

# CS 344 ASSIGNMENT - 1

## GROUP MEMBERS :-

- Jwalit Bharatkumar Devalia 200123026
- Akshat Jain 200123005
- Varun Bhardwaj 200123068

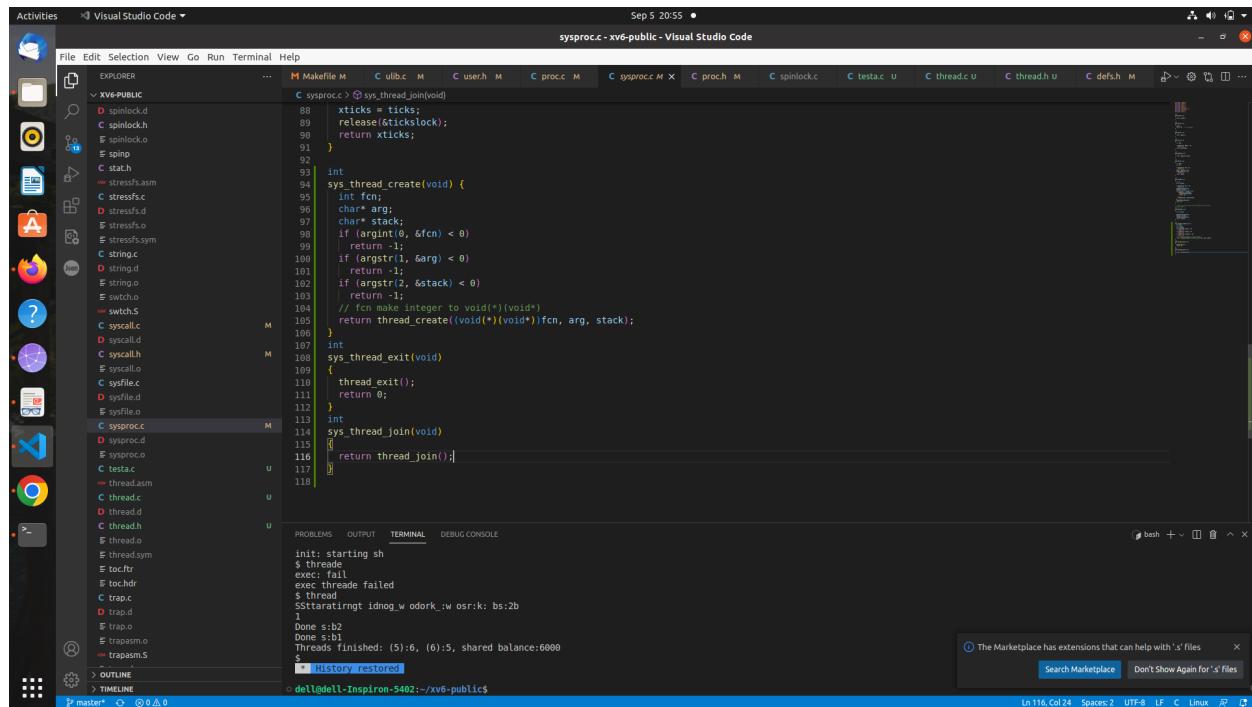
We have implemented kernel threads and then built spinlocks and mutexes to perform synchronization among the threads.

## PART 1: Implementation of kernel threads

We have made system calls for **thread\_create**, **thread\_join** and **thread\_exit**. For this we have made changes in sysproc.c, user.h, syscall.h, proc.c, makefile, defs.h, usys.h.

### Sysproc.c

We have defined the functions which call the main functions in proc.c where we have implemented threads.



```
File Explorer: sysproc.c
Sep 5 20:55 • sysproc.c - xv6-public - Visual Studio Code

sys_thread_join(void)
{
    ticks = ticks;
    release(&tickslock);
    return ticks;
}

int
sys_thread_create(void) {
    int fcn;
    char *arg;
    char *stack;
    if (argc[1] < 6fcn) < 0)
        return -1;
    if (argstr[1], &arg) < 0)
        return -1;
    if (argstr[2], &stack) < 0)
        return -1;
    // fcn make integer to void(*)(void*)
    return thread_create((void(*)(void*))fcn, arg, stack);
}

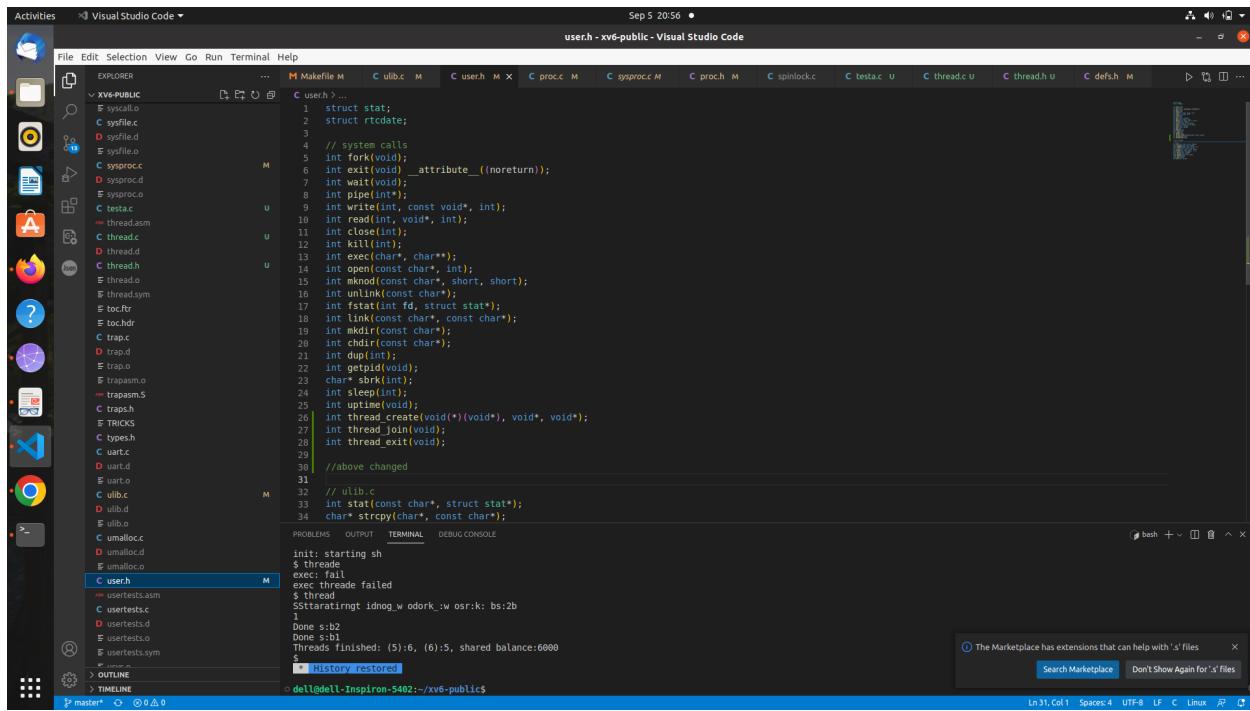
int
sys_thread_exit(void)
{
    thread_exit();
    return 0;
}

int
sys_thread_join(void)
{
    return thread_join();
}

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
init: starting sh
$ threads
exec: fail
exec:reade failed
$ threads
$Starting idnog_w odork:_w osr:k: bs:2b
1
Done s:2b
Done s:1b
Threads finished: (5):6, (6):5, shared balance:6000
$ * History restored
dell@dell-Inspiron-5402:~/xv6-public$
```

## User.h

Here we have declared our functions, `thread_create`, `thread_join` and `thread_exit`.



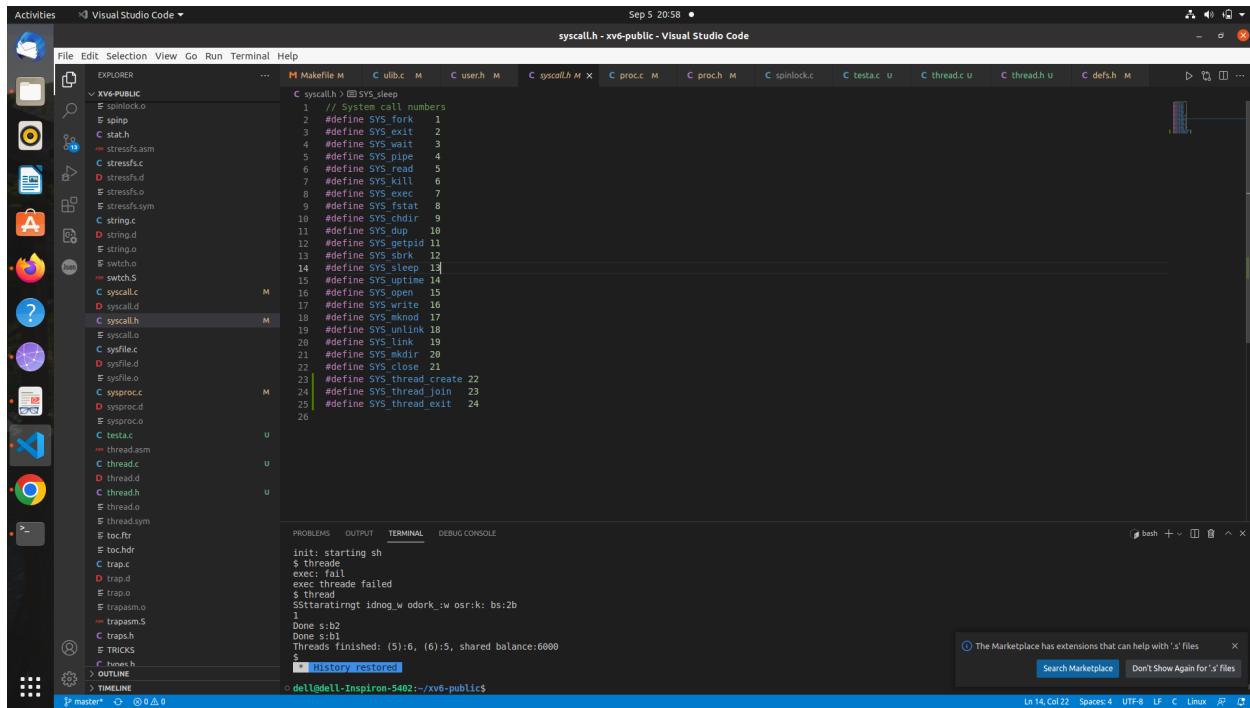
The screenshot shows the Visual Studio Code interface with the file `user.h` open in the editor. The code defines several system calls and their prototypes:

```
user.h ...
1 struct stat;
2 struct rtdate;
3 // system calls
4 int fork(void);
5 int exit(void) attribute_(noreturn);
6 int wait(void);
7 int pipe(int*);  
8 int write(int, const void*, int);
9 int read(int, void*, int);
10 int close(int);
11 int kill(int);
12 int exec(const char**, char**);
13 int open(const char*, int);
14 int mknod(const char*, short, short);
15 int unlink(const char*);
16 int link(const char*);
17 int fstat(int fd, struct stat*);
18 int link(const char*, const char*);
19 int mkdir(const char*);  
20 int chdir(const char*);  
21 int getpid();  
22 int getpid(void);  
23 int sleep(int);  
24 int uptime(void);  
25 int thread_create(void(*)(void*), void*, void*);  
26 int thread_join(void);  
27 int thread_exit(void);  
28  
29 //above changed  
30  
31 // libc.c  
32 // libc.h  
33 int stat(const char*, struct stat*);  
34 char* strcpy(char*, const char*);
```

The terminal below shows the boot process and a history restoration message.

## Syscall.h

We have just assigned numbers to system calls namely `SYS_thread_create`, `SYS_thread_exit` and `SYS_thread_join`.



The screenshot shows the Visual Studio Code interface with the file `syscall.h` open in the editor. The code defines system call numbers and their corresponding symbols:

```
syscall.h ...
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_chdir 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sleep 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_execve 20
22 #define SYS_close 21
23 #define SYS_thread_create 22
24 #define SYS_thread_join 23
25 #define SYS_thread_exit 24
```

The terminal below shows the boot process and a history restoration message.

## **Proc.c**

### **thread\_create():**

This function creates a new thread. This function is similar to the inbuilt fork function used for making new processes.

The following changes have been made:-

The function `thread_create` takes in 3 arguments:

1. The function that will use this thread for implementation. The new thread starts execution at the address space specified by this function.
2. The arguments required for the above function.
3. The user stack pointer of the stack to be used by this thread.

We are ensuring that there is a new process to be allocated to our thread. Few of the next lines have been commented on because they copy the process state of the parent process to the new process . We don't need to do this because we are spawning a new thread and not a process. The size of the new thread is set as the parent's size, the value of the trapframe of the parent process is given to that of the trapframe of our thread ,the page directory(address space) for the new thread is set to that of the parent. Then, we are setting the extended instruction pointer to our function `fcn`, and we are setting `np->tf->esp-4 = (uint)0xFFFFFFFF` , by which we are setting a fake program counter for the user stack used to implement the threads. Now we have to see what all files will be required by the thread and open them. This is what is happening in the for loop. If the required file has already been opened by `curproc`, then it gets duplicated, otherwise the new thread opens the file. Now, because the thread has access to all the files opened by the parent process, we are spanning through all the files, and are duplicating and associating the files opened by the main process to the newly created thread. Finally the state is set to RUNNABLE indicating that this thread is ready to be executed. It is done inside locks to make sure that it is executed together and that context switching does not happen before the state has been completely set.

Activities > Visual Studio Code

Sep 5 21:01 • proc.c - xv6-public - Visual Studio Code

```
File Edit Selection View Go Run Terminal Help
M Makefile M C libc.c M C user.h M C syscall.h M C proc.c M X C proch.h M C spinlock.c C testa.c U C thread.c U C threadh.h U C defsh.h M
```

221 return pid;

222 }

223 }

224 int

225 thread\_create(void(\*fcn)(void\*), void\* arg, void\* stack)

226 {

227 // how to pass argument in

228 int i, pid;

229 struct proc \*np;

230 struct proc \*curproc = myproc();

231

232 // Allocate process.

233 if((np = allocproc()) == 0){

234 return -1;

235 }

236

237 // Copy process state from proc.

238 if(np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){

239 kfree(np->stack);

240 np->state = UNUSED;

241 return -1;

242 }

243

244 np->pgdir = curproc->pgdir;

245

246 np->sz = curproc->sz;

247 np->parent = curproc;

248 \*np->tf = \*curproc->tf;

249

250 // Set eip instruction pointer

251 // Set esp stack pointer

252 // fcn is address of function ?

253 np->tf->eax = 0;

254 np->tf->ip = (uint)fcn;

255 np->tf->esp = (uint)stack+4096;

256

257 // put arg into user stack

258 // set the context of np->tf->esp to the address of arg

259 np->tf->esp = 4;

260 \*(uint\*)(np->tf->esp) = (uint)(arg);

261

262 np->tf->esp -= 4;

263 \*(uint\*)(np->tf->esp) = (uint)0xFFFFFFFF;

264

265

266 // Clear %eax so that fork returns 0 in the child.

267 np->tf->eax = 0;

268

269

270 for(i = 0; i < NOFILE; i++)

271 {

272 if(curproc->ofile[i])

273 | np->ofile[i] = fildup(curproc->ofile[i]);

274 np->cwd = idup(curproc->cwd);

275

276 safestrcpy(np->name, curproc->name, sizeof(curproc->name));

277

278 pid = np->pid;

279

280 acquire(&ptable.lock);

281

282 np->state = RUNNABLE;

283

284 release(&ptable.lock);

285

286 return pid;

287 }

288

289 int

290 thread\_join(void)

291 }

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

457

458

459

460

461

462

463

464

465

466

467

468

469

470

471

472

473

474

475

476

477

478

479

480

481

482

483

484

485

486

487

488

489

490

491

492

493

494

495

496

497

498

499

500

501

502

503

504

505

506

507

508

509

510

511

512

513

514

515

516

517

518

519

520

521

522

523

524

525

526

527

528

529

530

531

532

533

534

535

536

537

538

539

540

541

542

543

544

545

546

547

548

549

550

551

552

553

554

555

556

557

558

559

560

561

562

563

564

565

566

567

568

569

570

571

572

573

574

575

576

577

578

579

580

581

582

583

584

585

586

587

588

589

590

591

592

593

594

595

596

597

598

599

600

601

602

603

604

605

606

607

608

609

610

611

612

613

614

615

616

617

618

619

620

621

622

623

624

625

626

627

628

629

630

631

632

633

634

635

636

637

638

639

640

641

642

643

644

645

646

647

648

649

650

651

652

653

654

655

656

657

658

659

660

661

662

663

664

665

666

667

668

669

670

671

672

673

674

675

676

677

678

679

680

681

682

683

684

685

686

687

688

689

690

691

692

693

694

695

696

697

698

699

700

701

702

703

704

705

706

707

708

709

710

711

712

713

714

715

716

717

718

719

720

721

722

723

724

725

726

727

728

729

730

731

732

733

734

735

736

737

738

739

740

741

742

743

744

745

746

747

748

749

750

751

752

753

754

755

756

757

758

759

760

761

762

763

764

765

766

767

768

769

770

771

772

773

774

775

776

777

778

779

780

781

782

783

784

785

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

834

835

836

837

838

839

840

841

842

843

844

845

846

847

848

849

850

851

852

853

854

855

856

857

858

859

860

861

862

863

864

865

866

867

868

869

870

871

872

873

874

875

876

877

878

879

880

881

882

883

884

885

886

887

888

889

890

891

892

893

894

895

896

897

898

899

900

901

902

903

904

905

906

907

908

909

910

911

912

913

914

915

916

917

918

919

920

921

922

923

924

925

926

927

928

929

930

931

932

933

934

935

936

937

938

939

940

941

942

943

944

945

946

947

948

949

950

951

952

953

954

955

956

957

958

959

960

961

962

963

964

965

966

967

968

969

970

971

972

973

974

975

976

977

978

979

980

981

982

983

984

985

986

987

988

989

990

991

992

993

994

995

996

997

998

999

1000

1001

1002

1003

1004

1005

1006

1007

1008

1009

1010

1011

1012

1013

1014

1015

1016

1017

1018

1019

1020

1021

1022

1023

1024

1025

1026

1027

1028

1029

1030

1031

1032

1033

1034

1035

1036

1037

1038

1039

1040

1041

1042

1043

1044

1045

1046

1047

1048

1049

1050

1051

1052

1053

1054

1055

1056

1057

1058

1059

1060

1061

1062

1063

1064

1065

1066

1067

1068

1069

1070

1071

1072

1073

1074

1075

1076

1077

1078

1079

1080

1081

1082

1083

1084

1085

1086

1087

1088

1089

1090

1091

1092

1093

1094

1095

1096

1097

1098

1099

1100

1101

1102

1103

1104

1105

1106

1107

1108

1109

1110

1111

1112

1113

1114

1115

1116

1117

1118

1119

1120

1121

1122

1123

1124

1125

1126

1127

1128

1129

1130

1131

1132

1133

1134

1135

1136

1137

1138

1139

1140

1141

1142

1143

1144

1145

1146

1147

1148

1149

1150

1151

1152

1153

1154

1155

1156

1157

1158

1159

1160

1161

1162

1163

1164

1165

1166

1167

1168

1169

1170

1171

1172

1173

1174

1175

1176

1177

1178

1179

1180

1181

1182

1183

1184

1185

1186

1187

1188

1189

1190

1191

1192

1193

1194

1195

1196

1197

1198

1199

1200

1201

1202

1203

1204

1205

1206

1207

1208

1209

1210

1211

1212

1213

1214

1215

1216

1217

1218

1219

1220

1221

1222

1223

1224

1225

1226

1227

1228

1229

1230

1231

1232

1233

1234

1235

1236

1237

1238

1239

1240

1241

1242

1243

1244

1245

1246

1247

1248

1249

1250

1251

1252

1253

1254

1255

1256

1257

1258

1259

1260

1261

1262

1263

1264

1265

1266

1267

1268

1269

1270

1271

1272

1273

1274

1275

1276

1277

1278

1279

1280

1281

1282

1283

1284

1285

1286

1287

1288

1289

1290

1291

1292

1293

1294

1295

1296

1297

1298

1299

1300

1301

1302

1303

1304

1305

1306

1307

1308

1309

1310

1311

1312

1313

1314

1315

1316

1317

1318

1319

1320

1321

1322

1323

1324

1325

1326

1327

1328

1329

1330

1331

1332

1333

1334

1335

1336

1337

1338

1339

1340

1341

1342

1343

1344

1345

1346

1347

1348

1349

1350

1351

1352

1353

1354

1355

1356

1357

1358

1359

1360

1361

1362

1363

1364

1365

1366

1367

1368

1369

1370

1371

1372

1373

1374

1375

1376

1377

1378

1379

1380

1381

1382

1383

1384

1385

1386

1387

1388

1389

1390

1391

1392

1393

1394

1395

1396

1397

1398

1399

1400

1401

1402

1403

1404

1405

1406

1407

1408

1409

1410

1411

1412

1413

1414

1415

1416

1417

1418

1419

1420

1421

1422

1423

1424

1425

1426

1427

1428

1429

1430

1431

1432

1433

1434

1435

1436

1437

1438

1439

1440

1441

1442

1443

1444

1445

1446

1447

1448

1449

1450

1451

1452

1453

1454

1455

1456

1457

1458

1459

1460

1461

1462

1463

1464

1465

1466

1467

1468

1469

1470

1471

1472

1473

1474

1475

1476

1477

1478

1479

1480

1481

1482

1483

1484

1485

1486

1487

1488

1489

1490

1491

1492

1493

1494

1495

1496

1497

1498

1499

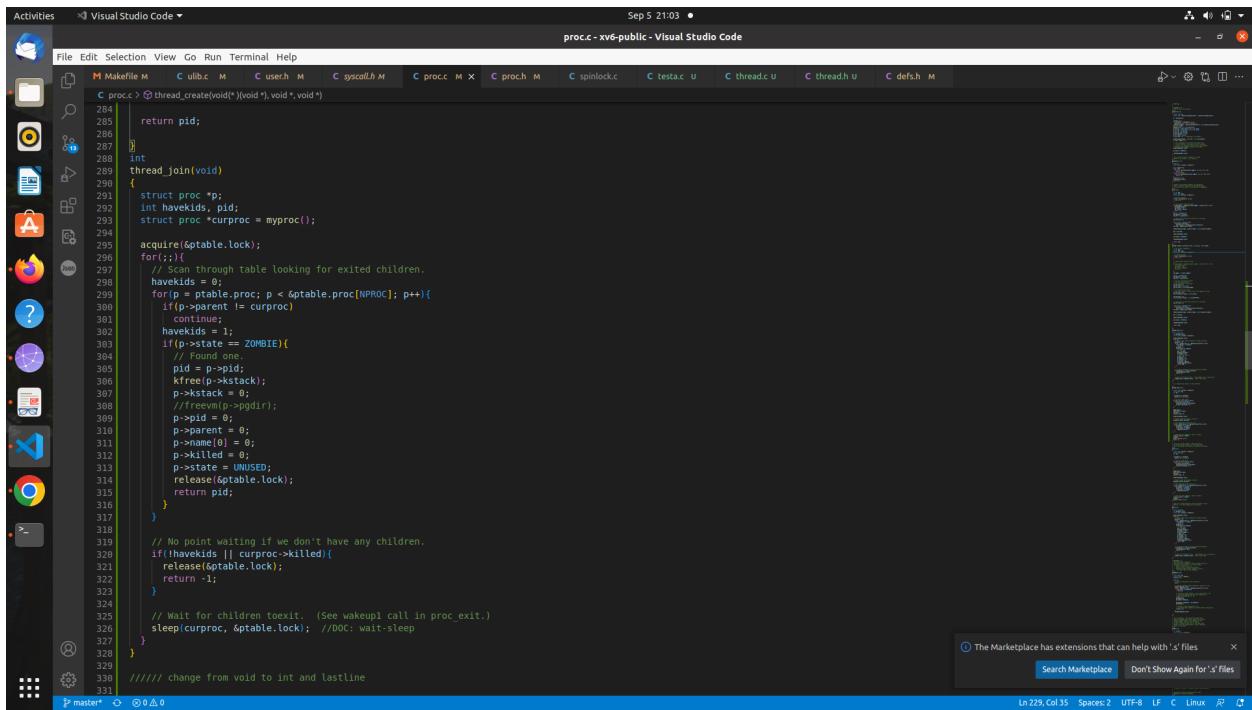
1500

1501

1502

### **thread\_join():**

This function takes no argument. It has been created with the help of the wait() function which is already available in xv6 inside the same file, *proc.c*. In the function, we scan through all the threads/processes available in the ptble and then, for all the processes(threads) which are child processes(threads) of our main process, we check if the thread has completed the execution and has entered the zombie state. If the thread has completed the execution and has entered the zombie state, then we deallocate all the resources given to that thread and finally kill it and return the pid of that killed thread. If there is no thread in the zombie state i.e. all are still executing, the *thread\_join()* function will wait for them to complete the execution. Once their execution is complete, they will enter in the zombie state so as to be killed, and therefore, enter the sleep state. At the end, when there is no thread alive, i.e. when all the threads are killed and when *have\_kids=0* or the current process is killed, we come out of the *thread\_join()* function returning -1 to indicate that the process is completed and *thread\_join()* has nothing to do now, otherwise, the pid of the thread killed is returned.



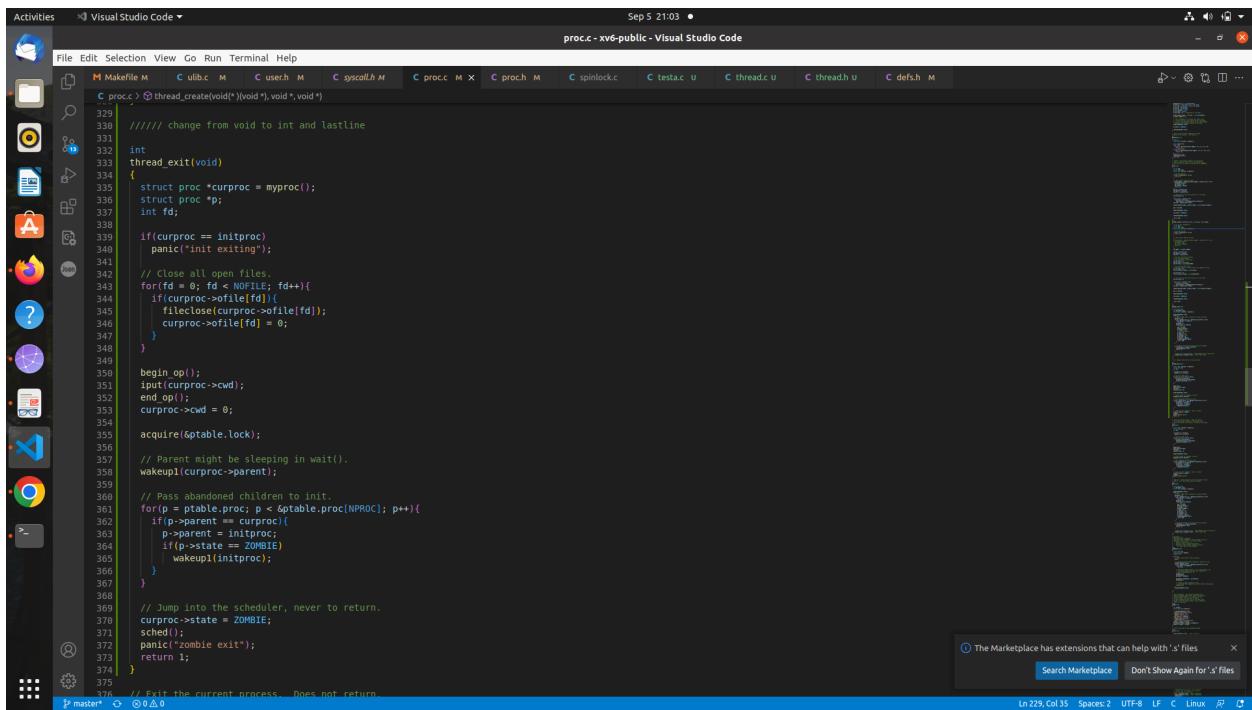
```
Sep 5 21:03 • proc.c - xv6-public - Visual Studio Code

File Edit Selection View Go Run Terminal Help
M Makefile M C libc.c M C user.h M C syscall.h M C proc.c M X C proch.c M C spinlock.c C testa.c U C thread.c U C thread.h U C defsh.c M

284     return pid;
285 }
286 }
287 }
288 int
289 thread_join(void)
290 {
291     struct proc *p;
292     int havekids, pid;
293     struct proc *curproc = myproc();
294
295     acquire(&ptable.lock);
296     for(;;)
297     {
298         /* Scan through table looking for exited children.
299         */
300         havekids = 0;
301         for(p = ptble.proc; p < &ptable.proc[NPROC]; p++)
302             if(p->parent == curproc)
303                 continue;
304             havekids = 1;
305             if(p->state == ZOMBIE)
306                 /* Found one.
307                 */
308                 pid = p->pid;
309                 p->exitp = p->parent;
310                 p->kstack = 0;
311                 p->freemv(p->pgdir);
312                 p->pid = 0;
313                 p->parent = 0;
314                 p->name[0] = 0;
315                 p->killed = 0;
316                 p->state = UNUSED;
317                 release(&ptable.lock);
318                 return pid;
319             }
320             // No point waiting if we don't have any children.
321             if(!havekids || curproc->killed)
322                 release(&ptable.lock);
323             return -1;
324         }
325         // Wait for children to exit. (See wakeup1 call in proc_exit.)
326         sleep(curproc, &ptable.lock); //DOC: wait-sleep
327     }
328 }
329 ///// change from void to int and lastline
330
331 // The Marketplace has extensions that can help with '.a' files
332 // Search Marketplace Don't Show Again for '.s' files
Ln 229, Col 35  Spaces: 2  UTF-8  LF  C  Linux  ⚙
```

### thread\_exit():

This function takes no argument. It allows a thread to terminate. It checks for all the processes in the ptable if they are completed. It has been created with the help of exit() function which is already available in xv6 inside the same file, *proc.c*. The first thing we do is make sure there are no open files associated with this thread, and if there are, we close them. Now we check if the parent is in sleep state. If the parent is indeed in sleep state, then we wake up the parent by running wakeup1 function. We then look through the ptable and if we find any threads which are associated with the parent process, and if the thread is in zombie state, then we wake up the parent using wakeup1 function call. When all children threads are completed, we then put the current process into Zombie state, and leave this function.



The screenshot shows the Visual Studio Code interface with the file *proc.c* open. The code implements the *thread\_exit()* function. It first changes the type of its argument from void\* to int and lastline. Then it initializes variables: *curproc* to *myproc*, *p* to NULL, and *fd* to 0. It checks if *curproc* is *initproc* and panics if so. It then iterates through all open files (from fd 0 to NFILE) and closes them. After closing files, it updates the cwd pointer and acquires the ptable lock. It then wakes up the parent process if it was sleeping. It iterates through the ptable to pass abandoned children to *init*. Finally, it sets the current process state to ZOMBIE, schedules, and panics with "zombie exit". The code ends with a note that it does not return.

```
329 ///// change from void* to int and lastline
330
331 int
332 thread_exit(void)
333 {
334     struct proc *curproc = myproc();
335     struct proc *p;
336     int fd;
337
338     if(curproc == initproc)
339         panic("init exiting");
340
341     // Close all open files.
342     for(fd = 0; fd < NFILE; fd++){
343         if(curproc->ofile[fd]){
344             fileclose(curproc->ofile[fd]);
345             curproc->ofile[fd] = 0;
346         }
347     }
348
349     begin_op();
350     put(curproc->cwd);
351     end_op();
352     curproc->cwd = 0;
353
354     acquire(&ptable.lock);
355
356     // Parent might be sleeping in wait().
357     wakeup1(curproc->parent);
358
359     // Pass abandoned children to init.
360     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
361         if(p->parent == curproc){
362             p->parent = initproc;
363             if(p->state == ZOMBIE)
364                 wakeup1(initproc);
365         }
366     }
367
368     // Jump into the scheduler, never to return.
369     curproc->state = ZOMBIE;
370     sched();
371     panic("zombie exit");
372     return 1;
373 }
374
375 // Exit the current process... Does not return.
```

## Makefile

In the UPROGS we are also including ‘threads’ to show that this is also a command that the user can enter.Under extra we are mentioning that the thread.c file also has to be compiled and run.

The screenshot shows a Visual Studio Code interface with the following details:

- Title Bar:** Activities > Visual Studio Code - Makefile - xv6-public - Visual Studio Code
- File Explorer:** Shows the project structure for "xv6-PUBLIC". The "Makefile" file is currently selected.
- Code Editor:** Displays the content of the "Makefile" file. The code defines various targets for building and running the xv6 kernel. Key sections include:
  - qemu-memfs:** A target that runs QEMU with a memory file system image.
  - qemu-nox:** A target that runs QEMU without graphical output.
  - .gdbinit:** A script for starting GDB with the kernel image.
  - qemu-gdb:** A target that runs QEMU with GDB attached.
  - qemu-nox-gdb:** A target that runs QEMU with a GDB image.
  - CUT HERE:** A comment indicating where to cut the Makefile for distribution.
  - dist:** A target that prepares a distribution directory by copying files and creating symbolic links.
  - dist-test:** A target that runs tests on the distribution files.
- Bottom Status Bar:** Shows the current file as "master" and other status indicators like "0 0 0".
- Bottom Right:** Marketplace notifications and search bar.

## Defs.h

The declarations for `thread_create`, `thread_join` and `thread_exit` were created in this file.

## Syscall.c

Here we are declaring new functions `sys_thread_create`, `sys_thread_join`, `sys_thread_exit`. We will be calling the main function which we have written in `proc.c`

File Edit Selection View Go Run Terminal Help

EXPLORER    ... M Makefile M C libc.b M C user.h M C syscall.c M X C proc.c M C proch M C spinlock.c C testac U C thread.c U C thread.h U C defsh M

xv6-PUBLIC

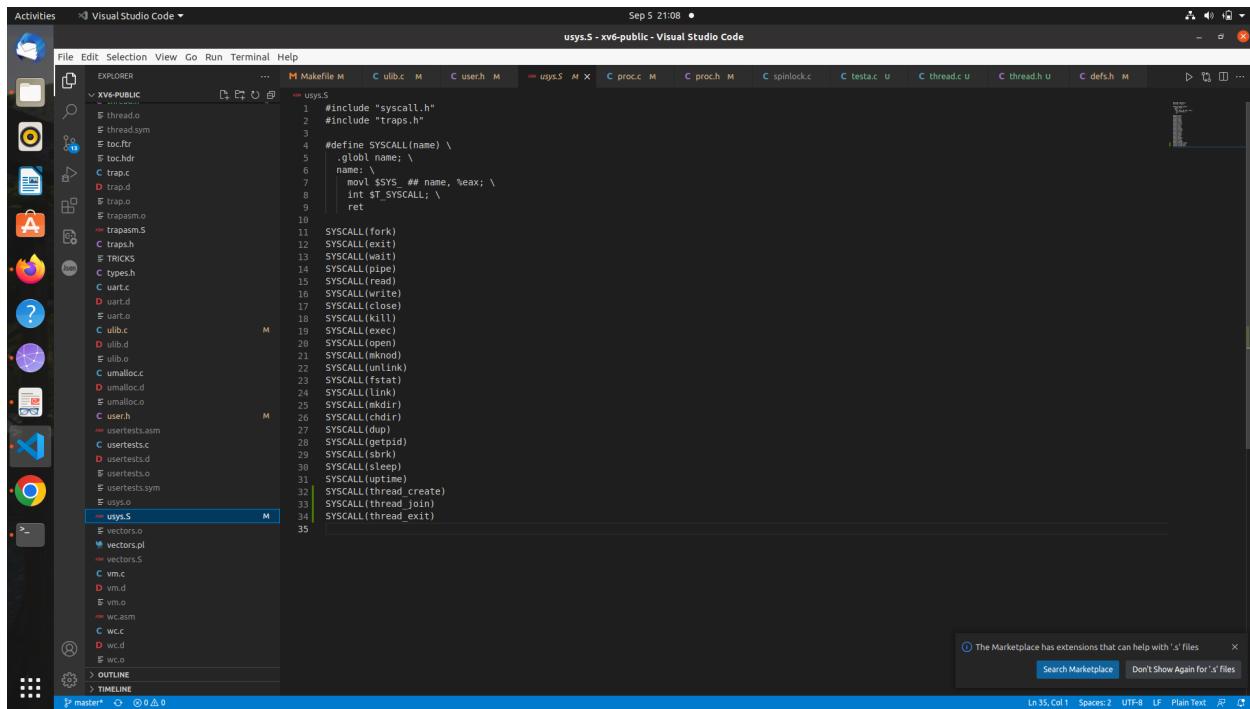
C sys\_thread\_exit(void);  
92 extern int sys\_sleep(void);  
93 extern int sys\_kill(void);  
94 extern int sys\_link(void);  
95 extern int sys\_mkdir(void);  
96 extern int sys\_mknod(void);  
97 extern int sys\_open(void);  
98 extern int sys\_pipe(void);  
99 extern int sys\_read(void);  
100 extern int sys\_readdir(void);  
101 extern int sys\_sleep(void);  
102 extern int sys\_unlink(void);  
103 extern int sys\_wait(void);  
104 extern int sys\_write(void);  
105 extern int sys\_uptime(void);  
106 extern int sys\_thread\_create(void);  
107 extern int sys\_thread\_join(void);  
108 extern int sys\_thread\_exit(void);  
109  
110 static int (\*syscalls[])()void = {  
111 [SYS\_fork] sys\_fork,  
112 [SYS\_exit] sys\_exit,  
113 [SYS\_wait] sys\_wait,  
114 [SYS\_pipe] sys\_pipe,  
115 [SYS\_read] sys\_read,  
116 [SYS\_write] sys\_write,  
117 [SYS\_exec] sys\_exec,  
118 [SYS\_fstat] sys\_fstat,  
119 [SYS\_chdir] sys\_chdir,  
120 [SYS\_dup] sys\_dup,  
121 [SYS\_getpid] sys\_getpid,  
122 [SYS\_sbrk] sys\_sbrk,  
123 [SYS\_sleep] sys\_sleep,  
124 [SYS\_uptime] sys\_uptime,  
125 [SYS\_open] sys\_open,  
126 [SYS\_write] sys\_write,  
127 [SYS\_mknod] sys\_mknod,  
128 [SYS\_unlink] sys\_unlink,  
129 [SYS\_link] sys\_link,  
130 [SYS\_mkdir] sys\_mkdir,  
131 [SYS\_close] sys\_close,  
132 [SYS\_thread\_create] sys\_thread\_create,  
133 [SYS\_thread\_join] sys\_thread\_join,  
134 [SYS\_thread\_exit] sys\_thread\_exit,  
135 };  
136 };  
137 void  
138 void  
139 syscall(void)

The Marketplace has extensions that can help with 's' files

Search Marketplace Don't Show Again for 's' files

## usys.S

In this file, we have added declaration of new functions to implement system calls for thread.



The screenshot shows the Visual Studio Code interface with the file 'usys.S' open in the center editor pane. The code defines several new system call declarations:

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

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mkdir)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sleep)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(thread_create)
SYSCALL(thread_join)
SYSCALL(thread_exit)
```

The code consists of approximately 35 lines of assembly-like syntax. The interface includes a sidebar with project files like 'Makefile', 'C libc.c', 'C userh.h', 'C proch.c', 'C spinlock.c', 'C testa.c', 'C thread.c', 'C thread.h', and 'C defsh.h'. A status bar at the bottom shows 'Ln 35 Col 1 Spaces:2 UTF-8 LF Plain Text'.

## PART 2: SYNCHRONIZATION

### Output without synchronization

We can clearly see that, without implementing locks, the total shared balance !=6000. Context switching keeps happening. Even if the entire critical section instructions of code doesn't complete its execution, context switching can happen. It might happen that both threads read an old value of the total\_balance at the same time, and then update it at almost the same time as well. As a result the deposit the increment of the balance) from one of the threads is lost.. Locks are useful in preventing this issue.

The delay statement gives time for context switching to happen and it can be used to show that not adding locks can cause a problem.

Activities QEMU

dell@dell-Inspiron-5402:~/xv6-public

File Edit Selection View Go Run Terminal Help

threadx - xv6-public - Visual Studio Code

Sep 5 22:47

dd if=bootblock of=xv6.1ng conv=notrunc  
1+0 records read  
0+0 records written  
512 bytes copied, 7.119e-05 s, 7.2 MB/s  
dd if=kernel of=xv6.1ng seek=1 conv=notrunc  
422+1 records read  
422+1 records written  
216108 bytes (216 kB, 211 KB) copied, 0.00052893 s, 391 MB/s  
raw -system=t386 -serial monstdio -drive file=fs.1mg,index=0,media=disk,format=raw -drive file=xv6.1mg,index=0,media=disk,format=xv6.  
cpu0: starting 1  
cpu0: starting 0  
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 imodestart 32 bmap star  
\$ init: starting sh  
\$ thread  
Starting do\_work: s:b1  
Starting do\_work: s:b2  
Done s:b2  
Done s:b1  
Threads finished: (4):5, (5):4, shared balance:3200  
\$ [REDACTED]

D init.d  
E init.o  
E init.sym  
E initcode  
- initcode.asm  
D initcode.d  
E initcode.o  
E initcode.out  
- initcode.S  
C logic.c  
D logicd.c  
E logicd.o  
C kalloc.c  
D kallocd.c  
E kalloc.o  
C kbdc.c  
D kbcd.c  
E kbdc.h  
E kernel  
- kernelasm  
C kernelasm  
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

init: starting sh  
\$ thread  
execve failed  
exec\_thread failed  
\$ thread  
\$Starting mt idnog\_w odork\_w osrk: bs:2b  
I  
Done s:b2  
Done s:b1  
Threads finished: (5):6, (6):5, shared balance:6000  
\$ [REDACTED]

Machine View  
SeaBIOS (version 1.13.0-Ubuntu1.1)  
IPXE (http://ipxe.org) 00:03:0 C000 PC12.10 FnP PMM+1FFBCB00+1FECCB00 C000  
Booting from Hard Disk...  
cpu0: starting 1  
cpu0: starting 0  
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 imodestart 32 bmap star  
\$ init: starting sh  
\$ thread  
Starting do\_work: s:b1  
Starting do\_work: s:b2  
Done s:b2  
Done s:b1  
Threads finished: (4):5, (5):4, shared balance:3200  
\$ [REDACTED]

The Marketplace has extensions that can help with 's' files  
Search Marketplace Don't Show Again for 's' files

Ln 41, Col 24 Spaces:2 UTF-8 LF C Linux

Activities QEMU

dell@dell-Inspiron-5402:~/xv6-public

File Edit Selection View Go Run Terminal Help

threadx - xv6-public - Visual Studio Code

Sep 5 22:48

File Explorer

XV6-PUBLIC

C Makefile M C lib.c M C user.c M C usys.S M C proc.c M C proch.c M C spinlock.c C testa.c U C thread.c U X C thread.h U C def.h M

thread.c > do\_work(void\*)

int i;

return i;

}

void do\_work(void \*arg){

int i;

int old;

struct balance \*b = (struct balance\*) arg;

thread\_spin\_lock(&lock);

printf("Starting do work: %s\n", b->name);

thread\_spin\_unlock(&lock);

for (i = 0; i < b->amount; i++) {

// thread\_spin\_lock(&lock);

// mutex\_lock(&lock);

old = total\_balance;

delay(100000);

total\_balance = old + 1;

// thread\_spin\_unlock(&lock);

// mutex\_unlock(&lock);

thread\_spin\_lock(&lock);

printf("Done %s\n", b->name);

thread\_spin\_unlock(&lock);

thread\_exit();

return;

}

int main(int argc, char \*argv[]){

init: starting sh  
\$ thread  
execve failed  
exec\_thread failed  
\$ thread  
\$Starting mt idnog\_w odork\_w osrk: bs:2b  
I  
Done s:b2  
Done s:b1  
Threads finished: (5):6, (6):5, shared balance:6000  
\$ [REDACTED]

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

The Marketplace has extensions that can help with 's' files  
Search Marketplace Don't Show Again for 's' files

Ln 43, Col 39 Spaces:2 UTF-8 LF C Linux

## Output with spinlocks

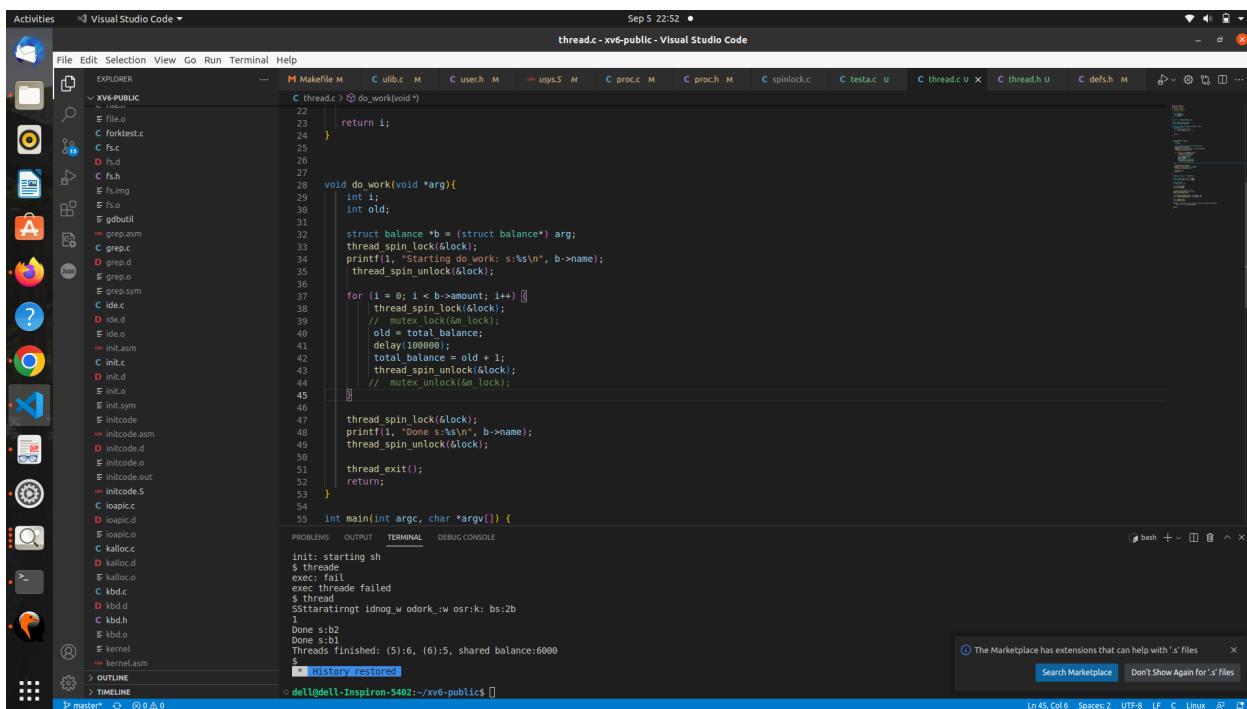
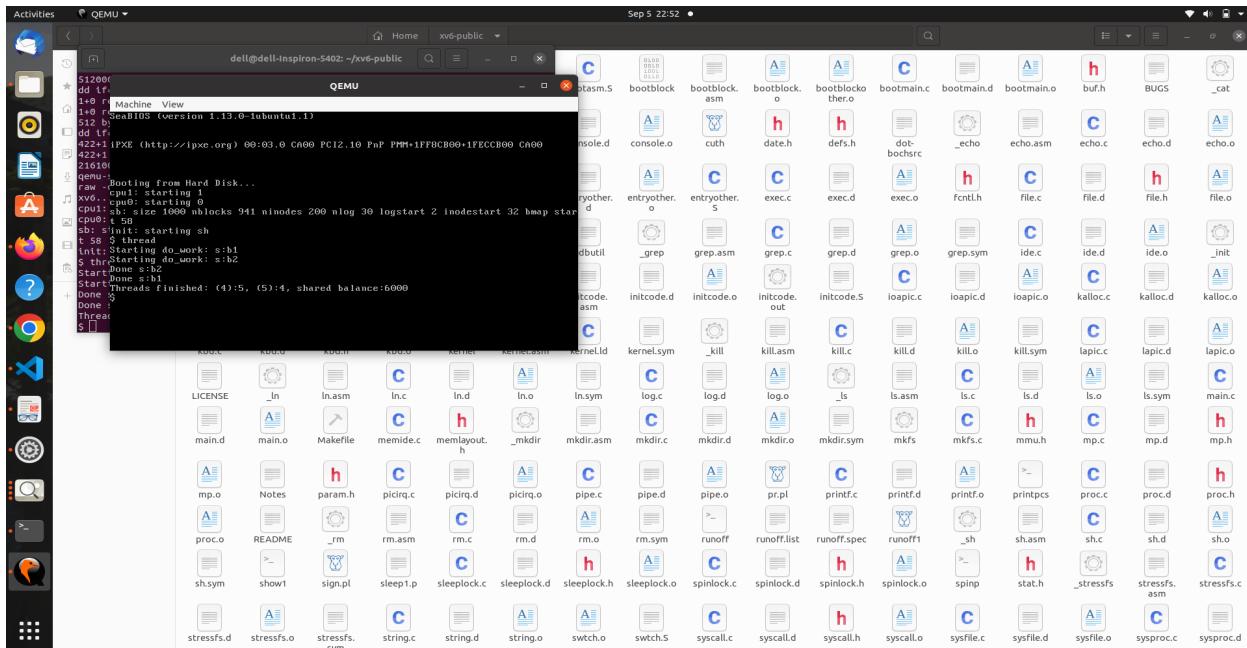
To fix this synchronization error we have to implement a spinlock that will allow us to execute the update atomically, i.e, We have to implement the `thread_spin_lock()` and `thread_spin_unlock()` functions and put them around our atomic section.

To implement spinlock we have used 3 functions:

(1)Function to Initialize the lock to the correct initial state

(2)Function to acquire a lock

(3)Function to release lock



## Output with mutexes

We define a simple mutex data structure and implement three functions that:

(1) Initialize the mutex to the correct initial state

(2) Function to acquire a mutex

(3) a function to release it

Mutexes implementation is very similar to spinlocks. We use sleep(1) instead as xv6 doesn't have an explicit yield(0)

