Processes
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
- Signals
- Alarms
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

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 different users

```
arlab02:~$ cat /proc/sys/kernel/pid_max
98304
```
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
Hardware and OS give each application process the illusion that it is the only process using memory

- Enables multiple simultaneous instances of one program!
Private Address Space: Reality

All processes use the same physical memory. Hardware and OS provide programs with a virtual view of memory, i.e. virtual memory (VM)

Memory is divided into pages
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 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.
Context Switch

**Context switch:**

- OS saves context of running process
- OS loads context of some ready process
- OS passes control to newly restored process
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
- Signals
- Alarms
### 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
- * Exception: the return value of the call to fork (wait 1 slide)

fork() is called once in parent process

fork() returns twice
- Once in parent process
- Once in child process
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 */
    ...
}
else {
    /* 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 run **serially**
  - OS provides the **illusion** of parallel execution
    - OS causes context switches between the two processes
    - (Recall *Exceptions and Processes* lecture)

**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?
#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;
}

Parent forks child

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;
}
```

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

From parent

From child

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;
}
```
#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”

```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

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
    printf("one\n");
    fork();
    printf("two\n");
    return 0;
}  
```
Simple `fork` Example Trace 2 (5)

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;
}
```
Simple fork Example Trace 2 Output

Output:

```
one
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

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);
    }
}
```
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);
    }
}
Assume OS gives CPU to 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);
    }
}
```
#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

#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
In parent, `fork()` returns process id of 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);
    }
}
```

Process id of child

```
x = 1
```
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>`

```c
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 exits

fork Example Trace 1 (8)

Example Trace 1 (8)
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);
    }
}
```
Parent forks child

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

int main(void)
{
    int x = 1;
    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);
    }
}
```

Executing concurrently

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

int main(void)
{
    int x = 1;
    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);
    }
}
```
fork Example Trace 2 (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--;
        printf("child: %d\n", x);
        exit(0);
    } else {
        x++;
        printf("parent: %d\n", x);
        exit(0);
    }
}
```

Assume OS gives CPU to parent

Process ID of child

Executing concurrently

`x = 1`
Parent increments its x and prints “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);
    }
}
```

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 2 (5)

Parent exits; OS gives CPU to 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);
    }
}
```
fork Example Trace 2 (6)

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);
    }
}
```
Child decrements its x and prints “child: 0”
fork Example Trace 2 (8)

Child exits

```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 Output

Example trace 2 output:

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

```
armlab01:~/Test$ for i in `seq 1 10000`; do ./fpe | head -n 1; done | sort | uniq -c
   56 child: 0
  9944 parent: 2
```
Q: Must we do `exit(0)` instead of `return 0;` here?

A. Yes, the program will not work with return statements
B. No, but it’s good programming practice for forking programs
C. No, but we need to in some other forking programs
D. No, this is actually a bug in this program and should be `return 0` instead

```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);
    }
}
```

B or C

(Consider if this fork were in a deeply nested function stack, not main.)
wait System-Level Function

Problem:
• How to control execution order?

Solution:
• Parent calls \texttt{wait()}

\texttt{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 \texttt{wait()}, so the parent should harvest (or reap) its children... more later)
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
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;
}
```

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(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

```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

```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 (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

```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 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

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

```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

```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
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;
}
```
Parent calls \texttt{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;
}
```
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
Aside: Orphans and Zombies

Terms inside boxes indicate condition of child process.

- Normal
  - Parent waits for child
  - Child exits
  - Parent exits

- Zombie
  - Parent waits for child
  - Parent exits

- Orphan扎
  - Process 1 adopts child
  - Process 1 detects that child has exited, and waits for child

- Orphan Zombie
  - Process 1 adopts child
  - Process 1 detects that child has exited, and waits for child

- Zombie
  - Child exits
  - Child never exits

Terms inside boxes indicate condition of child process.
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 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);
```
$ cat readme
This is my readme file.

#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;
}
```
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;
}
```
cat program executes in same process

```c
// cat program

with argv array:
    argv[0] = "cat"
    argv[1] = "readme"
    argv[2] = NULL
```
cat program writes "This is my\nreadme file."

cat program

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

cat program terminates

```
cat program

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

Output

This is my readme file.
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
• 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

```c
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`
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

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

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()
In child, `somepgm` overwrites shell program;
`main()` is called with `someargv` as `argv` parameter.
Somepgm executes in child, and eventually exits

```c
// 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

// Child Process
somepgm
With someargv
as argv param
executing concurrently
```
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
Aside: 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.
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
- Signals
- Alarms
Process Control Examples

Exactly what happens when you:

Type Ctrl-c?

• Keystroke generates interrupt
• OS handles interrupt
• OS sends a 2/SIGINT signal

Type Ctrl-z?

• Keystroke generates interrupt
• OS handles interrupt
• OS sends a 20/SIGTSTP signal
Sending Signals via Keystrokes

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 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
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 any signal by executing command:

`kill` command
- `kill -sig pid`
- Send a signal of type `sig` to process `pid`
- No `-sig` option specified ⇒ sends 15/SIGTERM signal
  - Default action for 15/SIGTERM is “terminate”
- You must own process `pid` (or have admin privileges)
- Commentary: Better command name would be `sendsig`

Examples
- `kill -2 1234`
- `kill -SIGINT 1234`
  - Same as pressing Ctrl-c if process 1234 is running in foreground
- `kill -2 %1`
  - Same as above, if process 1234 is running as background job 1
Exactly what happens when you:

Issue a `kill -sig pid` command?
- `kill` command executes `trap`
- OS handles trap
- OS sends a `sig` `signal` to the process whose id is `pid`

Issue a `fg` or `bg` command?
- `fg` or `bg` command executes `trap`
- OS handles trap
- OS sends a 18/SIGCONT `signal` (and does some other things too!)
Signals signals everywhere

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
<td>2</td>
<td>SIGINT</td>
<td>3</td>
<td>SIGQUIT</td>
<td>4</td>
<td>SIGILL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
<td>6</td>
<td>SIGABRT</td>
<td>7</td>
<td>SIGBUS</td>
<td>8</td>
<td>SIGFPE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>10</td>
<td>SIGUSR1</td>
<td>11</td>
<td>SIGSEGV</td>
<td>12</td>
<td>SIGUSR2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
<td>14</td>
<td>SIGALRM</td>
<td>15</td>
<td>SIGTERM</td>
<td>17</td>
<td>SIGCHLD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>SIGCONT</td>
<td>19</td>
<td>SIGSTOP</td>
<td>20</td>
<td>SIGTSTP</td>
<td>21</td>
<td>SIGTIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>SIGTTOU</td>
<td>23</td>
<td>SIGURG</td>
<td>24</td>
<td>SIGXCPU</td>
<td>25</td>
<td>SIGXFSZ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>SIGVTALRM</td>
<td>27</td>
<td>SIGPROF</td>
<td>28</td>
<td>SIGWINCH</td>
<td>29</td>
<td>SIGIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>SIGPWR</td>
<td>31</td>
<td>SIGSYS</td>
<td>34</td>
<td>SIGRTMIN+1</td>
<td>35</td>
<td>SIGRTMIN+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>SIGRTMIN+2</td>
<td>37</td>
<td>SIGRTMIN+3</td>
<td>38</td>
<td>SIGRTMIN+4</td>
<td>39</td>
<td>SIGRTMIN+5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>SIGRTMIN+6</td>
<td>41</td>
<td>SIGRTMIN+7</td>
<td>42</td>
<td>SIGRTMIN+8</td>
<td>43</td>
<td>SIGRTMIN+9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>SIGRTMIN+10</td>
<td>45</td>
<td>SIGRTMIN+11</td>
<td>46</td>
<td>SIGRTMIN+12</td>
<td>47</td>
<td>SIGRTMIN+13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>SIGRTMIN+14</td>
<td>49</td>
<td>SIGRTMIN+15</td>
<td>50</td>
<td>SIGRTMAX-14</td>
<td>51</td>
<td>SIGRTMAX-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>SIGRTMAX-12</td>
<td>53</td>
<td>SIGRTMAX-11</td>
<td>54</td>
<td>SIGRTMAX-10</td>
<td>55</td>
<td>SIGRTMAX-9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>SIGRTMAX-8</td>
<td>57</td>
<td>SIGRTMAX-7</td>
<td>58</td>
<td>SIGRTMAX-6</td>
<td>59</td>
<td>SIGRTMAX-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>SIGRTMAX-4</td>
<td>61</td>
<td>SIGRTMAX-3</td>
<td>62</td>
<td>SIGRTMAX-2</td>
<td>63</td>
<td>SIGRTMAX-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$ kill -l
$ man 7 signal

See Bryant & O’Hallaron book for more actions, triggering exceptions, and how the application program can define signals with unused values.
Sending Signals via Function Calls

Program can send any signal by calling function:

**raise() function**

- int raise(int iSig);
- Commands OS to send a signal of type iSig to calling process
- Returns 0 to indicate success, non-0 to indicate failure

**Example:**

- iRet = raise(SIGINT);
  - Send a 2/SIGINT signal to calling process

**One clever use case:**

**Sending Signals via Function Calls**

**kill() function**
- int kill(pid_t iPid, int iSig);
- Sends a iSig signal to the process iPid
- Equivalent to raise(iSig) when iPid is the id of current process
- You must own process pid (or have admin privileges)
- Commentary: Better function name would be sendsig()

**Example**
- iRet = kill(1234, SIGINT);
  - Send a 2/SIGINT signal to process 1234
Handling Signals

Each signal type has a default action
• For most signal types, default action is “terminate”

A program can install a signal handler
• To change action of (almost) any signal type
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`
SIG_DFL

Predefined value: **SIG_DFL**

Use as argument to `signal()` to **restore default action**

```c
int main(void)
{
    signal(SIGINT, somehandler);
    ...
    signal(SIGINT, SIG_DFL);
    ...
}
```

Subsequently, process will handle 2/SIGINT signals using default action for 2/SIGINT signals (“terminate”).
SIG_IGN

Predefined value: **SIG_IGN**

Use as argument to `signal()` to ignore signals

```c
int main(void)
{
    ...  
    signal(SIGINT, SIG_IGN);
    ...  
}
```

Subsequently, process will ignore 2/SIGINT signals
Uncatchable Signals

Special cases: A program cannot install a signal handler for signals of type:

- **9/SIGKILL**
  - Default action is “terminate”

- **19/SIGSTOP**
  - Default action is “stop until next 18/SIGCONT”
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. */
}
```

Error handling code omitted in this and all subsequent programs in this lecture
Program testsignalall.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <signal.h>

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

int main(void)
{
    int i;
    /* Install myHandler as the handler
       for all kinds of signals. */
    for (i = 1; i < 65; i++)
        signal(i, myHandler);
    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0; /* Never get here. */
}
```

Will fail:
- `signal(9, myHandler)`
- `signal(19, myHandler)`
- `signal(32, myHandler)`
- `signal(33, myHandler)`

```
$ ./a.out
signal 9 not handled
signal 19 not handled
signal 32 not handled
signal 33 not handled
Entering an infinite loop
^C
```

```
^Z
```

```
Killed
```

```
$ ps axu | grep 'a.out'
cmoretti 64220 101 0.0 2432 ...
$ kill -9 64220
```
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 3 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 3 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. */  
}  
```
Alarms

alarm() function
  • unsigned int alarm(unsigned int uiSec);
  • Send 14/SIGALRM signal after uiSec seconds
  • Cancel pending alarm if uiSec is 0
  • Use **wall-clock time**
    • Time spent executing other processes counts
    • Time spent waiting for user input counts
  • Return value is irrelevant for our purposes

Used to implement time-outs
Alarm Example 1

Program testalarm.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{
    printf("In myHandler with argument %d\n", iSig);
    alarm(2); /* Set another alarm */
}

int main(void)
{
    signal(SIGALRM, myHandler);
    alarm(2); /* Set an alarm. */
    printf("Entering an infinite loop\n");
    for (;;)
    {
        return 0; /* Never get here. */
    }
}  
```
Alarm Example 1

Program testalarm.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{
    printf("In myHandler with argument \%d\n", iSig);
    alarm(2); /* Set another alarm */
}

int main(void)
{
    signal(SIGALRM, myHandler);
    alarm(2); /* Set an alarm */
    printf("Entering an infinite loop\n");
    for (;;)
    {
        return 0; /* Never get here. */
    }
}
```
Program testalarmtimeout.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void
myHandler(int iSig)
{
    printf("\nSorry. You took too long.\n");
    exit(EXIT_FAILURE);
}

int main(void)
{
    int i;
    signal(SIGALRM, myHandler);
    printf("Enter a number: ");
    alarm(5);
    scanf("%d", &i);
    alarm(0);
    printf("You entered the number %d.\n", i);
    return 0;
}
```
Program testalarmtimeout.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{
    printf("Sorry. You took too long.\n");
    exit(EXIT_FAILURE);
}

int main(void)
{
    int i;
    signal(SIGALRM, myHandler);
    printf("Enter a number: ");
    alarm(5);
    scanf("%d", &i);
    alarm(0);
    printf("You entered the number %d.\n", i);
    return 0;
}
```

```
armlab01:~/Test$ echo 5 | > ./a.out
Enter a number: 
You entered the number 5.

armlab01:~/Test$ (sleep 10; > echo 5) | > ./a.out
Enter a number: 
Sorry. You took too long.
```
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**
  - Signals
  - Alarms