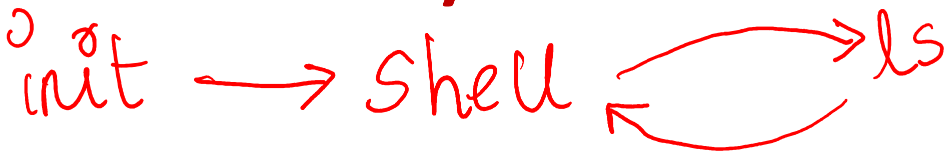
Lecture 23: System calls for process management in xv6

Mythili Vutukuru

IIT Bombay

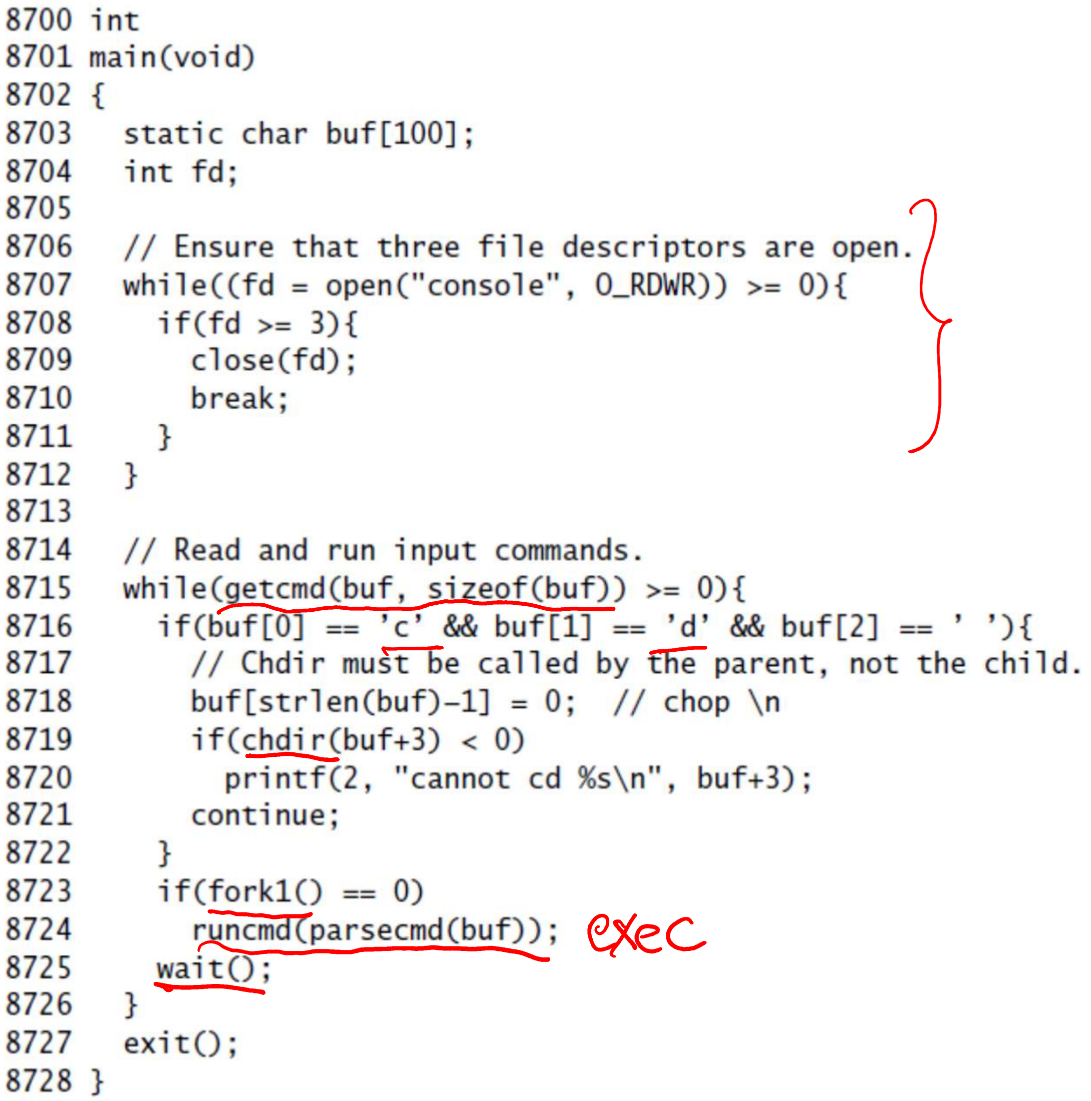
https://www.cse.iitb.ac.in/~mythili/os/

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

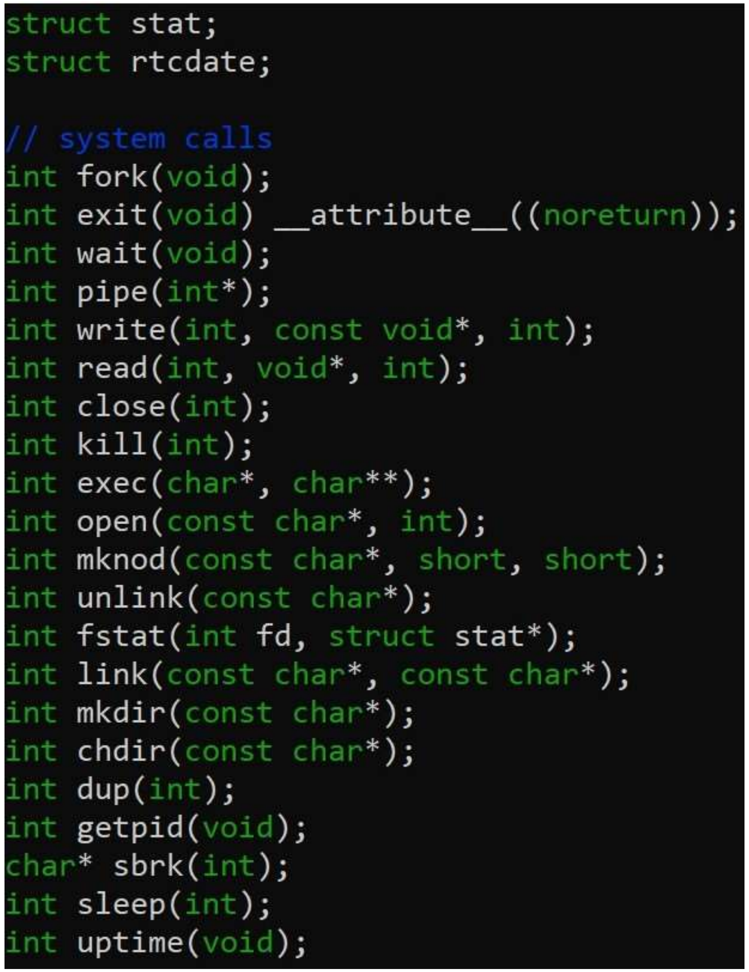


```
int
main(void)
{
  static char buf[100];
  int fd;

  // 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));
    wait();
  }
  exit();
}
```

exec

What happens on a system call? (1)

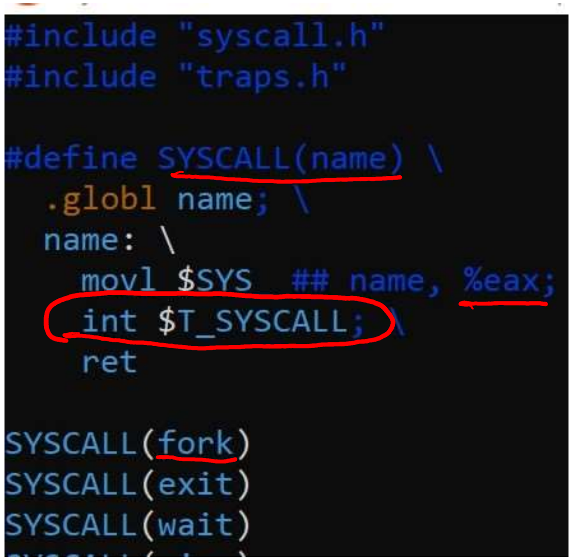
- System calls available to user programs are defined in user library header “user.h”
  - Equivalent to C library headers (xv6 doesn’t use standard C library)
  - Note that this user code is not available in the PDF source code (which covers only kernel code)



```
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)

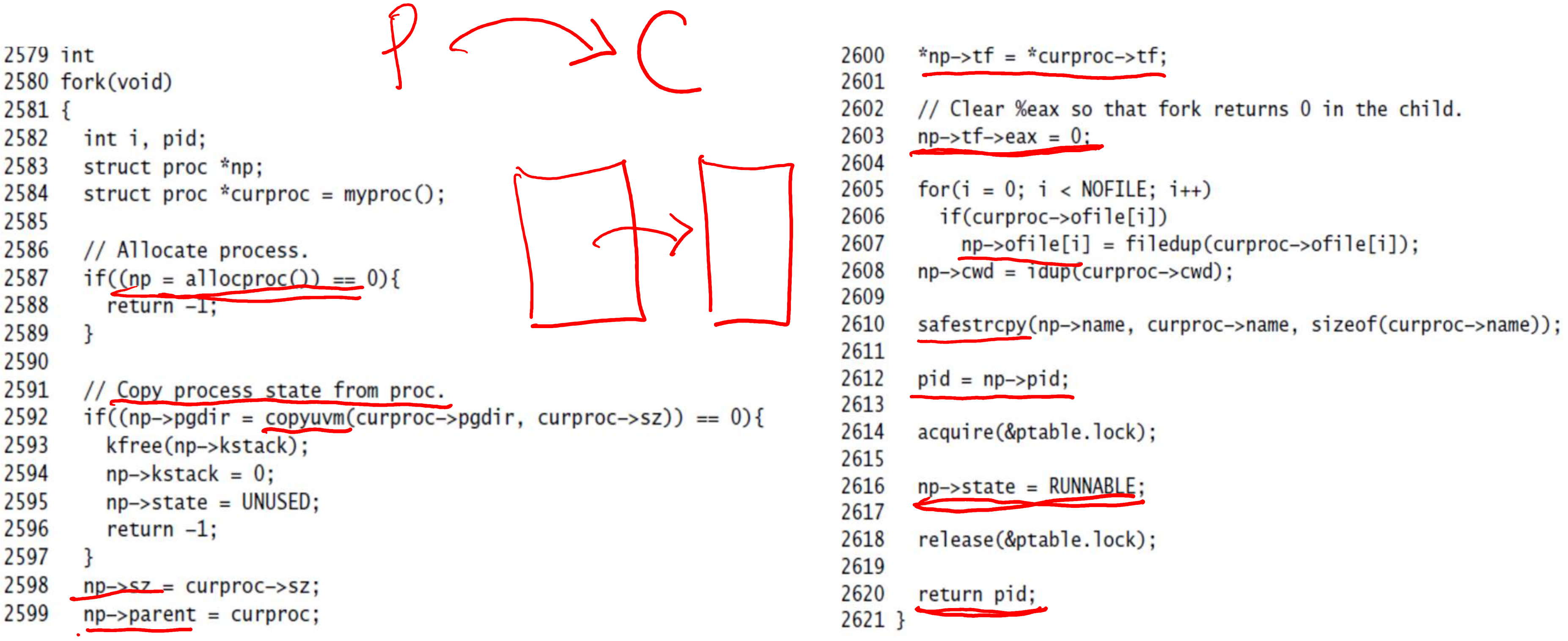
- System call implementation invokes special "trap" instruction called "int" in x86 (see usys.S)
- The trap (int) instruction causes a jump to kernel code that handles the system call
  - System call number moved into eax, to let kernel run the suitable code
  - More on trap instruction later



```
#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)
```

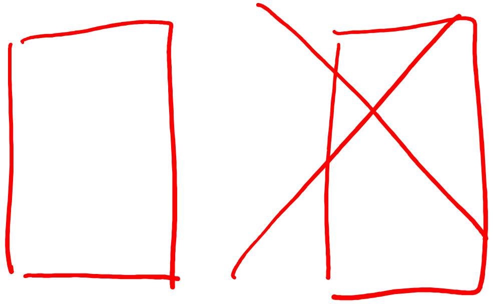
Fork system call: overview

- 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



```
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;
}
```

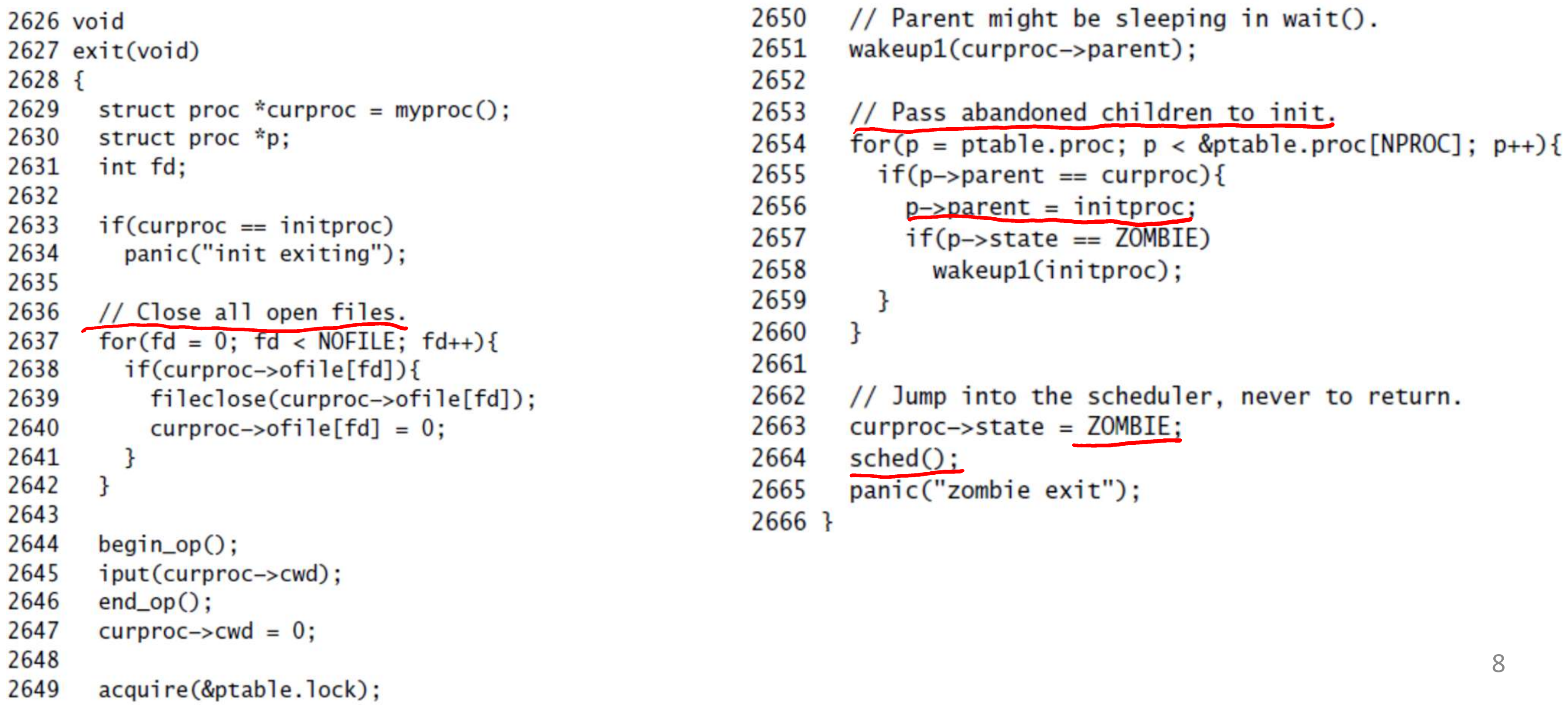
Exec system call: overview



- 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

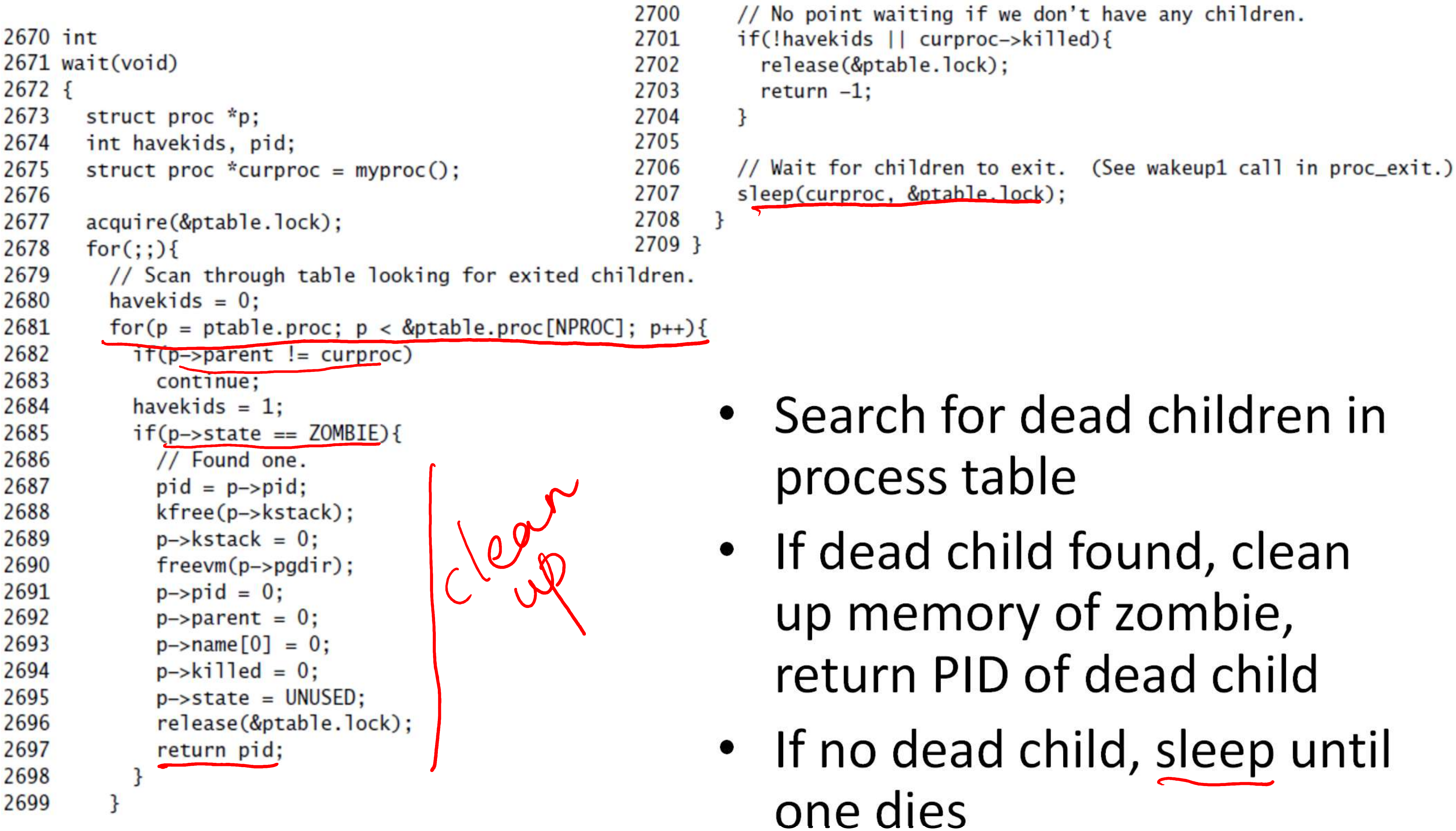
Exit system call: overview

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



```
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");
}
```

Wait system call overview



```
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

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)