System calls for process management

Mythili Vutukuru

CSE, IIT Bombay

API for process management

- What API does OS provide to user programs to manage processes?
  - How to create, run, terminate processes?
- API = Application Programming Interface

  = functions available to write user programs
- API provided by OS is a set of “system calls”
  - System call is a function call into OS code that runs at higher CPU privilege level
  - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
- Some “blocking” system calls cause the process to be blocked and context switched out (e.g., read() from disk), while others (e.g., getpid() to get PID) can return immediately

Portability of code across OS

- POSIX API: standard set of system calls (and some C library functions) available to user programs, defined for portability
  - Programs written using POSIX API can run on any POSIX compliant OS
  - Most modern OSes are POSIX compliant
  - Program may still need to be recompiled for different architectures
- Program language libraries hide the details of invoking system calls
  - The printf function in libc calls the `write` system call to write to screen
  - User programs usually do not need to worry about invoking system calls
- ABI (application binary interface) is the interface between machine code and underlying hardware: ISA, calling convention, …

What happens on a system call? (1)

- System calls are usually made by user library functions, e.g., C library
  - User code invokes library function only
- Example in xv6: system calls available to user programs are defined in user library header “user.h”
  - Equivalent to C library headers (xv6 doesn’t use standard C library)

```
struct stat;
struct rtcdate;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
```

What happens on a system call? (2)

- The user library makes the actual system call to invoke OS code
- NOT a regular function call to OS code as it involves CPU privilege level change
- User library invokes special “trap” instruction called “int” in x86 (see usys.S) to make system call
- The trap (int) instruction causes a jump to kernel code that handles the system call
  - More on trap instruction later

```
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
```

Process related system calls (in Unix)

- `fork()` creates a new child process
  - All processes are created by forking from a parent
  - OS starts `init` process after boot up, which forks other processes
  - The `init` process is ancestor of all processes
- `exec()` makes a process execute a given executable
- `exit()` terminates a process
- `wait()` causes a parent to block until child terminates
- Many variants of the above system calls exist in language libraries with different arguments

Process creation: fork

- Parent process calls "fork" system call to create (spawn) a new process
- New child process created with new PID
- Memory image of parent is copied into that of child
- Parent and child run different copies of same code

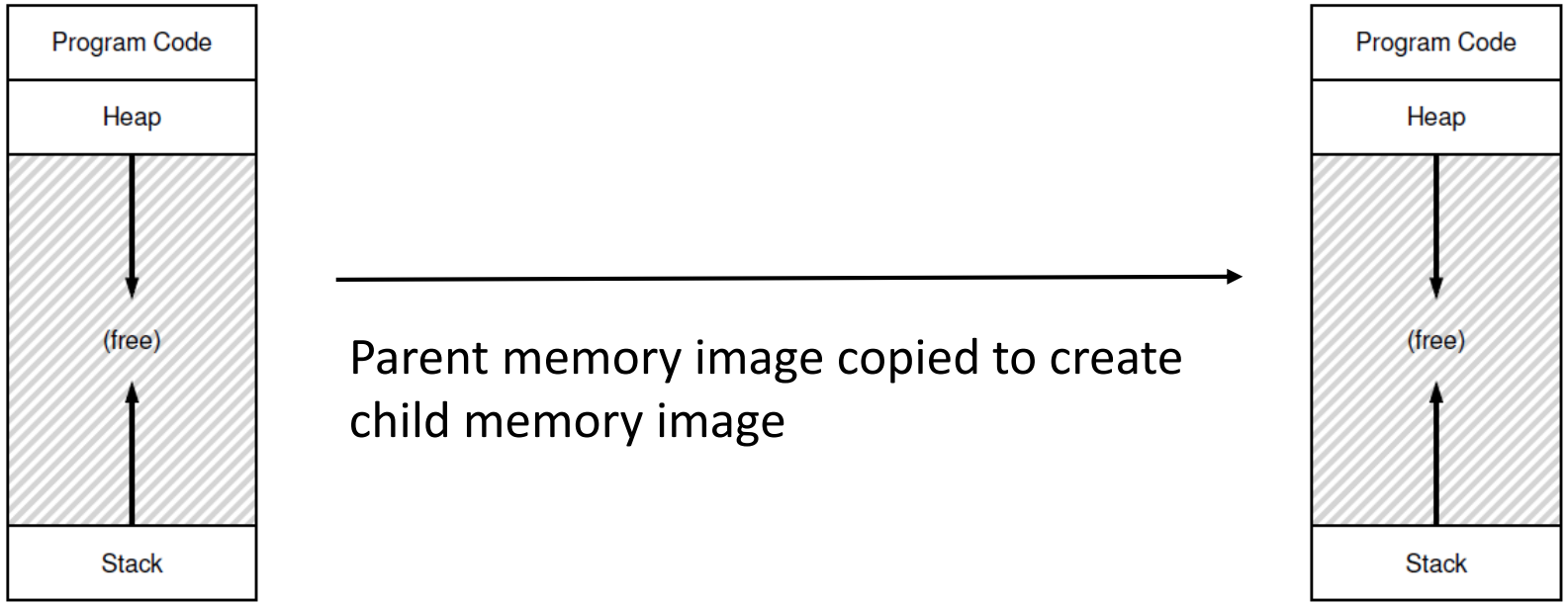
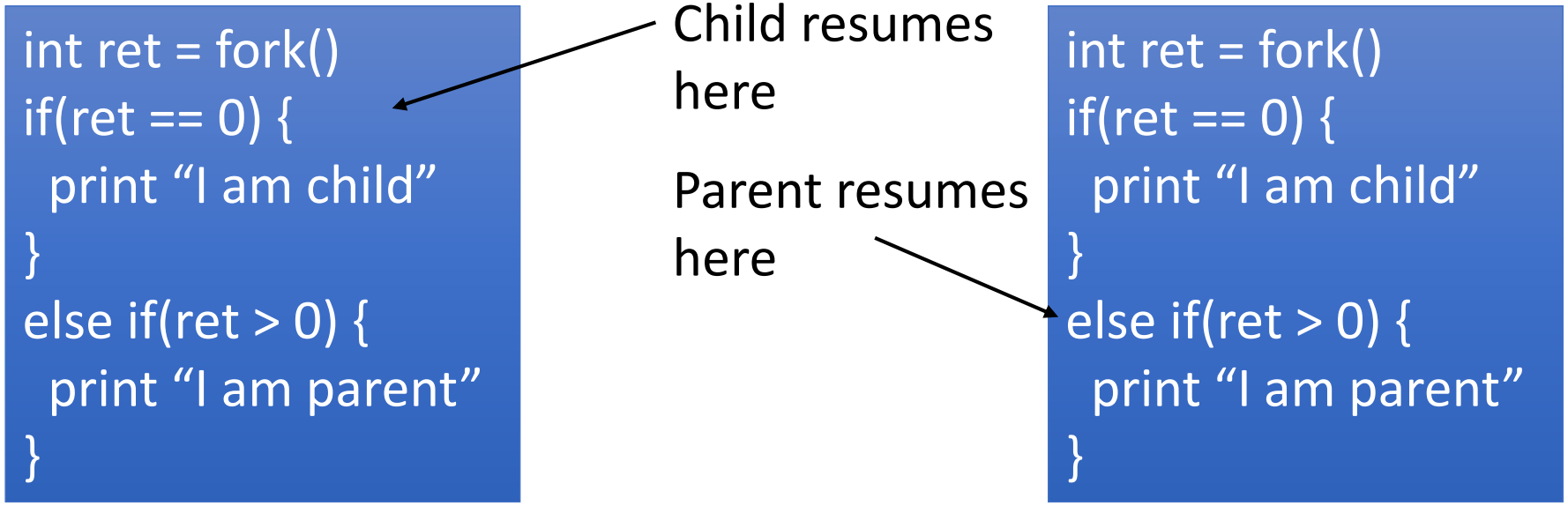


Image credit: OSTEP

What happens after fork?

- Parent and child resume execution in their copies of the code
- Child starts executing with a return value of 0 from fork
- Parent resumes executing with a return value equal to child PID
- Parent and child run independently
- Any changes in parent's data after fork does not impact child



```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4
5   int
6   main(int argc, char *argv[])
7   {
8       printf("hello world (pid:%d)\n", (int) getpid());
9       int rc = fork();
10      if (rc < 0) {          // fork failed; exit
11          fprintf(stderr, "fork failed\n");
12          exit(1);
13      } else if (rc == 0) { // child (new process)
14          printf("hello, I am child (pid:%d)\n", (int) getpid());
15      } else {               // parent goes down this path (main)
16          printf("hello, I am parent of %d (pid:%d)\n",
17                  rc, (int) getpid());
18      }
19      return 0;
20  }
```

Figure 5.1: **Calling `fork()` (`p1.c`)**

Image credit: OSTEP

Example code with fork

- Parent and child run independently and print to screen
- Order of execution of parent and child can vary

When you run this program (called `p1.c`), you'll see the following:

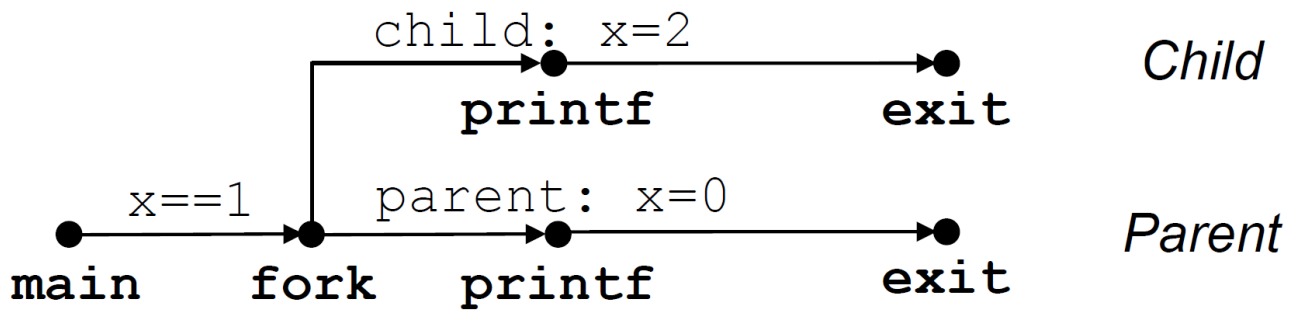
```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>

prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

Image credit: OSTEP

Example code with fork

- What values of x are printed?
- Parent and child both start with their own independent copies of variable x in their memory images
- Child increments its copy of x, prints 2
- Parent decrements its copy of x, prints 0




```
int ret = fork()
int x = 1
if(ret == 0) {
  print “I am child”
  x = x+1
  print x
}
else if(ret > 0) {
  print “I am parent”
  x = x -1
  print x
}
```

Image credit: CSAPP

Example code with nested fork

- Total 4 processes (1 parent + 3 child)
- Hello printed 4 times

```
fork()
fork()
print hello
exit
```

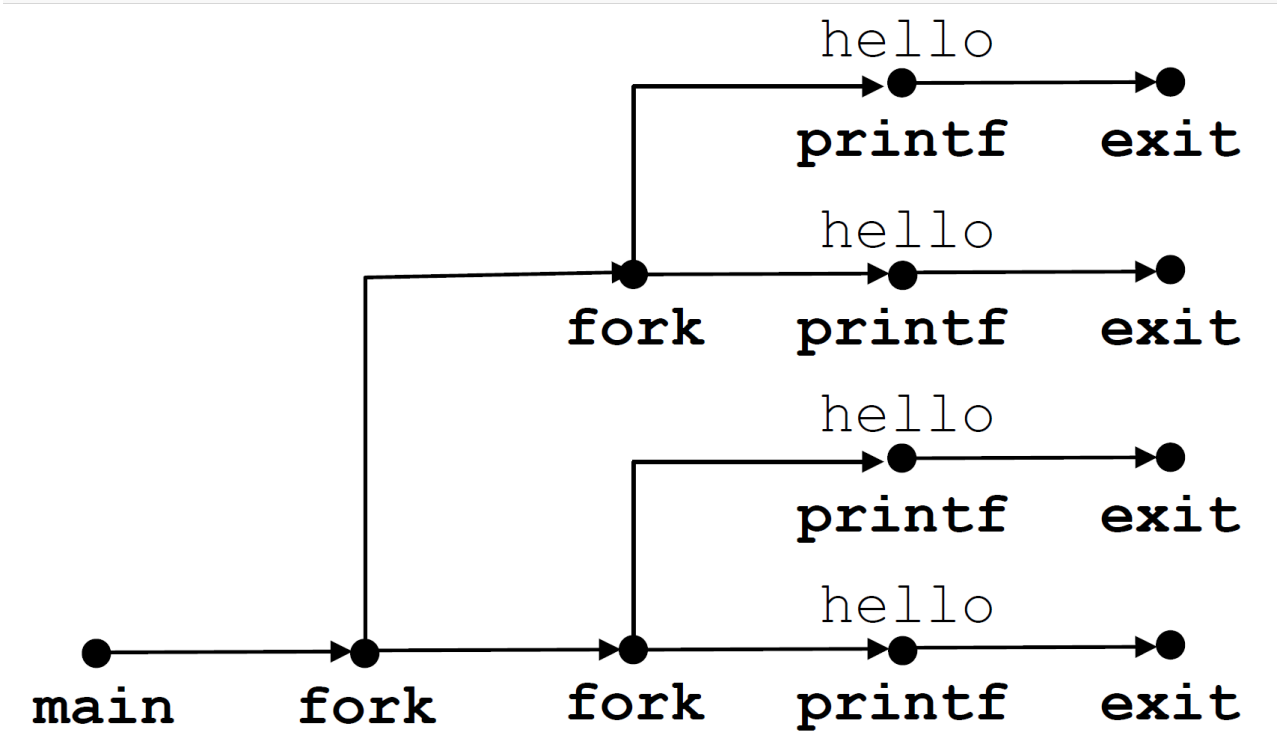


Image credit: CSAPP

xv6: fork system call implementation

```
2579 int
2580 fork(void)
2581 {
2582   int i, pid;
2583   struct proc *np;
2584   struct proc *curproc = myproc();
2585
2586   // Allocate process.
2587   if((np = allocproc()) == 0){
2588     return -1;
2589   }
2590
2591   // Copy process state from proc.
2592   if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
2593     kfree(np->kstack);
2594     np->kstack = 0;
2595     np->state = UNUSED;
2596     return -1;
2597   }
2598   np->sz = curproc->sz;
2599   np->parent = curproc;
2600   *np->tf = *curproc->tf;
2601
2602   // Clear %eax so that fork returns 0 in the child.
2603   np->tf->eax = 0;
2604
2605   for(i = 0; i < NOFILE; i++)
2606     if(curproc->ofile[i])
2607       np->ofile[i] = filedup(curproc->ofile[i]);
2608   np->cwd = idup(curproc->cwd);
2609
2610   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
2611
2612   pid = np->pid;
2613
2614   acquire(&ptable.lock);
2615
2616   np->state = RUNNABLE;
2617
2618   release(&ptable.lock);
2619
2620   return pid;
2621 }
```

xv6: fork system call implementation

- Parent process invokes fork to create new child
  - Allocates new process in ptable, get new PID for child
  - Variable “np” is pointer to newly allocated struct proc of child
  - Variable “currproc” is pointer to struct proc of parent
  - Copies information (memory, files, size, …) from currproc to np
- Child process set to runnable, scheduler runs it at a later time
- Return value in parent is PID of child
- Return value in child is set to 0 (by changing child’s EAX register)

Exit system call

- When a process finishes execution, it calls exit system call to terminate
  - OS switches the process out and never runs it again
  - Exit is automatically called at end of main
- Exiting process cannot clean up its memory, and memory must be freed up by someone else (why? More on this later.)
- Terminated process exists in a zombie state
- How are zombies cleaned up?

Wait system call

- Parent calls wait system call to reap (clean up memory of) a zombie child
- Wait cleans up memory of one terminated child and returns in parent process
- If child still running, wait system call blocks parent until child exits
  - If child terminated already, wait reaps child and returns immediately
  - If parent with no child calls wait, it returns immediately without reaping anything

```
…
int ret = fork()
if(ret == 0) {
  print “I am child”
  exit()
}
else if(ret > 0) {
  print “I am parent”
  wait()
}
…
```

More on wait

- Wait system call variant waitpid reaps a specific child with a given PID, while regular wait reaps any terminated child
  - Read man pages for more details on arguments to waitpid and wait
- Wait system call “reaps” one dead child at a time (in any order)
  - Every fork must be followed by call to wait at some point in parent
- What if parent has exited while child is still running?
  - Child will continue to run, becomes orphan
  - Orphans adopted by init process, reaped by init when they terminate
- If parent forks children, but does not bother calling wait for long time, system memory fills up with zombies
  - Common programming error, exhausts system memory

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <sys/wait.h>
5
6   int
7   main(int argc, char *argv[])
8   {
9       printf("hello world (pid:%d)\n", (int) getpid());
10      int rc = fork();
11      if (rc < 0) {          // fork failed; exit
12          fprintf(stderr, "fork failed\n");
13          exit(1);
14      } else if (rc == 0) { // child (new process)
15          printf("hello, I am child (pid:%d)\n", (int) getpid());
16      } else {               // parent goes down this path (main)
17          int wc = wait(NULL);
18          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19                 rc, wc, (int) getpid());
20      }
21      return 0;
22  }
```

Figure 5.2: **Calling `fork()` And `wait()` (`p2.c`)**

Image credit: OSTEP

Example code with fork and wait

- Order of printing of child and parent is deterministic now
- Why? Parent waits until child prints and exits, then prints

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

Image credit: OSTEP

xv6: exit system call implementation

```
2626 void
2627 exit(void)
2628 {
2629   struct proc *curproc = myproc();
2630   struct proc *p;
2631   int fd;
2632
2633   if(curproc == initproc)
2634     panic("init exiting");
2635
2636   // Close all open files.
2637   for(fd = 0; fd < NOFILE; fd++){
2638     if(curproc->ofile[fd]){
2639       fileclose(curproc->ofile[fd]);
2640       curproc->ofile[fd] = 0;
2641     }
2642   }
2643
2644   begin_op();
2645   iput(curproc->cwd);
2646   end_op();
2647   curproc->cwd = 0;
2648
2649   acquire(&ptable.lock);
2650   // Parent might be sleeping in wait().
2651   wakeup1(curproc->parent);
2652
2653   // Pass abandoned children to init.
2654   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2655     if(p->parent == curproc){
2656       p->parent = initproc;
2657       if(p->state == ZOMBIE)
2658         wakeup1(initproc);
2659     }
2660   }
2661
2662   // Jump into the scheduler, never to return.
2663   curproc->state = ZOMBIE;
2664   sched();
2665   panic("zombie exit");
2666 }
```

xv6: exit system call implementation

- Exiting process cleans up some state (e.g., close files)
- Wakes up parent process that may be waiting to reap
- Passes abandoned children (orphans) to init
- Marks itself as zombie and invokes scheduler, never gets scheduled again

xv6: wait system call implementation

```
2670 int
2671 wait(void)
2672 {
2673   struct proc *p;
2674   int havekids, pid;
2675   struct proc *curproc = myproc();
2676
2677   acquire(&ptable.lock);
2678   for(;;){
2679     // Scan through table looking for exited children.
2680     havekids = 0;
2681     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2682       if(p->parent != curproc)
2683         continue;
2684       havekids = 1;
2685       if(p->state == ZOMBIE){
2686         // Found one.
2687         pid = p->pid;
2688         kfree(p->kstack);
2689         p->kstack = 0;
2690         freevm(p->pgdir);
2691         p->pid = 0;
2692         p->parent = 0;
2693         p->name[0] = 0;
2694         p->killed = 0;
2695         p->state = UNUSED;
2696         release(&ptable.lock);
2697         return pid;
2698       }
2699     }
2700     // No point waiting if we don't have any children.
2701     if(!havekids || curproc->killed){
2702       release(&ptable.lock);
2703       return -1;
2704     }
2705
2706     // Wait for children to exit.  (See wakeup1 call in proc_exit.)
2707     sleep(curproc, &ptable.lock);
2708   }
2709 }
```

xv6: wait system call implementation

- Search for dead children in process table
- If dead child found, clean up memory of zombie, return its PID
- If no children, return -1, no need to wait
- If children exist but haven’t terminated yet, wait until one dies

Exec system call

- Isn’t it impractical to run the same code in all processes?
  - Sometimes parent creates child to do similar work..
  - .. but other times, child may want to run different code
- Child process uses “exec” system call to get a new “memory image”
  - Allows a process to switch to running different code
  - Exec system call takes another executable as argument
  - Memory image is reinitialized with new executable, new code, data, stack, heap, …

```
…
int ret = fork();
if(ret == 0) {
  exec(“some_executable”)
}
else if(ret > 0) {
  print “I am parent”
}
…
```

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <sys/wait.h>
6
7   int
8   main(int argc, char *argv[])
9   {
10      printf("hello world (pid:%d)\n", (int) getpid());
11      int rc = fork();
12      if (rc < 0) {          // fork failed; exit
13          fprintf(stderr, "fork failed\n");
14          exit(1);
15      } else if (rc == 0) { // child (new process)
16          printf("hello, I am child (pid:%d)\n", (int) getpid());
17          char *myargs[3];
18          myargs[0] = strdup("wc");   // program: "wc" (word count)
19          myargs[1] = strdup("p3.c"); // argument: file to count
20          myargs[2] = NULL;           // marks end of array
21          execvp(myargs[0], myargs);  // runs word count
22          printf("this shouldn't print out");
23      } else {                // parent goes down this path (main)
24          int wc = wait(NULL);
25          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                 rc, wc, (int) getpid());
27      }
28      return 0;
29  }
```

Figure 5.3: Calling `fork()`, `wait()`, And `exec()` (`p3.c`)

Image credit: OSTEP

Example code with exec

- Many variants of exec system call (execvp used in example), which differ in the arguments provided (read more in man pages)
- If exec successful, child gets new memory image, never comes back to the code in old memory image after exec
  - Print statement after exec doesn't run if exec successful
- If exec unsuccessful, reverts back to original memory image

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

Image credit: OSTEP

xv6: exec system implementation overview

- Copy new executable into memory from disk
- Create new stack, heap
- Copy command line arguments to new stack
- Switch process page table to use new memory image
- Process begins to run new code after system call ends
- Revert back to old memory image in case of any error

Summary

Process P spawns a child C one day
Then waits for it to die and cleans it away
Oh, what a cruel mother, some might say
But they are processes, not humans, so all okay!

Albeit, humanity is not a rare find in processes
For proof, look no further than the init
Which adopts orphan children sired by others
Without asking "For me, what's in it?"