

Exceptions, Processes and Signals

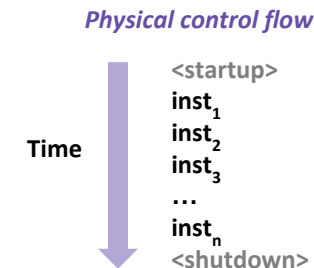
Computer Systems Organization (Spring 2017)
CSCI-UA 201, Section 3

Instructor: Joanna Klukowska

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU)

Control Flow

- Processors 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 CPU's **control flow** (or flow of control)



2

Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
 - Jumps and branches
 - Call and returnReact to changes in program state - both triggered by the program itself
- Insufficient for a useful system:
Difficult to react to changes in system state
 - Data arrives from a disk or a network adapter
 - Instruction divides by zero
 - User hits Ctrl-C at the keyboard
 - System timer expires
- System needs mechanisms for **exceptional control flow**

3

Exceptional Control Flow

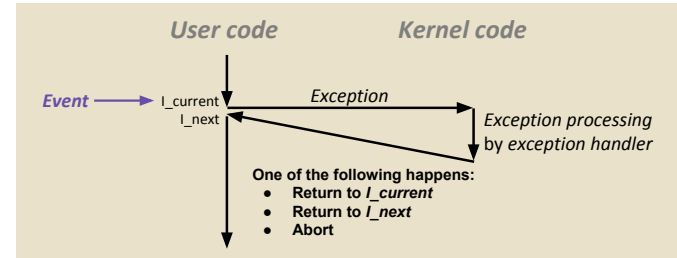
- Exists at all levels of a computer system
- Low level mechanisms
 - 1. **Exceptions**
 - Change in control flow in response to a system event (i.e., change in system state)
 - Implemented using combination of hardware and OS software
- Higher level mechanisms
 - 2. **Process context switch**
 - Implemented by OS software and hardware timer
 - 3. **Signals**
 - Implemented by OS software
 - 4. **Nonlocal jumps: setjmp() and longjmp()**
 - Implemented by C runtime library

4

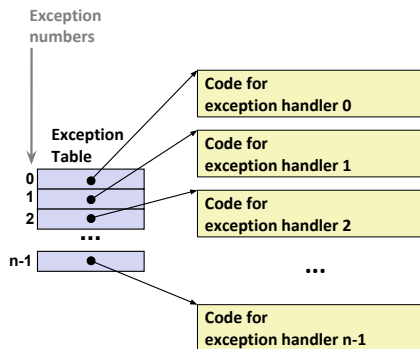
Exceptions

Exceptions

- An **exception** is a transfer of control to the OS kernel in response to some event (i.e., change in processor state)
 - Kernel is the memory-resident part of the OS
 - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
 - Indicated by setting the processor's interrupt pin
 - Handler returns to "next" instruction
- Examples:
 - Timer interrupt
 - Every few ms, an external timer chip triggers an interrupt
 - Used by the kernel to take back control from user programs
 - I/O interrupt from external device
 - Hitting Ctrl-C at the keyboard
 - Arrival of a packet from a network
 - Arrival of data from a disk

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
 - **Traps**
 - Intentional
 - Examples: system calls (requests for services from the kernel)
 - Returns control to **"next" instruction**
 - **Faults**
 - Unintentional but possibly recoverable
 - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
 - Either re-executes faulting (**"current"**) instruction or **aborts**
 - **Aborts**
 - Unintentional and unrecoverable
 - Examples: illegal instruction, parity error (data error/inconsistency detected), machine check (hardware issue detected)
 - **Aborts current program**

System Calls

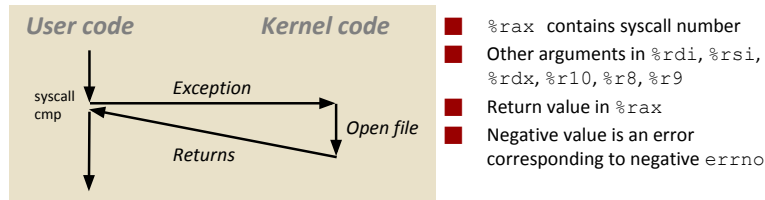
- Each x86-64 system call has a unique ID number (assigned by the operating system)
- Examples:

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

System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
0000000000e5d70 <__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 $0xffffffffffff001,%rax
...
e5dfa: c3               retq
```

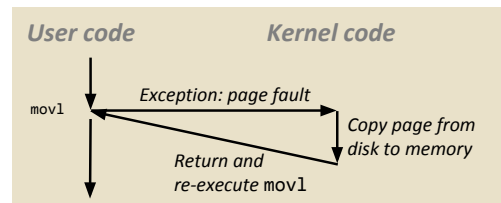


Fault Example: Page Fault

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

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

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

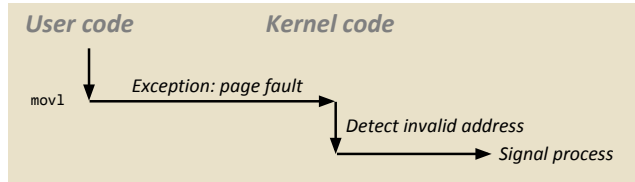


Fault Example: Invalid Memory Reference

- Sends **SIGSEGV** signal to user process
- User process exits with "segmentation fault"

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

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



13

Processes

14

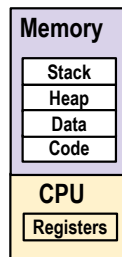
Processes

■ 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"

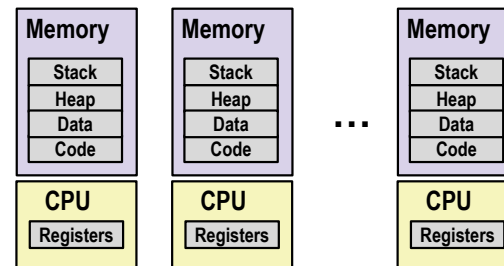
■ Process provides each program with two key abstractions:

- **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



15

Multiprocessing: The Illusion



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

16

Multiprocessing Example

```

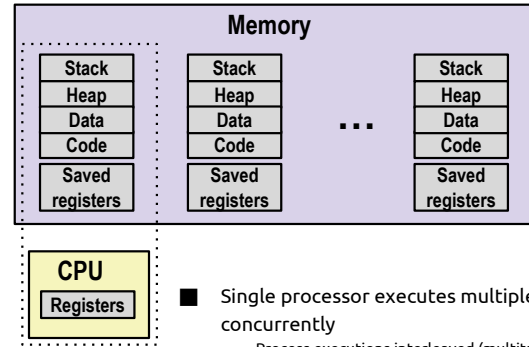
asia@asia:~$ top
top - 20:41:44 up 10 days, 11:33, 5 users, load average: 0.66, 0.85, 0.87
Tasks: 326 total, 3 running, 323 sleeping, 0 stopped, 0 zombie
%Cpu(s): 7.7 us, 3.0 sy, 0.1 ni, 89.2 id, 0.0 wa, 0.0 hi, 0.1 si, 0.0 st
KiB Mem: 16386048 total, 15656876 used, 729172 Free, 861248 buffers
KiB Swap: 34815996 total, 48164 used, 34767832 Free, 6291524 cached Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM     TIME  COMMAND
 5543 asia    20   0 1636780 357876 39500 S 31.8  2.2  28:20.55 chrome
27392 asia    20   0 926812 204128 107680 S 21.9  1.2  10:16.53 chrome
1411  root    20   0 678100 283832 75196 R 15.6  1.7   91:34.09 Xorg
3155 asia    20   0 522524 192820 94432 S 13.9  1.2   40:08.51 chrome
2414  asia    20   0 2046324 571996 99804 S 12.6  3.5  169:00.60 compiz
3082  asia    20   0 1648580 374544 105060 S 10.0  2.3   47:48.00 chrome
27762 asia    20   0 1406936 505220 86788 S  3.6  3.1   8:32.51 chrome
2330  asia    20   0 653084  9776   6780 S  2.7  0.1   1:02.96 pulseaudio
2434  asia    20   0 592468 155152 32322 S  2.3  0.9   90:21.57 skype
5238  asia    20   0 1576188 318424 83220 S  2.3  1.9  188:49.78 chrome
2167  asia    20   0 382790  39676  5752 S  1.3  0.2  11:49.75 ibus-daemon
5578  asia    20   0 652112  34648  24128 S  1.3  0.2  1:39.11 gnome-terminal
2242  asia    20   0 602420 105168 25008 S  1.0  0.6  31:14.12 unity-panel-ser
28270 asia    20   0 627668  27708 22352 S  1.0  0.2   0:00.22 gnome-screensho
  7 root    20   0  0 0 0 R 0.7 0.0  3:01.81 rcu_sched
2196  asia    20   0 489936  41008 20220 S  0.7  0.3  1:25.25 ibus-ui-gtk3
2439  asia    20   0 413900  19064 15712 S  0.7  0.1  42:53.79 indicator-multi
  33 root    rt    0  0 0 0 S 0.3 0.0  0:04.33 migration/0
  96 root    39  19  0 0 0 S 0.3 0.0  0:12.93 khugepaged
2142  asia    20   0 41068  4272  2172 S  0.3  0.0  12:38.68 dbus-daemon
2171  asia    20   0 549564  30360 21284 S  0.3  0.2   0:45.65 banfaemon
2233  asia    20   0 844188 188004 42948 S  0.3  0.8  1:44.73 hud-service
19790 asia    20   0 1283868 45788 36772 S  0.3  0.3  1:15.15 ibus-daemon
25409 root    20   0  0 0 0 S 0.3 0.0  0:00.00 kthreadd
27654 asia    20   0 1737088 255052 68772 S  0.3  1.6  1:00.00 ksoftirqd/0
28097 asia    20   0 29284  3176  2504 R  0.3  0.0  0:00.00 kworker/0:0
28393 asia    20   0 1332268 122160 68772 S  0.3  0.7  1:00.00 kworker/0:0
  1 root    20   0  33900  4068  2444 S  0.3  0.0  0:00.00 kworker/0:0
  2 root    20   0  0 0 0 S 0.3 0.0  0:00.00 kworker/0:0
  3 root    20   0  0 0 0 S 0.3 0.0  0:00.00 kworker/0:0
    
```

Running program top on my Ubuntu desktop

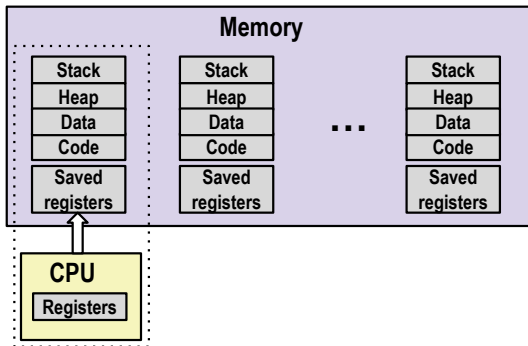
- System has 326 processes, 3 are running, 323 are sleeping
- Identified by Process ID (PID)

Multiprocessing: The One-Core Reality



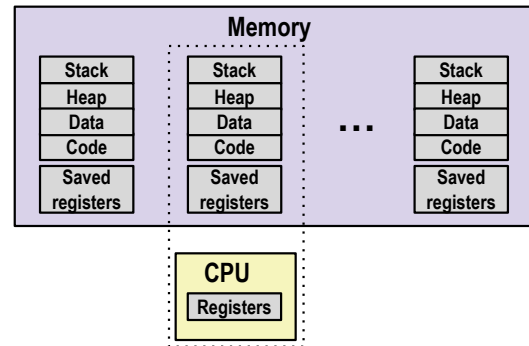
- Single processor executes multiple processes concurrently
 - Process executions interleaved (multitasking)
 - Address spaces managed by virtual memory system (later in course)
 - Register values for non-executing processes saved in memory

Multiprocessing: The One-Core Reality



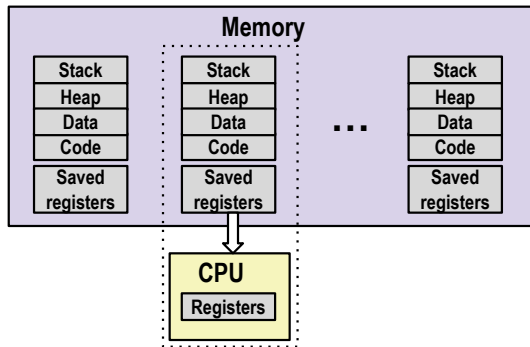
- Save current registers in memory

Multiprocessing: The One-Core Reality



- Schedule next process for execution

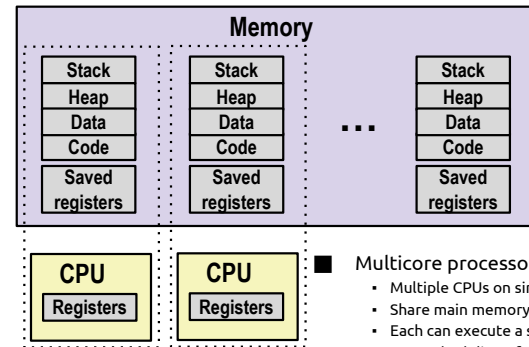
Multiprocessing: The One-Core Reality



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

21

Multiprocessing: The Multi-Core Reality

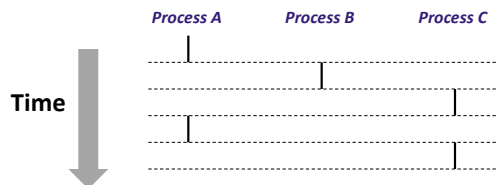


- Multicore processors
 - Multiple CPUs on single chip
 - Share main memory (and some of the caches)
 - Each can execute a separate process
 - Scheduling of processors onto cores done by kernel
 - (But we still will have more processes running than there are cores on a machine.)

22

Concurrent Processes

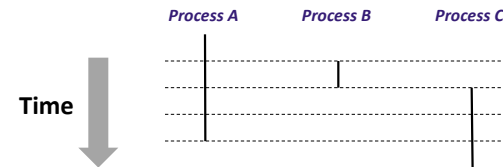
- Each process is a logical control flow.
- Two processes run **concurrently** (are concurrent) if their flows overlap in time
- Otherwise, they are **sequential**
- Examples (running on a single core):
 - Concurrent: A & B, A & C
 - Sequential: B & C



23

User View of Concurrent Processes

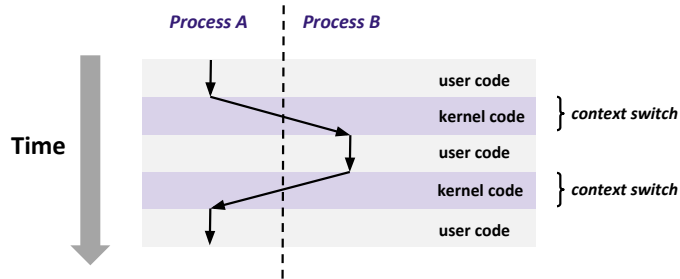
- Control flows for concurrent processes are physically disjoint in time
- However, we can think of concurrent processes as running in parallel with each other



24

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**



25

Process Control

(ways of creating, manipulating and terminating processes)

26

System Call Error Handling

- On error, Linux system-level functions typically return -1 and set global variable `errno` to indicate cause.
- Hard and fast rule:
 - You must check the return status of every system-level function
 - Only exception is the handful of functions that return void
- Example:

```
if ((pid = fork()) < 0) {  
    fprintf(stderr, "fork error: %s\n", strerror(errno));  
    exit(0);  
}
```

27

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(0);  
}
```

```
if ((pid = fork()) < 0)  
    unix_error("fork error");
```

28

Error-handling Wrappers

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

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

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

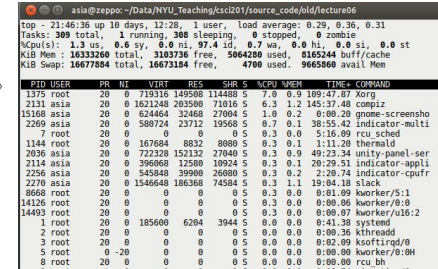
```
pid = Fork();
```

Used in the textbook. You can use those wrappers in your own code (assuming you include the book's code and header files) and on the exams. If you do not use the wrapper, you must check for errors.

29

Obtaining Process IDs

- Each process has a unique identifier known as a process id and abbreviated as PID
- `pid_t getpid(void)`
 - Returns PID of **current** process
- `pid_t getppid(void)`
 - Returns PID of **parent** process
 - The parent process is the process that created the current process.



```
asia@zeppo:~/Data/NYU_Teaching/csc201/source_code/old/lecture06
top - 21:46:36 up 10 days, 12:28, 1 user, load average: 0.29, 0.36, 0.31
Tasks: 309 total, 1 running, 388 sleeping, 0 stopped, 0 zombie
%cpu(s): 1.3 us, 0.6 sy, 0.0 ni, 97.4 id, 0.7 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 16333260 total, 3103736 free, 5864288 used, 8165244 buff/cache
KiB Swap: 16677864 total, 14673184 free, 4708 used, 9465560 avail Mem

  PID TIDTT  PPID  PFS  S  SHR  SBR  %CPU  MEM%  WCHAN
1375  root   20    0  719316 149508 114488  5  7.0  0.9 189:47.97  fork
2131  asia   20    0 1621248 203500  71016  5  6.3  1.2 145:37.48  compiz
15160  asia  20    0  624464  32460 27084  5  1.0  0.2  0:00.20  gnome-screensho
2269  asia  20    0  580724 23712 19568  5  0.7  0.1 38:55.42  indicator-multi
  7  root   20    0  0  0  0  0  0.3  0.0 5:16.09  rcu_sched
1144  root   20    0 167694  8832  8080  5  0.3  0.1 1:11.20  thermald
2036  asia  20    0  722328 152132 27040  5  0.3  0.9 49:23.34  unity-panel-ser
2114  asia  20    0  396060 12580 10924  5  0.3  0.1 20:29.51  indicator-appli
2256  asia  20    0  545848 39600 26600  5  0.3  0.2 2:20.74  indicator-cupfr
2270  asia  20    0 1546648 186368 74584  5  0.3  1.1 19:04.18  slack
8668  root  20    0  0  0  0  0  0.3  0.0 0:01.99  kworker/5:1
14126  root  20    0  0  0  0  0  0.3  0.0 0:00.06  kworker/0:0
14493  root  20    0  0  0  0  0  0.3  0.0 0:00.07  kworker/u16:2
  1  root   20    0 185600  6204  3944  5  0.0  0.0 0:41.38  systemd
  2  root   20    0  0  0  0  0  0.0  0.0 0:00.36  kthreadd
  3  root   20    0  0  0  0  0  0.0  0.0 0:02.99  ksoftirqd/0
  5  root   20    0  0  0  0  0  0.0  0.0 0:00.00  kworker/0:0H
  8  root   20    0  0  0  0  0  0.0  0.0 0:00.00  rcu_bh
```

Creating and Terminating Processes

From a programmer's perspective, we can think of a process as being in one of three states

- **Running**
 - Process is either executing, or waiting to be executed and will eventually be scheduled (i.e., chosen to execute) by the kernel
- **Stopped**
 - Process execution is suspended and will not be scheduled until further notice (next lecture when we study signals)
- **Terminated**
 - Process is stopped permanently

31

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(...)` function is called once but never returns

32

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 gets an identical (but separate) copy of the parent's virtual address space (this includes all the data on the stack and on the heap, and all the instructions).
 - Child gets identical copies of the parent's open file descriptors
 - Child has a different PID than the parent

`fork(...)` function is interesting (and often confusing) because it is called once but returns twice

33

fork Example

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

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

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

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

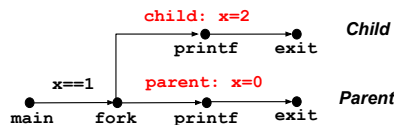
- Call once, return twice
- Concurrent execution
 - Can't predict execution order of parent and child
- Duplicate but separate address space
 - x has a value of 1 when `fork` returns in parent and child
 - subsequent changes to x are independent
- Shared open files
 - stdout is the same in both parent and child

34

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 in-edges



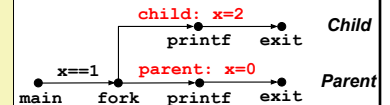
35

Process Graph Example

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

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

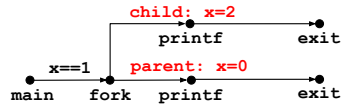
    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```



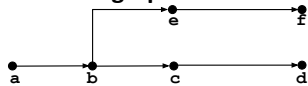
36

Interpreting Process Graphs

Original graph:



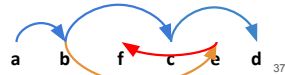
Re-labeled graph:



Feasible total ordering:

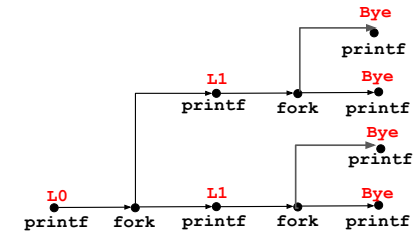


Infeasible total ordering:



fork 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
Bye
```

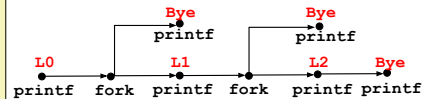
Infeasible output:

```
L0
Bye
L1
Bye
L1
Bye
Bye
```

38

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");
}
```



Feasible output:

```
L0
L1
Bye
Bye
L2
Bye
Bye
```

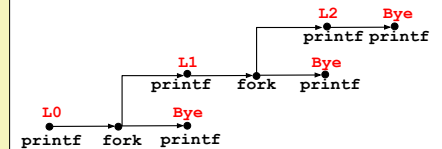
Infeasible output:

```
L0
Bye
L1
Bye
Bye
L2
```

39

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");
}
```



Feasible output:

```
L0
Bye
L1
L2
Bye
Bye
```

Infeasible output:

```
L0
Bye
L1
Bye
Bye
L2
```

40

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 (it is notified that the child process terminated and, by receiving the exit status, it acknowledges the termination)
 - 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`)
 - So, only need explicit reaping in long-running processes
 - e.g., shells and servers
 - (although you should be a good citizen and collect your zombies if possible)

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)
- `kill` command terminates a process (we cannot use `Ctrl+C`) when the process runs in a background
- Killing parent allows child to be reaped by `init`
 - Has to kill the parent
 - Cannot kill the zombie process

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 the parent process has terminated
- Must kill the child process explicitly, or else will keep running indefinitely

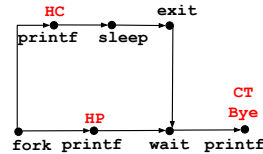
wait: Synchronizing with Children

- Parent reaps a child by calling the `wait` function
- `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 integer it points to will be set to a value that indicates reason the child terminated and the exit status:
 - checked using macros defined in `wait.h`
 - `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED`, `WSTOPSIG`, `WIFCONTINUED`
 - see textbook for details

wait: Synchronizing with Children

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        sleep(5);
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```



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

45

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

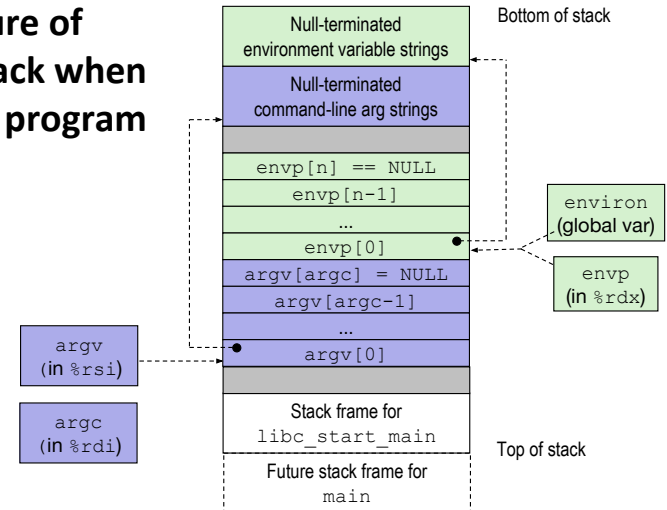
46

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, printenv`
- Overwrites code, data, and stack
 - Retains PID, open files and signal context
 - (the current process is gone, it is now running different program)
- Called once and never returns
 - ...except if there is an error

47

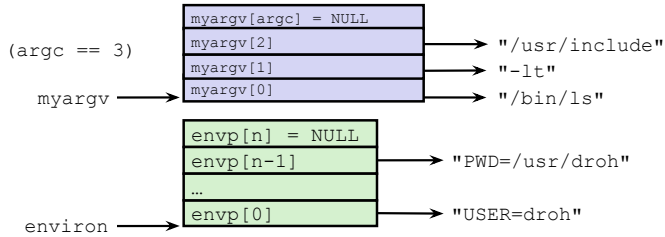
Structure of the stack when a new program starts



48

execve Example

- Executes `"/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: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

49

Summary

- Exceptions
 - Events that require nonstandard control flow
 - Generated externally (interrupts) or internally (traps and faults)
- Processes
 - At any given time, system has multiple active processes
 - Only one can execute at a time on a single core, though
 - Each process appears to have total control of processor + private memory space
- 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

50