

CS 344 ASSIGNMENT - 1

GROUP MEMBERS :

Arya Avinash Phadke 200101020

Mansi 200101064

Pranshu Pandya 200101073

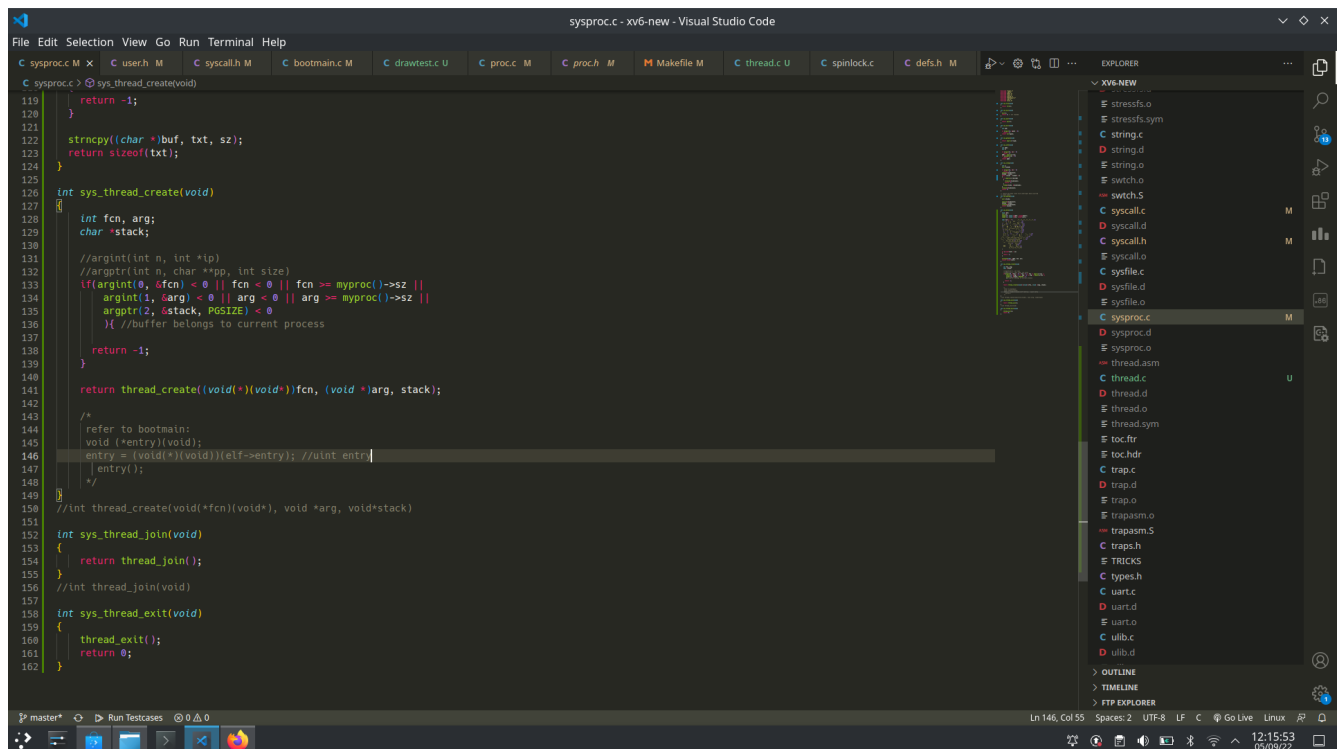
In this assignment, we had to implement kernel threads and then had to build spinlocks and mutexes to synchronize the access among the threads.

PART 1: Implementation of kernel threads

We have implemented system calls for **thread_create**, **thread_join** and **thread_exit**. For this we had to make changes in sysproc.c, user.h, syscall.h, syscall.c, proc.c, proc.h, makefile, defs.h, usys.S.

Sysproc.c

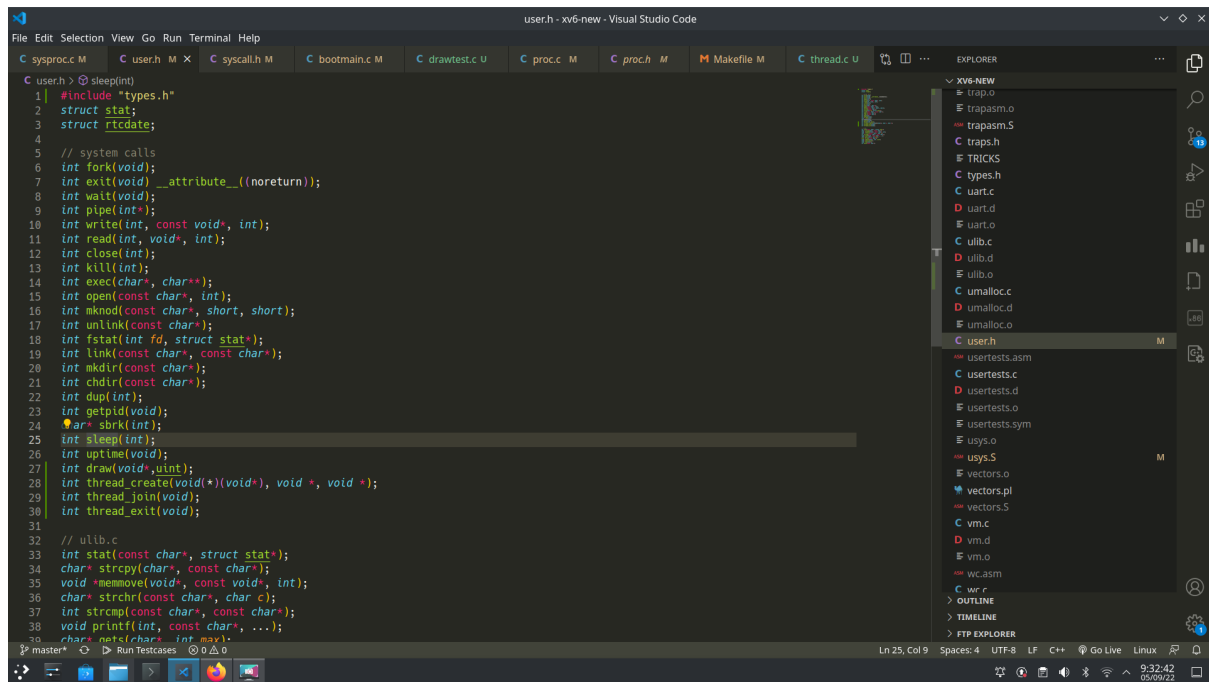
Here we have defined the system calls which call the main functions in proc.c where we have implemented thread_create, thread_join, thread_exit.



```
119     return -1;
120 }
121
122 strcpy((char *)buf, txt, sz);
123 return sizeof(txt);
124 }
125
126 int sys_thread_create(void)
127 {
128     int fcn, arg;
129     char *stack;
130
131     //argint(int n, int *ip)
132     //argptr(int n, char **pp, int size)
133     if(argint(0, &fcn) < 0 || fcn < 0 || fcn >= myproc()->sz ||
134        argint(1, &arg) < 0 || arg < 0 || arg >= myproc()->sz ||
135        argptr(2, &stack, PGSIZE) < 0
136        ){ //buffer belongs to current process
137         return -1;
138     }
139
140     return thread_create((void*)(void*)fcn, (void *)arg, stack);
141 }
142
143 /*
144  refer to bootmain:
145  void (*entry)(void);
146  entry = (void*)(void*)(elf->entry); //uint entry
147  entry();
148  */
149 //int thread_create(void(*fcn)(void*), void *arg, void*stack)
150
151 int sys_thread_join(void)
152 {
153     return thread_join();
154 }
155 //int thread_join(void)
156
157 int sys_thread_exit(void)
158 {
159     thread_exit();
160     return 0;
161 }
162 }
```

User.h

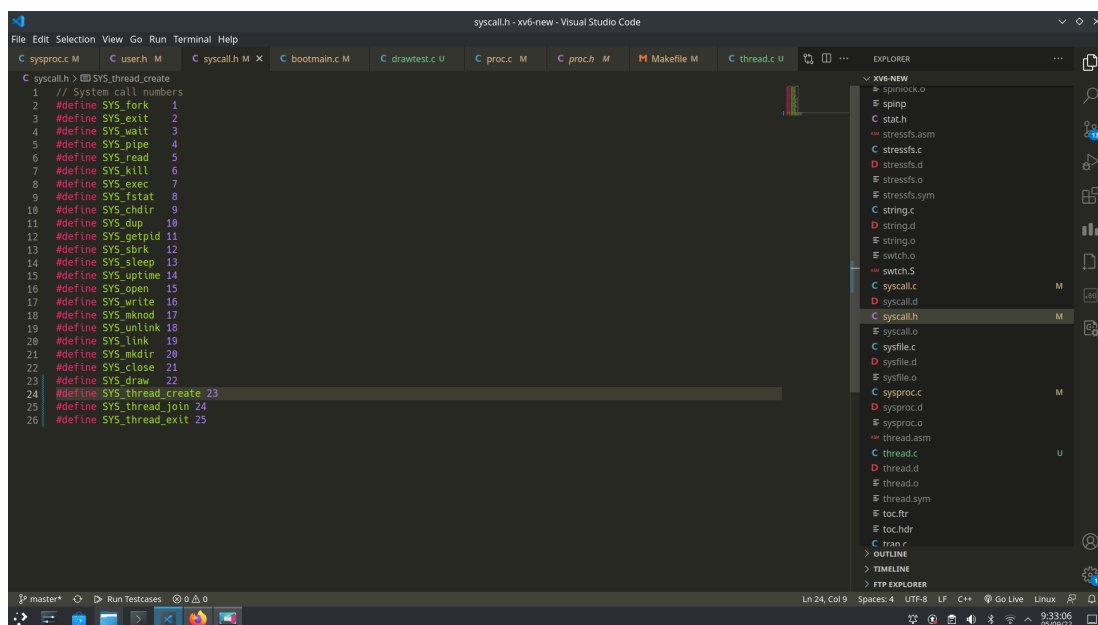
In this file, we have just declared our functions, `thread_create`, `thread_join` and `thread_exit`, in lines 28,29 and 30.



```
1 #include "types.h"
2 struct stat;
3 struct rtcdate;
4
5 // system calls
6 int fork(void);
7 int exit(void) __attribute__((noreturn));
8 int wait(void);
9 int pipe(int*);
10 int write(int, const void*, int);
11 int read(int, void*, int);
12 int close(int);
13 int kill(int);
14 int exec(char*, char**);
15 int open(const char*, int);
16 int mknod(const char*, short, short);
17 int unlink(const char*);
18 int fstat(int fd, struct stat*);
19 int link(const char*, const char*);
20 int mkdir(const char*);
21 int chdir(const char*);
22 int dup(int);
23 int getpid(void);
24 char* sbrk(int);
25 int sleep(int);
26 int uptime(void);
27 int draw(void, uint);
28 int thread_create(void*(*)(void*), void *, void *);
29 int thread_join(void);
30 int thread_exit(void);
31
32 // ulib.c
33 int stat(const char*, struct stat*);
34 char* strcpy(char*, const char*);
35 void *memmove(void*, const void*, int);
36 char* strchr(const char*, char c);
37 int strcmp(const char*, const char*);
38 void printf(int, const char*, ...);
39 char* nets(char*, int max);
```

Syscall.h

In this file, we have just defined the names of our functions as `SYS_thread_create`, `SYS_thread_join`, and `SYS_thread_exit`.



```
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_read 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_sbrk 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_mkdir 20
22 #define SYS_close 21
23 #define SYS_thread_create 23
24 #define SYS_thread_join 24
25 #define SYS_thread_exit 25
```

Proc.c

thread_create()

This function creates a new thread by forking the main process. This function has been taken up from the existing fork() function. The procedure for creating a new thread is almost the same with some changes which are explained below:

The function thread_create takes in, the following arguments:

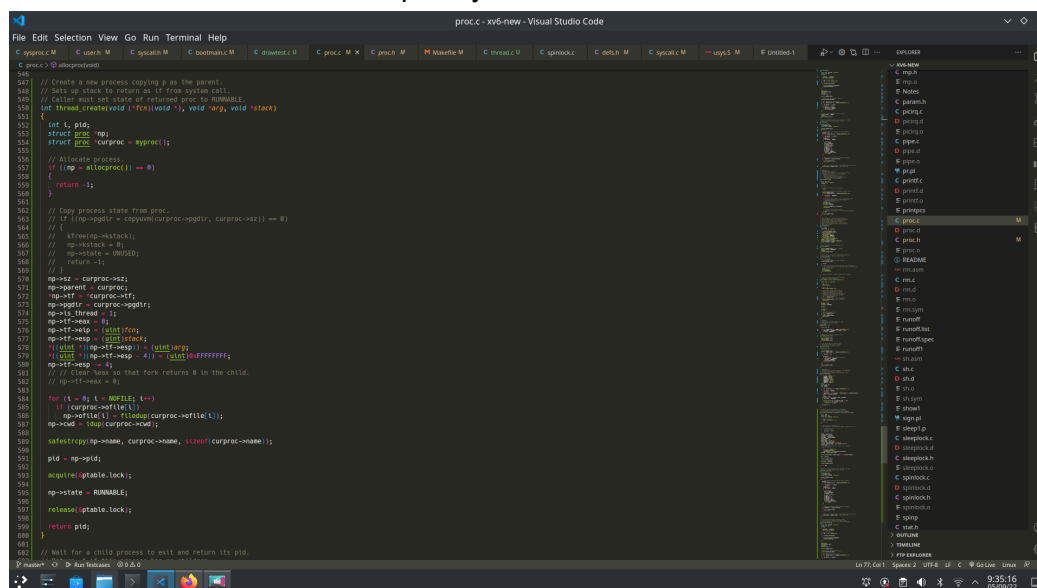
1. The function which will be using this thread for implementation. The new thread starts executing at the address space specified by this function.
2. The arguments required for the above function.
3. The user stack pointer of the stack to be used by this thread.

Firstly, we check if a process has been allocated. We are ensuring that there is a new process to be allocated to our thread. We have commented out the next few lines. In this they basically 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. Then a few things are initialized for the new thread we are making : the size is set as the parent's size, the trapframe for the new thread is set to that of the parent, the page directory(address space) for the new thread is set to that of the parent. Note here we are setting the is_thread flag as 1, to denote that we are creating a new thread.

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.

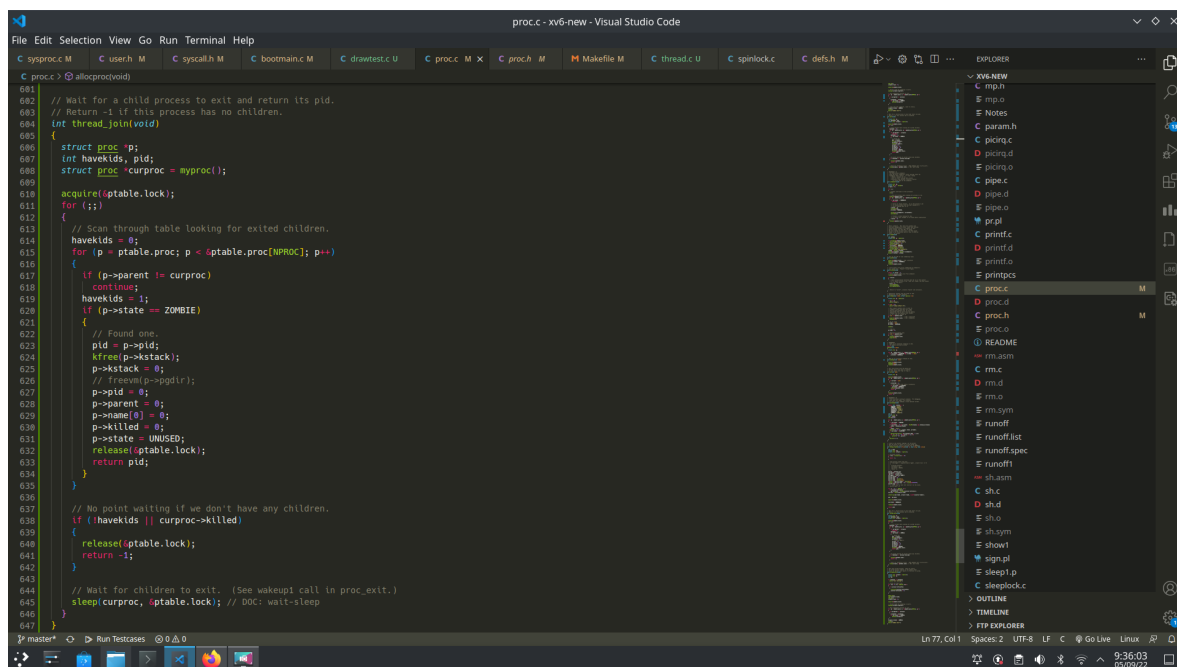
Now, because the thread has access to all the files opened by the parent process, we are spanning through all the files, and are duplicating and associating the files opened by the main process to the newly created thread.

Finally the state of the new thread is set. Here the state is set to RUNNABLE indicating that this thread is ready to be executed, and the thread is put in the ready queue. 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.



thread_join()

This function has been created using the wait() function which was already present in xv6. It takes nothing as the argument as it has to wait for the thread to complete its execution and then has to terminate the thread. In the function, we scan through all the threads/processes in the ptable and then, for the processes which are the child process(threads) of our main process, we check if the thread has completed the execution and has entered the zombie state, we deallocate all the resources given to the zombie thread and finally kill it and return the pid of the killed thread. If there is no thread in the zombie state i.e. all are still executing, it will wait for them to complete the execution and enter in the zombie state so as to be killed, and therefore, enters the sleep state. At the end, when all the threads are killed i.e. 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 killed thread is returned.

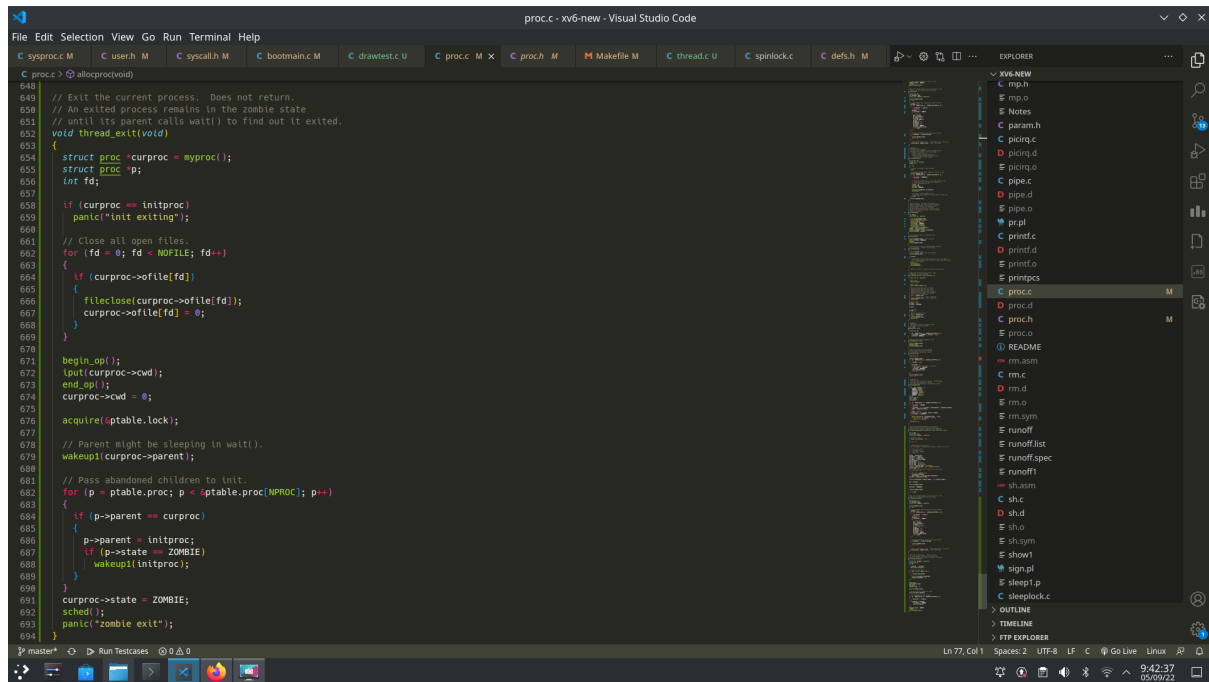


```
601 // Wait for a child process to exit and return its pid.
602 // Return -1 if this process has no children.
603 int thread_join(void)
604 {
605     struct proc *p;
606     int havekids, pid;
607     struct proc *curproc = myproc();
608     acquire(&ptable.lock);
609     for (;;)
610     {
611         // Scan through table looking for exited children.
612         havekids = 0;
613         for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
614         {
615             if (p->parent != curproc)
616                 continue;
617             havekids = 1;
618             if (p->state == ZOMBIE)
619             {
620                 // Found one.
621                 pid = p->pid;
622                 kfree(p->stack);
623                 p->stack = 0;
624                 // Free(p->pgdir);
625                 p->pid = 0;
626                 p->parent = 0;
627                 p->name[0] = 0;
628                 p->state = UNUSED;
629                 release(&ptable.lock);
630                 return pid;
631             }
632         }
633     }
634     // No point waiting if we don't have any children.
635     if (!havekids || curproc->killed)
636     {
637         release(&ptable.lock);
638         return -1;
639     }
640     // Wait for children to exit. (See wakeup1 call in proc_exit.)
641     sleep(curproc, &ptable.lock); // DOC: wait-sleep
642 }
```

thread_exit()

This function allows a thread to terminate. Again, it takes no argument as it checks for all the processes in the ptable to check if they are completed. This has also been picked up from the already existing exit function. 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 it is then we run the wakeup1 function, and then we set the state of the thread

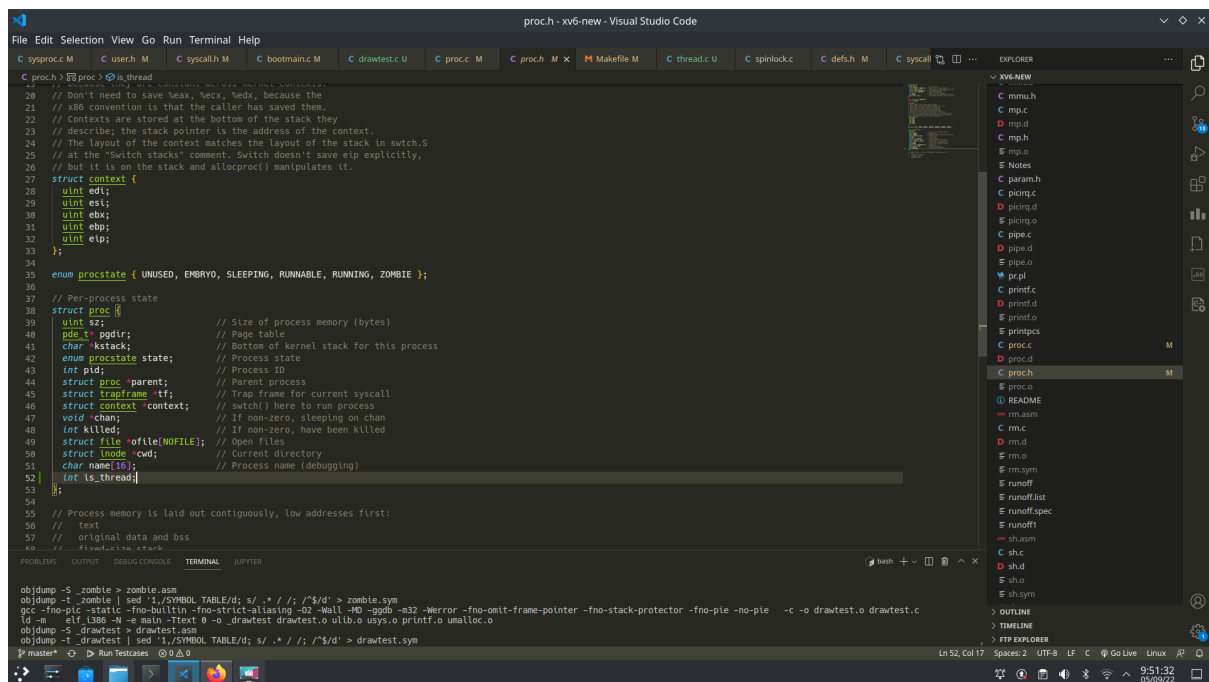
as zombie, indicating that the process has completed.



```
648 // Exit the current process. Does not return.
649 // An exited process remains in the zombie state
650 // until its parent calls wait() to find out it exited.
651 void thread_exit(void)
652 {
653     struct proc *curproc = myproc();
654     struct proc *p;
655     int fd;
656
657     if (curproc == initproc)
658         panic("init exiting");
659
660     // Close all open files.
661     for (fd = 0; fd < NOFILE; fd++)
662     {
663         if (curproc->ofile[fd])
664         {
665             fileclose(curproc->ofile[fd]);
666             curproc->ofile[fd] = 0;
667         }
668     }
669
670     begin_op();
671     puti(curproc->cwd);
672     end_op();
673     curproc->cwd = 0;
674
675     acquire(&table.lock);
676
677     // Parent might be sleeping in wait().
678     wakeup(curproc->parent);
679
680     // Pass abandoned children to init.
681     for (p = table.proc; p < &table.proc[NPROC]; p++)
682     {
683         if (p->parent == curproc)
684         {
685             p->parent = initproc;
686             if (p->state == ZOMBIE)
687                 wakeup(initproc);
688         }
689     }
690
691     curproc->state = ZOMBIE;
692     sched();
693     panic("Zombie exit");
694 }
```

Proc.h

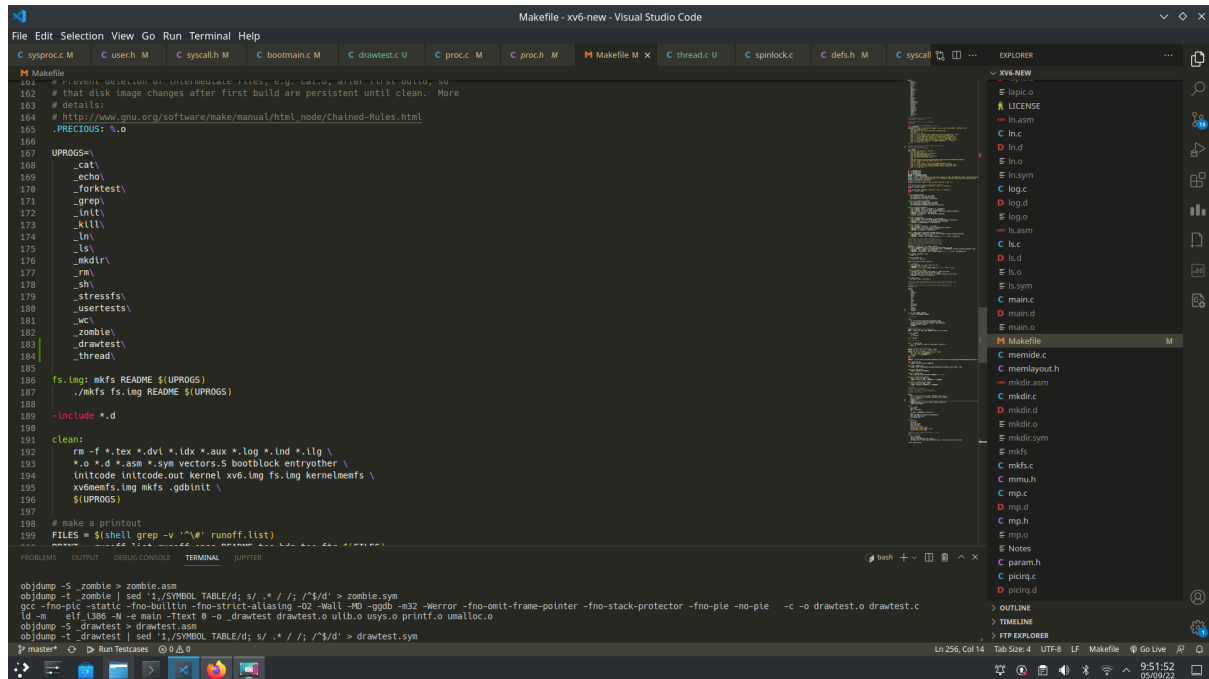
In this file, we are making a change to the struct proc - we are adding another int field called is_thread. This is done so we do not have to implement an entire new struct to represent a thread. If the value of is_thread is 1, then the struct represents a thread, and if it is 0, then the struct represents a process.



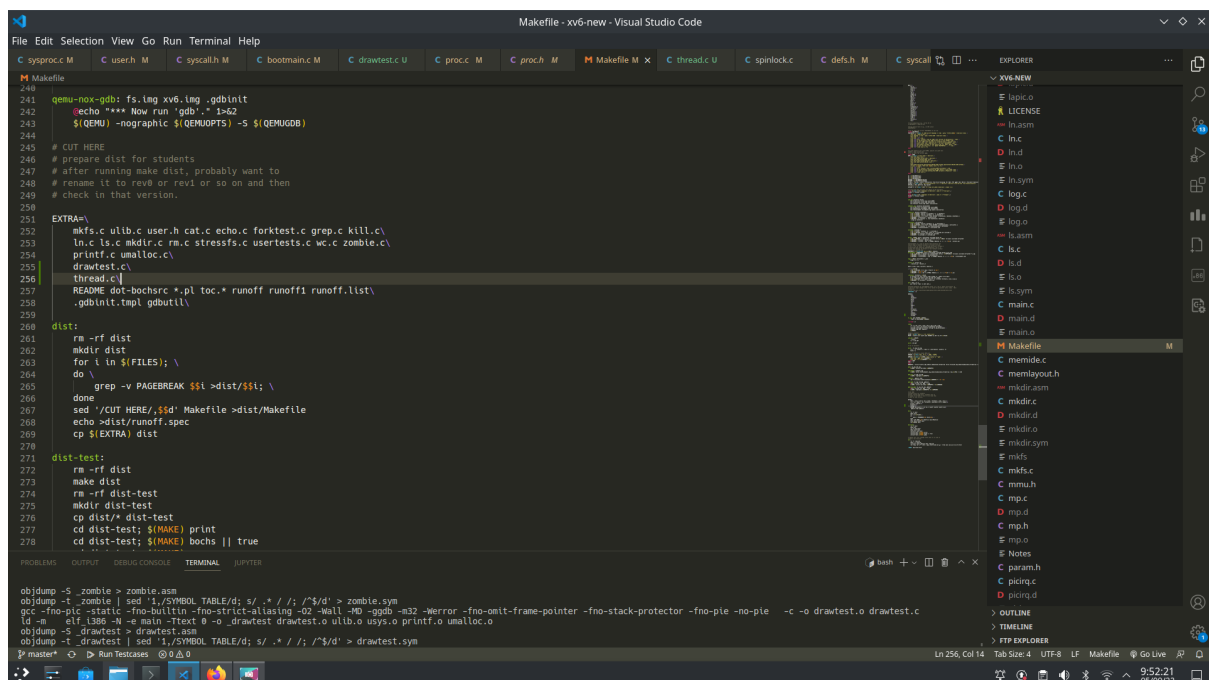
```
28 // Don't need to save %eax, %ecx, %edx, because the
29 // x86 convention is that the caller has saved them.
30 // Contexts are stored at the bottom of the stack in
31 // describe; the stack pointer is the address of the context.
32 // The layout of the context matches the layout of the stack in switch.S
33 // at the "switch stacks" comment. Switch doesn't save eip explicitly,
34 // but it is on the stack and allocproc() manipulates it.
35 struct context {
36     uint edi;
37     uint esi;
38     uint ebx;
39     uint ebp;
40     uint eip;
41 };
42
43 enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
44
45 // Per-process state
46 struct proc {
47     uint sz; // Size of process memory (bytes)
48     pde_t *pgdir; // Page table
49     char *kstack; // Bottom of kernel stack for this process
50     enum procstate state; // Process state
51     int pid; // Process ID
52     struct proc *parent; // Parent process
53     struct trapframe *tf; // Trap frame for current syscall
54     struct context *context; // switch() here to run process
55     void *chan; // If non-zero, sleeping on chan
56     int killed; // If non-zero, have been killed
57     struct file *ofile[NOFILE]; // Open files
58     struct inode *cwd; // Current directory
59     char name[10]; // Process name (debugging)
60     int is_thread;
61 };
62
63 // Process memory is laid out contiguously, low addresses first:
64 // text
65 // original data and bss
66 // fixed-size no-stack
67 // variable-size no-stack
68 // stack
```

Makefile

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



```
162 # that disk image changes after first build are persistent until clean. More
163 # details:
164 # http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
165 .PRECIOUS: %.o
166
167 UPROGS=\
168     _cat\
169     _echo\
170     _forktest\
171     _grep\
172     _init\
173     _kill\
174     _ln\
175     _ls\
176     _mkdir\
177     _rm\
178     _sh\
179     _stressfs\
180     _userstests\
181     _wc\
182     _zombie\
183     _drawtest\
184     _thread\
185
186 fs.img: mkfs README $(UPROGS)
187     ./mkfs fs.img README $(UPROGS)
188
189 -include *.d
190
191 clean:
192     rm -f *.tex *.dvi *.idx *.aux *.log *.ind *.ilg \
193         *.o *.d *.asm *.sym vectors.S bootblock entryother \
194         initcode initcode.out kernel xv6.img fs.img kernelmemfs \
195         xv6memfs.img mkfs.gdbinit \
196         $(UPROGS)
197
198 # make a printout
199 FILES = $(shell grep -v '^#' runoff.list)
200
201 objdump -S _zombie > zombie.asm
202 gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o drawtest.o drawtest.c
203 ld -m elf_i386 -N -e main -Ttext 0 -o _drawtest drawtest.o ulib.o usys.o printf.o umalloc.o
204 objdump -S _drawtest > drawtest.asm
205 objdump -t _drawtest | sed '1,/SYMBOL TABLE/d; s/ .*/ /; /"/%d/' > drawtest.sym
206 master* @0.0.0
```



```
241 qemu-nox-gdb: fs.img xv6.img .gdbinit
242     echo "*** Now run 'gdb' -s 162
243     $(QEMU) -nographic -s $(QEMUOPTS) -S $(QEMUGDB)
244
245 # CUT HERE
246 # prepare dist for students
247 # after running make dist, probably want to
248 # rename it to rev0 or rev1 or so on and then
249 # check in that version.
250
251 EXTRA=\
252     mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
253     ln.c ls.c mkdir.c rm.c stressfs.c userstests.c wc.c zombie.c\
254     printf.c umalloc.c\
255     drawtest.c\
256     thread.c\
257     README dot-bochsrc *.pl toc.* runoff runoff1 runoff.llist\
258     .gdbinit.tmpl gdbutil\
259
260 dist:
261     rm -rf dist
262     mkdir dist
263     for i in $(FILES); \
264     do \
265         grep -v PAGEBREAK $$i >dist/$$i; \
266     done
267     sed '/CUT HERE/,$$d' Makefile >dist/Makefile
268     echo >dist/runoff.spec
269     cp $(EXTRA) dist
270
271 dist-test:
272     rm -rf dist
273     make dist
274     rm -rf dist-test
275     mkdir dist-test
276     cp dist/* dist-test
277     cd dist-test; $(MAKE) print
278     cd dist-test; $(MAKE) bochs || true
```

Defs.h

We are defining all the functions which we have created. For part1, we have created three new functions : thread_create, thread_join and thread_exit.

```

101 int piperead(struct pipe*, char*, int);
102 int pipewrite(struct pipe*, char*, int);
103
104 //PAGEBREAK: 16
105 // proc.c
106 int cpuid(void);
107 void exit(void);
108 int fork(void);
109 int groupproc(int);
110 int kill(int);
111 struct cpu* mycpu(void);
112 struct proc* myproc();
113 void print(void);
114 void procmap(void);
115 void scheduler(void) __attribute__((noreturn));
116 void sched(void);
117 void setproc(struct proc*);
118 void sleep(void, struct spinlock*);
119 void userinit(void);
120 int wait(void);
121 void wakeup(void*);
122 void yield(void);
123 int thread_create(void (*)(void *), void *, void *);
124 int thread_join(void);
125 void thread_exit(void);
126 // switch.S
127 void switch(struct context*, struct context*);
128
129 // spinlock.c
130 void acquire(struct spinlock*);
131 void getcallerpcs(void*, uint*);
132 int holding(struct spinlock*);
133 void initlock(struct spinlock*, char*);
134 void release(struct spinlock*);
135 void pushcli(void);
136 void popcli(void);
137
138 // sleeplock.c
139 void acquiresleep(struct sleeplock*);

```

Syscall.c

Here we are defining our function and we will call the main function which we have written in proc.c

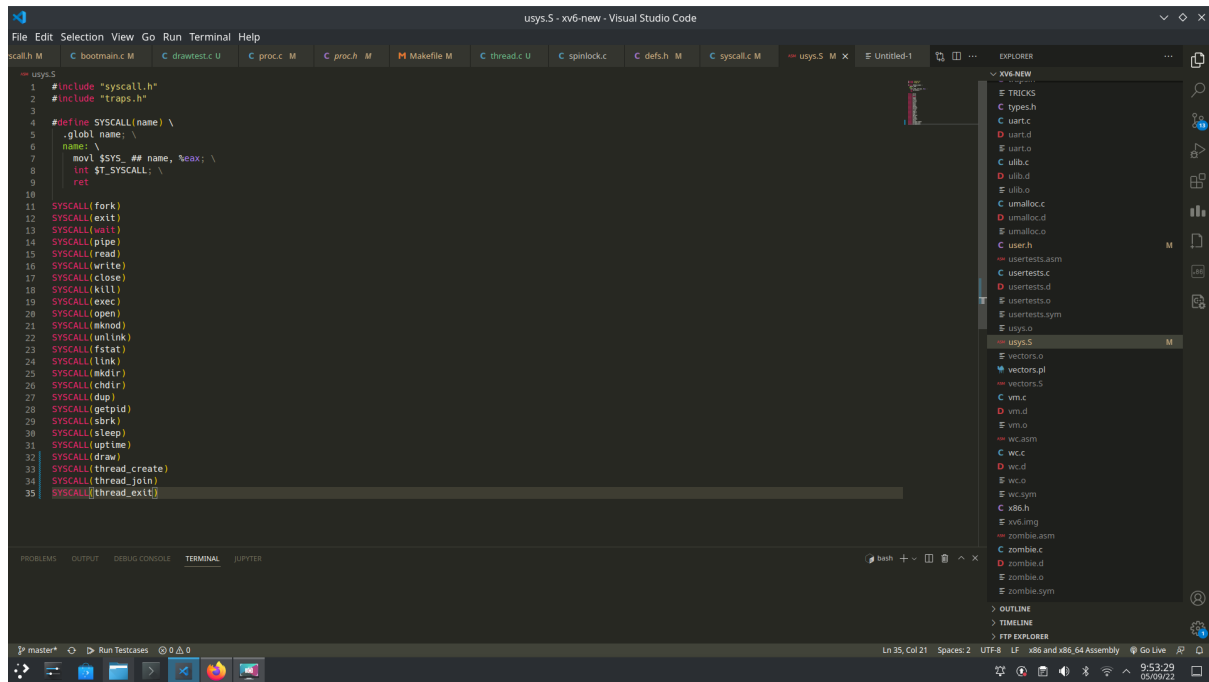
```

105 extern int sys_uptime(void);
106 extern int sys_draw(void);
107 extern int sys_thread_create(void);
108 extern int sys_thread_join(void);
109 extern int sys_thread_exit(void);
110
111 static int (*syscalls[])(void) = {
112     [SYS_fork] sys_fork,
113     [SYS_exit] sys_exit,
114     [SYS_wait] sys_wait,
115     [SYS_pipe] sys_pipe,
116     [SYS_read] sys_read,
117     [SYS_kill] sys_kill,
118     [SYS_exec] sys_exec,
119     [SYS_fstat] sys_fstat,
120     [SYS_chdir] sys_chdir,
121     [SYS_dup] sys_dup,
122     [SYS_getpid] sys_getpid,
123     [SYS_sbrk] sys_sbrk,
124     [SYS_sleep] sys_sleep,
125     [SYS_uptime] sys_uptime,
126     [SYS_open] sys_open,
127     [SYS_write] sys_write,
128     [SYS_mknod] sys_mknod,
129     [SYS_unlink] sys_unlink,
130     [SYS_link] sys_link,
131     [SYS_mkdir] sys_mkdir,
132     [SYS_close] sys_close,
133     [SYS_draw] sys_draw,
134     [SYS_thread_create] sys_thread_create,
135     [SYS_thread_join] sys_thread_join,
136     [SYS_thread_exit] sys_thread_exit,
137 };
138
139 void
140 syscall(void)
141 {
142     int num;

```

usys.S

This is added to implement the system call for thread.

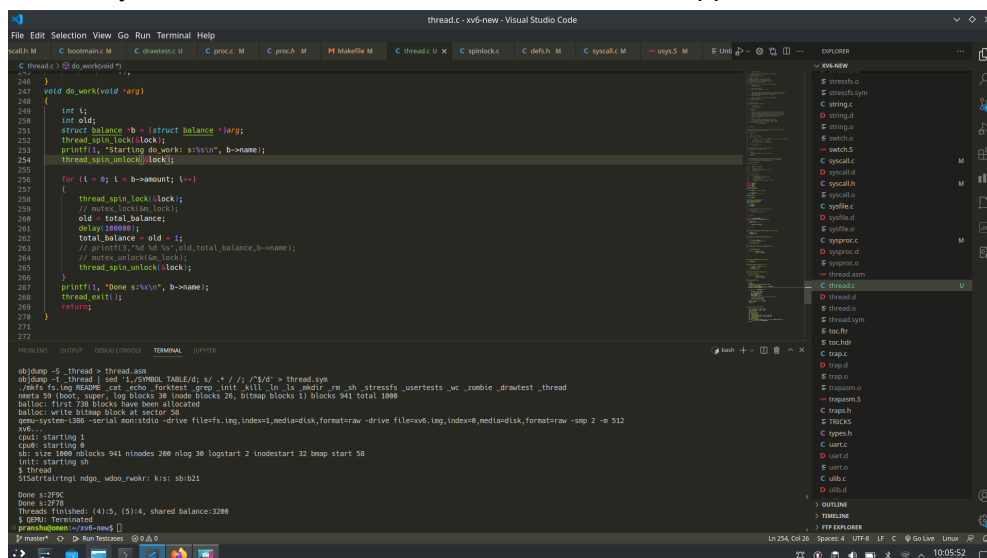


Part 2: Synchronization

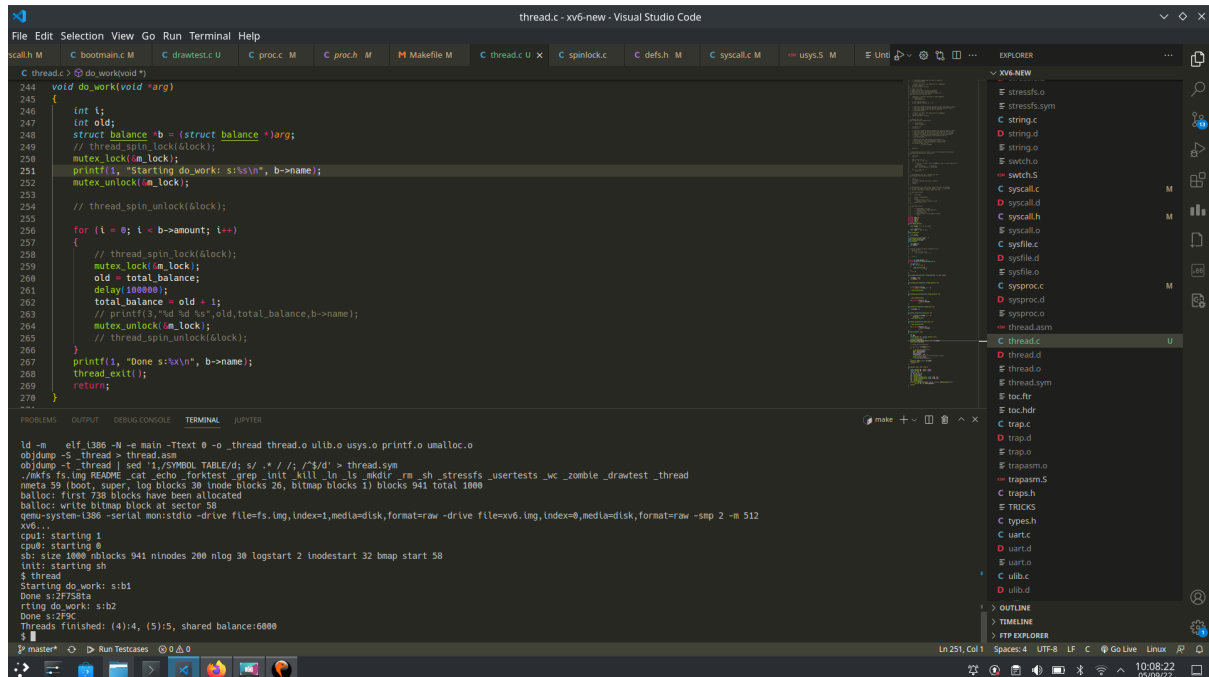
Output without synchronization

Here after executing the thread command without implementing any locks, we can see that the total balance is not 6000. This happens because there is no synchronization and context switching keeps happening. Here total balance is the shared variable, and due to context switching, before the entire set of instructions in the critical section of one process gets executed, context switching occurs and the other process uses the outdated value of the variable only hence the total does not add up to 6000. This can be prevented by adding locks.

The delay statement has been added here to show that not adding locks causes a problem. The delay statement ensures that a context switch happens in between the critical section.



Output with spinlocks



We have defined a struct called lock - it has two fields : a name given to the lock and the int which holds the value of the lock.

```
struct thread_spinlock
{
    uint locked; // Is the lock held?

    // For debugging
    char *name; // Name of lock.
};
```

thread_initlock - this is the initialization function. We are setting the name of the lock variable and setting the value of lk to 0. This means that no thread is executing its critical section right now.

```
void thread_initlock(struct thread_spinlock *lk, char *name)
{
    lk->name = name;
    lk->locked = 0;
}
```

thread_spin_lock - the xchg instruction in the while loop condition basically compares the value of the parameters passed, here it compares the value of the lock and 1. It returns the value of lock and sets the value of the lock to 1(the second parameter). We use this special instruction as it is atomic and hence there is no risk of context switching. So the thread will be stuck in the while loop if the value of lock is 1, meaning some other thread is executing their critical section. The while loop keeps running until the value of lock becomes 0 (busy waiting) and then we come out of the while loop. Now the value of the lock is 1 again and the current thread has 'acquired' the lock, meaning it can execute its critical section safely.

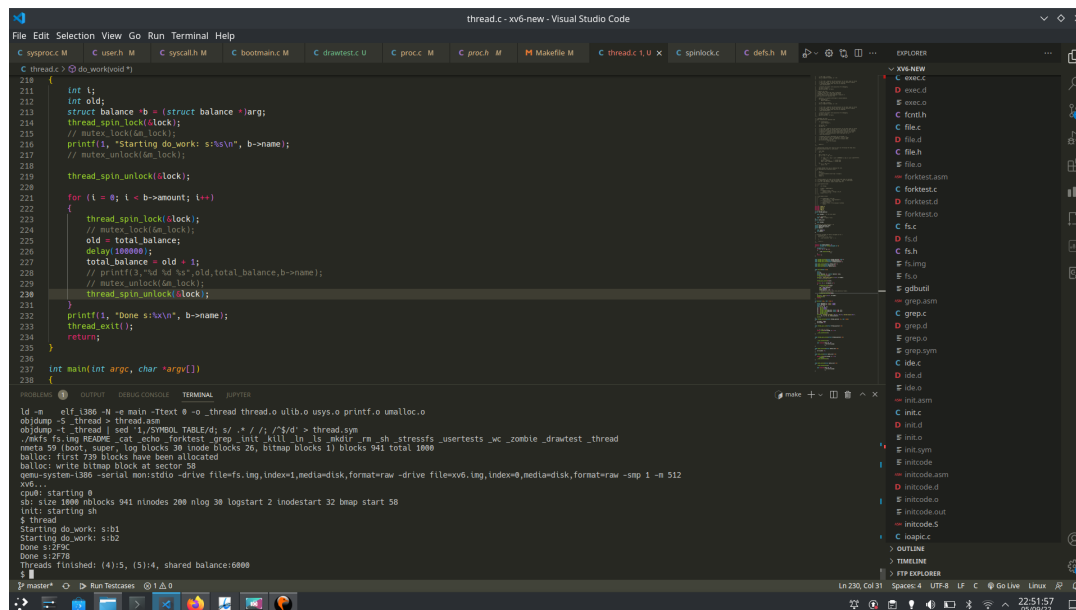
__sync_synchronize() is used to ensure that no memory operand will be moved across the operation, either forward or backward. Further, instructions will be issued as necessary to prevent the processor from speculating loads across the operation and from queuing stores after the operation.

```
void thread_spin_lock(struct thread_spinlock *lk)
{
    // The xchg is atomic.
    while (xchg(&lk->locked, 1) != 0)
        ;
    __sync_synchronize();
}
```

thread_unlock - This function sets the value of lock back to 0. This is executed after the critical section has been completed to set the lock back to 0, now other processes can execute their critical sections.

```
void thread_spin_unlock(struct thread_spinlock *lk)
{
    __sync_synchronize();

    asm volatile("movl $0, %0"
                  : "+m"(lk->locked)
                  :);
}
```

A screenshot of the Visual Studio Code editor with a C program using pthread_spinlock_t. The code defines a shared variable 'balance' and a 'do_work' function that increments it in a loop, protected by a spinlock. The main function creates two threads, 's1' and 's2', each calling 'do_work' 5000 times. The terminal output shows the execution of the program, including the creation of threads and the final shared balance of 6000.

```
218 int i;  
219 int old;  
220 struct balance *b = (struct balance *)arg;  
221 pthread_spin_lock(&lock);  
222 // mutex_lock(&lock);  
223 printf(i, "Starting do_work: s:%s\n", b->name);  
224 // mutex_unlock(&lock);  
225  
226 pthread_spin_unlock(&lock);  
227  
228 for (i = 0; i < b->amount; i++)  
229 {  
230     pthread_spin_lock(&lock);  
231     // mutex_lock(&lock);  
232     old = total_balance;  
233     delay(100000);  
234     total_balance = old + 1;  
235     // printf(i, "old %d\n", old, total_balance, b->name);  
236     // mutex_unlock(&lock);  
237     pthread_spin_unlock(&lock);  
238 }  
239 printf(i, "Done s:%s\n", b->name);  
240 pthread_exit(i);  
241  
242 int main(int argc, char *argv[])  
243 {  
244     ...  
245     ...  
246     ...  
247     ...  
248     ...  
249     ...  
250     ...  
251     ...  
252     ...  
253     ...  
254     ...  
255     ...  
256     ...  
257     ...  
258     ...  
259     ...  
260     ...  
261     ...  
262     ...  
263     ...  
264     ...  
265     ...  
266     ...  
267     ...  
268     ...  
269     ...  
270     ...  
271     ...  
272     ...  
273     ...  
274     ...  
275     ...  
276     ...  
277     ...  
278     ...  
279     ...  
280     ...  
281     ...  
282     ...  
283     ...  
284     ...  
285     ...  
286     ...  
287     ...  
288     ...  
289     ...  
290     ...  
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    ...
```

Output with spinlock

Mutexes

Now, we will implement synchronization with the help of mutex. The advantage that mutex has over spin locks is that it won't wait in the while loop while another process is running the critical section. Instead, it goes into sleep, and frees up the processor.

We have defined the following functions for mutex implementation:

Firstly, we define the structure of mutex_lock where, we define just one parameter, which is the value of the lock, 1 represents lock is activated, and 0 represents the lock is free.

```
struct mutex_lock  
{  
    uint locked;  
};
```

mutex_initlock

This function initializes the lock, and sets it to 0.

```
void mutex_initlock(struct mutex_lock *lk)  
{  
    lk->locked = 0;  
}
```

mutex_lock

This function takes in the lock as the argument. Here, the xchg is the swap and compare function. Therefore, if the lock has a value of 0, we won't enter the while loop as in the condition for the while loop, we compare the value of lock with 1, as it was initially 0, the value returned will be 0, and the value of lock will be set to 1. If the value of lock is 1 when the mutex_lock is called, it would enter the while loop and wait there until the value of lock is not reset to 0 as, until then, it would return 1 only, and therefore, would be stuck in the while loop. Now, after this, the process would sleep and the critical region would be executed.

```
void mutex_lock(struct mutex_lock *lk)
{
    while (xchg(&lk->locked, 1) != 0)
        sleep(1);
    __sync_synchronize();
}
```

mutex_unlock

This function releases the lock by setting the value of lock back to 0. The asm volatile("movl \$0,%0"

```
    : "+m"(lk->locked)
    : );
```

part sets the lock to 0 atomically.

```
void mutex_unlock(struct mutex_lock *lk)
{
    __sync_synchronize();

    asm volatile("movl $0, %0"
                  : "+m"(lk->locked)
                  : );
}
```

Output

Now, using this mutex lock around the critical section (lines 225-228), we find that the total balance comes out to be 6000, which shows that the lock was implemented correctly. The screenshot of the output is shown below:

```
thread.c - xv6-new - Visual Studio Code
File Edit Selection View Go Run Terminal Help
C sysproc.c M C user.h M C syscall.h M C bootmain.c M C drawtest.c U C proc.c M C proc.h M M Makefile M C thread.c 1 U X C spinlock.c C defs.h M ... EXPLORER
C thread.c > Q mutex_unlock(mutex_lock *)
218 {
219     int i;
220     int old;
221     struct balance *b = (struct balance *)arg;
222     // thread_spin_lock(&lock);
223     mutex_lock(&m_lock);
224     printf(1, "Starting do_work: s:%s\n", b->name);
225     mutex_unlock(&m_lock);
226     // thread_spin_unlock(&lock);
227
228     for (i = 0; i < b->amount; i++)
229     {
230         // thread_spin_lock(&lock);
231         mutex_lock(&m_lock);
232         old = total_balance;
233         delay(10000);
234         total_balance = old + 1;
235         // printf(3, "hd %d %s", old, total_balance, b->name);
236         mutex_unlock(&m_lock);
237         // thread_spin_unlock(&lock);
238     }
239     printf(1, "Done s:%s\n", b->name);
240     thread_exit();
241     return;
242 }
243
244 int main(int argc, char *argv[])
245 {
246     // ...
247 }
248
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
objdump -S thread > thread.asm
objdump -t _thread | sed '1,/SYMBOL TABLE/d; s/ / /; /"/s/d' > thread.sym
./mkfs fs.img README cat_echo_forktest_grep_init_kill_ln_ls_mkdir_rm_sh_stressfs_userstests_wc_zombie_drawtest_thread
meta: 59 boot, super, log blocks 38 inode blocks 26, bitmap blocks 1 blocks 941 total 1000
ballocc: first 728 blocks have been allocated
ballocc: write bitmap block at sector 58
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ thread
Starting do_work: s:b1
Starting do_work: s:b2
Done s:2F78
Done s:2F9C
Threads finished: (4):4, (5):5, shared balance:0000
$ qemu Terminated
pranshugomen ~/xv6-new
```

Ln 299, Col 2 (54 selected) Spaces: 4 UTF-8 LF C Go Live Linux A?

22:51:06 09/09/22