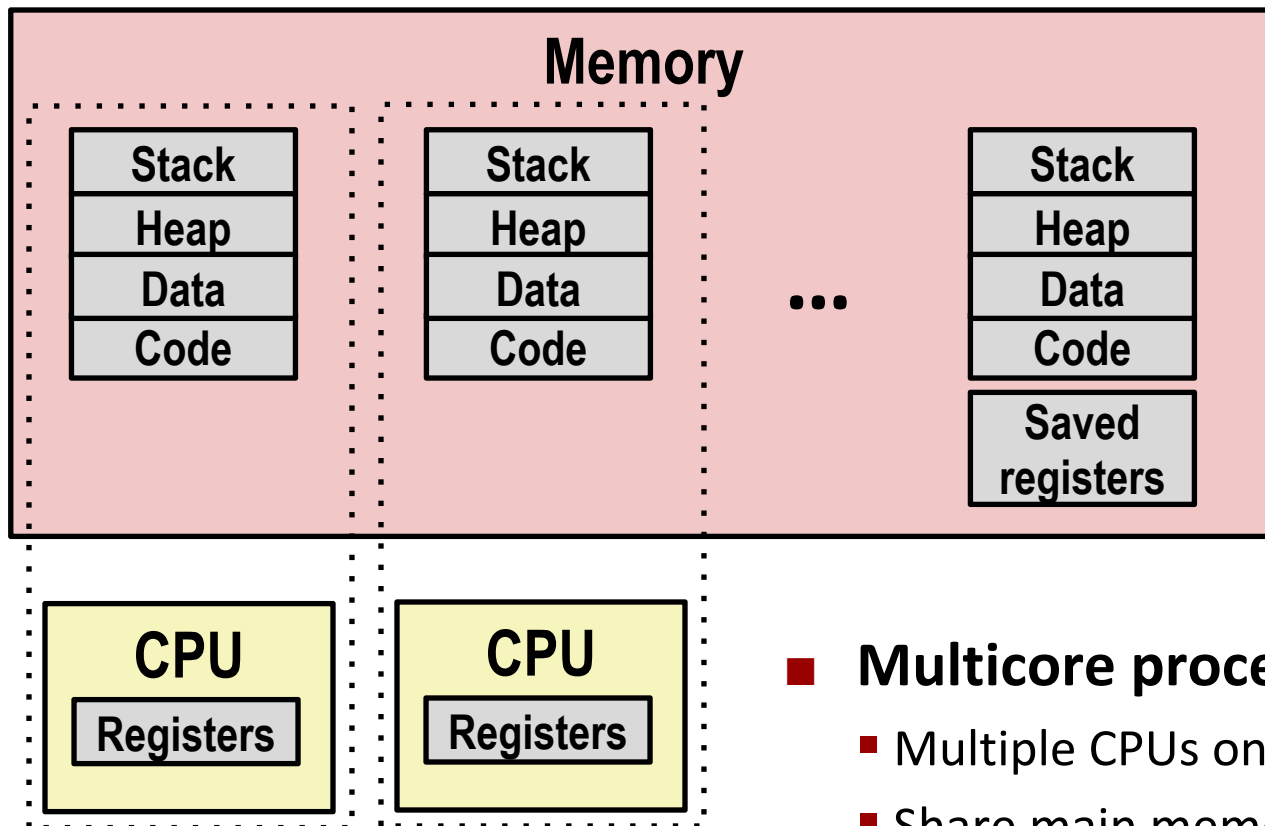
- **Schedule next process for execution**

Context Switching (Uniprocessor)



- Load saved registers and switch address space (context switch)

Context Switching (Multicore)

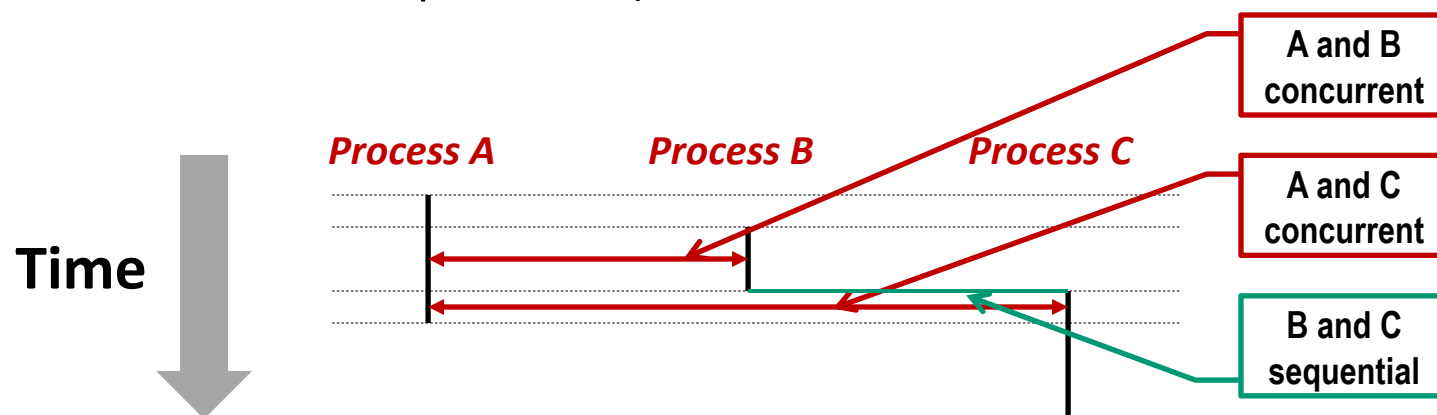


■ Multicore processors

- Multiple CPUs on single chip
- Share main memory (and some caches)
- Each can execute a separate process
 - Scheduling of processors onto cores done by kernel

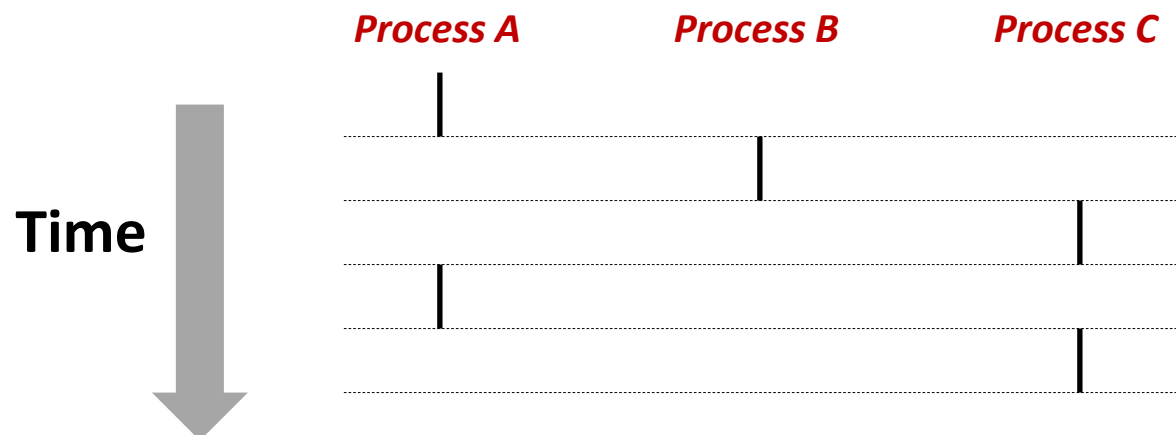
User View of Concurrent Processes

- Two processes *run concurrently* (are concurrent) if their execution overlaps in time
- Otherwise, they are *sequential*
- Appears as if concurrent processes run in parallel with each other
 - This means they can interfere with each other (more on that in a couple weeks)



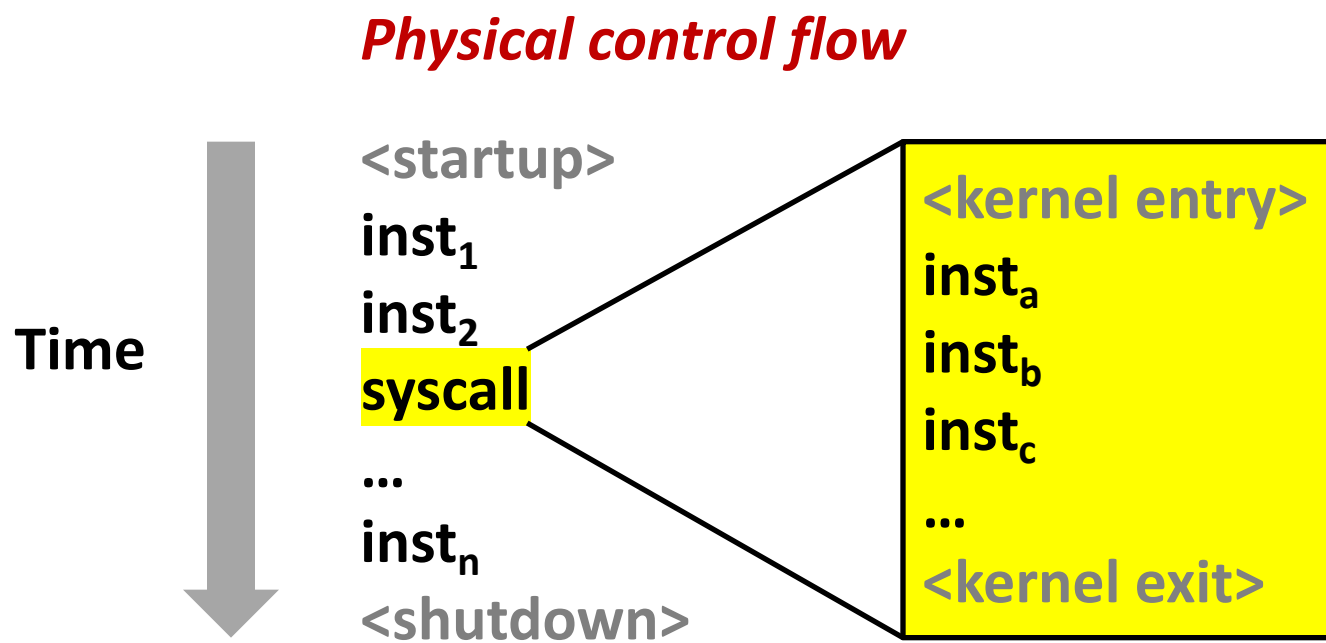
Traditional (Uniprocessor) Reality

- Only one process runs at a time
- A and B execution is *interleaved*, not truly concurrent
- Similarly for A and C
- Still possible for A and B / A and C to interfere with each other



How does the kernel take control?

- The CPU executes instructions in sequence
- We don't write "now run kernel code" in our programs...
 - *Or do we??*



Today

- Processes
- **System Calls**
- Process Control
- Shells

System Calls

- Whenever a program wants to cause an effect outside its own process, it must ask the kernel for help
- Examples:
 - Read/write files
 - Get current time
 - Allocate RAM (sbrk)
 - Create new processes

```
// fopen.c
FILE *fopen(const char *fname,
            const char *mode) {
    int flags = mode2flags(mode);
    if (!flags) return NULL;
    int fd = open(fname, flags,
                  DEFPERMS);
    if (fd == -1) return NULL;
    return fdopen(fd, mode);
}

// open.S
.global open
open:
    mov $SYS_open, %eax
    syscall
    cmp $SYS_error_thresh, %rax
    ja  __syscall_error
    ret
```

All the system calls

accept	fanotify_init	getresuid	llistxattr	nfsservctl	recvmsg	set_mempolicy_home_node	sync_file_range
accept4	fanotify_mark	getrlimit	lookup_dcookie	open_by_handle_at	recvmsg	set_robust_list	sync_file_range2
acct	fchdir	getrusage	lremovexattr	open_tree	remap_file_pages	set_tid_address	syncfs
add_key	fchmod	getsid	lsetxattr	openat	removexattr	setdomainname	sysinfo
adjtimex	fchmodat	getsockname	madvis	openat2	renameat	setfsuid	syslog
bind	fchown	getsockopt	mbind	perf_event_open	renameat2	setfsuid	tee
bpf	fchownat	gettid	membarrier	personality	request_key	setgid	tgkill
brk	fdatasync	gettimeofday	memfd_create	pidfd_getfd	restart_syscall	setgroups	timer_create
capget	fgetxattr	getuid	memfd_secret	pidfd_open	rseq	sethostname	timer_delete
capset	finit_module	getxattr	migrate_pages	pidfd_send_signal	rt_sigaction	setitimer	timer_getoverrun
chdir	flistxattr	init_module	mincore	pipe2	rt_sigpending	setns	timer_gettime
chroot	flock	inotify_add_watch	mknodat	pivot_root	rt_sigprocmask	setpgid	timer_settime
clock_adjtime	fremovexattr	inotify_init1	mknodat	pkey_alloc	rt_sigqueueinfo	setpriority	timerfd_create
clock_getres	fsconfig	inotify_rm_watch	mlock	pkey_free	rt_sigreturn	setregid	timerfd_gettime
clock_gettime	fsetxattr	io_cancel	mlock2	pkey_mprotect	rt_sigsuspend	setresgid	timerfd_settime
clock_nanosleep	fsmount	io_destroy	mlockall	ppoll	rt_sigtimedwait	setresuid	times
clock_settime	fsopen	io_getevents	mount	prctl	rt_tgsigqueueinfo	setreuid	tkill
clone	fspick	io_pgetevents	mount_setattr	pread64	sched_get_priority_max	setrlimit	umask
clone3	fsync	io_setup	move_mount	preadv	sched_get_priority_min	setsid	umount2
close	futex	io_submit	move_pages	preadv2	sched_getaffinity	setsockopt	uname
close_range	futex_waitv	io_uring_enter	mprotect	prlimit64	sched_getattr	settimeofday	unlinkat
connect	get_mempolicy	io_uring_register	mq_getsetattr	process_madvise	sched_getparam	setuid	unshare
copy_file_range	get_robust_list	io_uring_setup	mq_notify	process_mrelease	sched_getscheduler	setxattr	userfaultfd
delete_module	getcpu	ioctl	mq_open	process_vm_readv	sched_rr_get_interval	shmat	utimensat
dup	getcwd	ioprio_get	mq_timedreceive	process_vm_writev	sched_setaffinity	shmctl	vhangup
dup3	getdents64	ioprio_set	mq_timedsend	pselect6	sched_setattr	shmdt	vmsplice
epoll_create1	getegid	kcmp	mq_unlink	ptrace	sched_setparam	shmget	wait4
epoll_ctl	geteuid	kexec_file_load	mremap	pwrite64	sched_setscheduler	shutdown	waitid
epoll_pwait	getgid	kexec_load	msgctl	pwritev	sched_yield	sigaltstack	write
epoll_pwait2	getgroups	keyctl	msgget	pwritev2	seccomp	signalfd4	writev
eventfd2	getitimer	kill	msgrcv	quotactl	semctl	socket	
execve	getpeername	landlock_add_rule	msgsnd	quotactl_fd	semget	socketpair	
execveat	getpgid	landlock_create_ruleset	msync	read	semop	splice	
exit	getpid	landlock_restrict_self	munlock	readahead	semtimedop	statx	
exit_group	getppid	lgetxattr	munlockall	readlinkat	sendmsg	swapoff	
faccessat	getpriority	linkat	munmap	readv	sendmsg	swapon	
faccessat2	getrandom	listen	name_to_handle_at	reboot	sendto	symlinkat	
fallocate	getresgid	listxattr	nanosleep	recvfrom	set_mempolicy	sync	

System Call Error Handling

- **Almost all system-level operations can fail**
 - Only exception is the handful of functions that return `void`
 - You must explicitly check for failure
- **On error, most system-level functions return `-1` and set global variable `errno` to indicate cause.**
- **Example:**

```
pid_t pid = fork();  
if (pid == -1) {  
    fprintf(stderr, "fork error: %s\n", strerror(errno));  
    exit(1);  
}
```

Error-reporting functions

- Can simplify somewhat using an *error-reporting function*:

```
void unix_error(char *msg) /* Unix-style error */
{
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(1);
}
```

```
pid_t pid = fork();
if (pid == -1)
    unix_error("fork error");
```

Note: csapp.c exits with 0.

- Not always appropriate to exit when something goes wrong.

Error-handling Wrappers

- We simplify the code we present to you even further by using Stevens¹-style error-handling wrappers:

```
pid_t Fork(void)
{
    pid_t pid = fork();

    if (pid == -1)
        unix_error("Fork error");
    return pid;
}
```

```
pid = Fork(); // Only returns if successful
```

- NOT what you generally want to do in a real application

¹e.g., in "UNIX Network Programming: The sockets networking API" W. Richard Stevens

Today

- Processes
- System Calls
- **Process Control**
- Shells

Obtaining Process IDs

- `pid_t getpid(void)`
 - Returns PID of current process
- `pid_t getppid(void)`
 - Returns PID of parent process

Process States

At any time, each process is either:

■ Running

- Process is either executing instructions, or it *could be* executing instructions if there were enough CPU cores.

■ Blocked / Sleeping

- Process cannot execute any more instructions until some external event happens (usually I/O).

■ Stopped

- Process has been prevented from executing by user action (control-Z).

■ Terminated / Zombie

- Process is finished. Parent process has not yet been notified.

Terminating Processes

- **Process becomes terminated for one of three reasons:**
 - Receiving a signal whose default action is to terminate (next lecture)
 - Returning from the **main** routine
 - Calling the **exit** function

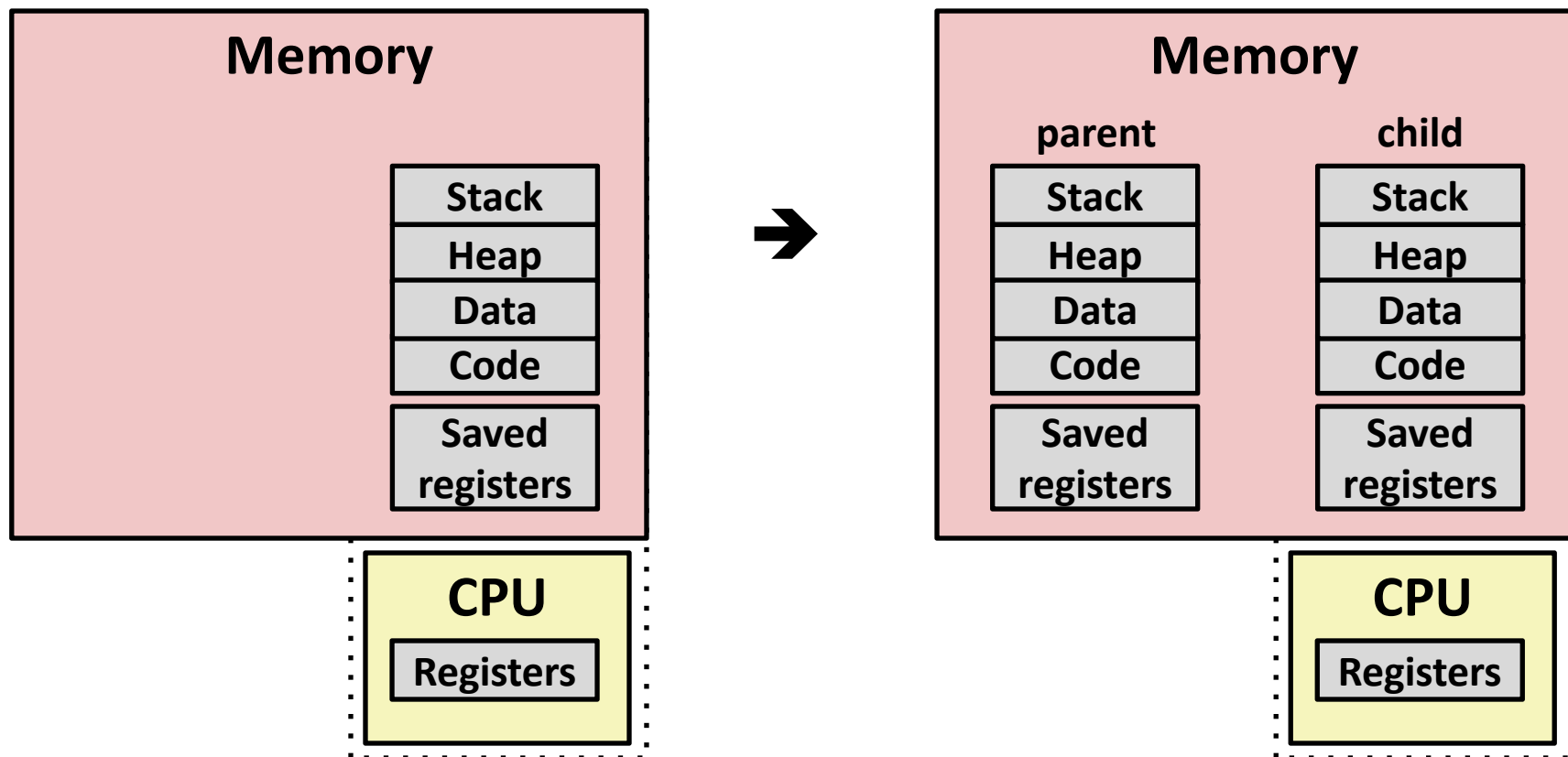
- **`void exit(int status)`**
 - Terminates with an *exit status* of **status**
 - Convention: normal return status is 0, nonzero on error
 - Another way to explicitly set the exit status is to return an integer value from the main routine

- **`exit` is called **once** but **never** returns.**

Creating Processes

- *Parent process* creates a new running *child process* by calling `fork`
- `int fork(void)`
 - Returns 0 to the child process, child's PID to parent process
 - Child is *almost* identical to parent:
 - Child get an identical (but separate) copy of the parent's virtual address space.
 - Child gets identical copies of the parent's open file descriptors
 - Child has a different PID than the parent
- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*

Conceptual View of fork



■ Make complete copy of execution state

- Designate one as parent and one as child
- Resume execution of parent or child
- (Optimization: Use copy-on-write to avoid copying RAM)

fork Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

fork.c

- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
child : x=2
parent: x=0
```

```
linux> ./fork
parent: x=0
child : x=2
```

```
linux> ./fork
parent: x=0
child : x=2
```

fork Example

```
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}
```

```
linux> ./fork
parent: x=0
child : x=2
```

- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child
- Duplicate but separate address space
 - `x` has a value of 1 when fork returns in parent and child
 - 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
 - $a \rightarrow b$ 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

Process Graph Example

```

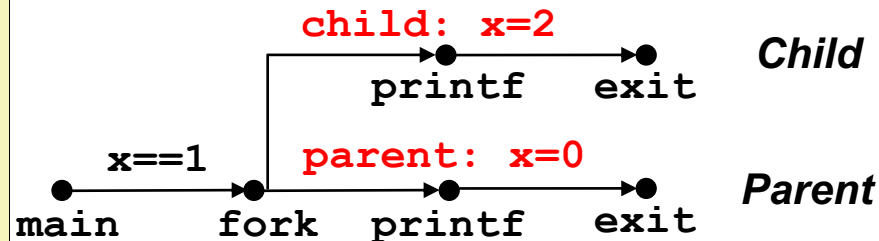
int main(int argc, char** argv)
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        return 0;
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    return 0;
}

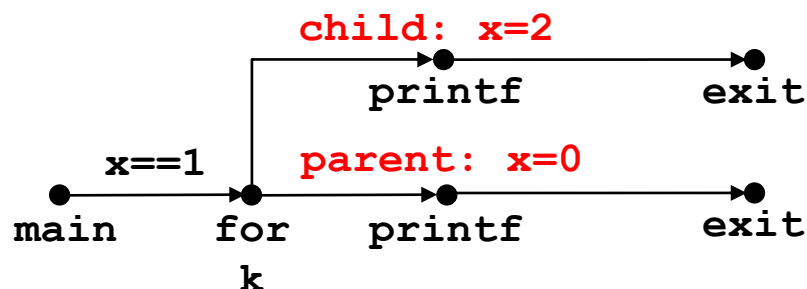
```

fork.c

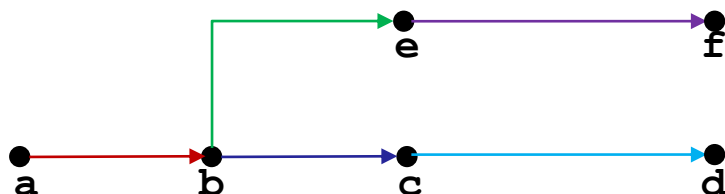


Interpreting Process Graphs

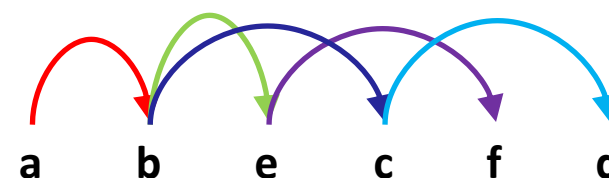
■ Original graph:



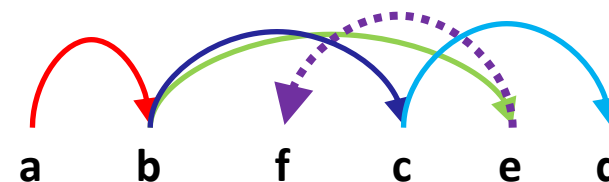
■ Relabelled graph:



Feasible total ordering:



Feasible or Infeasible?

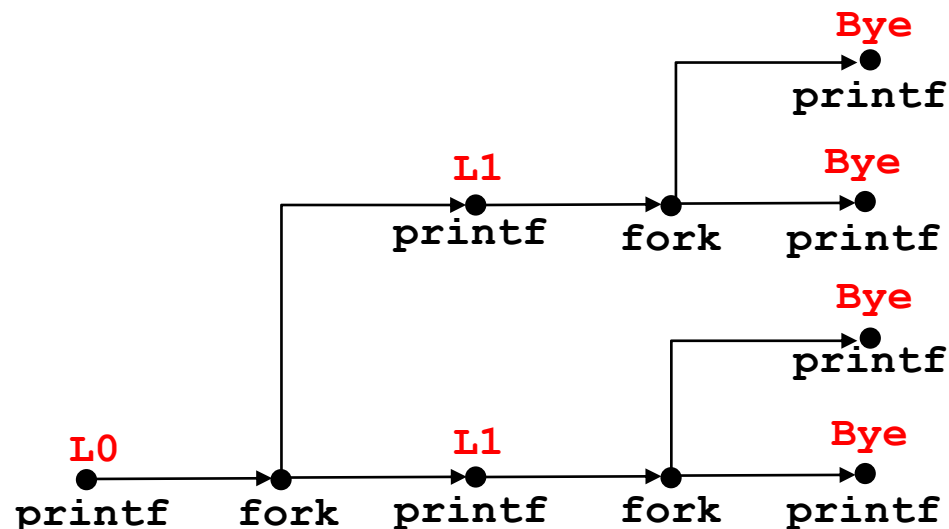


Infeasible: not a topological sort

fork Example: Two consecutive forks

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

forks.c



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

Infeasible output:

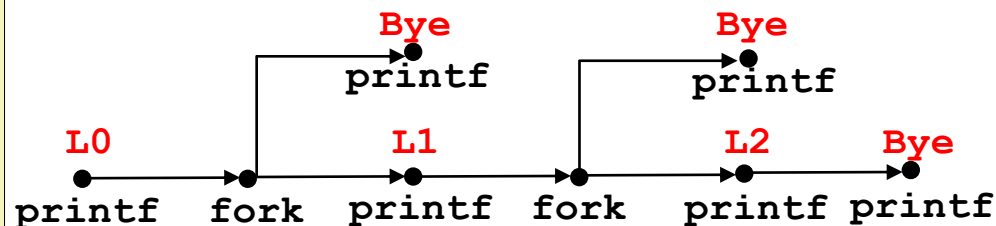
L0
Bye
L1
Bye
L1
Bye
Bye

fork Example: Nested forks in parent

```

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

```



Feasible or Infeasible?

L0

Bye

L1

Bye

Bye

L2

Infeasible

Feasible or Infeasible?

L0

L1

Bye

Bye

L2

Bye

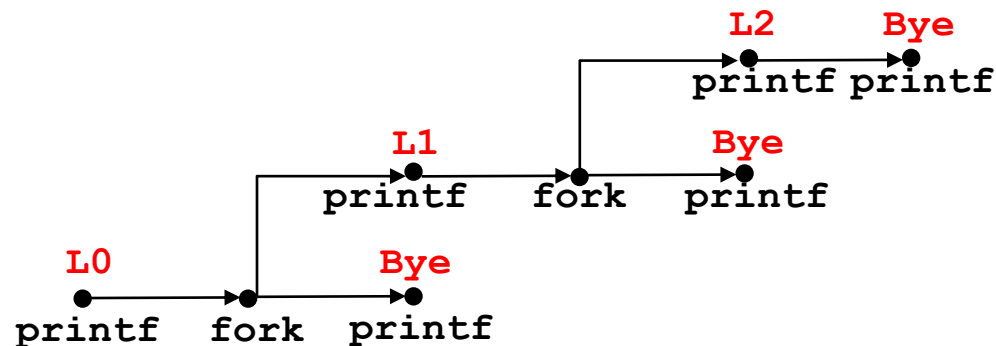
Feasible

fork Example: Nested forks in children

```

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

```



Feasible or Infeasible?

L0

Bye

L1

Bye

Bye

L2

Infeasible

Feasible or Infeasible?

L0

Bye

L1

L2

Bye

Bye

Feasible

Reaping Child Processes

■ Idea

- When process terminates, it still consumes system resources
 - Examples: Exit status, various OS tables
- Called a “zombie”
 - Living corpse, half alive and half dead

■ Reaping

- Performed by parent on terminated child (using `wait` or `waitpid`)
- Parent is given exit status information
- Kernel then deletes zombie child process

■ What if parent doesn't reap?

- If any parent terminates without reaping a child, then the orphaned child should be reaped by `init` process (`pid == 1`)
 - Unless it was `init` that terminated! Then need to reboot...
- So, only need explicit reaping in 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” (i.e., a zombie)

■ Killing parent allows child to be reaped by **init**

Non-terminating 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 child explicitly, or else will keep running indefinitely

`wait`: Synchronizing with Children

- Parent reaps a child with one of these system calls:
- `pid_t wait(int *status)`
 - Suspends current process until one of its children terminates
 - Returns PID of child, records exit status in `status`
- `pid_t waitpid(pid_t pid, int *status, int options)`
 - More flexible version of `wait`:
 - Can wait for a specific child or group of children
 - Can be told to return immediately if there are no children to reap

wait: Synchronizing with Children

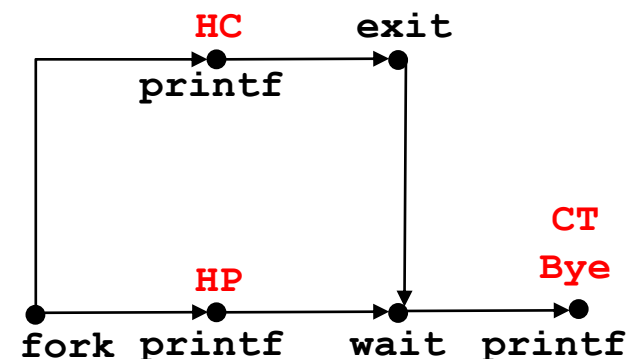
```

void fork9() {
    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(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

HP
CT
Bye
HC

wait: Status codes

- Return value of `wait` is the pid of the child process that terminated
- If `status != NULL`, then the integer it points to will be set to a value that indicates the exit status
 - More information than the value passed to `exit`
 - Must be decoded, using macros defined in `sys/wait.h`
 - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
 - See textbook for details

Another 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, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(100+i); /* Child */
        }
    for (i = 0; i < N; i++) { /* Parent */
        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);
    }
}
```

forks.c

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 (see textbook)

```
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 = N-1; i >= 0; 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 terminate abnormally\n", wpid);
    }
}
```

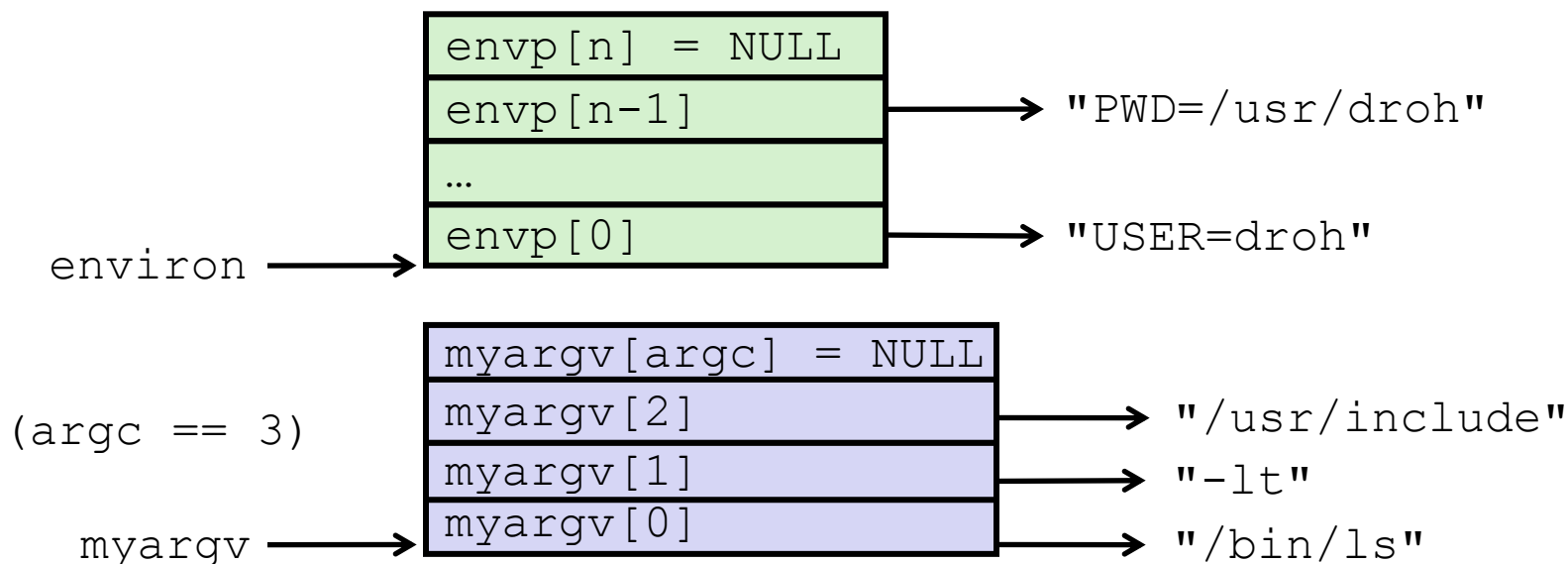
forks.c

execve : Loading and Running Programs

- `int execve(char *filename, char *argv[], char *envp[])`
- **Loads and runs in the current process:**
 - Executable file `filename`
 - Can be object file or script file beginning with `#!interpreter` (e.g., `#!/bin/bash`)
 - ...with argument list `argv`
 - By convention `argv[0]==filename`
 - ...and environment variable list `envp`
 - “name=value” strings (e.g., `USER=droh`)
 - `getenv`, `putenv`, `putenv`
- **Overwrites code, data, and stack**
 - Retains PID, open files and signal context
- **Called **once** and **never** returns**
 - ...except if there is an error

execve Example

- Execute `"/bin/ls -lt /usr/include"` in child process using current environment:

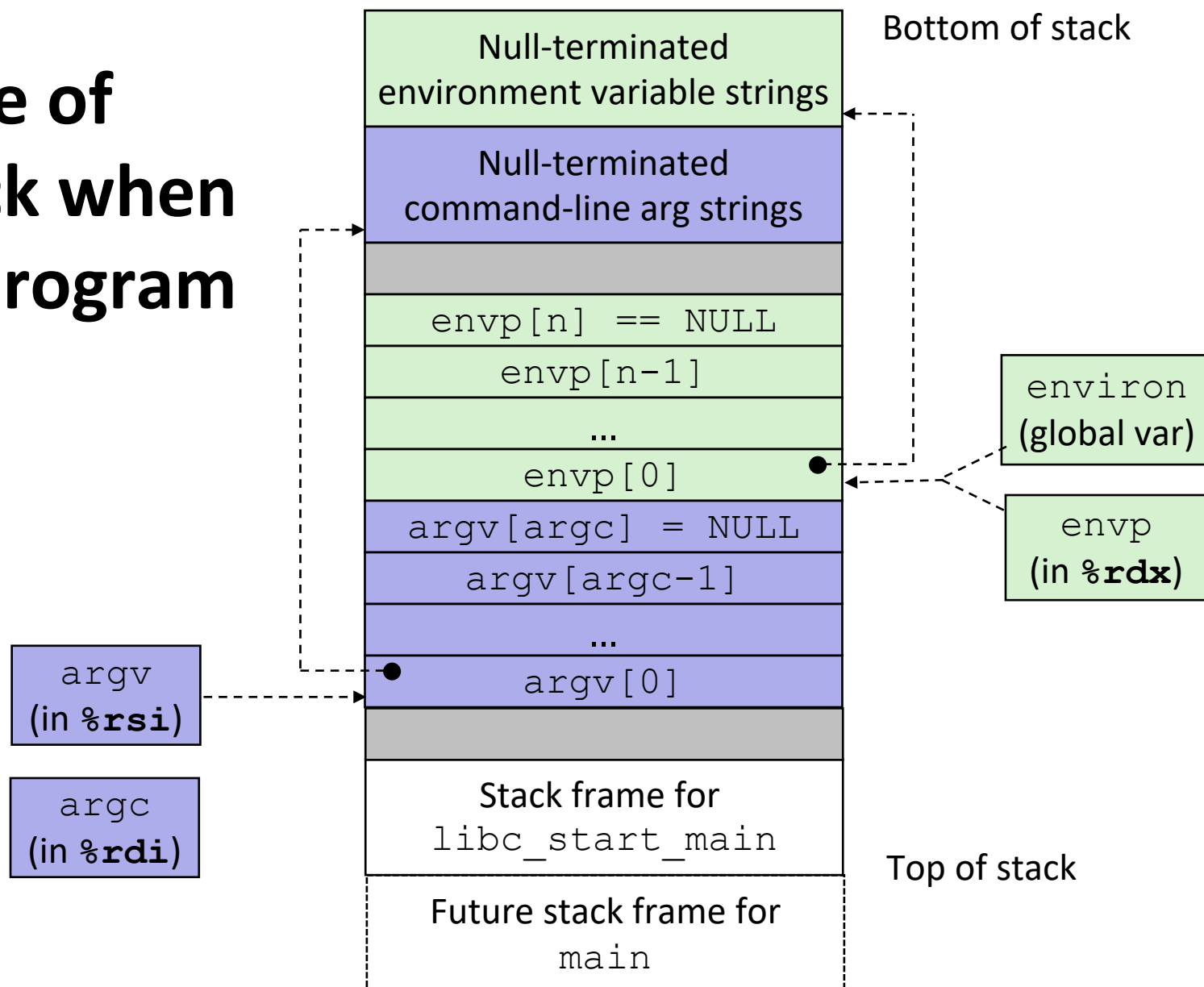


```

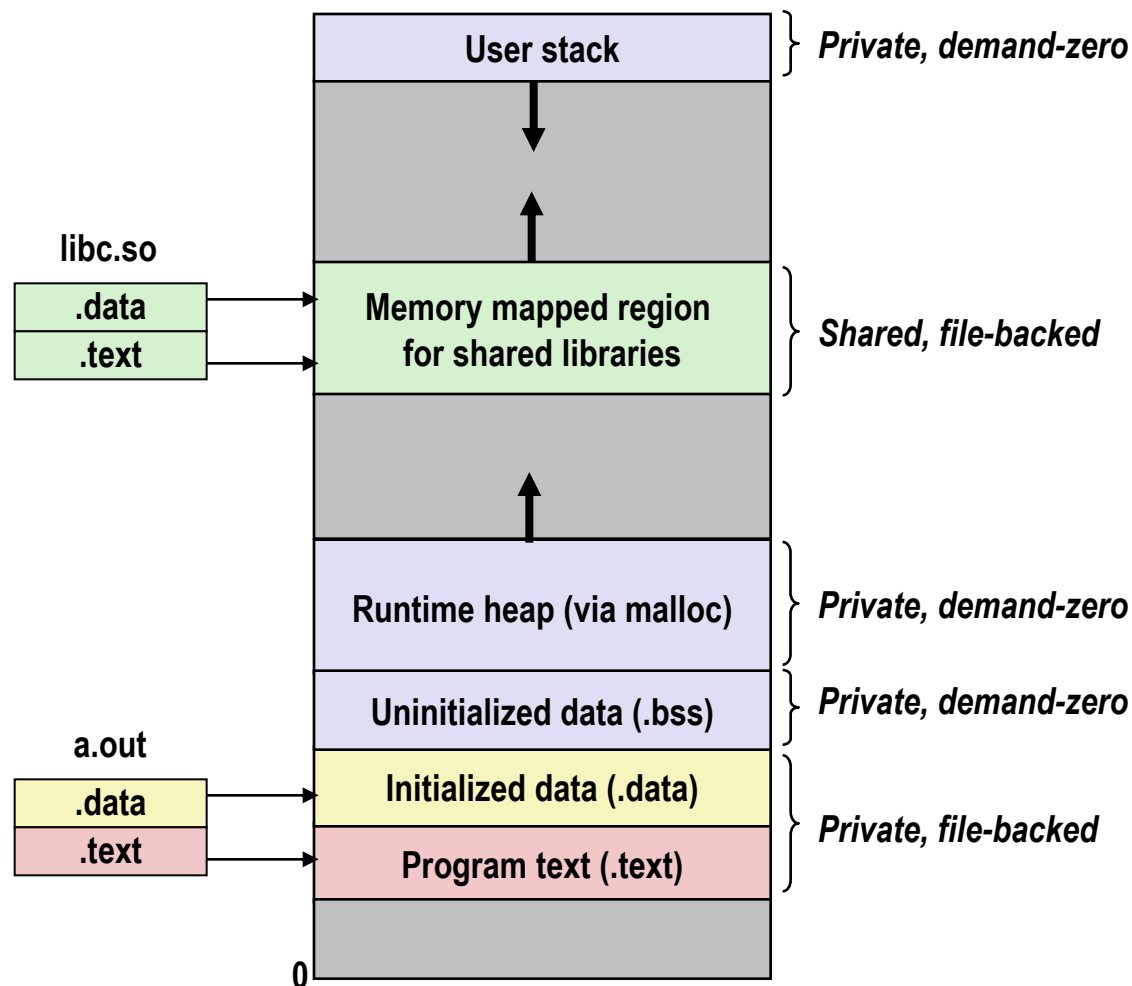
if ((pid = Fork()) == 0) {    /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: %s\n", myargv[0], strerror(errno));
        exit(1);
    }
}

```

Structure of the stack when a new program starts



execve and process memory layout



- To load and run a new program `a.out` in the current process using `execve`:
- Free `vm_area_struct`'s and `page tables` for old areas
- Create `vm_area_struct`'s and `page tables` for new areas
 - Programs and initialized data backed by object files.
 - `.bss` and stack backed by anonymous files.
- Set PC to entry point in `.text`
 - Linux will fault in code and data pages as needed.

Quiz

<https://canvas.cmu.edu/courses/49105/quizzes/150044/>

Today

- Processes
- System Calls
- Process Control
- **Shells**

Shell Programs

- A *shell* is an application program that runs programs on behalf of the user
 - **sh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
 - **csh/tcsh** BSD Unix C shell
 - **bash** “Bourne-Again” Shell (default Linux shell)



```
sharkics.cs.cmu.edu - PuTTY
zweinber@hammerheadshark ~
$ ls
15213      career  kilordle-strats  test      test.o      test.s
15410      gnu     oldFiles         test2.c   test.rs
15410-local iclab   shark-audit.txt  test.c    test.rs~
```

Shell Programs

■ Simple shell

- Described in the textbook, starting at p. 753
- Implementation of a very elementary shell
- Purpose
 - Understand what happens when you type commands
 - Understand use and operation of process control operations

Simple Shell Example

```
linux> ./shellex
> /bin/ls -l csapp.c    Must give full pathnames for programs
-rw-r--r-- 1 bryant users 23053 Jun 15  2015 csapp.c
> /bin/ps
  PID TTY          TIME CMD
 31542 pts/2        00:00:01 tcsh
 32017 pts/2        00:00:00 shellex
 32019 pts/2        00:00:00 ps
> /bin/sleep 10 &    Run program in background
32031 /bin/sleep 10 &
> /bin/ps
  PID TTY          TIME CMD
 31542 pts/2        00:00:01 tcsh
 32024 pts/2        00:00:00 emacs
 32030 pts/2        00:00:00 shellex
 32031 pts/2        00:00:00 sleep    Sleep is running
 32033 pts/2        00:00:00 ps       in background
> quit
```

Simple Shell Implementation

■ Basic loop

- Read line from command line
- Execute the requested operation
 - Built-in command (only one implemented is `quit`)
 - Load and execute program from file

```
int main(int argc, char** argv)
{
    char cmdline[MAXLINE]; /* command line */

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

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

shellex.c

*Execution is a
sequence of
read/evaluate
steps*

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
```

`parseline` will parse 'buf' into 'argv' and return whether or not input line ended in '&'

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */
```

Ignore empty lines.

Simple Shell eval Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
```

If it is a 'built in' command, then handle it here in this program. Otherwise fork/exec the program specified in argv[0]

Simple Shell eval Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* Child runs user job */
```

Create child

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* Child runs user job */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }
    }
}
```

Start `argv[0]`.
`execve` only returns on error.

Simple Shell eval Function

```

void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* Child runs user job */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
    }
}

```

If running child in foreground, wait until it is done.

shellex.c

Simple Shell eval Function

```

void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* Child runs user job */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s\n", pid, cmdline);
    }
    return;
}

```

If running child in background, print pid and continue doing other stuff.

Simple Shell eval Function

```

void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];    /* Holds modified command line */
    int bg;               /* Should the job run in bg or fg? */
    pid_t pid;            /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return; /* Ignore empty lines */

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) { /* Child runs user job */
            execve(argv[0], argv, environ);
            // If we get here, execve failed.
            printf("%s: %s\n", argv[0], strerror(errno));
            exit(127);
        }

        /* Parent waits for foreground job to terminate */
        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}

```

Oops. There is a problem with this code.

shellex.c

Problem with Simple Shell Example

■ Shell designed to run indefinitely

- Should not accumulate unneeded resources
 - Memory
 - Child processes
 - File descriptors

■ Our example shell correctly waits for and reaps foreground jobs

■ But what about background jobs?

- Will become zombies when they terminate
- Will never be reaped because shell (typically) will not terminate
- Could run the entire computer out of memory
 - More likely, run out of PIDs

Summary

■ Processes

- At any given time, system has multiple active processes
- Only one can execute at a time on any single core
- Each process appears to have total control of processor + private memory space

Summary (cont.)

■ Spawning processes

- Call `fork`
- One call, two returns

■ Process completion

- Call `exit`
- One call, no return

■ Reaping and waiting for processes

- Call `wait` or `waitpid`

■ Loading and running programs

- Call `execve` (or variant)
- One call, (normally) no return