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

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

System-Level Functions

As noted in the Exceptions and Processes lecture...

Linux system-level functions for process management

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>exit()</td>
<td>Terminate the process</td>
</tr>
<tr>
<td>57</td>
<td>fork()</td>
<td>Create a child process</td>
</tr>
<tr>
<td>7</td>
<td>wait()</td>
<td>Wait for child process termination</td>
</tr>
<tr>
<td>11</td>
<td>execvp()</td>
<td>Execute a program in current process</td>
</tr>
<tr>
<td>20</td>
<td>getpid()</td>
<td>Return the process id of the current process</td>
</tr>
</tbody>
</table>

Agenda

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

fork System-Level Function

```c
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”

Simple fork Example

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

What is the output?

Simple fork Example Trace 1 (1)

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

Parent prints “one”

Execute concurrently

Simple fork Example Trace 1 (2)

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

Parent forks child

OS gives CPU to child; child prints “two”

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

Child exits

Execute concurrently

Simple fork Example Trace 1 (4)

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

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

Execlute concurrently
Simple fork Example Trace 1 (5)

OS gives CPU to parent; parent prints “two”

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

Simple fork Example Trace 1 (6)

OS gives CPU to parent; parent prints “two”

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

Simple fork Example Trace 1 Output

Output:

```
one
```

From parent

```
two
```

From child

```
two
```

From parent

Simple fork Example Trace 2 (1)

Parent prints “one”

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

Simple fork Example Trace 2 (2)

Parent forks child

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

Executing concurrently

Simple fork Example Trace 2 (3)

OS gives CPU to parent; parent prints “two”

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

Executing concurrently
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

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;
}

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;
}

Simple fork Example Trace 2 Output

Output:

one
two
From parent
From parent
From child

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

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
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

```c
pid = fork();
if (pid == 0)
    { /* in child */
    ... }   // x = 1
else
    { /* in parent */
    ... }
```

fork Example

```c
#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 = 1
        { x--;     // x = 0
          printf("child: %d\n", x);
          exit(0);  // x = 0
        }
      else
        { x++;      // x = 2
          printf("parent: %d\n", x);
          exit(0);  // x = 2
        }
    }
```

fork Example Trace 1 (1)

```c
#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 = 1
        { x--;     // x = 0
          printf("child: %d\n", x);
          exit(0);  // x = 0
        }
      else
        { x++;      // x = 2
          printf("parent: %d\n", x);
          exit(0);  // x = 2
        }
    }
```

fork Example Trace 1 (2)

```c
#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 = 1
        { x--;     // x = 0
          printf("child: %d\n", x);
          exit(0);  // x = 0
        }
      else
        { x++;      // x = 2
          printf("parent: %d\n", x);
          exit(0);  // x = 2
        }
    }
```

fork Example Trace 1 (3)

```c
#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 = 1
        { x--;     // x = 0
          printf("child: %d\n", x);
          exit(0);  // x = 0
        }
      else
        { x++;      // x = 2
          printf("parent: %d\n", x);
          exit(0);  // x = 2
        }
    }
```

fork Example Trace 1 (4)

```c
#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 = 1
        { x--;     // x = 0
          printf("child: %d\n", x);
          exit(0);  // x = 0
        }
      else
        { x++;      // x = 2
          printf("parent: %d\n", x);
          exit(0);  // x = 2
        }
    }
```
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
", x);
      exit(0);
  }
  else
  {  x++;
      printf("parent: %d
", x);
      exit(0);
  }
}
```

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
", x);
      exit(0);
  }
  else
  {  x++;
      printf("parent: %d
", x);
      exit(0);
  }
}
```

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
", x);
      exit(0);
  }
  else
  {  x++;
      printf("parent: %d
", x);
      exit(0);
  }
}
```

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
", x);
      exit(0);
  }
  else
  {  x++;
      printf("parent: %d
", x);
      exit(0);
  }
}
```

fork Example 1 Output

Example trace 1 output:

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

fork Example 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
", x);
      exit(0);
  }
  else
  {  x++;
      printf("parent: %d
", x);
      exit(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);
    }
}

// Examples of fork() usage

// Example Trace 2 (2)
Parent forks child

// Example Trace 2 (3)
Assume OS gives CPU to parent

// Example Trace 2 (4)
Parent increments its x and prints "parent: 2"

// Example Trace 2 (5)
Parent exits; OS gives CPU to child

// Example Trace 2 (6)
In child, fork() returns 0

// Example Trace 2 (7)
Child decrements its x and prints "child: 0"
fork Example Trace 2 (8)

```c
#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
", x);
       exit(0);
    }
    else
    {  x++;
       printf("parent: %d
", x);
       exit(0);
    }
}
```

Example trace 2 output:

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

fork Example Trace 2 Output

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)
  ```c
  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

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
");
       exit(0);
    }
    wait(NULL);
    printf("parent
");
    return 0;
}
```

Wait Example Trace 1 (1)

```
#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;
}
```

Agenda

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

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;
}
```

Executing concurrently
Example Trace 1 (2)

OS gives CPU to parent

```c
#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

Example Trace 1 (3)

Parent calls `wait()`

```c
#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

Example Trace 1 (4)

OS gives CPU to child

```c
#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

Example Trace 1 (5)

Child prints “child” and exits

```c
#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

Example Trace 1 (6)

Parent returns from call of `wait()`, prints “parent”, exits

```c
#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;
}
```

Example trace 1 output
wait Example Trace 2 (1)

Parent forks child

```c
#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;
}
```

wait Example Trace 2 (2)

OS gives CPU to child

```c
#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

wait Example Trace 2 (3)

Child prints “child” and exits

```c
#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;
}
```

wait Example Trace 2 (4)

OS gives CPU to parent

```c
#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

wait Example Trace 2 (5)

Parent calls `wait()`; returns immediately

```c
#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;
}
```

wait Example Trace 2 (6)

Parent prints “parent” and exits

```c
#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;
}
```
**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

**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

**Agenda**

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

**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
execvp System-Level Function

Example: Execute “cat readme”

```c
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

execvp Failure

```c
fork()
- If successful, returns two times
  - Once in parent
  - Once in child
execvp()
- If successful, returns zero times
  - Calling program is overwritten with new program
  - Corollary:
    - If execvp() returns, then it must have failed
```

execvp Example

```c
#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;
}
```

execvp Example Trace (1)

Process creates arguments to be passed to execvp()

```c
#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;
}
```

execvp Example Trace (2)

Process executes execvp()

```c
#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;
}
```
execvp Example Trace (3)

*cat* program executes in same process

```c
#include <stdio.h>

int main() {
    printf("cat program\n");
    return 0;
}
```

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

execvp Example Trace (4)

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

```c
#include <stdio.h>

int main() {
    printf("cat program\n");
    return 0;
}
```

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

execvp Example Trace (5)

*cat* program terminates

```c
#include <stdio.h>

int main() {
    printf("cat program\n");
    return 0;
}
```

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

execvp Example Trace (6)

Output

```
This is my\nreadme file.
```

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
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
```

Simple Shell Trace (1)

```
Parent Process (1)
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
```

Simple Shell Trace (2)

```
Parent Process (2)
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
```

Simple Shell Trace (3)

```
Parent Process (3)
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
```

```
Child Process (3)
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
```

Simple Shell Trace (4)

```
Parent Process (4)
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
```

```
Child Process (4)
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, somepgm overwrites shell program;
main() is called with someargv as argv parameter
```

Simple Shell Trace (5)

```
Parent Process (5)
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
```

```
Child Process (5)
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…

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

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

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

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

```
In child, somepgm overwrites shell program;
main() is called with someargv as argv parameter
```
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
");
    exit(EXIT_FAILURE);
} /* in parent */
wait(NULL);
Repeat the previous

Child Process

somepgm
With someargv
as argv param

Executing concurrently

Somepgm executes in child, and eventually exits

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
");
    exit(EXIT_FAILURE);
} /* in parent */
wait(NULL);
Repeat the previous

Parent returns from wait() and repeats

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.

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

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

Aside: fork Efficiency

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

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")
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

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

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!

Summary

Creating new processes
  - `fork()`

Executing new programs
  - `execvp()`

Waiting for processes to terminate
  - `wait()`

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