# Princeton University

Computer Science 217: Introduction to Programming Systems

## Process Management



## Goals of this Lecture

Help you learn about:

- Creating new processes
- Waiting for processes to terminate
- Executing new programs
- Shell structure

Why?

- Creating new processes and executing new programs are fundamental tasks of a Unix **shell**
  - See Assignment 7
- A power programmer knows about Unix shells



## System-Level Functions

As noted in the ***Exceptions and Processes*** lecture…

Linux system-level functions for **process management**

| Number | Function | Description |
|---|---|---|
| 60 | exit() | Terminate the process |
| 57 | fork() | Create a child process |
| 7 | wait() | Wait for child process termination |
| 11 | execvp() | Execute a program in current process |
| 20 | getpid() | Return the process id of the current process |



## Agenda

**Creating new processes**

Waiting for processes to terminate

Executing new programs

Shell structure (continued next time)



## Why Create New Processes?

Why create a new process?

- Scenario 1: Program wants to run an additional instance of itself
  - E.g., **web server** receives request; creates additional instance of itself to handle the request; original instance continues listening for requests
- Scenario 2: Program wants to run a different program
  - E.g., **shell** receives a command; creates an additional instance of itself; additional instance overwrites itself with requested program to handle command; original instance continues listening for commands

How to create a new process?

- A "parent" process **forks** a "child" process
- (Optionally) child process overwrite itself with a new program



## `fork` System-Level Function

```
pid_t fork(void);
```

- Create a new process by duplicating the calling process
- New (child) process is an exact duplicate of the calling (parent) process
- In the child, return 0
- In the parent, return the process id of the child

fork() is called once in parent process

fork() returns twice

- Once in parent process
- Once in child process

## Creating New Processes

Parent process and child process run **concurrently**
- Two CPUs available ⇒
  - Parent process and child process run in **parallel**
- Fewer than two CPUs available ⇒
  - Parent process and child process run **serially**
  - OS provides the **illusion** of parallel execution
    - OS causes context switches between the two processes
    - (Recall ***Exceptions and Processes*** lecture)

Reality: Each CourseLab computer has 24 CPUs

Simplifying assumption: there is only one CPU
- We'll speak of "which process gets **the** CPU"

7

## Simple `fork` Example

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

What is the output?

8

## Simple `fork` Example Trace 1 (1)

Parent prints "one"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

9

## Simple `fork` Example Trace 1 (2)

Parent forks child

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

10

## Simple `fork` Example Trace 1 (3)

OS gives CPU to child; child prints "two"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

11

## Simple `fork` Example Trace 1 (4)

Child exits

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

12

## Simple `fork` Example Trace 1 (5)

OS gives CPU to parent; parent prints "two"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

13

## Simple `fork` Example Trace 1 (6)

OS gives CPU to parent; parent prints "two"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

14

## Simple `fork` Example Trace 1 Output

Output:

```
one
two
two
```

From parent

From child

From parent

15

## Simple `fork` Example Trace 2 (1)

Parent prints "one"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

16

## Simple `fork` Example Trace 2 (2)

Parent forks child

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

17

## Simple `fork` Example Trace 2 (3)

OS gives CPU to parent; parent prints "two"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

18

## Simple `fork` Example Trace 2 (4)

Parent exits

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

19

## Simple `fork` Example Trace 2 (5)

OS gives CPU to child; child prints "two"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

20

## Simple `fork` Example Trace 2 (6)

Child exits

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

21

## Simple `fork` Example Trace 2 Output

Output:

```
one
two
two
```

From parent
From parent
From child

22

## Fact 1: `fork` and Process State

Immediately after **`fork()`**, parent and child have identical but distinct process states

- Contents of registers
- Contents of memory
- File descriptor tables
  - (Relevant later)
- Etc.
  - See Bryant & O'Hallaron book for details

23

## Fact 2: `fork` and Process Ids

Any process has a unique nonnegative integer id

- Parent process and child processes have different process ids
- No process has process id 0

24

## Fact 3: `fork` and Return Values

Return value of fork has meaning
- In child, `fork()` returns 0
- In parent, `fork()` returns process id of child

```
pid = fork();
if (pid == 0)
{
   /* in child */
   …
}
else
{
   /* in parent */
   …
}
```

25

## `fork` Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

What is the output?

26

## `fork` Example Trace 1 (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

27

## `fork` Example Trace 1 (2)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

28

## `fork` Example Trace 1 (3)

Assume OS gives CPU to child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

0

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

29

## `fork` Example Trace 1 (4)

Child decrements its x, and prints "child: 0"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 0

30

## fork Example Trace 1 (5)

Child exits; OS gives CPU to parent

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <sys/types.h>
int  main(void)
{  pid_t  pid;
   int  x = 1;

   pid  = fork();
   if  (pid  == 0)
   {  x--;
      printf("child:  %d\n",  x);
      exit(0);
   }
   else
   {  x++;
      printf("parent:  %d\n",  x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <sys/types.h>
int  main(void)
{  pid_t  pid;
   int  x = 1;

   pid  = fork();
   if  (pid  == 0)
   {  x--;
      printf("child:  %d\n",  x);
      exit(0);
   }
   else
   {  x++;
      printf("parent:  %d\n",  x);
      exit(0);
   }
}
```

x = 0

31

## fork Example Trace 1 (6)

In parent, fork() returns process id of child

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <sys/types.h>
int  main(void)
{  pid_t  pid;
   int  x = 1;

   pid  = fork();
   if  (pid  == 0)
   {  x--;
      printf("child:  %d\n",  x);
      exit(0);
   }
   else
   {  x++;
      printf("parent:  %d\n",  x);
      exit(0);
   }
}
```

x = 1

Process id of child

32

## fork Example Trace 1 (7)

Parent increments its x, and prints “parent: 2”

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <sys/types.h>
int  main(void)
{  pid_t  pid;
   int  x = 1;

   pid  = fork();
   if  (pid  == 0)
   {  x--;
      printf("child:  %d\n",  x);
      exit(0);
   }
   else
   {  x++;
      printf("parent:  %d\n",  x);
      exit(0);
   }
}
```

x = 2

33

## fork Example Trace 1 (8)

Parent exits

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <sys/types.h>
int  main(void)
{  pid_t  pid;
   int  x = 1;

   pid  = fork();
   if  (pid  == 0)
   {  x--;
      printf("child:  %d\n",  x);
      exit(0);
   }
   else
   {  x++;
      printf("parent:  %d\n",  x);
      exit(0);
   }
}
```

x = 2

34

## fork Example Trace 1 Output

Example trace 1 output:

```
Child:  0
Parent:  2
```

35

## fork Example Trace 2 (1)

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <sys/types.h>
int  main(void)
{  pid_t  pid;
   int  x = 1;

   pid  = fork();
   if  (pid  == 0)
   {  x--;
      printf("child:  %d\n",  x);
      exit(0);
   }
   else
   {  x++;
      printf("parent:  %d\n",  x);
      exit(0);
   }
}
```

x = 1

36

## `fork` Example Trace 2 (2)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

37

## `fork` Example Trace 2 (3)

Assume OS gives CPU to parent

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

Process ID of child

x = 1

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

38

## `fork` Example Trace 2 (4)

Parent increments its x and prints "parent: 2"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 2

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

39

## `fork` Example Trace 2 (5)

Parent exits; OS gives CPU to child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 2

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

40

## `fork` Example Trace 2 (6)

In child, fork() returns 0

0

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

41

## `fork` Example Trace 2 (7)

Child decrements its x and prints "child: 0"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 0

42

## `fork` Example Trace 2 (8)

Child exits

```
#include   <stdio.h>
#include   <stdlib.h>
#include   <unistd.h>
#include   <sys/types.h>
int  main(void)
{   pid_t  pid;
    int  x = 1;

    pid = fork();
    if (pid == 0)
    {  x--;
       printf("child:  %d\n",  x);
       exit(0);
    }
    else
    {  x++;
       printf("parent:  %d\n",  x);
       exit(0);
    }
}
```

x = 0

43

## `fork` Example Trace 2 Output

Example trace 2 output:

```
Parent:  2
Child:  0
```

44

## Agenda

Creating new processes

**Waiting for processes to terminate**

Executing new programs

Shell structure (continued next time)

45

## `wait` System-Level Function

Problem:

- How to control execution order?

Solution:

- Parent should call `wait()`
- (child is a "zombie" until parent does the `wait()`, so the parent should **harvest** (or **reap**) its children... more later)

`pid_t wait(int *status);`

- Suspends execution of the calling process until one of its children terminates
- If status is not NULL, stores status information in the int to which it points; this integer can be inspected with macros [see man page for details].
- On success, returns the process ID of the terminated child
- On error, returns -1

Paraphrasing man page

46

## `wait` Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

What is the output?

47

## `wait` Example Trace 1 (1)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

48

## `wait` Example Trace 1 (2)

OS gives CPU to parent

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

49

## `wait` Example Trace 1 (3)

Parent calls `wait()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

50

## `wait` Example Trace 1 (4)

OS gives CPU to child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

51

## `wait` Example Trace 1 (5)

Child prints "child" and exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

52

## `wait` Example Trace 1 (6)

Parent returns from call of wait(), prints "parent", exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

53

## `wait` Example Trace 1 Output

Example trace 1 output

```
child
parent
```

54

## `wait` Example Trace 2 (1)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

55

## `wait` Example Trace 2 (2)

OS gives CPU to child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

56

## `wait` Example Trace 2 (3)

Child prints "child" and exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

57

## `wait` Example Trace 2 (4)

OS gives CPU to parent

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

58

## `wait` Example Trace 2 (5)

Parent calls `wait()`; returns immediately

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

59

## `wait` Example Trace 2 (6)

Parent prints "parent" and exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

60

## `wait` Example Trace 2 Output

Example trace 2 output

```
child
parent
```

Same as trace 1 output!



## Aside: Orphans and Zombies

Question:

- What happens if parent process does not wait for (reap/harvest) child process?

Answer 1:

- In shell, could cause sequencing problems
- E.g, parent process running shell writes prompt for next command before current command is finished executing

Answer 2:

- In general, child process becomes **zombie** and/or **orphan**



## Aside: Orphans and Zombies

**Orphan**

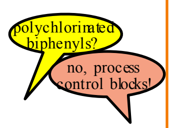
- A process that has no parent

**Zombie**

- A process that has terminated but has not been waited for (reaped)

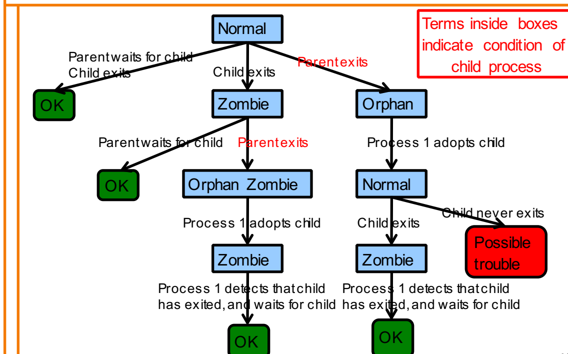
Orphans and zombies

- Clutter Unix data structures unnecessarily
  - OS maintains unnecessary PCBs
- Can become long-running processes
  - Consume CPU time unnecessarily






## Aside: Orphans and Zombies






## Agenda

Creating new processes

Waiting for processes to terminate

**Executing new programs**

Shell structure (continued next time)



## `execvp` System-Level Function

Problem: How to execute a new program?

- Usually, in the newly-created child process

Solution: **`execvp()`**

```
int execvp(const char *file, char *const argv[]);
```

- Replaces the current process image with a new process image
- Provides an array of pointers to null-terminated strings that represent the argument list available to the new program
  - The first argument, by convention, should point to the filename associated with the file being executed
  - The array of pointers must be terminated by a NULL pointer

Paraphrasing man page

## execvp System-Level Function

Example: Execute "cat readme"

```
char *newCmd;
char *newArgv[3];
newCmd = "cat";
newArgv[0] = "cat";
newArgv[1] = "readme";
newArgv[2] = NULL;
execvp(newCmd, newArgv);
```

- First argument: name of program to be executed
- Second argument: argv to be passed to main() of new program
  - Must begin with program name, end with NULL

67

## execvp Failure

**fork()**
- If successful, returns **two** times
  - Once in parent
  - Once in child

```
char *newCmd;
char *newArgv[3];
newCmd = "cat";
newArgv[0] = "cat";
newArgv[1] = "readme";
newArgv[2] = NULL;
execvp(newCmd, newArgv);
fprintf(stderr, "exec failed\n");
exit(EXIT_FAILURE);
```

**execvp()**
- If successful, returns **zero** times
  - Calling program is overwritten with new program
- Corollary:
  - If **execvp()** returns, then it must have failed

68

## execvp Example

```
$ cat readme
This is my
readme file.
```

69

## execvp Example

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{  char *newCmd;
   char *newArgv[3];
   newCmd = "cat";
   newArgv[0] = "cat";
   newArgv[1] = "readme";
   newArgv[2] = NULL;
   execvp(newCmd, newArgv);
   fprintf(stderr, "exec failed\n");
   return EXIT_FAILURE;
}
```

What is the output?

70

## execvp Example Trace (1)

Process creates arguments to be passed to **execvp()**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{  char *newCmd;
   char *newArgv[3];
   newCmd = "cat";
   newArgv[0] = "cat";
   newArgv[1] = "readme";
   newArgv[2] = NULL;
   execvp(newCmd, newArgv);
   fprintf(stderr, "exec failed\n");
   return EXIT_FAILURE;
}
```

71

## execvp Example Trace (2)

Process executes **execvp()**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{  char *newCmd;
   char *newArgv[3];
   newCmd = "cat";
   newArgv[0] = "cat";
   newArgv[1] = "readme";
   newArgv[2] = NULL;
   execvp(newCmd, newArgv);
   fprintf(stderr, "exec failed\n");
   return EXIT_FAILURE;
}
```

72

## `execvp` Example Trace (3)

`cat` program executes in same process

```
cat program

with argv array:
   argv[0] = "cat"
   argv[1] = "readme"
   argv[2] = NULL
```

73

## `execvp` Example Trace (4)

`cat` program writes “This is my\nreadme file.”

```
cat program

with argv array:
   argv[0] = "cat"
   argv[1] = "readme"
   argv[2] = NULL
```

74

## `execvp` Example Trace (5)

`cat` program terminates

```
cat program

with argv array:
   argv[0] = "cat"
   argv[1] = "readme"
   argv[2] = NULL
```

75

## `execvp` Example Trace (6)

Output

```
This is my
readme file.
```

76

## Agenda

Creating new processes

Waiting for processes to terminate

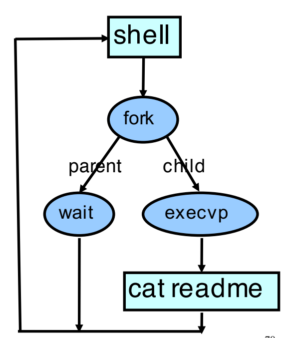
Executing new programs

**Shell structure (continued next time)**

77

## Shell Structure

- Parent (shell) reads & parses the command line
  - E.g., “cat readme”
- Parent forks child
- Parent waits
- Child calls execvp to execute command
- Child exits
- Parent returns from wait
- Parent repeats




78

## Simple Shell Code

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

79

## Simple Shell Trace (1)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

Parent reads and parses command line
Parent assigns values to **somepgm** and **someargv**

80

## Simple Shell Trace (2)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

**fork()** creates child process
Which process gets the CPU first? Let's assume the parent…

81

## Simple Shell Trace (3)

Parent Process

child's pid

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

In parent, pid != 0; parent waits; OS gives CPU to child

82

## Simple Shell Trace (4)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

0

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

In child, pid == 0; child calls **execvp()**

83

## Simple Shell Trace (5)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

```
somepgm
With someargv
as argv param
```

In child, somepgm overwrites shell program;
**main()** is called with **someargv** as **argv** parameter

84

## Simple Shell Trace (6)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

~~*somepgm*~~
~~*With someargv*~~
~~*as argv param*~~

Somepgm executes in child, and eventually exits

85

## Simple Shell Trace (7)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

Parent returns from **`wait()`** and repeats

86

## Background processes

Unix shell lets you run a process "in the background"

**`$ compute <my-input >my-output &`**

How it's implemented in the shell:

Don't wait() after the fork!

But: must clean up zombie processes

**`waitpid(0, &status, WNOHANG)`** (more info: "man 2 wait")

When to do it?

Every time around the main loop, or

When parent receives the SIGCHLD signal.

One or the other, don't need both!

## Aside: `system` Function

Common combination of operations
- **`fork()`** to create a new child process
- **`execvp()`** to execute new program in child process
- **`wait()`** in the parent process for the child to complete

Single call that combines all three
- **`int system(const char *cmd);`**

Example

```
#include <stdlib.h>
int main(void)
{  system("cat readme");
   return 0;
}
```

88

## Aside: `system` Function

Question:
- Why not use **`system()`** instead of **`fork()/execvp()/wait()`** in Assignment 7 shell?

Shallow answer:
- Assignment requirements!

Deeper answer:
- Using **`system()`**, shell could not handle **signals** as specified
- See ***Signals*** reference notes

89

## Aside: `fork` Efficiency

Question:
- **`fork()`** duplicates an entire process (text, bss, data, rodata, stack, heap sections)
- Isn't that *very* inefficient???!!!

Answer:
- Using virtual memory, not really!
- Upon **`fork()`**, OS creates virtual pages for child process
- Each child virtual page maps to physical page (in memory or on disk) of parent
- OS duplicates physical pages incrementally, and only if/when "write" occurs ("copy-on-write")

90

## Aside: `exec` Efficiency

Question:
- **`execvp()`** loads a new program from disk into memory
- Isn't that somewhat inefficient?

Answer:
- Using virtual memory, not really!
- Upon **`execvp()`**, OS changes process's virtual page table to point to pages on disk containing the new program
- As page faults occur, OS swaps pages of new program into memory incrementally as needed

91

## Aside: `fork/exec` Efficiency

The bottom line…

**`fork()`** and **`execvp()`** are efficient
- Because they were designed with virtual memory in mind!

Commentary: A **beautiful** intersection of three **beautiful** abstractions

92

## Assignment 7 Suggestion

A shell is mostly a big loop
- **Read char array** from `stdin`
- **Lexically** analyze char array to create **token array**
- **Parse** token array to create **command**
- **Execute** command
  - Fork child process
  - Parent:
    - Wait for child to terminate
  - Child:
    - Exec new program

Start with code from earlier slides and from precepts
- And edit until it becomes a Unix shell!

93

## Summary

Creating new processes
- **`fork()`**

Executing new programs
- **`execvp()`**

Waiting for processes to terminate
- **`wait()`**

Shell structure
- Combination of **`fork()`**, **`execvp()`**, **`wait()`**

94