Process system calls in xv6

Mythili Vutukuru

CSE, IIT Bombay

xv6 system calls

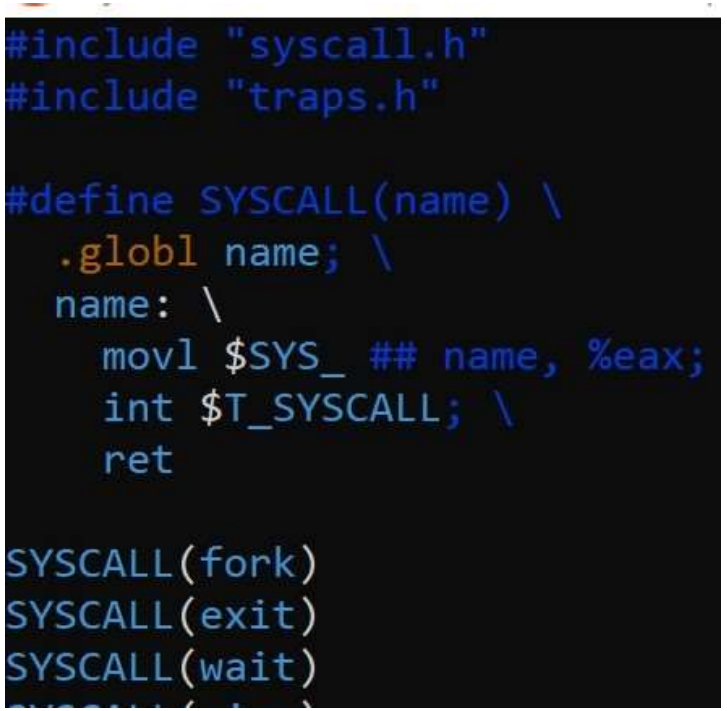
- In xv6, as in other systems, system calls are made by user library functions
  - User code invokes library function only
- System calls available to user programs are defined in user library header “user.h”
  - Equivalent to C library headers (xv6 doesn’t use standard C library)

```
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?

- 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)
```

Kernel implementation of syscalls

- The trap instruction invokes (via a series of events that we will see later) the various syscall implementations in the OS
- The user code pushes syscall number into eax (notice prev slide)
- Once we trap into the OS syscall code, the corresponding syscall function is invoked based on the value of eax
- Note that the userspace function names and kernel functions names of syscalls may be the same, but do not confuse them
  - Userspace library has a function “fork” which simply traps into the OS
  - When in the OS, we invoke OS function “fork” which runs at a higher privilege level and does the actual child process creation

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 explanation

- 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)

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 explanation

- 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 explanation

- 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

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