CS 140 Lab Report 4

Dale Sealtiel T. Flores
2023-11373
THX/WXY

1. Using annotated xv6 code snippets or screenshots (include filenames), answer the following:

   (a) Explain how the `main` function of xv6 is able to context switch into the `init` process by going through relevant function calls.

   Answer: the `main` function will call different functions to initialize the kernel. It will then run `userinit()` to create the first user process. Going into `userinit()`, it will then call `allocproc()` to allocate a process. After this, it will set the state of the process to `RUNNABLE`. In here, it will also set `p->context.ra` by running `forkret`. After returning to `main`, it will call `scheduler()`. Going into it, this will turn the `userinit` process to `RUNNING` state and will switch to the process using `swtch()`. After this, it will go back to what we got in `forkret` and will then run `prepare_return()` to switch back to user mode. Since the process is now in user mode, it will then execute `init`

   The flow of code will be as follows: `main` -> `userinit` -> `allocproc` -> `scheduler` -> `swtch` -> `forkret` -> `prepare_return` -> `init`

```c
void    ■ Return type of 'main' is not 'int' (fix available)
main()
{
  if(cpuid() == 0){
    consoleinit();
    printfinit();
    printf("\n");
    printf("xv6 kernel is booting\n");
    printf("\n");
    kinit();         // physical page allocator
    kvminit();       // create kernel page table
    kvminithart();   // turn on paging
    procinit();      // process table
    trapinit();      // trap vectors
    trapinithart();  // install kernel trap vector
    plicinit();      // set up interrupt controller
    plicinithart();  // ask PLIC for device interrupts
    binit();         // buffer cache
    iinit();         // inode table
    fileinit();      // file table
    virtio_disk_init(); // emulated hard disk
    userinit();      // first user process
    __sync_synchronize();
    started = 1;
  } else {
    while(started == 0)
      ;
    __sync_synchronize();
    printf("hart %d starting\n", cpuid());
    kvminithart();    // turn on paging
    trapinithart();   // install kernel trap vector
    plicinithart();   // ask PLIC for device interrupts
  }

  scheduler();
}
```

Figure 1: Relevant code snippet 1 (`main` in `main.c`)

```c
void
userinit(void)
{
  struct proc *p;

  p = allocproc();
  initproc = p;

  p->cwd = namei("/");

  p->state = RUNNABLE;

  release(&p->lock);
}
```

Figure 2: Relevant code snippet 2 (`userinit` in `proc.c`)

```c
  p->context.ra = (uint64)forkret;
```

Figure 3: Relevant code snippet 3 (`forkret` in `proc.c`)

```c
  scheduler();
```

Figure 4: Relevant code snippet 4 (`scheduler` in `main.c`)

```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for(;;){
    // The most recent process to run may have had interrupts
    // turned off; enable them to avoid a deadlock if all
    // processes are waiting. Then turn them back off
    // to avoid a possible race between an interrupt
    // and wfi.
    intr_on();
    intr_off();

    int found = 0;
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
        found = 1;
      }
      release(&p->lock);
    }
    if(found == 0) {
      // nothing to run; stop running on this core until an interrupt.
      asm volatile("wfi");
    }
  }
}
```

Figure 5: Relevant code snippet 5 (`scheduler` in `proc.c`)

```c
void
forkret(void)
{
  extern char userret[];
  static int first = 1;
  struct proc *p = myproc();

  // Still holding p->lock from scheduler.
  release(&p->lock);

  if (first) {
    // File system initialization must be run in the context of a
    // regular process (e.g., because it calls sleep), and thus ca
    // be run from main().
    fsinit(ROOTDEV);

    first = 0;
    // ensure other cores see first=0.
    __sync_synchronize();

    // We can invoke exec() now that file system is initialized.
    // Put the return value (argc) of exec into a0.
    p->trapframe->a0 = exec("/init", (char *[]){ "/init", 0 });
    if (p->trapframe->a0 == -1) {
      panic("exec");
    }
  }

  // return to user space, mimicing usertrap()'s return.
  prepare_return();
  uint64 satp = MAKE_SATP(p->pagetable);
  uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline);
  ((void (*)(uint64))trampoline_userret)(satp);
}
```

Figure 6: Relevant code snippet 6 (`forkret` in `proc.c`)

(b) Explain why it is important for `exit` to wake a possibly sleeping process up.

Answer: It is important for `exit` to wake a possibly sleeping process up because the parent process might be sleeping while waiting for the child to terminate. The parent process will be in `wait()` and will be in `SLEEPING` state. If the child process calls `exit()`, it will have to wake up the parent process so that it can continue its execution and properly handle the termination of the child process.

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

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

  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

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

  acquire(&wait_lock);

  // Give any children to init.
  reparent(p);

  // Parent might be sleeping in wait().
  wakeup(p->parent);

  acquire(&p->lock);

  p->xstate = status;
  p->state = ZOMBIE;

  release(&wait_lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```

Figure 7: Relevant code snippet 1 (`exit` in `proc.c`)

(c) Explain what happens when a child process calls `exit`, but the parent process does not call `wait` and why this situation must be avoided.

Answer: When a child process calls `exit`, but the parent process does not call `wait`, the child process will stay in `ZOMBIE` state. This means that the process we want to exit is already done, but it is still there. This means that the parent process will not be able to know that the child process is already done and clean up. This must be avoided because it could lead to memory problems. Even though the process is already done, it is still in memory.

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

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

  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      struct file *f = p->ofile[fd];
      fileclose(f);
      p->ofile[fd] = 0;
    }
  }

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

  acquire(&wait_lock);

  // Give any children to init.
  reparent(p);

  // Parent might be sleeping in wait().
  wakeup(p->parent);

  acquire(&p->lock);

  p->xstate = status;
  p->state = ZOMBIE;

  release(&wait_lock);

  // Jump into the scheduler, never to return.
  sched();
  panic("zombie exit");
}
```

Figure 8: Relevant code snippet 1 (`exit` in `proc.c`)

(d) The `kill` syscall allows a process to terminate another process using its PID. Despite this, the `kill` function in `kernel/proc.c` simply sets the value of the `killed` field of the target process to `1`.
Explain how setting `killed` to `1` results in the associated process being terminated. Your explanation is expected to relate this termination mechanism to context switching.

Answer: Setting `killed` to `1` will not immediately terminate the process. Instead, it will mark the process as killed. In the next context switch, whenever it calls `usertrap`, it will check if the process is killed. If it is it will call `exit(-1)` to finally terminate the process.

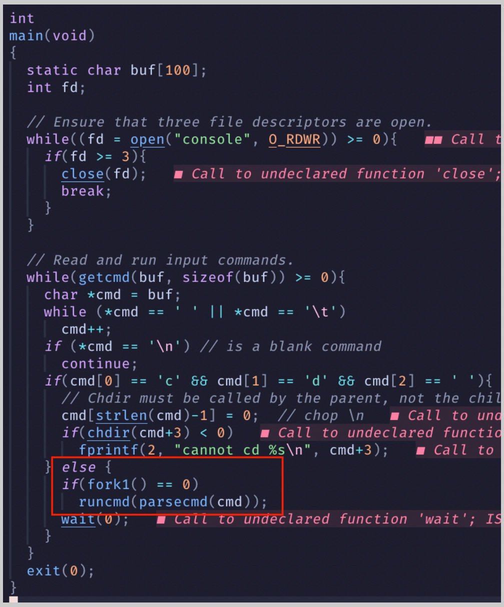
```c
uint64
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(killed(p))
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sepc, scause, and sstatus,
    // so enable only now that we're done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if((r_scause() == 15 || r_scause() == 13) &&
            vmfault(p->pagetable, r_stval(), (r_scause() == 13)? 1 : 0) != 0) {
    // page fault on lazily-allocated page
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid);
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }

  if(killed(p))
    exit(-1);
```

Figure 9: Relevant code snippet 1 (`usertrap` in `trap.c`)

(e) Go through the code of the xv6 shell may be found in `user/sh.c` and explain using `fork`, `exec`, and `wait` how:

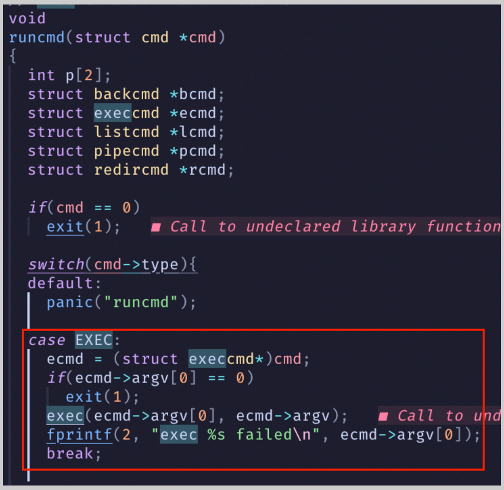
- Typing `ls` will result in the `ls` user program being executed
- `sh` is able to pause execution until `ls` ends
- `sh` is able to continue execution when `ls` ends

Answer: When we type `ls`, the shell will first call `fork1()` to create a new process. The new process will then go to `runcmd()`. In there, it will go to `case EXEC` and will call `exec()`. This will replace the process with the implementation of `ls`. The parent process will then go to `wait()` and wait for the child process (the one running `ls`) to finish. When the child process finishes, it will call `exit()` and will go to `wait()` in the parent process. This will then return and the shell will continue its execution.



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

  // Ensure that three file descriptors are open.
  while((fd = open("console", O_RDWR)) >= 0){   ■■ Call t
    if(fd >= 3){
      close(fd);   ■ Call to undeclared function 'close';
      break;
    }
  }

  // Read and run input commands.
  while(getcmd(buf, sizeof(buf)) >= 0){
    char *cmd = buf;
    while (*cmd == ' ' || *cmd == '\t')
      cmd++;
    if (*cmd == '\n') // is a blank command
      continue;
    if(cmd[0] == 'c' && cmd[1] == 'd' && cmd[2] == ' '){
      // Chdir must be called by the parent, not the chil
      cmd[strlen(cmd)-1] = 0;  // chop \n   ■ Call to und
      if(chdir(cmd+3) < 0)   ■ Call to undeclared functio
        fprintf(2, "cannot cd %s\n", cmd+3);   ■ Call to
    } else {
      if(fork1() == 0)
        runcmd(parsecmd(cmd));
      wait(0);   ■ Call to undeclared function 'wait'; IS
    }
  }
  exit(0);
}
```

Figure 10: Relevant code snippet 1 (`main` in `sh.c`)



```
void
runcmd(struct cmd *cmd)
{
  int p[2];
  struct backcmd *bcmd;
  struct execcmd *ecmd;
  struct listcmd *lcmd;
  struct pipecmd *pcmd;
  struct redircmd *rcmd;

  if(cmd == 0)
    exit(1);   ■ Call to undeclared library function

  switch(cmd->type){
  default:
    panic("runcmd");

  case EXEC:
    ecmd = (struct execcmd*)cmd;
    if(ecmd->argv[0] == 0)
      exit(1);
    exec(ecmd->argv[0], ecmd->argv);   ■ Call to und
    fprintf(2, "exec %s failed\n", ecmd->argv[0]);
    break;
```

Figure 11: Relevant code snippet 2 (`runcmd` in `sh.c`)

2. Create a user progam `user/formbomb.c` with the code in Code Block 1, run xv6 via `CPUS=1 make qemu`, execute `forkbomb`, and observe its behavior. Commit all changes made related to this item.
   a Explain what the code in Code Block 1 does and how it is recursive in nature.

   Answer: The code in Code Block 1 calls `fork()` in an infinite loop. The first `fork()` will create a child process, then both the parent and child will execute a new `forkbomb` process which will then again call `fork()`.

   b Describe the output of running `forkbomb`, how it affects the process table, and why it goes on indefinitely.

> **Answer:**
> The output of running `forkbomb` are continuous lines of `fork failed for PID #`, where the number is increasing. This is because the process table is getting filled more and more because of `fork()`. It goes indefinitely since there is no condition to stop the loop like `wait()` or `exit()`.

3. Draw a state diagram where there is a one-to-one mapping between states in the diagram and the six xv6 process states with state transition containing the name of the xv6 kernel function that performs the corresponding state change (i.e, which function containts `p->state = PROCESS_STATE_HERE`).
   If there are multiple xv6 functions resulting in the same transition, use a single arrow with all function names separated by commas as its label (e.g, `f1`, `f2`).

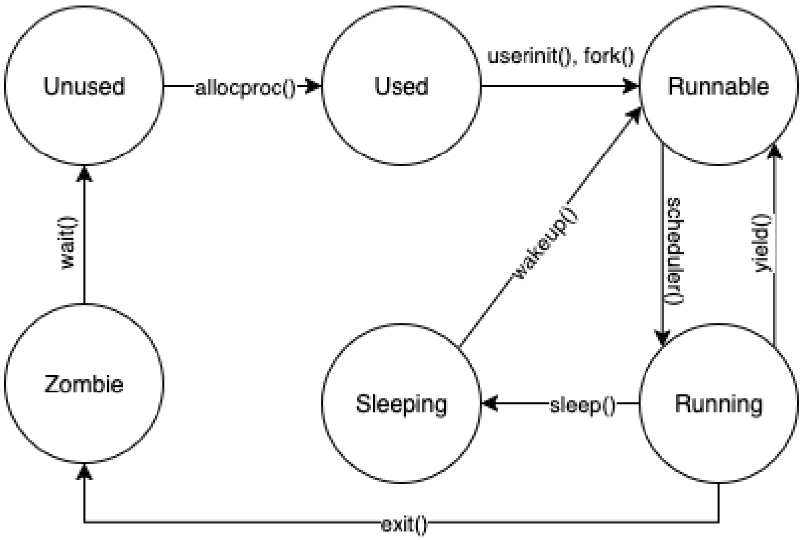
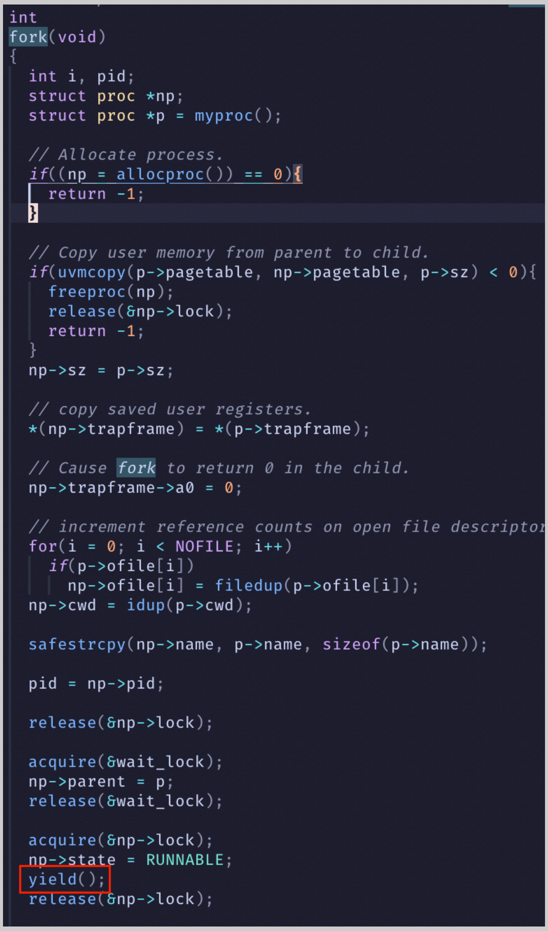


Figure 12: State diagram of xv6 process states

4. The xv6 implementation of `fork` has the invoking process continue execution after invoking `fork`. Modify xv6 such that calling `fork` instead causes a context switch to the next available process right after the process control block of the child process has been initialized.
   Show all relevant changes made (with corresponding filename) via code screenshots or snippets, and briefly describe what each change does. Ensure that your changes are properly commited and pushed to your Github Classroom repository

Answer: In order to make `fork` switch to the next available process right after the child process has been initialized, I added a call to `yield()` at the end of the `fork()` function. This will cause the current process to yield the CPU and allow the scheduler to pick the next process to run.



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

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

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;

  // increment reference counts on open file descripto
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

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

  pid = np->pid;

  release(&np->lock);

  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);

  acquire(&np->lock);
  np->state = RUNNABLE;
  yield();
  release(&np->lock);
```

Figure 13: Relevant code snippet 1 (`fork` in `proc.c`)