LECTURE 6: SYSTEM CALL FOR PROCESS MANAGEMENT IN XV6

xv6 book:Pg 7-10
And refer to xv6 code

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));
8725     wait();
8726   }
8727   exit();
8728 }
```

buf is a string.

using "cd" to change directory

changing the directory.

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)

function which user can use to invoke system calls are define in user.h.

```
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)
  - ("interrupt" in assembly code.)
- 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

int 0x80

The term "int" can have different meanings depending on the context in which it is used.

1. `int` as a Trap Instruction:
   - In assembly language or low-level programming, particularly in x86 architecture, `int` stands for "interrupt." It's an instruction used to generate a software interrupt. For example, `int 0x80` is commonly used in Linux assembly programs to make system calls. The `int` instruction is used to transfer control to an interrupt handler, which is part of the operating system.
2. `int` as a Data Type (Integer):
   - In high-level programming languages like C, C++, Java, and others, `int` stands for "integer." It is a data type that represents whole numbers, both positive and negative, including zero. The size of an `int` (in terms of bytes) can vary depending on the programming language and the architecture of the machine.

To sum up, `int` in the context of a trap instruction refers to an "interrupt," whereas `int` as a data type refers to an "integer" number. The key difference lies in the context—whether you're dealing with low-level assembly language or high-level programming language data types.

```
#include "sy
#include "tr

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

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

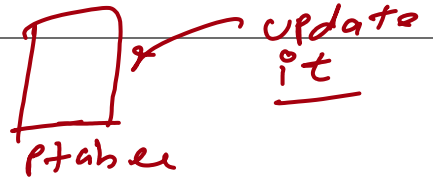
trap instruction (int)

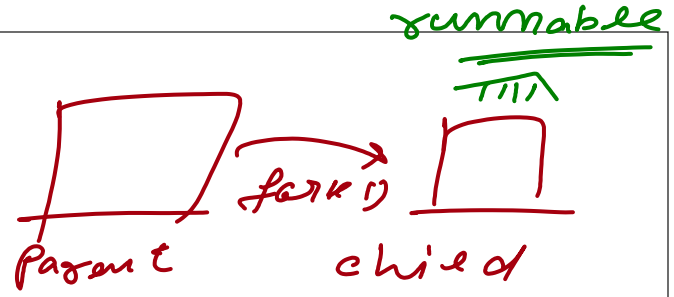
Causes Jump to kernel code.

system call's number is stored in "eax" register.

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
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 }
```

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

close file → child to init → mark self as zombie → Sched()

- 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]){
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 }
```

p → iterator

Wait system call: overview

```
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     }
```

clean if dead children found.

assuming only 1 process?

```
2700     // No point waiting if we don't have any child
2701     if(!havekids || curproc->killed){
2702       release(&ptable.lock);
2703       return -1;
2704     }
2705
2706     // Wait for children to exit.  (See wakeup1 ca
2707     sleep(curproc, &ptable.lock);
2708   }
2709 }
```

if no child sleep.

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)

Thank You