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 many utilities and end-user applications
  - Assignment 7...

System-Level Functions
As noted in the Exceptions and Processes lecture...
Linux system-level functions for process management

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exit()</td>
<td>Terminate the process</td>
</tr>
<tr>
<td>fork()</td>
<td>Create a child process</td>
</tr>
<tr>
<td>wait()</td>
<td>Wait for child process termination</td>
</tr>
<tr>
<td>execvp()</td>
<td>Execute a program in current process</td>
</tr>
<tr>
<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

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 overwrites itself with a new program, after performing appropriate setup

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

Reality: Each ArmLab computer has 96 CPUs
Simplifying assumption: there is only one CPU
- We'll speak of "which process gets the CPU"
- But which process gets the CPU first? Unknown!

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)

Parent prints "one"

```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 1 (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 1 (3)

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;
}
```

Executing concurrently

Simple fork Example Trace 1 (4)

Child exits

```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 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;
}
```

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;
}
```

Simple fork Example Trace 1 Output

Output:

```
one
two
two
```

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;
}
```

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

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
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

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

iClicker Question
Q: What is the output?
A. child: 0
   parent: 2
B. parent: 2
   child: 0
C. child: 0
   parent: 1
D. A or B
E. A or C

```
```
fork Example
```
```
fork Example Trace 1 (1)
```
```
fork Example Trace 1 (2)
```
```
fork Example Trace 1 (3)
```
```
fork Example Trace 1 (3)
```
```
Assume OS gives CPU to child
```
```
Executing concurrently
```
```
Executing concurrently
```
```
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
", 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
", 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
", 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
", x);
        exit(0);
    }
    else
    {  x++;
        printf("parent: %d
", x);
        exit(0);
    }
}
Example Trace 2 (1)

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main(void) {
    pid_t pid;
    int x = 1;
    pid = fork();
    if (pid == 0) {
        x++;
        exit(0);
    } else {
        printf("parent: %d
", x);
        printf("child: %d
", x);
        exit(0);
    }
}
```

Example Trace 2 (2)

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main(void) {
    pid_t pid;
    int x = 1;
    pid = fork();
    if (pid == 0) {
        x++;
        exit(0);
    } else {
        printf("parent: %d
", x);
        printf("child: %d
", x);
        exit(0);
    }
}
```

Example Trace 2 (3)

Assume OS gives CPU to parent

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main(void) {
    pid_t pid;
    int x = 1;
    pid = fork();
    if (pid == 0) {
        x++;
        printf("child: %d
", x);
    } else {
        printf("parent: %d
", x);
        wait(0);
    }
}
```

Example Trace 2 (4)

Parent increments its x and prints "parent: 2"

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.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 {
        printf("parent: %d
", x);
        printf("child: %d
", x);
        exit(0);
    }
}
```

Example Trace 2 (5)

Parent exits; OS gives CPU to child

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main(void) {
    pid_t pid;
    int x = 1;
    pid = fork();
    if (pid == 0) {
        x++;
        printf("child: %d
", x);
    } else {
        printf("parent: %d
", x);
        wait(0);
    }
}
```

Example Trace 2 (6)

In child, fork() returns 0

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.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 {
        printf("parent: %d
", x);
        wait(0);
    }
}
```
# Problem

**Problem:**
- How to control execution order?

**Solution:**
- Parent calls `wait()`:
  ```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.
    - (a child that has exited is a "zombie" until parent does the `wait()`, so the parent should `wait` (or `reap`) its children... more later)

## wait System-Level Function

**Example**

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

What is the output?
Q: What is the output of this program?

A. child
B. parent
C. something other than A or B
D. A or B
E. A or C

```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;
}
```

Parent forks child

OS gives CPU to parent

OS gives CPU to child

Child prints "child" and exits
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;
}

Example Trace 1 Output

Example trace 1 output

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;
}

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

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;
}

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;
}
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;
  }
```

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;
  }
```

Example Trace 2 Output

Example trace 2 output

```
child
parent
```

Same as trace 1 output!

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

Terms inside boxes indicate condition of child process
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()

```c
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 Example

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

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

execvp Example Trace (4)
cat program writes "This is my/readme file."
```c
```
cat program
with argv array:
    argv[0] = "cat"
    argv[1] = "readme"
    argv[2] = NULL
```

execvp Example Trace (5)
cat program terminates
```c
```
cat program
with argv array:
    argv[0] = "cat"
    argv[1] = "readme"
    argv[2] = NULL
```

execvp Example Trace (6)
Output
```
This is my
readme file.
```
Agenda

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

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

Shell Structure Diagram:

```
shell
    ↓
   fork
       ↓
      parent
        ↓
       child
        ↓
      wait
         ↓
      execvp
         ↓
cat readme
```

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
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
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
```

fork() creates child process
Which process gets the CPU first? Let’s assume the parent...
In child, pid == 0; child calls `execvp()`.

Somepgm executes in child, and eventually exits.

Parent returns from `wait()` and repeats.

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:

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

Question:
• Why not use system() instead of fork()/execvp()/wait() in applications (e.g. Assignment 7)?

Shallow answer:
• Assignment requirements!

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

Even deeper answer:
• fork/exec allows arbitrary setup for child between fork and exec
  • cf. CreateProcess() on Windows, which has a zillion params

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”)

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()