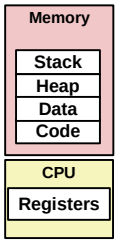
Processes

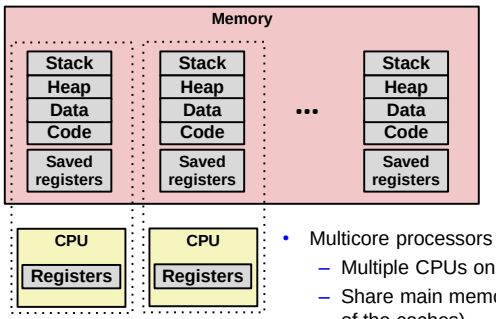
- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - Not the same as "program" or "processor"



- Process provides each program with two key abstractions:
  - ***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*

63

Multiprocessing: The (Modern) Reality

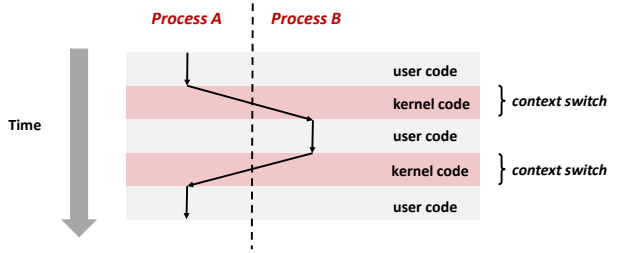


- Multicore processors
  - Multiple CPUs on single chip
  - Share main memory (and some of the caches)
  - Each can execute a separate process
    - Scheduling of processors onto cores done by kernel

64

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*

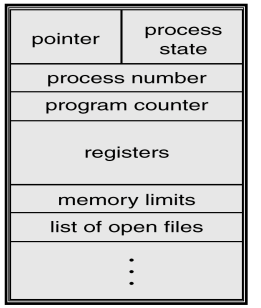


65

Process Control Block (PCB)

OS data structure (in kernel memory) maintaining information associated with each process.

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- Information about open files
- maybe kernel stack?



66

Today

- Exceptional Control Flow
- Exceptions
- Processes
- Process Control

67

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*

68

`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);
}
```

fork.c

```
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**

69

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);
}
```

70

Error-handling Wrappers

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

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

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

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

71

Obtaining Process IDs

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

72

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

73

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.

74

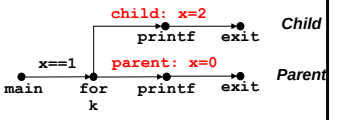
Modeling `fork` with Process Graphs

- 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

75

Process Graph Example

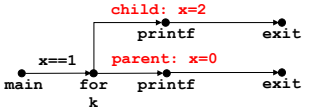
```
int main()
{
   pid_t pid;
   int x = 1;

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

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



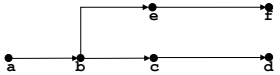
76

Interpreting Process Graphs

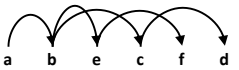
- Original graph:



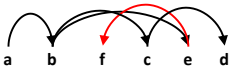
- Relabled graph:



Feasible total ordering:



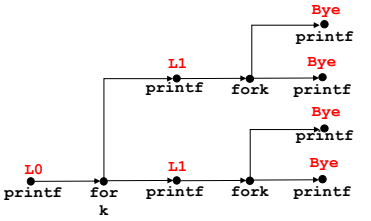
Infeasible total ordering:



77

`fork` Example: Two consecutive `fork`s

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



Feasible output:

L0
L1
Bye
Bye
L1
Bye
Bye

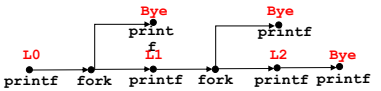
Infeasible output:

L0
Bye
L1
Bye
L1
Bye
Bye

78

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 output:

L0
L1
Bye
Bye
L2
Bye

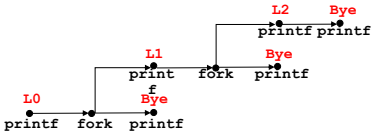
Infeasible output:

L0
Bye
L1
Bye
Bye
L2

79

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 output:

L0
Bye
L1
L2
Bye
Bye

Infeasible output:

L0
Bye
L1
Bye
Bye
L2

80

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 will be reaped by **`init`** process (pid == 1)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

81

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 */
   }
}
                                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
```

- `ps` shows child process as "defunct" (i.e., a zombie)
- Killing parent allows child to be reaped by `init`

82

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);
  }
}
```
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 child explicitly, or else will keep running indefinitely

83

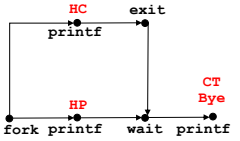
`wait`: Synchronizing with Children

- 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

84

`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:	Infeasible output:
HC	HP
HP	CT
CT	Bye
Bye	HC

85

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

86

`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

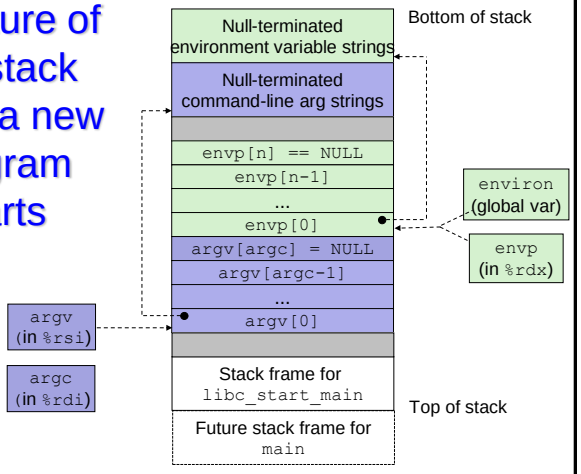
87

`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=sandhya`)
    - `getenv, putenv, printenv`
- Overwrites code, data, and stack
  - Retains PID, open files and signal context
- Called once and never returns
  - …except if there is an error

88

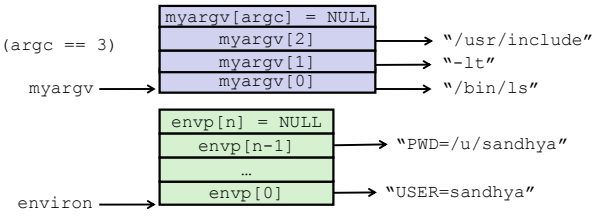
Structure of the stack when a new program starts



89

`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);
   }
}
```

90

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

91

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

92

Breakout/Quiz 7

- If the following program were executed, how many times would “hello” be printed?

```
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    exit(0);
}
```

93