

Lecture 23: System calls for process management in xv6

Mythili Vutukuru

IIT Bombay

<https://www.cse.iitb.ac.in/~mythili/os/>

Process system calls: Shell



- When xv6 boots up, it starts init process (first user process)
- Init forks shell (another user process, which prompts for input)
- Shell executes user commands as follows
 - Shell reads command from terminal
 - Shell forks child (new process created in ptable)
 - When child runs, it calls exec (rewrite code/data with that of command)
 - Shell (parent) waits for child to terminate
 - The whole process repeats again
- Some commands have to be executed by parent process itself, and not by child.
 - For example, “cd” command should change the current directory of parent (shell), not of child
 - Such commands are directly executed by shell itself without forking a child

Main function of shell

```
8700 int
8701 main(void)
8702 {
8703     static char buf[100];
8704     int fd;
8705
8706     // Ensure that three file descriptors are open.
8707     while((fd = open("console", O_RDWR)) >= 0){
8708         if(fd >= 3){
8709             close(fd);
8710             break;
8711         }
8712     }
8713
8714     // Read and run input commands.
8715     while(getcmd(buf, sizeof(buf)) >= 0){
8716         if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){
8717             // Chdir must be called by the parent, not the child.
8718             buf[strlen(buf)-1] = 0; // chop \n
8719             if(chdir(buf+3) < 0)
8720                 printf(2, "cannot cd %s\n", buf+3);
8721             continue;
8722         }
8723         if(fork1() == 0)
8724             runcmd(parsecmd(buf)); exec
8725         wait();
8726     }
8727     exit();
8728 }
```

What happens on a system call? (1)

- 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)
 - Note that this user code is not available in the PDF source code (which covers only kernel code)

```
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)

- System call implementation invokes special “trap” instruction called “int” in x86 (see usys.S)
- The trap (int) instruction causes a jump to kernel code that handles the system call
 - System call number moved into `eax`, to let kernel run the suitable code
 - More on trap instruction later

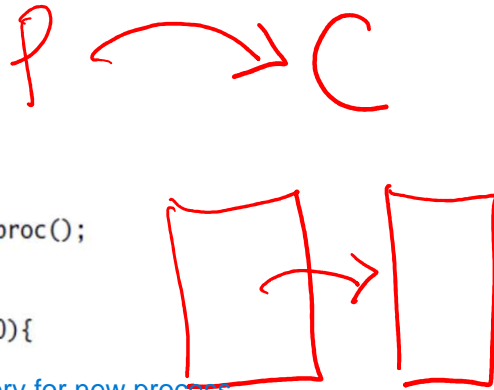
```
#include "syscall.h"
#include "traps.h"

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

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

Fork system call: overview

- 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**



```
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     // Copy process state from proc.
2591     if((np->pgdir = copyvm(curproc->pgdir, curproc->sz)) == 0){
2592         kfree(np->kstack);
2593         np->kstack = 0;
2594         np->state = UNUSED;
2595         return -1;
2596     }
2597     np->sz = curproc->sz;
2598     np->parent = curproc;
```

allocates memory for new process.

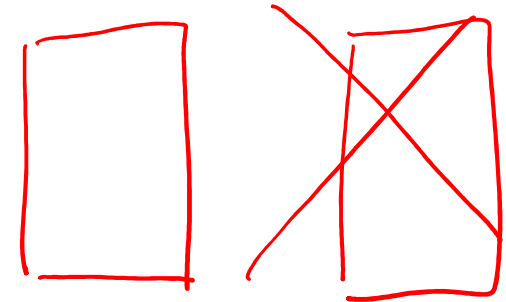
```
2600     *np->tf = *curproc->tf;
2601     // Clear %eax so that fork returns 0 in the child.
2602     np->tf->eax = 0;
2603     for(i = 0; i < NOFILE; i++)
2604         if(curproc->ofile[i])
2605             np->ofile[i] = filedup(curproc->ofile[i]);
2606     np->cwd = idup(curproc->cwd);
2607     safestrcpy(np->name, curproc->name, sizeof(curproc->name));
2608     pid = np->pid;
2609     acquire(&ptable.lock);
2610     np->state = RUNNABLE;
2611     release(&ptable.lock);
2612     return pid;
```

Copying the trapframe to resume execution from the same point.

same openfiles and current directory in child.

Mark the process runnable.

Exec system call: overview



- Key steps:
 - Copy new executable into memory
 - Create new stack, heap
 - Switch process page table to use new memory image
 - Process begins to run new code after system call ends
- See page 66 of source code PDF for full implementation

Exit system call: overview

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

```
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]){close all the file descriptors
2639             fclose(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     Give signal to reap the child.
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         } tell init to reap the abandoned child
2660     }
2661
2662     // Jump into the scheduler, never to return.
2663     curproc->state = ZOMBIE;
2664     sched();
2665     panic("zombie exit");
2666 }
```


Wait system call overview

```
2670 int
2671 wait(void)
2672 {
2673     struct proc *p;
2674     int havekids, pid;
2675     struct proc *curproc = myproc();
2676     acquire(&ptable.lock);
2677     for(;;){
2678         // Scan through table looking for exited children.
2679         havekids = 0;
2680         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
2681             if(p->parent != curproc)
2682                 continue;
2683             havekids = 1;
2684             if(p->state == ZOMBIE){
2685                 // Found one.
2686                 pid = p->pid;
2687                 kfree(p->kstack);
2688                 p->kstack = 0;
2689                 freevm(p->pgdir);
2690                 p->pid = 0;
2691                 p->parent = 0;
2692                 p->name[0] = 0;
2693                 p->killed = 0;
2694                 p->state = UNUSED;
2695                 release(&ptable.lock);
2696                 return pid;
2697             }
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 }
```

clean up

See it is not entirely gone just memory freed up and put into the unused state.

- 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

Summary of process management system calls in xv6

- Fork – process marks new child's struct proc as RUNNABLE, initializes child memory image and other state that is needed to run when scheduled
- Exec – process reinitializes memory image of user code, data, stack, heap and returns to run new code
- Exit – process marks itself as ZOMBIE, cleans up some of its state, and invokes scheduler
- Wait – parent finds any ZOMBIE child and cleans up all its state. If no dead child yet, it sleeps (marks itself as SLEEPING and invokes scheduler)