Processes
Where We Are in the Course...

Goal:
- Brief overview of “greatest hits”
- You won’t be responsible for details
- Whet your appetite for COS 316, 318, 375
- Today, processes and exceptions; next week, VM and caching
Agenda

Processes
- Illusion: Private address space
- Illusion: Private control flow

Process management in C
- Creating new processes
- Waiting for termination
- Executing new programs

Unix Process Control
- Exceptions
- Signals
Processes

Program
- Executable code
- A static entity

Process
- An instance of a program in execution
- A dynamic entity: has a time dimension
- Each process runs one program
  - E.g. the process with Process ID 12345 might be running emacs
- One program can run in multiple processes
  - E.g. PID 12345 might be running emacs, and PID 23456 might also be running emacs – for the same user or for a different user
Processes Significance

Process abstraction provides two key illusions:

• Processes believe they have a *private address space*
• Processes believe they have *private control flow*

*Process is a profound abstraction in computer science*
Private Address Space: Illusion

Hardware and OS give each application process the illusion that it is the *only* process using memory

- Enables multiple simultaneous instances of one program!
All processes use the same physical memory.
Hardware and OS provide programs with a virtual view of memory, i.e. virtual memory (VM) (Details in VM lecture next week!)
Private Control Flow: Illusion

Simplifying assumption: only one CPU / core

Hardware and OS give each application process the illusion that it is the *only* process running on the CPU
Multiple processes are **time-sliced** to run concurrently.

OS occasionally **preempts** running process to give other processes their fair share of CPU time.
Process Status

More specifically...

At any time, a process has a status:

- **Running**: a CPU is executing instructions for the process
- **Ready**: Process is ready for OS to assign it to a CPU
- **Blocked**: Process is waiting for some requested service (typically I/O) to finish

Modern machines may have multiple CPUs or “cores”, but the same principles apply if \#processes > \#cores

- For simplicity, we will speak of “the” CPU
Process Status Transitions

Scheduled for execution: OS selects some process from ready set and assigns CPU to it.
Time slice expired: OS moves running process to ready set because process consumed its fair share of CPU time.
Service requested: OS moves running process to blocked set because it requested a (time consuming) system service (often I/O).
Service finished: OS moves blocked process to ready set because the requested service finished.
Throughout its lifetime, a process’s status switches between running, ready, and blocked.
Each process has a **context**

- The process’s state, that is...
- Contents of registers (internal CPU state – we’ll talk a lot about these later in the semester)
- Memory contents
  - TEXT, RODATA, DATA, BSS, HEAP, and STACK
Context Switch

**Context switch:**
- OS saves context of running process
- OS loads context of some ready process
- OS passes control to newly restored process

**Diagram:**
- Process X:
  - Running
  - Save X context
  - Load Y context
  - Ready
  - Running

- Process Y:
  - Running
  - Save Y context
  - Load X context
  - Ready
Agenda

Processes
- Illusion: Private address space
- Illusion: Private control flow

Process management in C
- Creating new processes
- Waiting for termination
- Executing new programs

Unix Process Control
- Exceptions
- Signals
## System-Level Process Management Functions

<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>
</tbody>
</table>
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

```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
  * Almost – the call to `fork` has a different return value (wait 1 slide)

fork() is called once in parent process

fork() returns twice
  - Once in parent process
  - Once in child process
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) {
    /* executed in child */
    ...
} else {
    /* executed in parent */
    ...
}
```
Programs With 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 time-sliced to run **serially**
  - OS provides the **illusion** of parallel execution

Reality: Each ArmLab computer has 96 CPUs
- But each student who is logged in might be concurrently running sshd, bash, emacs, make, gcc217, etc.

Simplifying assumption: there is only one CPU
- We’ll speak of “which process gets the **CPU**”
- But which process gets the CPU first? **Unknown**!
```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;
}
```
### 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

```c
#include <stdio.h>
#include <unistd.h>
int main(void)
{   printf("one\n");
    fork();
    printf("two\n");
    return 0;
}
```
Simple fork Example Trace 1 (3)

OS gives CPU to child; child prints “two”
Child exits

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

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

Executing concurrently
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
two
two
```

From parent

From child

From parent
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    printf("one\n");
    fork();
    printf("two\n");
    return 0;
}
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    printf("one\n");
    fork();
    printf("two\n");
    return 0;
}
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;
}
```
```c
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    printf("one\n");
    fork();
    printf("two\n");
    return 0;
}
```
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;
}
```
Simple fork Example Trace 2 (6)

Child exits

```c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    printf("one\n");
    fork();
    printf("two\n");
    return 0;
}

```
Output:

```
|   one   |
|---|---|
| two |
| two |
```

- From parent
- From parent
- From child
Q: What is the output of this program?

A. child: 0
   parent: 2

B. parent: 2
   child: 0

C. child: 0
   parent: 1

D. parent: 2
   child: 1

E. A or B

#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);
    }
}

The answer is E.

See following slides.
```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\n", x);
        exit(0);
    }
    else
    {
        x++;
        printf("parent: %d\n", x);
        exit(0);
    }
}
```
### fork Example Trace 1 (2)

**Parent forks child**

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

**Executing concurrently**

```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\n", x);
        exit(0);
    }
    else
    {
        x++;
        printf("parent: %d\n", x);
        exit(0);
    }
}
```
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);
    }
}
```
fork Example Trace 1 (4)

Child decrements its x, and prints “child: 0”

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

Executing concurrently

```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\n", x);
        exit(0);
    }
    else
    {
        x++;
        printf("parent: %d\n", x);
        exit(0);
    }
}
```
fork Example Trace 1 (5)

Child exits; OS gives CPU to parent

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

---

Exeuting concurrently

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

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);
    }
}

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);
    }
}
fork Example Trace 1 Output

Example trace 1 output:

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

 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);
    }
}
```
fork Example Trace 2 (3)

Assume OS gives CPU to parent

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

Parent exits; OS gives CPU to child

Executing concurrently
In child, `fork()` returns 0

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

`x = 1`
Child decrements its x and prints “child: 0”

```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\n", x);
        exit(0);
    }
    else
    {
        x++;
        printf("parent: %d\n", 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);
    }
}
Example trace 2 output:

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

```
armlab01:~$ for i in `seq 1 10000`; do ./fpe | head -n 1; done | sort | uniq -c
  56 child: 0
  9944 parent: 2
```
**wait System-Level Function**

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
Q: What is the output of this program?

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

The answer is A.  
See following slides.
wait Example Trace 1 (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;
}
```

Executing concurrently

```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 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;
}
```

```
#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 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;
}
```

```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 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
wait 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;
}
```

```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
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;
}
```
| child | parent |
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;
}
```

Executing concurrently
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

```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 (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;
}
```

Executing concurrently
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;
}
```
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;
}
```
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

```
child
parent
```

Same as trace 1 output!
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 program’s filename
    • The array of pointers must be terminated by a NULL pointer

Paraphrasing man page
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

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

```c
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);
```
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;
}
```
Shell Structure

- Parent (shell) reads & parses the command line
- Parent forks child and waits
- Child calls execvp to execute command
- Child exits, parent returns from wait and repeats

```c
while (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);
}
```
Agenda

Processes
- Illusion: Private address space
- Illusion: Private control flow

Process management in C
- Creating new processes
- Waiting for termination
- Executing new programs

Unix Process Control
- Exceptions
- Signals
Exception

- An abrupt change in control flow of a running program corresponding to a change in process state

*Note: Exceptions in OS ≠ exceptions in Java

Implemented using

**try/catch** and **throw** statements
Synchronous Exceptions

Some exceptions are **synchronous**

- Occur as result of actions of executing program
- Examples:
  - **System call**: Application requests I/O
  - **System call**: Application requests more heap memory
  - Application pgm attempts integer division by 0
  - Application pgm attempts to access privileged memory
  - Application pgm accesses variable that is not in physical memory
Asynchronous Exceptions

Some exceptions are **asynchronous**

- Do not occur (directly) as result of actions of executing program
- Examples:
  - User presses key on keyboard
  - Packet received over network
  - Disk controller finishes reading data
  - Hardware timer expires
Exceptional Control Flow

Application program

Exception handler in operating system

exception

exception return (sometimes)

exception handler
Exceptions vs. Function Calls

Handling an exception is **similar to** calling a function
- Control transfers from original code to other code
- Other code executes
- Control returns to some instruction in original code

Handling an exception is **different from** calling a function
- CPU saves **additional data**
  - E.g. values of all registers
- CPU pushes data onto **OS’s stack**, not application pgm’s stack
- Handler runs in **kernel/privileged mode**, not in **user mode**
  - Handler can execute all instructions and access all memory
- Control **might return** to some instruction in original code
  - Sometimes control returns to **next** instruction
  - Sometimes control returns to **current** instruction
  - Sometimes control does not return at all!
Classes of Exceptions

There are 4 classes of exceptions...
(1) Interrupts

**Application program**

(1) CPU interrupt pin goes high

**Exception handler**

(2) After current instr finishes, control passes to exception handler

(3) Exception handler runs

(4) Exception handler returns control to next instr

**Occurs when:** External (off-CPU) device requests attention

**Examples:**
- User presses key
- Disk controller finishes reading/writing data
- Network packet arrives
(2) Traps

**Occurs when:** Application pgm requests OS service

**Examples:**
- Application pgm requests I/O
- Application pgm requests more heap memory

Traps provide a function-call-like interface between application pgm and OS.
(3) Faults

**Occurs when:** Application pgm causes a (possibly recoverable) error

**Examples:**
- Application pgm divides by 0
- Application pgm accesses privileged memory (seg fault)
- Application pgm accesses data that is not in physical memory (page fault)
(4) Aborts

Occurs when: HW detects a non-recoverable error

Example:
Parity check indicates corruption of memory bit (overheating, cosmic ray!, etc.)
### Summary of Exception Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Occurs when</th>
<th>Asynch/Synch</th>
<th>Return Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>External device requests attention</td>
<td>Asynch</td>
<td>Return to next instr</td>
</tr>
<tr>
<td>Trap</td>
<td>Application pgm requests OS service</td>
<td>Synch</td>
<td>Return to next instr</td>
</tr>
<tr>
<td>Fault</td>
<td>Application pgm causes (maybe recoverable) error</td>
<td>Synch</td>
<td>Return to current instr (maybe)</td>
</tr>
<tr>
<td>Abort</td>
<td>HW detects non-recoverable error</td>
<td>Synch-ish</td>
<td>Do not return</td>
</tr>
</tbody>
</table>
Process Control Examples

Exactly what happens when you:

Type Ctrl-c?
• Keystroke generates interrupt
• OS handles interrupt
• OS sends process a 2/SIGINT signal

Type Ctrl-z?
• Keystroke generates interrupt
• OS handles interrupt
• OS sends process a 20/SIGTSTP signal
Signals Overview

**Signal**: A notification of an exception

Typical signal sequence:

- Process P is executing
- Exception occurs (interrupt, trap, fault, or abort)
- OS gains control of CPU
- OS wishes to inform process P that something happened
- OS sends a signal to process P
  - OS sets a bit in **pending** bit vector of process P
  - Indicates that OS is sending a signal of type X to process P
  - A signal of type X is **pending** for process P
Signals Overview (cont.)

Typical signal sequence (cont.):

• Sometime later...
• OS is ready to give CPU back to process P
• OS checks pending for process P, sees that signal of type X is pending
• OS forces process P to receive signal of type X
  • OS clears bit in process P’s pending
• Process P executes action for signal of type X
  • Normally process P executes default action for that signal
  • If signal handler was installed for signal of type X, then process P executes signal handler
  • Action might terminate process P; otherwise...
• Process P resumes where it left off
User can send three signals from keyboard:

- **Ctrl-c** ⇒ **2/SIGINT** signal
  - Default action is “terminate”

- **Ctrl-z** ⇒ **20/SIGTSTP** signal
  - Default action is “stop until next 18/SIGCONT”

- **Ctrl-\** ⇒ **3/SIGQUIT** signal
  - Default action is “terminate”
Examples of Non-keyboard Signals

Process makes illegal memory reference

- Segmentation fault occurs
- OS gains control of CPU
- OS sends 11/SIGSEGV signal to process
- Process receives 11/SIGSEGV signal
- Default action for 11/SIGSEGV signal is “terminate”

https://xkcd.com/371/
Signals, signals everywhere

List of the predefined signals, learn many details with these commands:

```
$ kill -l
 1) SIGHUP
 3) SIGQUIT
 5) SIGTRAP
 7) SIGBUS
 9) SIGKILL
11) SIGSEGV
13) SIGPIPE
15) SIGTERM
18) SIGSTOP
22) SIGTTOU
26) SIGVTALRM
30) SIGPWR
40) SIGRTMIN+6
44) SIGRTMIN+10
48) SIGRTMIN+14
52) SIGRTMAX-12
56) SIGRTMAX-8
60) SIGRTMAX-4
64) SIGRTMAX
 2) SIGINT
 4) SIGILL
 6) SIGABRT
 8) SIGFPE
10) SIGUSR1
12) SIGUSR2
14) SIGALRM
17) SIGCHLD
23) SIGURG
28) SIGWINCH
31) SIGSYS
34) SIGRTMIN
37) SIGRTMIN+3
41) SIGRTMIN+6
45) SIGRTMIN+10
49) SIGRTMIN+14
53) SIGRTMAX-12
57) SIGRTMAX-8
61) SIGRTMAX-4
 3) SIGQUIT
 7) SIGBUS
11) SIGSEGV
14) SIGALRM
17) SIGCHLD
24) SIGXCPU
28) SIGWINCH
34) SIGRTMIN
38) SIGRTMIN+4
42) SIGRTMIN+7
46) SIGRTMIN+11
50) SIGRTMIN+15
54) SIGRTMAX-11
58) SIGRTMAX-7
62) SIGRTMAX-3
 4) SIGILL
 8) SIGFPE
12) SIGUSR2
15) SIGTERM
20) SIGTSTP
24) SIGXCPU
28) SIGWINCH
35) SIGRTMIN
39) SIGRTMIN+5
43) SIGRTMIN+8
47) SIGRTMIN+12
51) SIGRTMAX-14
55) SIGRTMAX-10
59) SIGRTMAX-6
63) SIGRTMAX-1
```

$ man 7 signal
Installing a Signal Handler

**signal() function**

- `sighandler_t signal(int iSig, sighandler_t pfHandler);`
  - Install function `pfHandler` as the handler for signals of type `iSig`
  - `pfHandler` is a function pointer:
    ```c
    typedef void (*sighandler_t)(int);
    ```
  - Return the old handler on success, `SIG_ERR` on error
  - After call, `(*pfHandler)` is invoked whenever process receives a signal of type `iSig`
Signal Handling Example 1

Program testsignal.c:

```c
#define _GNU_SOURCE /* Use modern handling style */
#include <stdio.h>
#include <signal.h>

static void myHandler(int iSig)
{
    printf("In myHandler with argument %d\n", iSig);
}

int main(void)
{
    signal(SIGINT, myHandler);
    printf("Entering an infinite loop\n");
    for (;;)
    {
        return 0; /* Never get here. */
    }
```
Program generates lots of temporary data

- Stores the data in a temporary file
- Must delete the file before exiting

```c
... int main(void)
{
    FILE *psFile;
    psFile = fopen("temp.txt", "w");
    ...
    fclose(psFile);
    remove("temp.txt");
    return 0;
}
```
Example 2 Problem

What if user types Ctrl-c?
• OS sends a 2/SIGINT signal to the process
• Default action for 2/SIGINT is “terminate”

Problem: The temporary file is not deleted
• Process terminates before remove("temp.txt") is executed

Challenge: Ctrl-c could happen at any time
• Which line of code will be interrupted???

Solution: Install a signal handler
• Define a “clean up” function to delete the file
• Install the function as a signal handler for 2/SIGINT
Example 2 Solution

```c
...  
static FILE *psFile; /* Must be global. */
static void cleanup(int iSig)
{
    fclose(psFile);
    remove("temp.txt");
    exit(0);
}
int main(void)
{
    ...  
    psFile = fopen("temp.txt", "w");
    signal(SIGINT, cleanup);
    ...
    cleanup(0); /* or raise(SIGINT); */
    return 0; /* Never get here. */
}
```