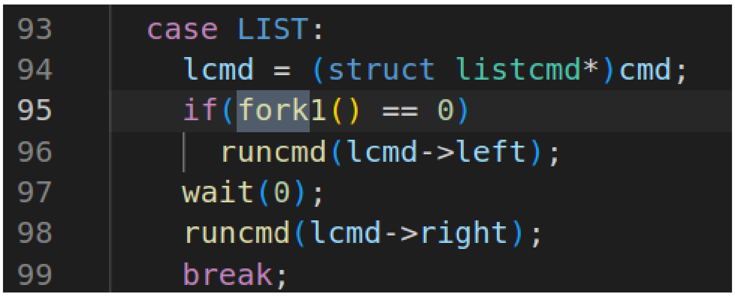
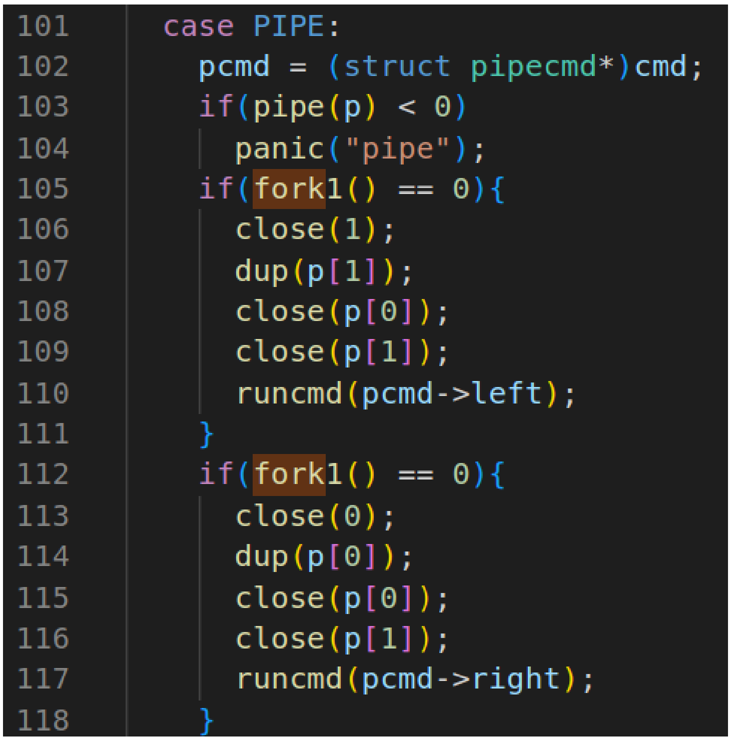
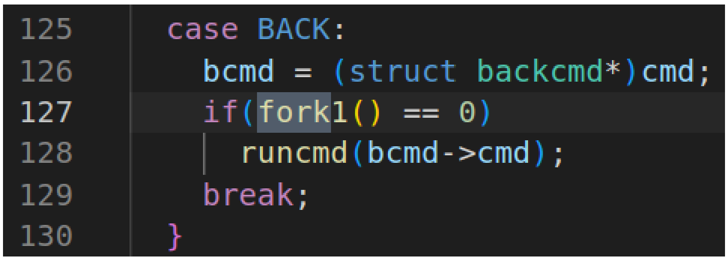
On which line in sh.c does the shell invoke the fork() system call?



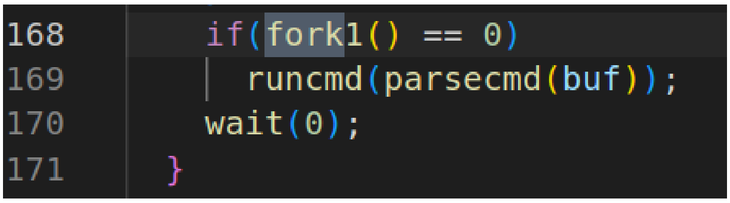
```
93    case LIST:
94      lcmd = (struct listcmd*)cmd;
95      if(fork1() == 0)
96        runcmd(lcmd->left);
97      wait(0);
98      runcmd(lcmd->right);
99      break;
```



```
101   case PIPE:
102     pcmd = (struct pipecmd*)cmd;
103     if(pipe(p) < 0)
104       panic("pipe");
105     if(fork1() == 0){
106       close(1);
107       dup(p[1]);
108       close(p[0]);
109       close(p[1]);
110       runcmd(pcmd->left);
111     }
112     if(fork1() == 0){
113       close(0);
114       dup(p[0]);
115       close(p[0]);
116       close(p[1]);
117       runcmd(pcmd->right);
118     }
```

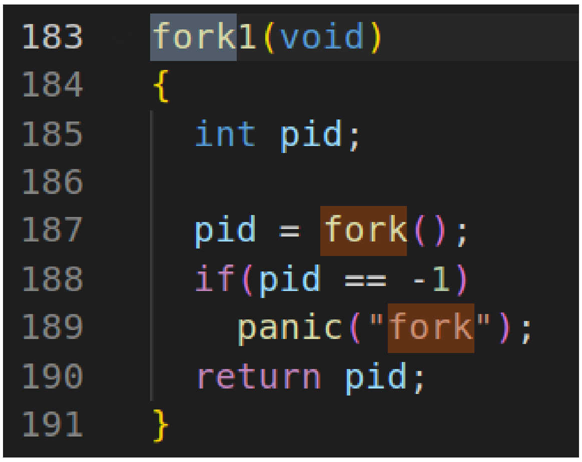


```
125   case BACK:
126     bcmd = (struct backcmd*)cmd;
127     if(fork1() == 0)
128       runcmd(bcmd->cmd);
129     break;
130   }
```



```
168       if(fork1() == 0)
169         runcmd(parsecmd(buf));
170       wait(0);
171     }
```

In line 95(case LIST)、105、112(case PIPE)、127(case BACK) and 168(main function) are have the call of fork1() and actually fork1() almost the same as fork().

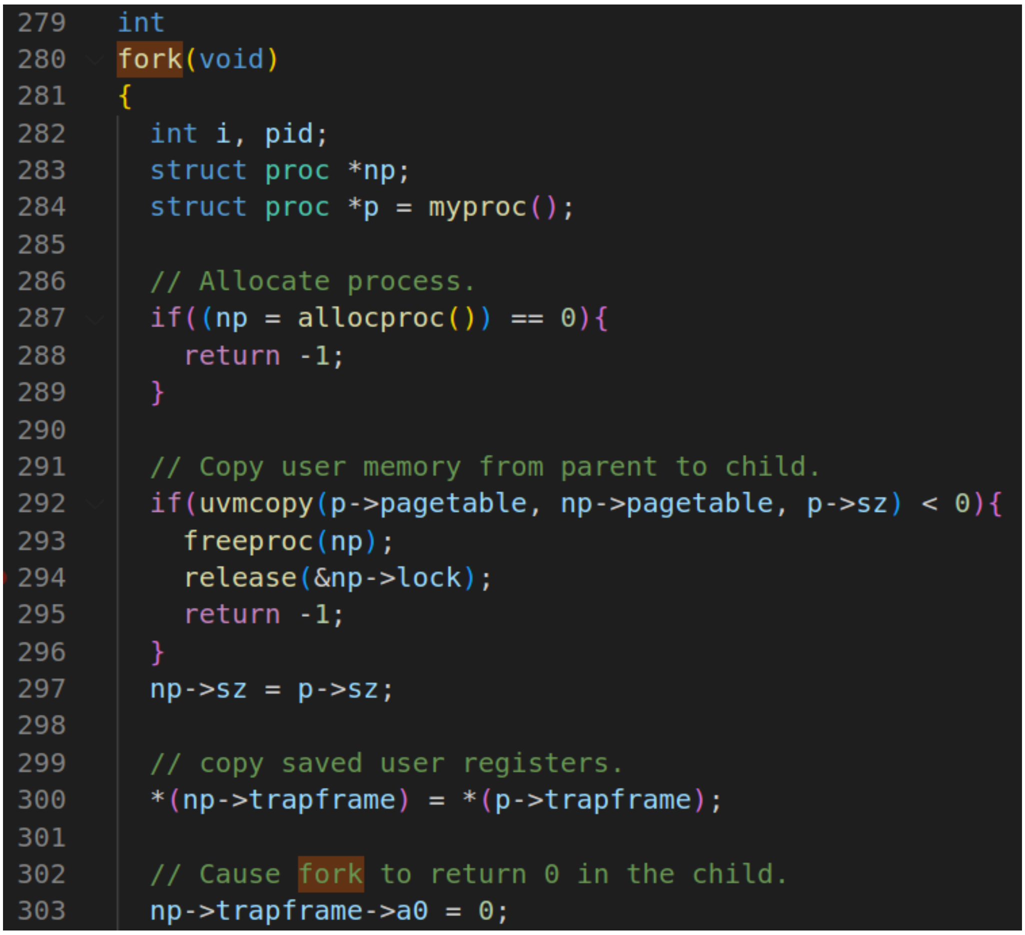


```
183  fork1(void)
184  {
185    int pid;
186
187    pid = fork();
188    if(pid == -1)
189      panic("fork");
190    return pid;
191  }
```

And the real call of fork() is at line 187.

In which file and on which line is the fork() system call implemented?

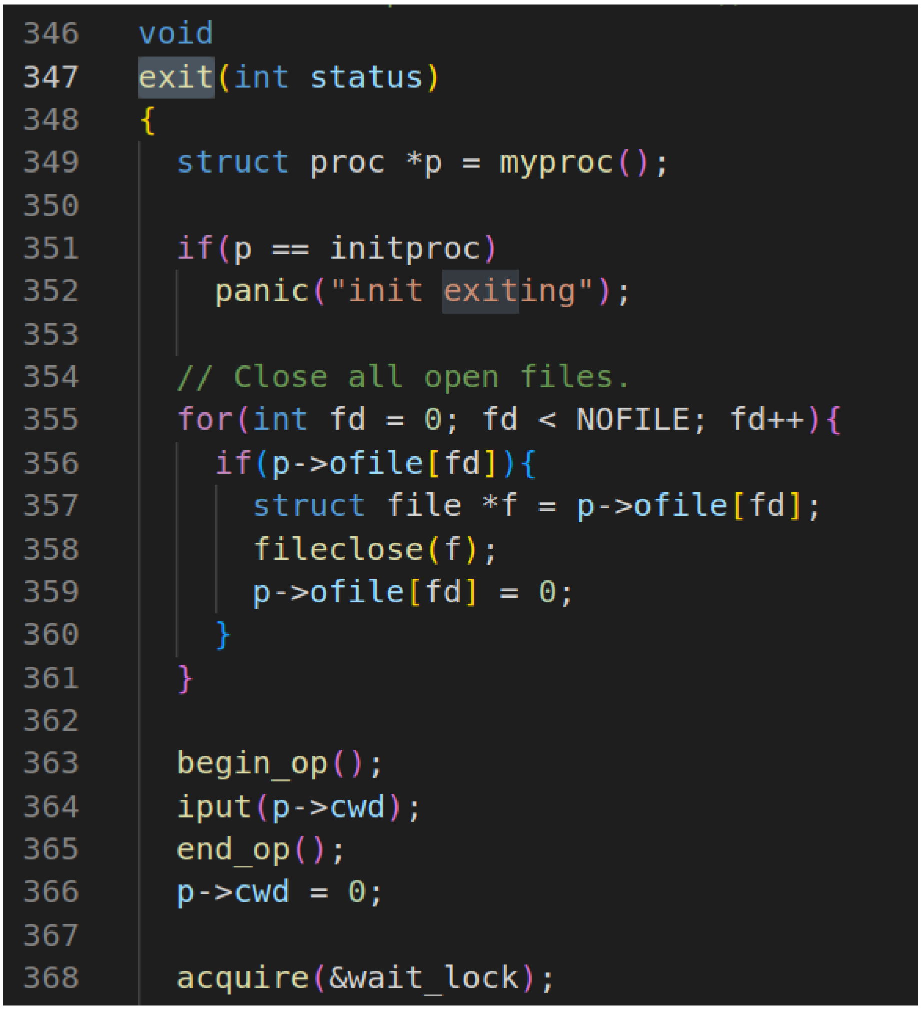
In line 280 of proc.c in kernel.



```
279  int
280  fork(void)
281  {
282    int i, pid;
283    struct proc *np;
284    struct proc *p = myproc();
285
286    // Allocate process.
287    if((np = allocproc()) == 0){
288      return -1;
289    }
290
291    // Copy user memory from parent to child.
292    if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
293      freeproc(np);
294      release(&np->lock);
295      return -1;
296    }
297    np->sz = p->sz;
298
299    // copy saved user registers.
300    *(np->trapframe) = *(p->trapframe);
301
302    // Cause fork to return 0 in the child.
303    np->trapframe->a0 = 0;
```

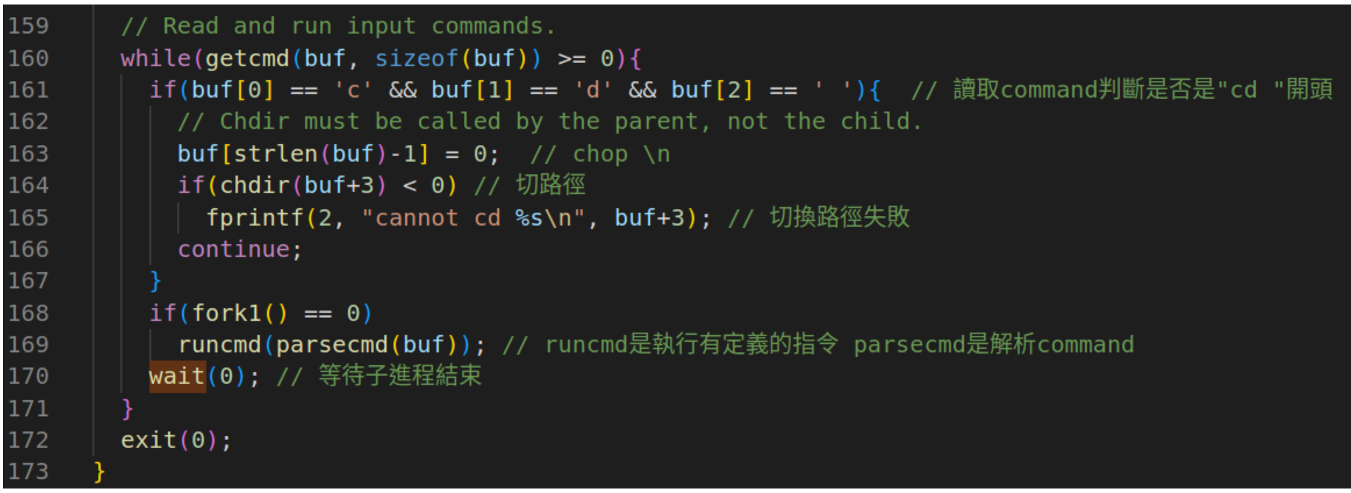
In which file and on which line is the exit() system call implemented?

In line 347 of proc.c in kernel.



```
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);
```

Do some code tracing and use the code to explain the implementation of a background process. To be specific, explain how the shell enables the user to run the next command before the previous one finishes. The wait(0) on line 170 in sh.c seems to prevent that from working, no?



```
159      // Read and run input commands.
160      while(getcmd(buf, sizeof(buf)) >= 0){
161        if(buf[0] == 'c' && buf[1] == 'd' && buf[2] == ' '){  // 讀取command判斷是否是"cd "開頭
162          // Chdir must be called by the parent, not the child.
163          buf[strlen(buf)-1] = 0;  // chop \n
164          if(chdir(buf+3) < 0) // 切路徑
165            fprintf(2, "cannot cd %s\n", buf+3); // 切換路徑失敗
166          continue;
167        }
168        if(fork1() == 0)
169          runcmd(parsecmd(buf)); // runcmd是執行有定義的指令 parsecmd是解析command
170        wait(0); // 等待子進程結束
171      }
172      exit(0);
173    }
```

The parent process will wait child process until they finished no matter whether child process is in the background(If they are in background, then the command will contain & but it only affect runcmd 、parsecmd and so on). But when they wait(), they still doing while loop and read next command. So the wait() only make sure no child process isn’t finished or leave in the background and wouldn’t prevent user running the next command before the previous one finishes.