Documentation for xv6 Operating System Project

Project Title: Extending xv6 with Mutex, Threads, IPC, and Signal System Calls

Course Name: Operating Systems

Group Members: CS22B2020-Sriram,

CS22B2023-Madhan,

CS22B2024-Harsha,

CS22B2017-Dhanush.

Table of Contents

---

1. Introduction

The purpose of this project is to extend the functionality of the xv6 operating system by implementing and modifying system calls that support **mutexes**, **threads**, **inter-process communication (IPC)**, and **signals**. These enhancements will enable concurrent execution, synchronization, and communication between processes and threads in the operating system.

---

2. Overview of Existing System Calls in xv6

Xv6 provides basic system calls for process creation, termination, and basic process management. However, it lacks advanced features like thread management, synchronization, and process communication. This project aims to introduce the following system calls:

- **Mutex**: For mutual exclusion between threads.
- **Threads**: For managing lightweight processes within the operating system.
- **IPC (Inter-process Communication)**: For enabling communication between processes.
- **Signals**: For handling asynchronous notifications between processes.

---

3. System Calls Implemented

This section describes each of the newly implemented system calls.

System Call 1: Mutex

Functionality:
The mutex system call allows synchronization between threads by providing mutual exclusion, ensuring that only one thread accesses a critical section at a time.

To implement mutexes and threads in your project, you have modified and added several files to the system. Here's how you've organized your implementation:

Mutex Implementation:

1. syscall.h:

- Added system call numbers for SYS_mutex_lock and SYS_mutex_unlock to define unique identifiers for the new system calls.

2. syscall.c:

- Declared sys_mutex_lock and sys_mutex_unlock functions.

- Added these functions to the syscalls array to link system call numbers with their implementations.

3. sysproc.c:

- Implemented sys_mutex_lock and sys_mutex_unlock system call handler functions.

- Used a global volatile unsigned integer mutex as a simple spinlock to control access to shared resources.

4. usys.S:

- Added the assembly stubs

mutex_lock and mutex_unlock to enable user programs to invoke the system calls.

5. user.h:

- Declared the user-space functions

int mutex_lock(void);

and

int mutex_unlock(void);

for use in applications.

6. **defs.h**:

- Declared the prototypes for sys_mutex_lock and sys_mutex_unlock to make them visible across the kernel codebase.

7. **mutex_imple.c**:

- Created a user-level program to test mutex functionality.

- Implemented a shared counter incremented by multiple processes while using mutex_lock and mutex_unlock to protect the critical section.

Reasoning:

- System Calls Integration*: By adding new system calls and their handlers, user programs can safely request mutex operations from the kernel.

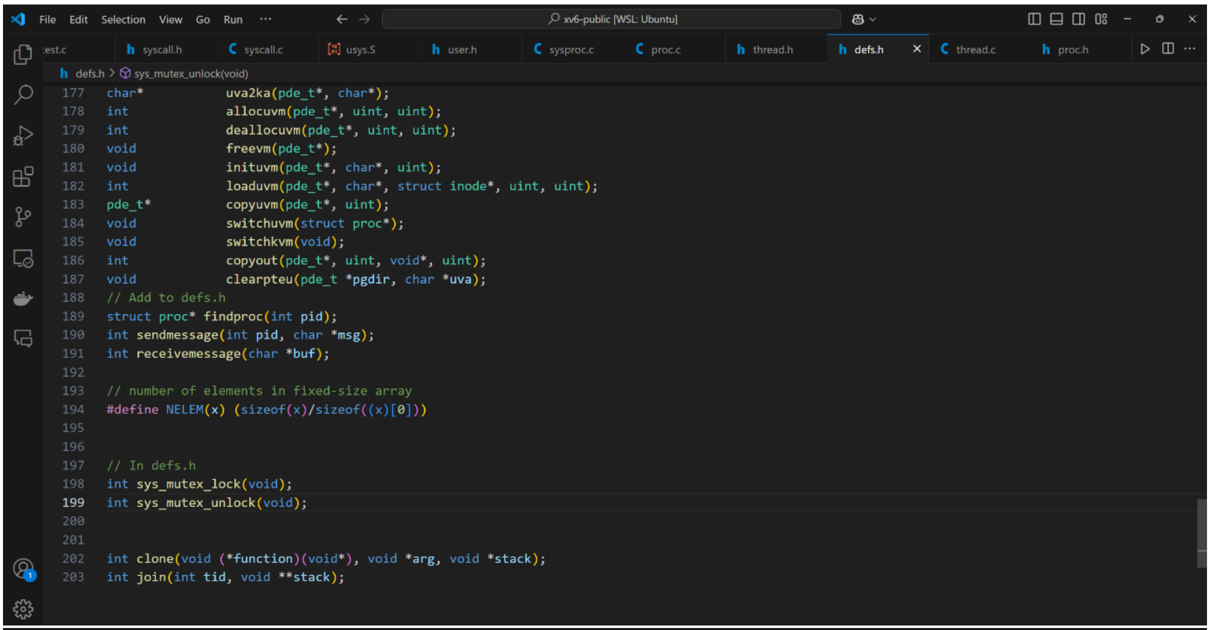
- Global Mutex Variable*: Using a global mutex provides a simple mechanism for mutual exclusion across processes.

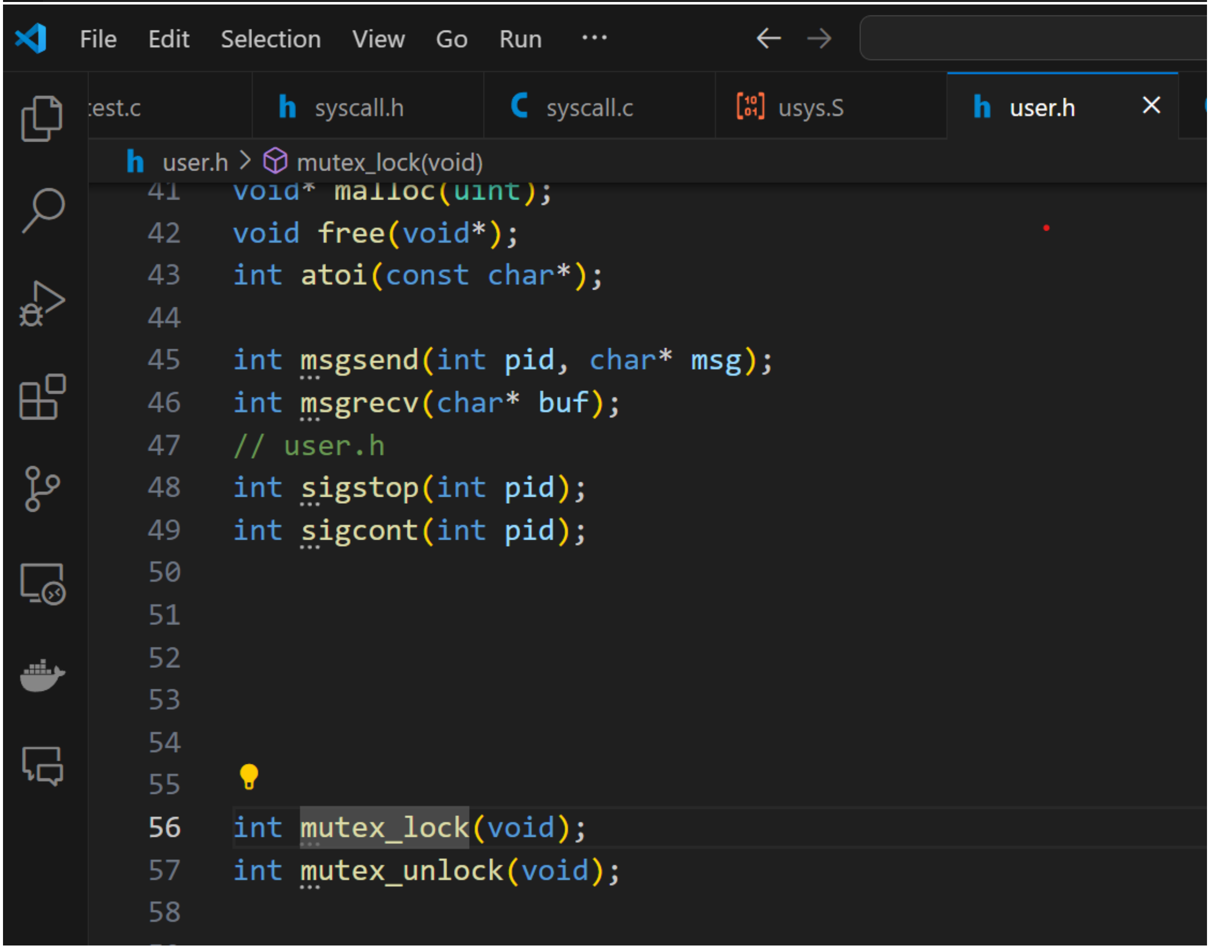
- User-Space Access: Updating

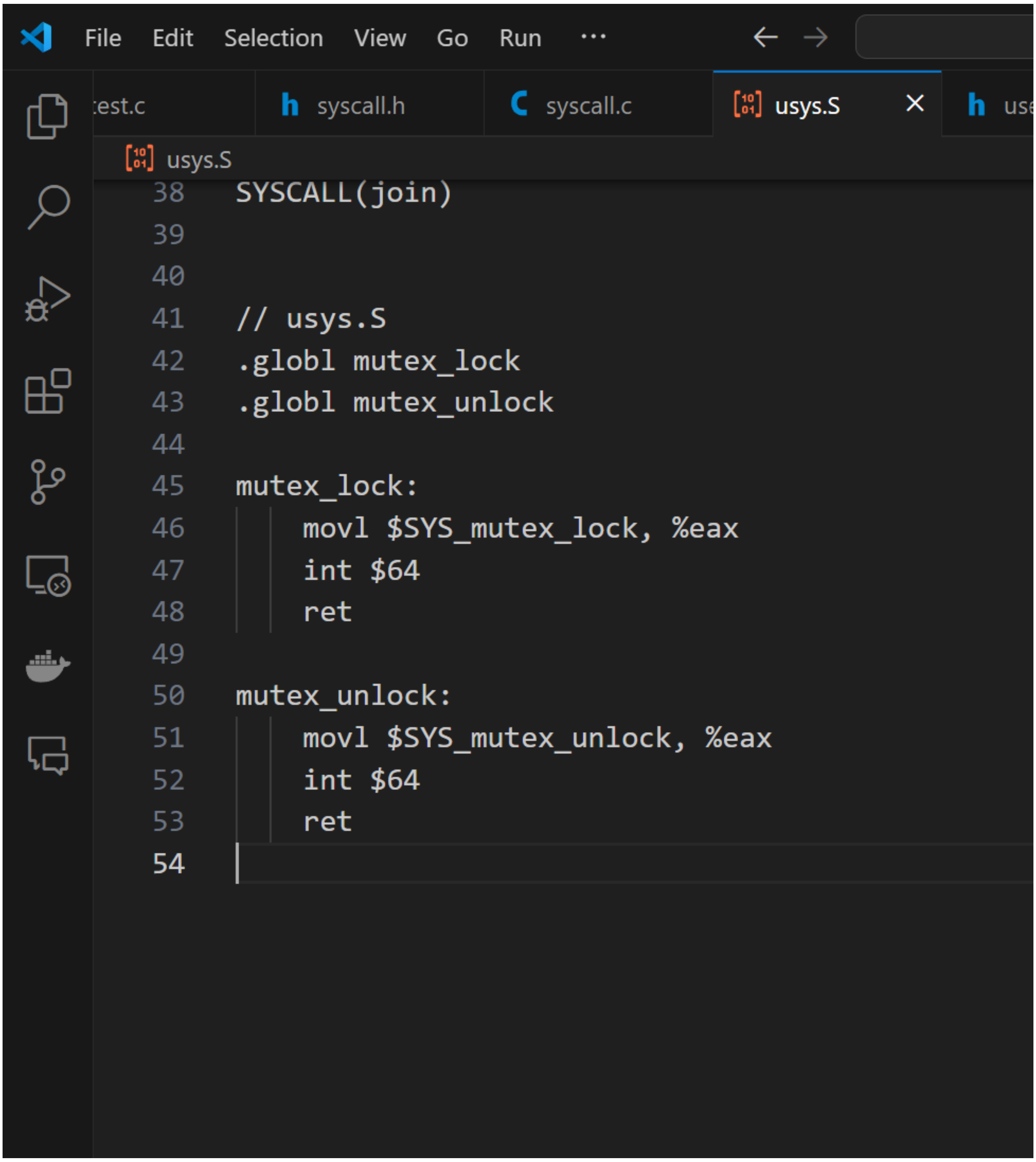
usys.S and user.h allows user programs to call mutex functions just like any other system call.

- Testing: The mutex_imple.c program demonstrates and validates the correct behavior of the mutex implementation.

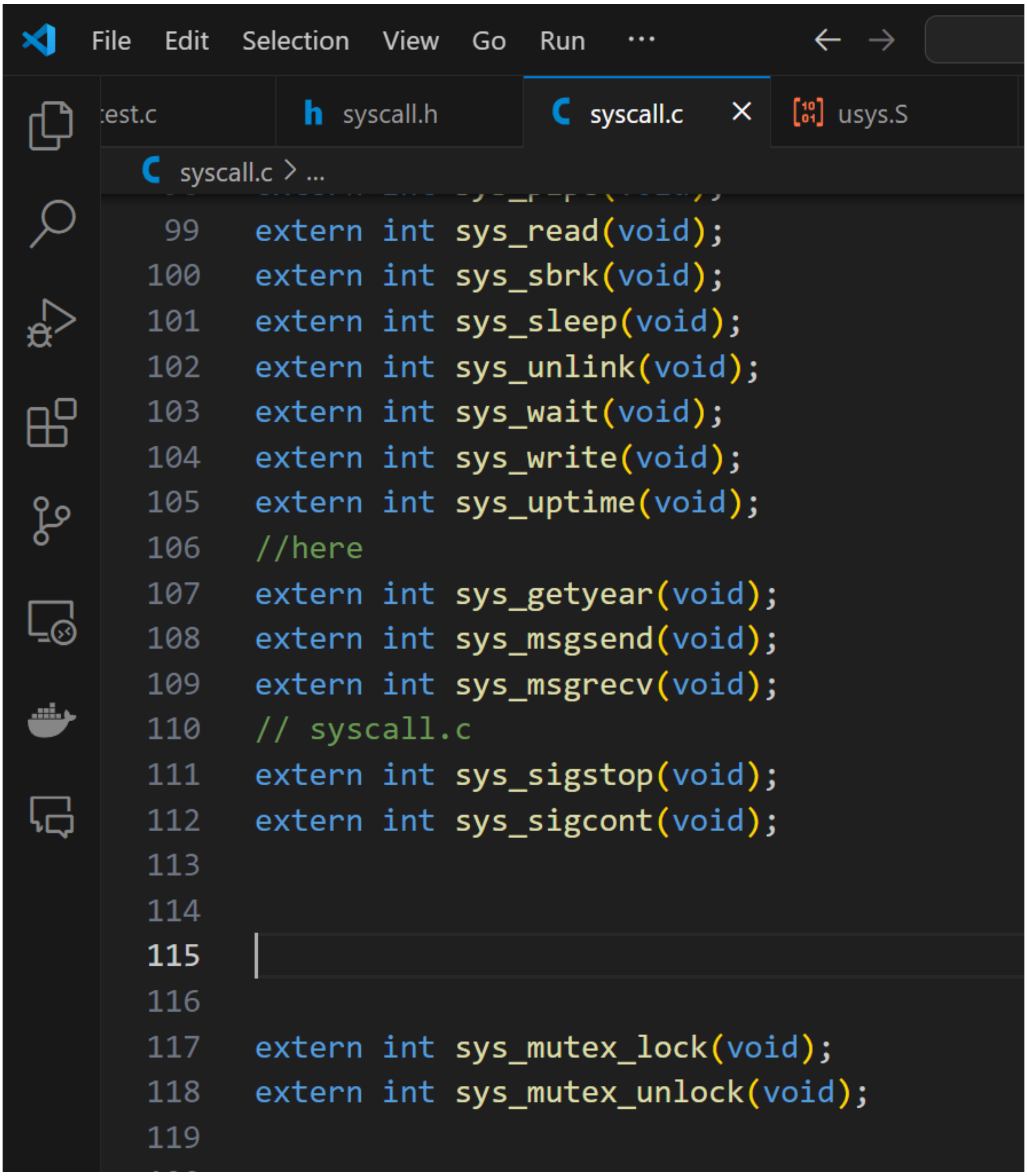
EXCUETION:

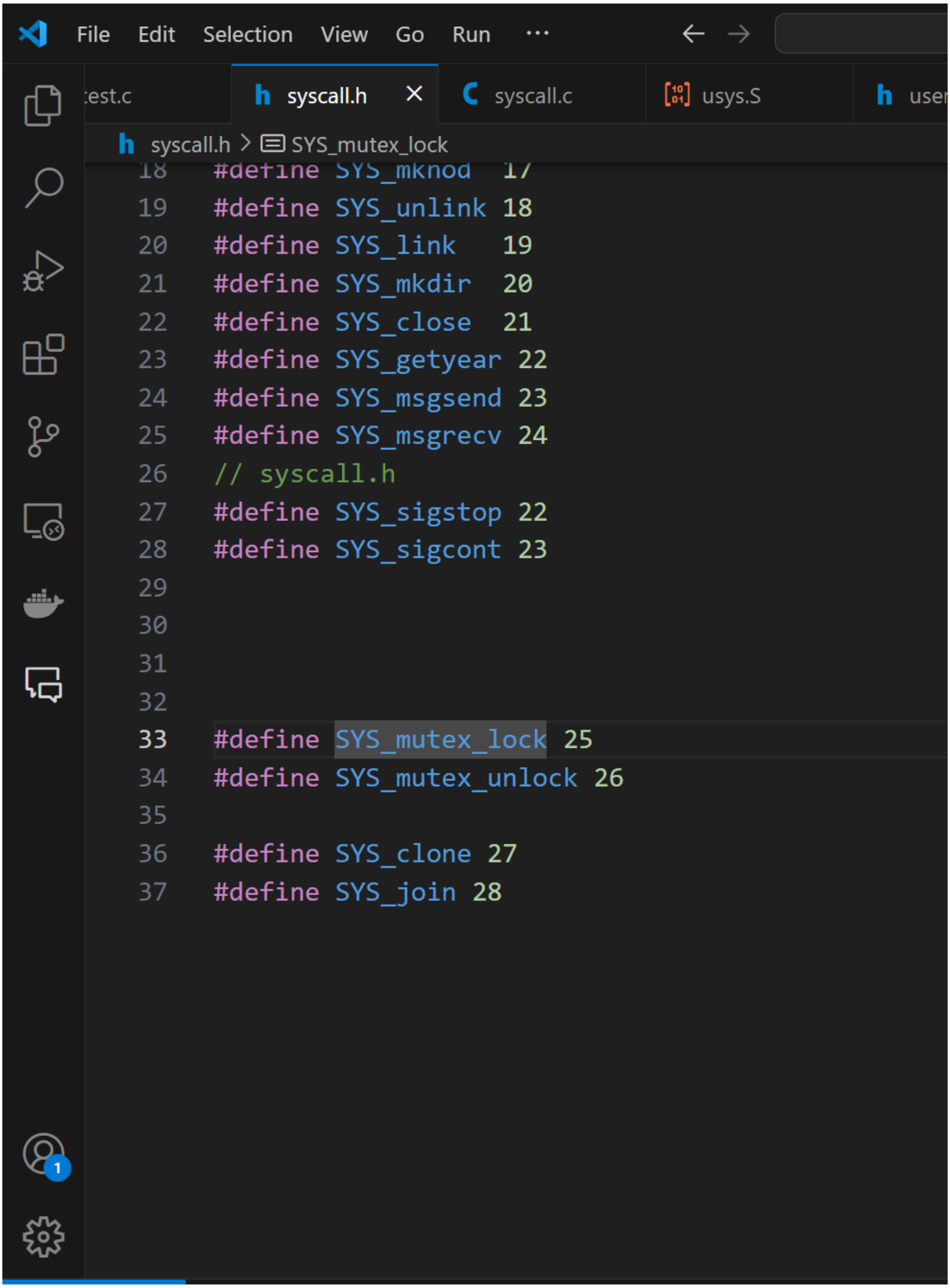

File Edit Selection View Go Run
xv6-public [WSL: Ubuntu]
est.c syscall.h syscall.c usys.S user.h sysproc.c proc.c thread.h defs.h thread.c proc.h
defs.h > sys_mutex_unlock(void)
char* uva2ka(pde_t*, char*);
int allocuvm(pde_t*, uint, uint);
int deallocuvm(pde_t*, uint, uint);
void freevm(pde_t*);
void inituvm(pde_t*, char*, uint);
int loaduvm(pde_t*, char*, struct inode*, uint, uint);
pde_t* copyuvm(pde_t*, uint);
void switchuvm(struct proc*);
void switchkvm(void);
int copyout(pde_t*, uint, void*, uint);
void clearpteu(pde_t *pgdir, char *uva);
// Add to defs.h
struct proc* findproc(int pid);
int sendmessage(int pid, char *msg);
int receivemessage(char *buf);
// number of elements in fixed-size array
#define NELEM(x) (sizeof(x)/sizeof((x)[0]))
// In defs.h
int sys_mutex_lock(void);
int sys_mutex_unlock(void);
int clone(void (*function)(void*), void *arg, void *stack);
int join(int tid, void **stack);


File Edit Selection View Go Run
est.c syscall.h syscall.c usys.S user.h
user.h > mutex_lock(void)
void* malloc(uint);
void free(void*);
int atoi(const char*);
int msgsend(int pid, char* msg);
int msgrecv(char* buf);
// user.h
int sigstop(int pid);
int sigcont(int pid);
int mutex_lock(void);
int mutex_unlock(void);


File
Edit
Selection
View
Go
Run
est.c
syscall.h
syscall.c
usys.S
usys.S
SYSCALL(join)
// usys.S
.globl mutex_lock
.globl mutex_unlock
mutex_lock:
movl $SYS_mutex_lock, %eax
int $64
ret
mutex_unlock:
movl $SYS_mutex_unlock, %eax
int $64
ret

```c
// Inside sysproc.c

volatile unsigned int mutex = 0; // Global mutex variable (simple binary lock)

int sys_mutex_lock(void)
{
    // Spin until the lock is acquired (basic spinlock)
    while (xchg(&mutex, 1) != 0)
        ; // wait
    return 0;
}

int sys_mutex_unlock(void)
{
    // set mutex back to unlocked
    mutex = 0;
    return 0;
}
```


File
Edit
Selection
View
Go
Run
syscall.h
syscall.c
usys.S
syscall.c > ...
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
//here
extern int sys_getyear(void);
extern int sys_msgsend(void);
extern int sys_msgrecv(void);
// syscall.c
extern int sys_sigstop(void);
extern int sys_sigcont(void);
extern int sys_mutex_lock(void);
extern int sys_mutex_unlock(void);


File
Edit
Selection
View
Go
Run
test.c
syscall.h
syscall.c
usys.S
syscall.h > SYS_mutex_lock
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_getyear 22
#define SYS_msgsend 23
#define SYS_msgrecv 24
// syscall.h
#define SYS_sigstop 22
#define SYS_sigcont 23
#define SYS_mutex_lock 25
#define SYS_mutex_unlock 26
#define SYS_clone 27
#define SYS_join 28

```c
#include "user.h"

int shared_counter = 0;

void
increment_counter(void)
{
    for (int i = 0; i < 10; i++) {
        mutex_lock();              // Lock
        shared_counter++;          // Critical section
        printf(1, "Counter: %d\n", shared_counter);
        mutex_unlock();            // Unlock
    }
}

int main(void)
{
    if (fork() == 0) {
        increment_counter();
        exit();
    } else {

        increment_counter();
    }
    wait();
    exit();
}

```

System Call 2: Threads

Functionality:
This system call allows the creation of threads within a process. Threads share the same memory space but can execute concurrently, improving resource utilization.
Code Modifications:

Threads Implementation:

1. proc.h:

 - Modified the struct proc to include a pointer void *tstack; representing the thread's user stack.

 - This change allows each thread to have its own stack while sharing the same address space.

2. proc.c :- Implemented the clone function to create a new thread that shares the address space with the parent process.

 - Modified allocproc to accommodate thread-specific initializations.

 - Implemented join to allow a process to wait for a thread to finish and clean up resources.

3.syscall.h: - Added system call numbers for SYS_clone and SYS_join to assign unique identifiers

4. syscall.c:

- Declared sys_clone and sys_join.

- Added them to the syscalls array to connect system call numbers with their implementations.

5. sysproc.c:

- Implemented sys_clone to handle the clone system call from user space.

- Implemented sys_join to handle the join system call, enabling synchronization with threads.

6. usys.S:

- Added assembly stubs for clone and join to facilitate user-level access to threading functionality.

7. user.h:

- Declared user-space functions

int clone(void (function)(void), void *arg, void *stack);

and

int join(int tid, void **stack);

.

8. thread.h and thread.c:

- Created a user-level threading library.

- Implemented thread_create, thread_join, and thread_exit functions.

- thread_create wraps around clone, handling stack allocation and alignment.

- thread_join wraps around join, managing stack deallocation after a thread finishes.

9. test.c:

- Wrote a test program to validate thread creation and execution.

- Demonstrated creating multiple threads, each performing computations and updating shared data.

Reasoning:

- Process Structure Enhancement: Modifying proc.h and proc.c allows threads to share the same memory space while maintaining separate execution contexts (stacks and registers).

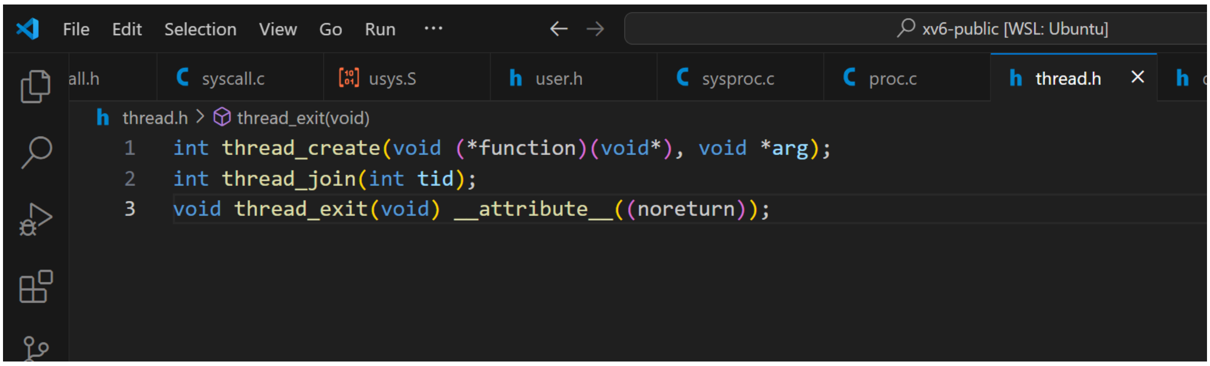
- System Call Additions: Introducing clone and join system call provides the necessary kernels support for threading .

- User-Level Thread Library: Abstracting thread operations in thread.h and thread .c simplifies thread management for applications.

- Testing: The test.c program ensures that threads are created, execute concurrently, and interact correctly with shared data.

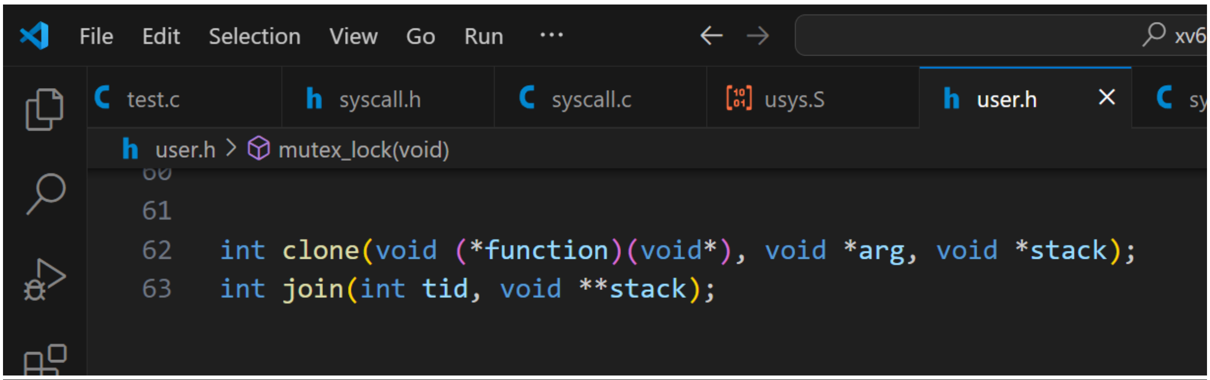
By organizing the code this way, you've extended the operating system to support threading and mutual exclusion, enabling concurrent execution of processes and safe access to shared resources.

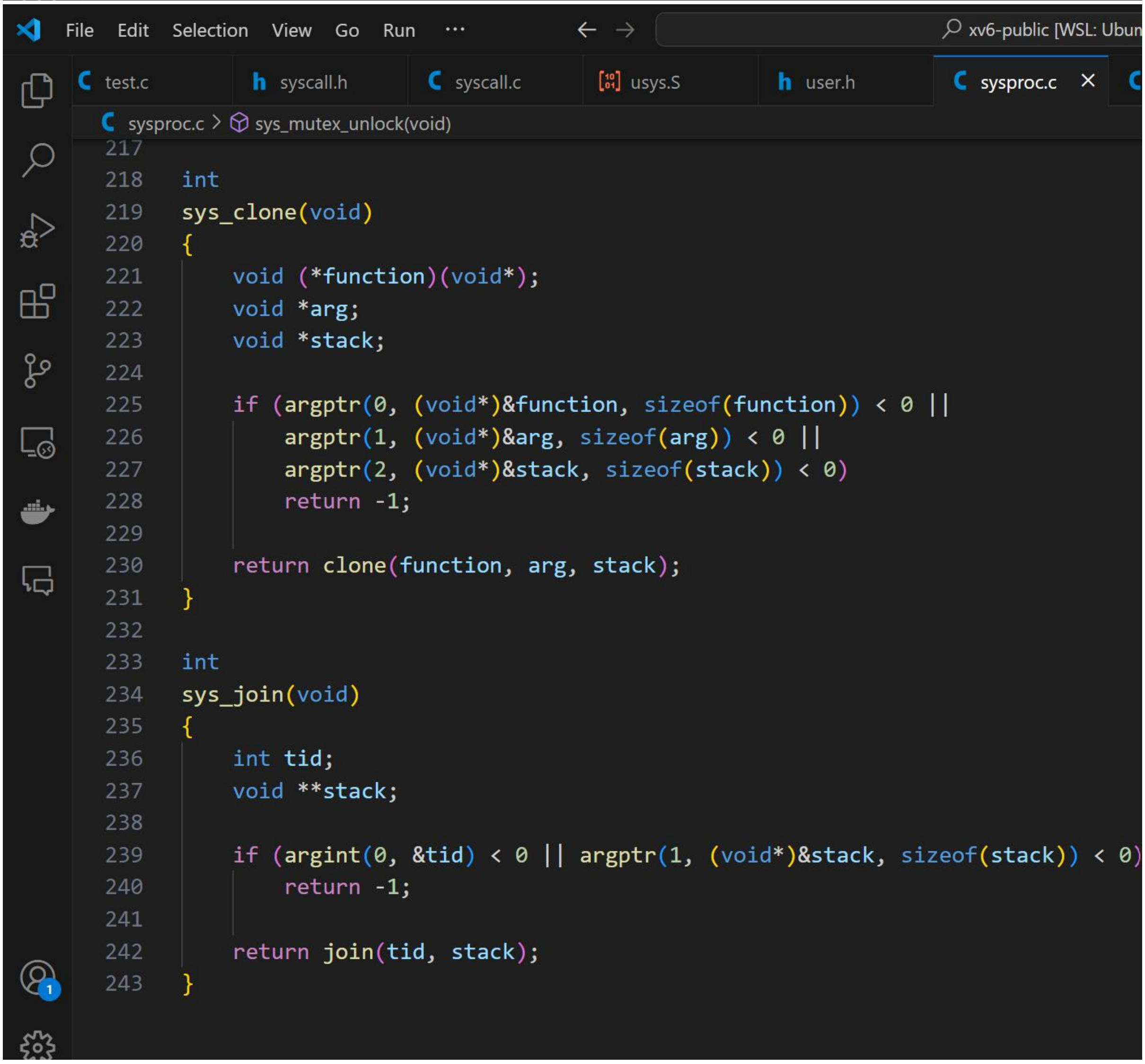
EXCUETION :


File Edit Selection View Go Run ···
xv6-public [WSL: Ubuntu]
all.h
syscall.c
usys.S
user.h
sysproc.c
proc.c
thread.h
thread.h > thread_exit(void)
int thread_create(void (*function)(void*), void *arg);
int thread_join(int tid);
void thread_exit(void) __attribute__((noreturn));

thread.c

```c
#include "types.h"
#include "user.h"
#include "mmu.h"
#include "thread.h"

int
thread_create(void (*function)(void*), void *arg)
{
    void *stack = malloc(PGSIZE);
    if (stack == 0)
        return -1;

    // Align stack to page boundary
    if ((uint)stack % PGSIZE)
        stack = stack + (PGSIZE - (uint)stack % PGSIZE);

    int tid = clone(function, arg, stack);
    if (tid < 0)
        free(stack);

    return tid;
}

int
thread_join(int tid)
{
    void *stack;
    int ret = join(tid, &stack);
    if (ret > 0)
        free(stack);
    return ret;
}


void thread_exit(void) {
    exit();
}
```


File Edit Selection View Go Run ···
test.c
syscall.h
syscall.c
usys.S
user.h
user.h > mutex_lock(void)
int clone(void (*function)(void*), void *arg, void *stack);
int join(int tid, void **stack);

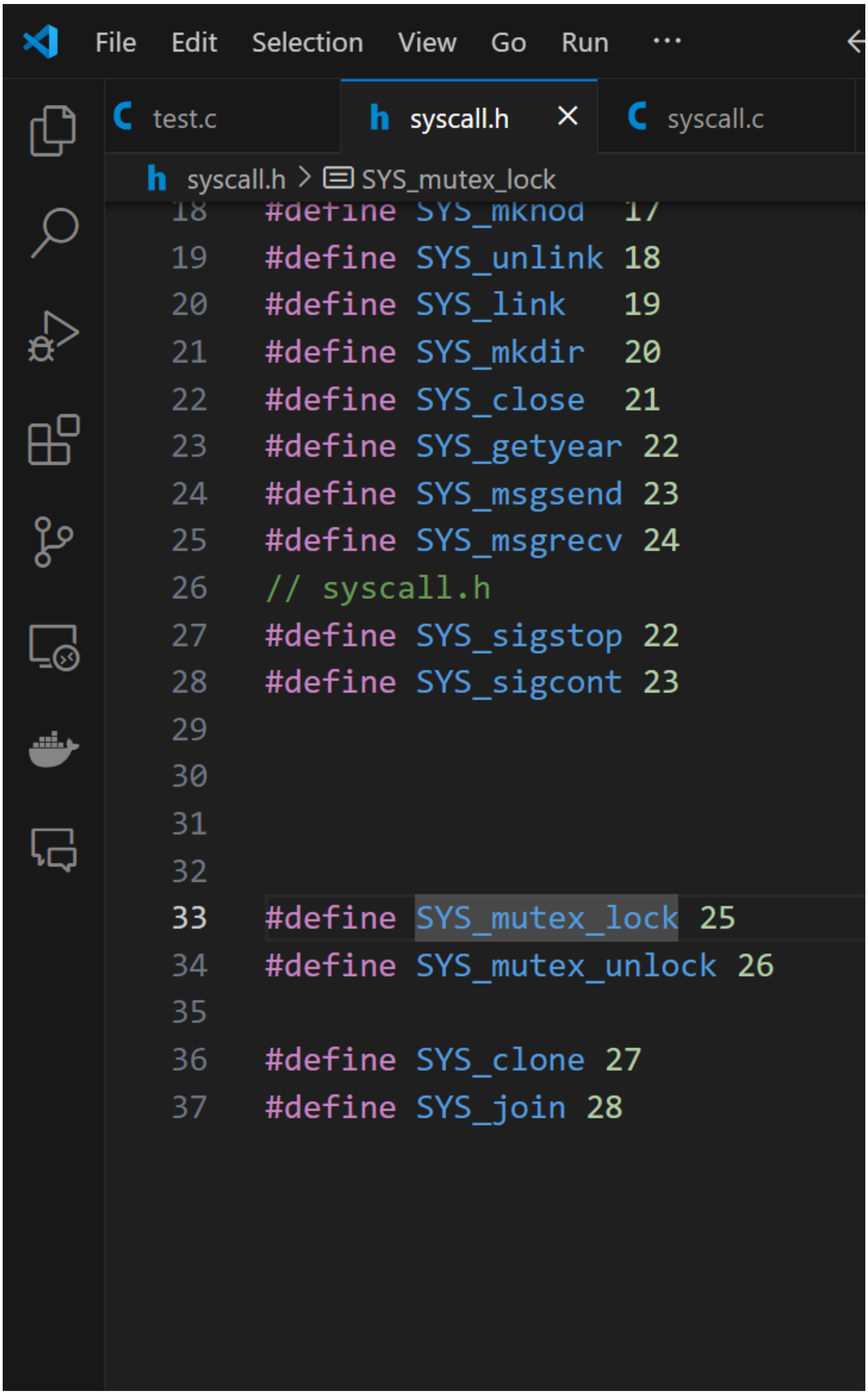

File Edit Selection View Go Run ···
xv6-public [WSL: Ubun
test.c
syscall.h
syscall.c
usys.S
user.h
sysproc.c
sysproc.c > sys_mutex_unlock(void)
int
sys_clone(void)
{
void (*function)(void*);
void *arg;
void *stack;
if (argptr(0, (void*)&function, sizeof(function)) < 0 ||
argptr(1, (void*)&arg, sizeof(arg)) < 0 ||
argptr(2, (void*)&stack, sizeof(stack)) < 0)
return -1;
return clone(function, arg, stack);
}
int
sys_join(void)
{
int tid;
void **stack;
if (argint(0, &tid) < 0 || argptr(1, (void*)&stack, sizeof(stack)) < 0)
return -1;
return join(tid, stack);
}

File Edit Selection View Go Run ···

test.c | syscall.h | syscall.c

syscall.c > ...

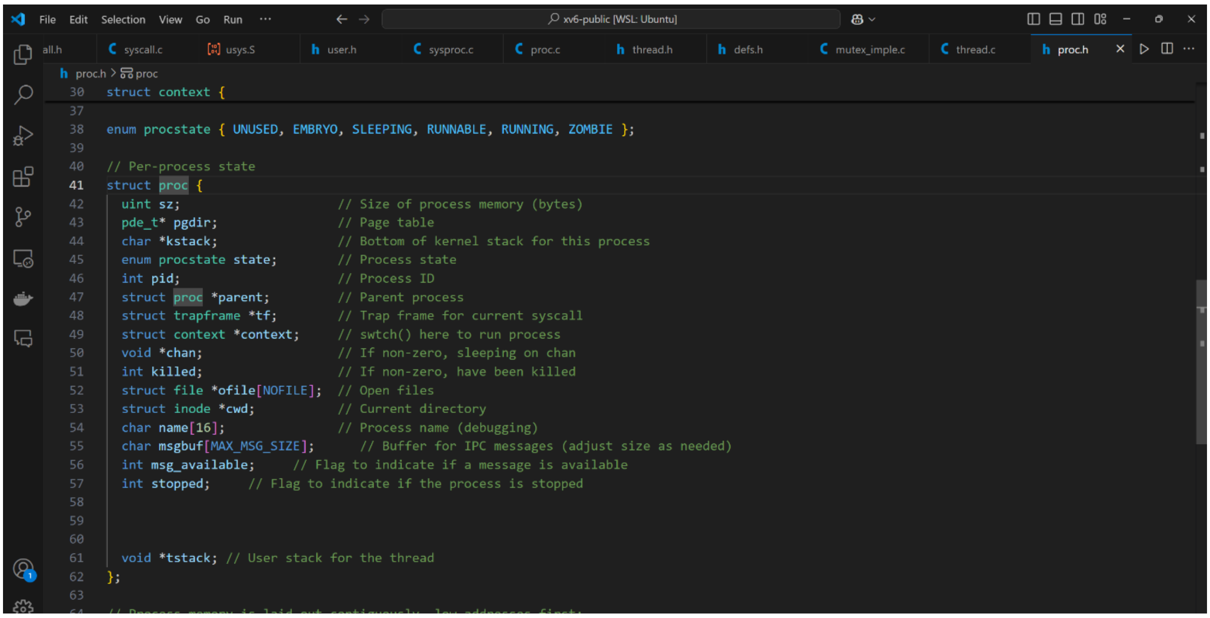
```c
static int (*syscalls[])(void) =
[SYS_write]    sys_write,
[SYS_mknod]    sys_mknod,
[SYS_unlink]   sys_unlink,
[SYS_link]     sys_link,
[SYS_mkdir]    sys_mkdir,
[SYS_close]    sys_close,
//here
[SYS_getyear] sys_getyear,
[SYS_msgsend] sys_msgsend,
[SYS_msgrecv] sys_msgrecv,
[SYS_sigstop]  sys_sigstop,
[SYS_sigcont]  sys_sigcont,


[SYS_mutex_lock] sys_mutex_lock,
[SYS_mutex_unlock] sys_mutex_unlo


[SYS_clone]   sys_clone,
[SYS_join]    sys_join

};

void
syscall(void)
```



File Edit Selection View Go Run ···
test.c
syscall.h
syscall.c
syscall.h > SYS_mutex_lock
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
#define SYS_getyear 22
#define SYS_msgsend 23
#define SYS_msgrecv 24
// syscall.h
#define SYS_sigstop 22
#define SYS_sigcont 23
#define SYS_mutex_lock 25
#define SYS_mutex_unlock 26
#define SYS_clone 27
#define SYS_join 28

proc.c

```c
int
clone(void (*function)(void*), void *arg, void *stack)
{
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

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

    // Share address space
    np->pgdir = curproc->pgdir;
    np->sz = curproc->sz;

    // Copy trap frame
    *np->tf = *curproc->tf;

    // Set up new user stack for thread
    uint sp = (uint)stack + PGSIZE;
    sp -= sizeof(void*);
    *(uint*)sp = (uint)arg;
    sp -= sizeof(void*);
    *(uint*)sp = 0xFFFFFFFF; // Fake return PC
    np->tf->esp = sp;
    np->tf->eip = (uint)function;

    // Clear %eax so that clone returns 0 in the child.
    np->tf->eax = 0;

    // Copy open file descriptors
    for (i = 0; i < NOFILE; i++)
        if (curproc->ofile[i])
            np->ofile[i] = filedup(curproc->ofile[i]);
    np->cwd = idup(curproc->cwd);

    np->parent = curproc;
    safestrcpy(np->name, curproc->name, sizeof(curproc->name));

    pid = np->pid;

    acquire(&ptable.lock);
    np->state = RUNNABLE;
    release(&ptable.lock);

    return pid;
}

int
join(int tid, void **stack)
{
    struct proc *p;
    int havekids, pid;
    struct proc *curproc = myproc();

    acquire(&ptable.lock);
    for (;;)
    {
        havekids = 0;
        for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        {
            if (p->parent != curproc || p->pid != tid)
                continue;
            havekids = 1;
            if (p->state == ZOMBIE)
            {
                // Found one.
                pid = p->pid;
                kfree(p->kstack);
                *stack = p->tstack;
                p->kstack = 0;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->state = UNUSED;
                release(&ptable.lock);
                return pid;
            }
        }

        if (!havekids || curproc->killed)
        {
            release(&ptable.lock);
            return -1;
        }

        sleep(curproc, &ptable.lock);
    }
}
```



System Call 3: IPC (Inter-process Communication)

Functionality:
The IPC system call allows processes to communicate with each other by sending and receiving messages. This is achieved through a shared memory buffer and semaphores to coordinate access.
Code Modifications:

- **Header File:** Defined functions in ipc.h.
- **Implementation:** Implemented message passing and synchronization in ipc.c.
  Code Example:

c

```
int

sys_msgsend(void)

{

  int pid;

  char *msg;

  if (argint(0, &pid) < 0 || argstr(1, &msg) < 0)

    return -1;

  return sendmessage(pid, msg);

}


int sys_msgrecv(void) {
```

```
    int pid;
    char *buf;

    // Get the first argument (pid)
    if (argint(0, &pid) < 0) {
        cprintf("msgrecv: failed to get pid argument\n");
        return -1;
    }

    // Get the second argument (buffer)
    if (argstr(1, &buf) < 0) {
        cprintf("msgrecv: failed to get buffer argument\n");
        return -1;
    }

    // Call receivemessage with both pid and buf
    return receivemessage(pid, buf);
}

int sendmessage(int pid, char *msg) {
    struct proc *p = findproc(pid);
    if (p == 0) {
        cprintf("sendmessage: No process found with pid %d\n", pid);
        return -1;
    }

    acquire(&ptable.lock);
    safestrcpy(p->msgbuf, msg, MAX_MSG_SIZE);
    p->msg_available = 1;
    release(&ptable.lock);
    return 0;
```

```
}


int receivemessage(int pid, char *buf) {
   struct proc *p = findproc(pid);
   if (p == 0) {
      cprintf("receivemessage: No process found with pid %d\n", pid);
      return -1;
   }

   acquire(&ptable.lock);
   if(p->msg_available == 0) {
   release(&ptable.lock);
         return -1;
      }
   safestrcpy(buf, p->msgbuf, MAX_MSG_SIZE);
   p->msg_available = 0;
   release(&ptable.lock);
   return 0;
}
```

EXCUETION:

```
$ test_ipc "hello"
Parent trying to send message Child process started with pid 6
to child pid 6
Message sent to child
Child trying to receive message...
Child received message: "hello"
$ |
```

System Call 4: Signals

Functionality:
The signals system call allows processes to send asynchronous signals to each other, such as for handling events or exceptions.
Code Modifications:

- **Header File:** Defined signal handling functions in signals.h.
- **Implementation:** Implemented signal registration and delivery mechanisms in signals.c.
  Code Example:

```c
// System call to stop a process
int sys_sigstop(void) {
   int pid;
   if (argint(0, &pid) < 0)
      return -1;

   struct proc *p = find_proc(pid);
   if (!p)
      return -1;

   acquire(&ptable.lock);
   p->stopped = 1;  // Mark process as stopped
   release(&ptable.lock);
   return 0;
}

// System call to continue a stopped process
int sys_sigcont(void) {
   int pid;
   if (argint(0, &pid) < 0)
      return -1;

   struct proc *p = find_proc(pid);
   if (!p)
      return -1;

   acquire(&ptable.lock);
```

```
    p->stopped = 0;  // Clear stopped flag
    release(&ptable.lock);
    return 0;
}
```

4. User Program Demonstration

To test the new system calls, a user program was written that demonstrates their usage.

Program Name: test_syscalls.c
Description: The program creates threads, utilizes mutexes for synchronization, sends IPC messages, and handles signals between processes.
Code Snippet:

c

```
int kill(int pid, int sig) {
    struct proc *p;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->pid == pid) {
            if (sig == SIGSTOP) {
                p->stopped = 1;  // Set stopped flag
            } else if (sig == SIGCONT) {
                p->stopped = 0;  // Clear stopped flag
            } else if (sig == SIGKILL) {
                p->killed = 1;
            }
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;  // No process found with that PID
}]
```
EXCUETION:

```
$ signal_test
Parent process: sending SIGSTOP to child
Child process running...
Child process running...
SIGSTOP sent, child should stop
Parent process: sending SIGCONT to child
SIGCONT sent, child should continue
Child process running...
Child process running...
Child process running...
Child process running...
zombie!
```

---

5. Challenges and Solutions

- **Challenge 1:** Ensuring thread synchronization with mutexes.
  - **Solution:** Used a busy-wait loop to implement the lock mechanism in the mutex_lock function.
- **Challenge 2:** Implementing IPC with synchronization to avoid race conditions.
  - **Solution:** Used semaphores to manage access to shared memory in the IPC system.
- **Challenge 3:** Handling signals asynchronously.
  - **Solution:** Implemented a signal queue for each process and ensured signals were delivered when processed.

---

6. Conclusion

The project successfully added four new system calls to the xv6 operating system: mutex, threads, IPC, and signals. These system calls were tested through a user program, demonstrating their intended functionality for synchronization, communication, and concurrent execution.