Forking Processes

<u>Computer Systems</u> Chapter 8.2, 8.4

Naming Clients

- Need a name/handle for each running program
  - Can't be program name, because I can run the same program concurrently
  - Must be created when program starts
  - Must be deleted when program ends

- ***process*** - An invocation of a program
  - Process ID: a numeric identifier associated with a process (PID)
  - C Standard library function calls can create new processes
  - Ended by "exit" library call (in stdlib.h)

What's in a process?

- Logical Control Flow
  - A process executes instructions
  - EIP points to the next instruction to execute
  - After an instruction is fetched, EIP points to the next sequential instruction
  - Control flow instructions modify EIP (jump, call, ret, etc.)
- Address Space
  - Memory starting at address 0x0000 0000 up to 0xFFFF FFFF
  - Contains OS, Code, Heap, Stack, bss, global data, shared libraries, etc.
- Registers / Register Values
- IO resources

Abstract, Single Process View

A process executes a single stream of sequential instructions on data in a single address space with a single set of registers. The process also controls the set of open IO resources.

Process Hierarchy

- Processes can create new processes
  - The creator is called the ***parent process*** or “ppid”
  - The spawned process is called a ***child process***
- Parent processes are responsible for their children
- In UNIX, when you log on, the OS process creates a child process and assigns that process to you
  - This is the interactive shell or GUI running on your behalf

Operating System Process Status Table

- Keeps track of every process
- Process added to OS process status table when a parent spawns a child process
- The child process is alive (running) as long as it continues to execute instructions
- When a child exits (or is killed), it becomes “dead”, **but it is still in the process table!**
  - Process table holds the return code from the process
- Process is removed from the OS process table when the parent “reaps” the process (reads the child’s return code)

Forks

When you “fork” a single process…

- A second process is created… a *child* of the existing process
- The process doing the forking is the *parent* process
- At the point of the fork, the parent’s address space is cloned
  - The child gets a FULL COPY of the parent’s address space
- At the point of the fork, the parent’s IO resources are cloned
  - The child inherits a copy of the parent’s IO resources
- At the point of the fork, the parent’s register values are cloned
  - Including EIP!

What does “clone” mean?

- Start out identical…

- … but as time goes by, clones diverge…

After the Fork

- Two independent copies of memory that start out identical, but diverge as parent and child write different things in their memory
- Two independent copies of IO resources that start out pointing to single IO resources, but may diverge as parent and child manipulate these resources independently
- Two independent copies of Register values that start out identical, but diverge as parent and child write different values
- **No communication between parent and child through memory!**
- Parent is still responsible for child.

How can you tell child from parent?

- Memory is cloned… parent and child are the same
- Register values are cloned… parent and child are the same
  - Same %EIP implies the same instruction(s) are executing
- The ONLY difference between parent and child is the return value from the “fork” function
  - %eax register is different!
  - For the parent, the “fork” function returns the PID of the child
  - For the child, the “fork” function returns zero (0)
    - Zero is not a valid PID

fork standard library call

```
#include <unistd.h>

pid_t pid;
...
pid = fork();
if (pid==0) { // This is the child
...
} else { // This is the parent... pid is the child pid
...
}
```

Only parent executes...
No child yet.

Both parent and child execute independently from here on...

Cleaning Up After Your Kids

- When a child process exits, it posts its return code
  - BUT IT STAYS ACTIVE
  - Must stay active until it's parent process reaps the child's return code
- Parent must read the return code from its children
  - Reading the return code is called *reaping* the child process
  - When a child process has been reaped, it can be removed from OS tables
  - Reaping a child process can be done with either "wait" or "waitpid" C system library calls
    - "wait" – allows you to reap any child process
    - "waitpid" – allows you to reap a specific child process
  - Both "wait" and "waitpid" make parent go idle until child exits

wait standard library function

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t pid;
int childStatus;
…
pid=wait(&childStatus);  // Blocks until any child pid’s status changes
if (pid==-1) { … } // error… no children to reap
else printf(“Child pid %d exited with status %d\n”,pid,childStatus);
```

waitpid standard library function

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t pid;
int childStatus;
pid=fork();
…
if (pid) {
   pid=waitpid(pid,&childStatus,0);  // Blocks until pid's status changes
   if (pid==-1) { … } // error… pid not an unreaped child
   printf("Child pid %d exited with status %d\n",pid,childStatus);
}
```

Forking Example

```
int common=do_common_calculation(); int cstat=-1;
pid_t pid = fork();
if (pid==0) { /* Child process */
        int resultA=do_A_calculation(common);
        printf("The result of A is %d\n",resultA);
} else { /* Parent process */
        int resultB=do_B_calculation(common);
        printf("The result of B is %d\n",resultB);
        wait(&cstat);
}
printf("%s is done\n",pid==0?"child":"parent");
exit(0);
```

This runs once

These run simultaneously

Wait for child to exit

This runs twice!

Forking Example

```
int common=do_common_calculation();
int cstat=-1; pid_t pid = fork();
```

Parent (pid=5673)

```
if (pid==0) { …
} else { /* Parent process */
        int resultB=do_B_calculation(common);
        printf("The result of B is %d\n",resultB);



        wait(&cstat);}
printf("%s is done\n",pid==0?"child":"parent");
exit(0);
```

Child (pid=4783)

```
if (pid==0) { /* Child process */
        int resultA=do_A_calculation(common);
        printf("The result of A is %d\n",resultA);
} else { …
}
printf("%s is done\n",pid==0?"child":"parent");
exit(0);
```

```
common=12
pid=4783
cstat=0
resultB=18
```

```
unix> ./forker
The result of B is 18
The result of A is 42
child is done
parent is done
unix>
```

```
common=12
pid=0
cstat=-1
resultA=42
```

"Automatic" clean-up

- C "exit" processing performs automated clean-up:
  - closes any files you have left open
  - free's any space you have malloc'ed
  - waits for any unreaped children
- Automatic clean-up is frowned on!
  - What happens if you never get there?
  - It might take days before the parent exits
  - It's messy – you know when you are done with a resource better than the OS

Missing Exits/Waits?

```
int common=do_common_calculation();
int cstat=-1; pid_t pid = fork();
if (pid==0) { /* Child process */
        int resultA=do_A_calculation(common);
        printf("The result of A is %d\n",resultA);
} else { /* Parent process */
        int result=do_B_calculation(common);
        printf("The result of B is %d\n",resultB);
        while(1); // infinite loop
}
printf("%s is done\n",pid==0?"child":"parent");
exit(0);
```

```
unix> ./forker
The result of B is 18
The result of A is 42
child is done
unix>ps
  PID TTY          TIME CMD
6585 ttyp9   00:00:00 tcsh
5673 ttyp9   00:00:03 forker
4783 ttyp9   00:00:00 forker <defunct>
6642 ttyp9   00:00:00 ps
```

Parent process keep running,
No wait – explicit or implicit!
Child is ***"Zombie"***

Child process returns from "main";
implicit exit

Missing Exits/Waits?

```
int common=do_common_calculation();
int cstat=-1; pid_t pid = fork();
if (pid==0) { /* Child process */
	int resultA=do_A_calculation(&common);
	printf("The result of A is %d\n",resultA);
	while(1); // infinite loop
} else { /* Parent process */
	int result=do_B_calculation(&common);
	printf("The result of B is %d\n",resultB);
}
printf("%s is done\n",pid==0?"child":"parent");
exit(0);
```

Child process keep running

Parent process returns from "main";
implicit exit
implicit "wait" for child
Parent remains active process!

```
unix> ./forker&
The result of B is 18
The result of A is 42
parent is done
unix>ps
  PID TTY          TIME CMD
6585 ttyp9   00:00:00 tcsh
5673 ttyp9   00:00:03 forker
4783 ttyp9   00:00:00 forker
6642 ttyp9   00:00:00 ps
```

Signals

- List of specific asynchronous messages to a process
- Message include (but not limited to…)
  - KILL - kill the process no matter what (with no exit)
  - TERM - kill the process with no exit (but can be caught)
  - INT - Interrupt – kill the process with exit
  - SEGV - Segmentation Violation – dump core and exit
  - STOP - Stop executing instructions
  - CONT – Resume executing instructions

Signals may be sent to a process

- Via keyboard:
  - Ctrl+C – INT sent to process whose stdin is keyboard
  - Ctrl+\ - QUIT sent to process
  - Ctrl+Z – STOP sent to process
  - Ctrl+B – Resume (in the background) sent to process
- Via kill command
  - kill -<signal> <pid>
  - Default <signal> is TERM

Signals can be “caught” and handled

- When a signal is received by a process,
  - it stops executing instructions
  - it checks to see if the signal can be caught
  - if so, it checks to see if process has registered a handler for that signal
  - if so, signal handler is invoked
    - instructions in the signal handler are executed
    - return code from the signal handler says…
      - Resume instruction processing or
      - Core dump
      - exit
      - terminate

Explicit kill

- Unix command "kill <pid>"
  - If you own <pid>, terminates process
  - While terminating, reaps child processes
- What would happen with

  >kill 4783

```
unix> ./forker&
The result of B is 18
The result of A is 42
child is done
unix>ps
  PID TTY          TIME CMD
6585 ttyp9   00:00:00 tcsh
5673 ttyp9   00:00:03 forker
4783 ttyp9   00:00:00 forker <defunct>
6642 ttyp9   00:00:00 ps
unix>kill 5673
[1]       Terminated
unix> ps
 PID TTY          TIME CMD
6585 ttyp9   00:00:00 tcsh
6648 ttyp9   00:00:00 ps
```

Forks and Standard Streams

- child process inherits the parent's stdin / stdout / stderr

- If stdin is redirected from a file
  - both parent and child read that file
  - Separate file pointers… both parent and child read same data
- If stdin is connected to the keyboard
  - parent and child both read from keyboard
  - Each gets separate / independent input (I think)
- stdout/stderr
  - Output from parent and child intermixed!
  - Unpredictable output

Loading and Running Programs

int execve(char * filename, char *argv[], char *envp[])

- Library function in unistd.h
- <filename> – Name of ELF executable file
- <argv> –> Null terminated array of arguments
- <envp> –> Null terminated array of environment variables

- Loads executable from <filename>
- Calls “main” function, but sets return value to OS
- Never returns to calling code! (unless error occurs loading)

Over-simplified "Shell"

```
char cbuf[256];
pid_t cpid; int cstat;
while(gets(cbuf)) {
	cpid=fork();
	if (cpid==0) { execve(qfile(cbuf),qargs(cbuf),NULL); }
	waitpid(cpid,&cstat,NULL);
}
exit(0);
```

Over-simplified “Shell” w redirection

```
char cbuf[256];
pid_t cpid; int cstat;
while(gets(cbuf)) {
	cpid=fork();
	if (cpid==0) {
		stdin=fopen(qinput(cbuf),"r");
		stdout=fopen(qoutput(cbuf),"w");
		execve(qfile(cbuf),qargs(cbuf),NULL);
	}
	waitpid(cpid,&cstat,NULL);
}
exit(0);
```

Over-simplified “Shell” for background

```
char cbuf[256];
pid_t cpid; int cstat;
while(gets(cbuf)) {
        cpid=fork();
        if (cpid==0) { execve(qfile(cbuf),qargs(cbuf),NULL); }
}
while(wait(&cstat) != -1) {}; // Reap all children
exit(0);
```