CS 347M (Operating Systems Minor)

Spring 2022

Lecture 3: System calls for process management

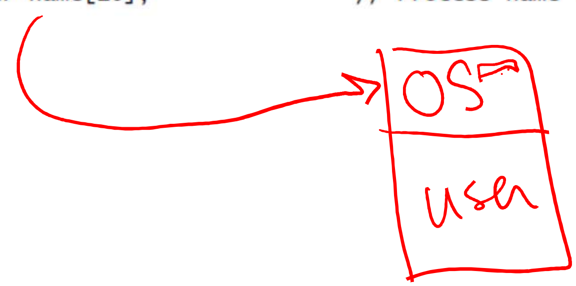
Mythili Vutukuru

CSE, IIT Bombay

Recap: Process and PCB

- OS manages multiple active processes
- Maintains one PCB for every active process (struct proc in xv6)
    - Process PID
    - Process state
    - Saved CPU context
    - Memory and I/O information

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```



Recap: Process table (ptable) in xv6

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- ptable: Fixed-size array of all processes
  - Real kernels have dynamic-sized data structures
- CPU scheduler in the OS loops over all runnable processes, picks one, and sets it running on the CPU

```
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
```

This lecture: API for process management

printf → syscall

user mode

low priv

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

user

syscall

OS

high priv

- 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

Kernel mode

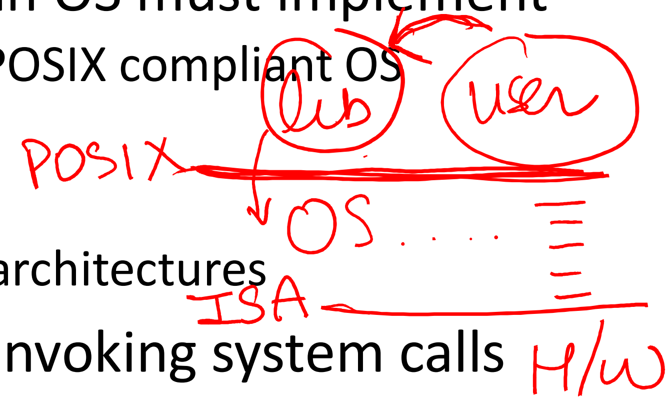
- Some "blocking" system calls cause the process to be blocked and descheduled (e.g., `read` from disk), while others can return immediately

P1 → OS → P2

P1 getpid() → OS

So, should we rewrite programs for each OS?

- POSIX API: a standard set of system calls that an OS must implement
  - Programs written using POSIX API can run on any POSIX compliant OS
  - Most modern OSes are POSIX compliant
  - Ensures program portability
  - Program may need to be recompiled for different architectures
- Program language libraries hide the details of invoking system calls
  - The printf function in the C library calls the `write` system call to write to screen
  - User programs usually do not need to worry about invoking system calls



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)

```c
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);
```

syscall

program → library → int → OS

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

interrupt

- 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)
```

~~int a=1~~ ✗

Process related system calls (in Unix)

P wait()

fork

C -> exit

- 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

init

- exec() makes a process execute a given ~~executable~~

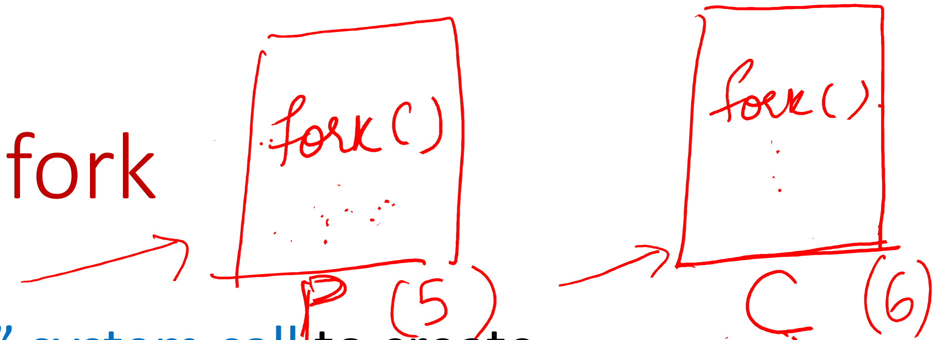
a.out

- 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

shell

firefox

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
  - Parent and child resume execution in the code after “fork”
  - Child starts executing with a return value of 0 from fork
  - Parent resumes executing with a return value of child PID
  - After fork, parent and child run independently (any changes in parent’s data after fork does not impact child)

a = 1

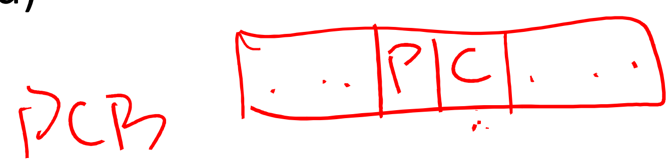
```
...
int ret = fork()
if(ret == 0) {
  print “I am child”
}
else if(ret > 0) {
  print “I am parent”
}
...
```

a = 3

ret = pid

a = 2

test.c



OSTEP

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
          int a
int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {         // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
a = 1   printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {              // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
a = 2           rc, (int) getpid());
    }
    return 0;
}
```

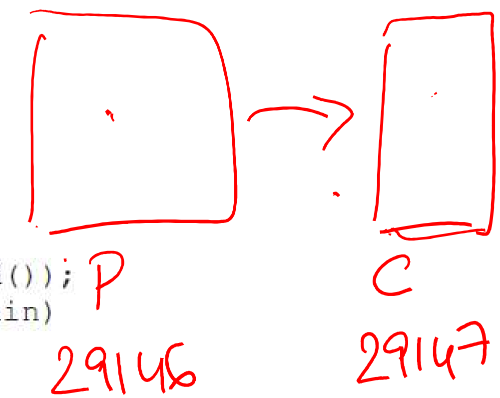
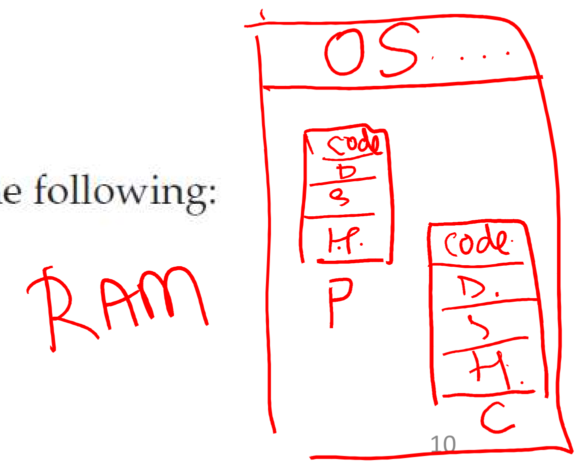


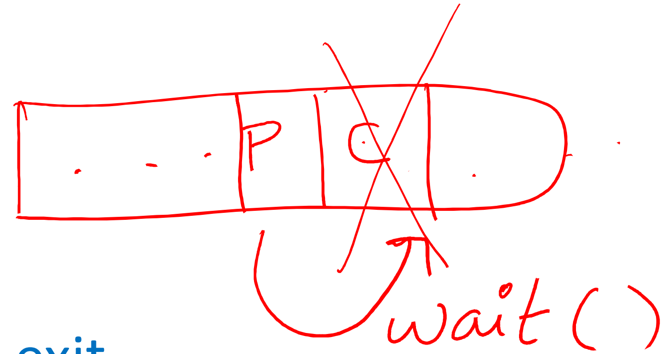
Figure 5.1: **Calling fork() (p1.c)**

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>
```



Exit and wait system calls



- When a process finishes execution, it called exit system call to terminate
  - OS switches the process out and never runs it again
  - Exit is automatically called at end of main
  - Process does not disappear, only becomes zombie
- Parent calls "wait" system call to reap (clean up memory of) a zombie child
  - Wait system call blocks parent until child exits
  - After child exit, wait cleans up memory of child and returns in parent process

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

x P C ✓

wait() ← exit

exit

More on zombies

P

wait → zombie

- Exiting child cannot clean up its memory during exit system call. Memory can be cleaned up only after a process stops executing on the CPU and is descheduled (why? more later)
- Wait system call "reaps" one dead child at a time
  - 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
  - When orphan dies, the zombie is reaped by init
- 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

fork ( )
fork ( )
wait ( )
wait ( )

P → C 1 (zombie)
P → C 2 (zombie)
0 of init

GP ← P → C orphan

man wait

Init

P1 → P2 → C

wait() (exit)

orphan

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

P → C

wait

print

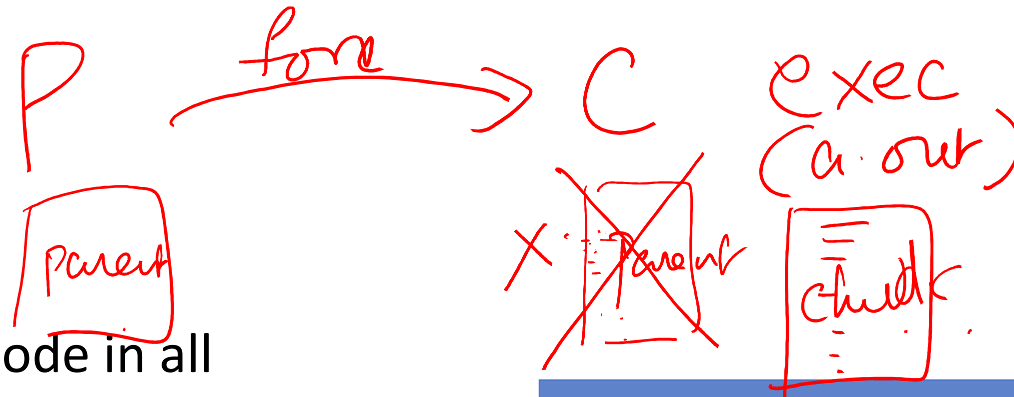
print

C

P

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

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, …
  - Child process does not return to old parent program (unless exec fails)
  - Print statement after exec never prints unless exec fails

```
…
int ret = fork();
if(ret == 0) {
  exec("some_executable")
  print "error: exec failed"
}
else if(ret > 0) {
  print "I am parent"
}
…
```

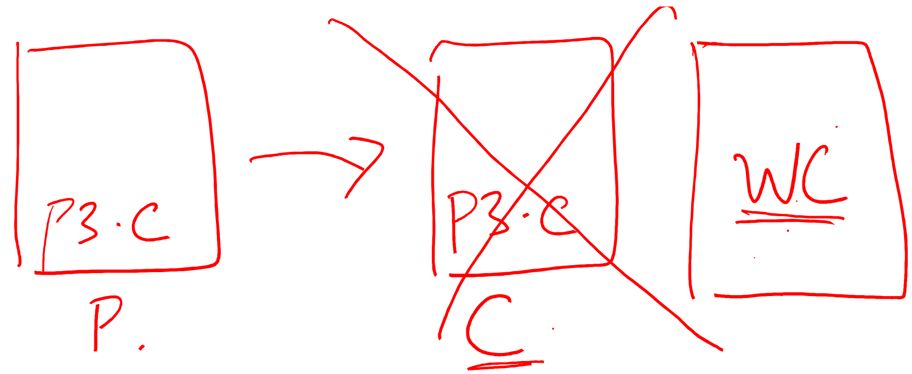
C

P

parent.c

(child.c → a.out)

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

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {              // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```



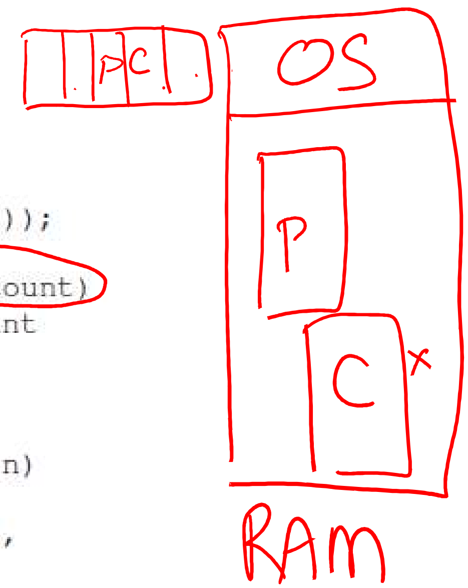


Figure 5.3: Calling **fork()**, **wait()**, And **exec()** (**p3.c**)

$ sleep 5 &

$

Shell $\xrightarrow{\text{fork}}$ C

wait

exec (echo)

a.out

How the shell works

- OS exposes a terminal/shell to run user programs
  - Can be created by first "init" process on boot up
- What happens when you type a command in the shell?
  - Shell runs command, returns back to command prompt again
- How does the shell work?
  - Shell reads input from user
  - Shell process forks a child process
  - Child process runs exec with "echo" program executable as argument (most Linux commands are programs written already for your convenience)
  - Child runs "echo" command, calls exit at end of program
  - Parent shell calls wait, blocks till child terminates, reaps it
  - Once child is done, reads next input command from user
- Think: why doesn't shell exec command directly?
  - Do we want the shell program code to be rewritten fully?

./a.out

```
$echo hello
hello
$
```

gcc

Shell C

```
do forever {
   input(command)
   int ret = fork()

   if(ret == 0) {
      exec(command)
   }
   else {
      wait()
   }
}
```

exec(...)

STD IN ←
OUT → screen
ERR → screen

Shell → C

stdout
↓
file

exec

More on Shell

- Shell can manipulate the child in strange ways
- Suppose you want to redirect output from a command to a file

$echo hello > foo.txt

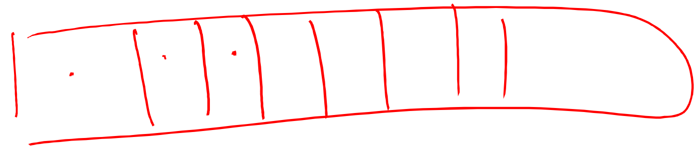
$

- Shell spawns a child, rewires its standard output to a file, then calls exec on the child

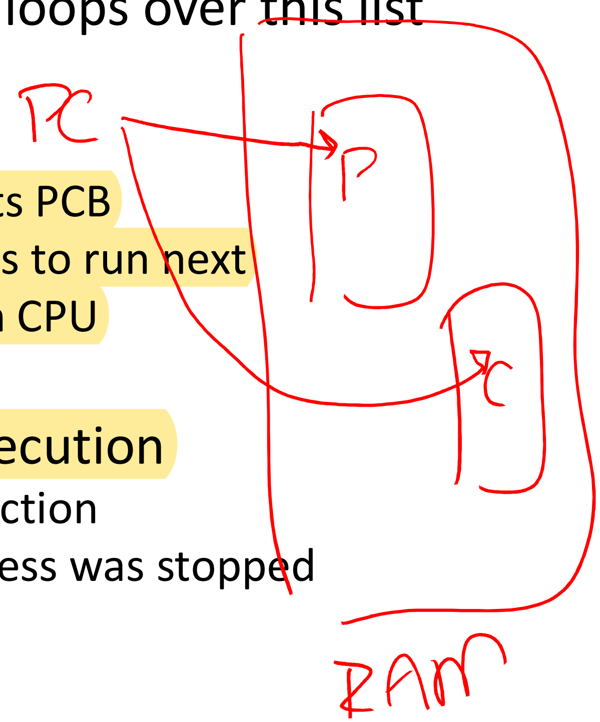
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {           // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child: redirect standard output to a file
        close(STDOUT_FILENO);
        open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

        // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {                // parent goes down this path (main)
        int wc = wait(NULL);
    }
    return 0;
}
```

Figure 5.4: All Of The Above With Redirection (p4.c)

OS scheduler



- OS maintains list of all active processes (PCBs) in a data structure
  - Processes added during fork, removed after clean up in wait
- OS scheduler is special code in the OS that periodically loops over this list and picks processes to run



- Basic outline of scheduler code
  - When invoked, save context of currently running process in its PCB
  - Loop over all ready/runnable processes and identify a process to run next
  - Restore context of new process from PCB and get it to run on CPU
  - Repeat this process as long as system is running
- Note that restoring context of a process resumes its execution
  - PC points to instruction in process code, starts running instruction
  - Other registers are filled with values that existed before process was stopped
  - Process continues execution without realizing it was paused

xv6: fork system call implementation

- Parent allocates new process in ptable, copies parent state to child
- 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

P | C

ready runnable

fork

int

OS

P -> fork()

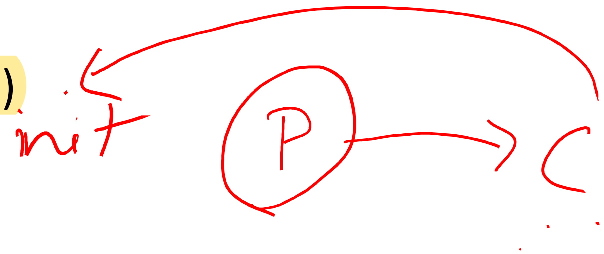
PCB entry

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

ready

xv6: exit system call implementation

- Exiting process cleans up state (e.g., close files)
- Pass abandoned children (orphans) to init
- Mark itself as zombie and invoke scheduler




```c
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

xv6: wait system call implementation

```
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);
  }
}
```

clean up

- Search for dead children in process table
- If dead child found, clean up memory of zombie, return PID of dead child
- If no dead child, sleep until one dies

xv6: main function of shell

```
int
main(void)
{
  static char buf[100];
  int fd;

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){
    if(fd >= 3){
      close(fd);
      break;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
      // Chdir must be called by the parent, not the child.
      buf[strlen(buf)-1] = 0;  // chop \n
      if(chdir(buf+3) < 0)
        printf(2, "cannot cd %s\n", buf+3);
      continue;
    }
    if(fork1() == 0)
      runcmd(parsecmd(buf));   exec
    wait();
  }
  exit();
}
```