2022083436 주예지

운영체제 Project 2 Wiki

A Simple Kernel-Level Thread

I. Design:

1.) system call clone(), join()

- The clone() system call must create a new thread sharing the parent's address space and resources but with an independent execution context and stack. The modifications ensure threads share one page table (true threading), correctly set up register state, and avoid lock races or lost wakeups.
- The join() system call must wait for any thread of the calling process to terminate, retrieve its user stack pointer, clean up the kernel TCB, and return the thread's PID. Modifications prevent lost wakeups and ensure safe user-space pointer copy.

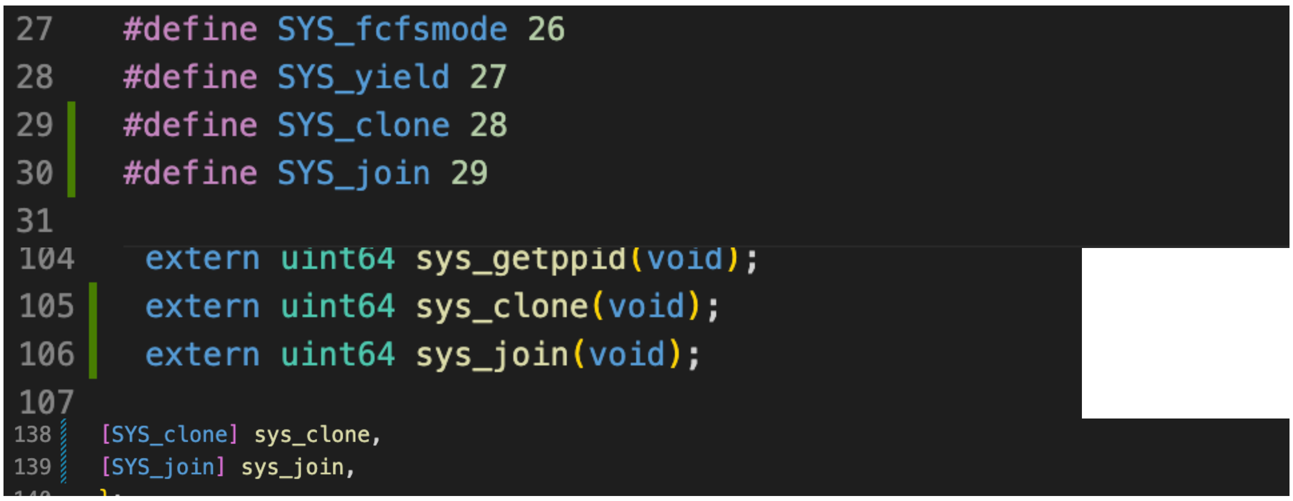
2.) thread_create() and thread_join()

- The thread_create() must allocate a user stack and invoke the clone() system call.
- I've allocated a page size user stack by calling malloc(), and called clone() directly.
- The thread_join() must block until a thread exists, retrieve and free its user stack, and return the thread's pid.
- I've added a join() call and made it return -1 if pid<0.

II. Implementation

1.) clone()

In syscall.c, I linked the SYS_clone system call to sys_clone().



```
#define SYS_fcfsmode 26
#define SYS_yield 27
#define SYS_clone 28
#define SYS_join 29

```

```
extern uint64 sys_getppid(void);
extern uint64 sys_clone(void);
extern uint64 sys_join(void);

```

```
[SYS_clone] sys_clone,
[SYS_join] sys_join,
```

sysproc.c

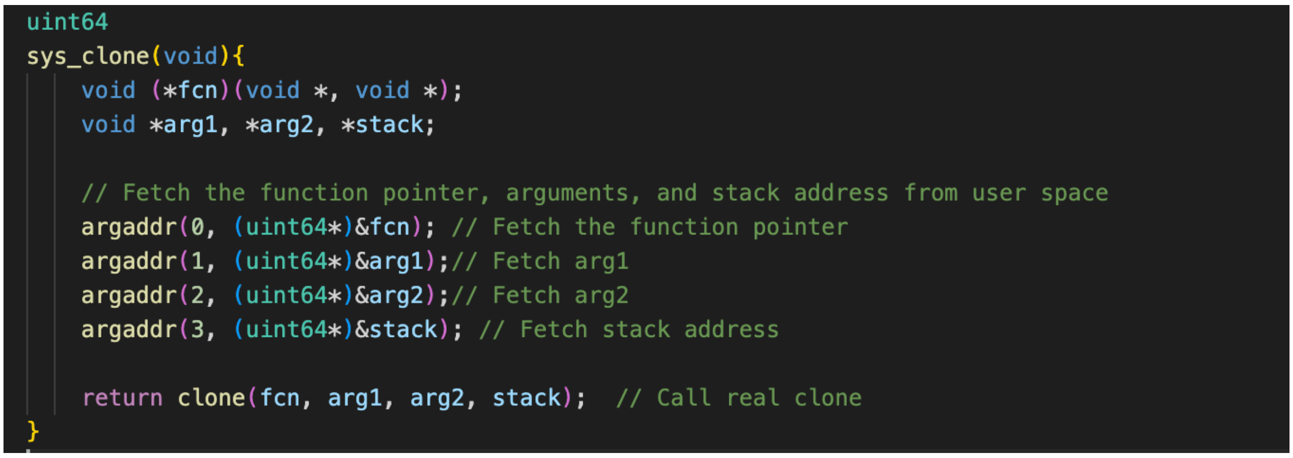
Fetches the function pointer, arguments, and stack pointer from the user program and calls the clone() function.

The argaddr() function is used to fetch the arguments passed from user space. It reads the address of each argument and stores them in local variables.

fcn: is the function pointer where new thread starts executing

arg1 and arg2: are the arguments to be passed to fcn

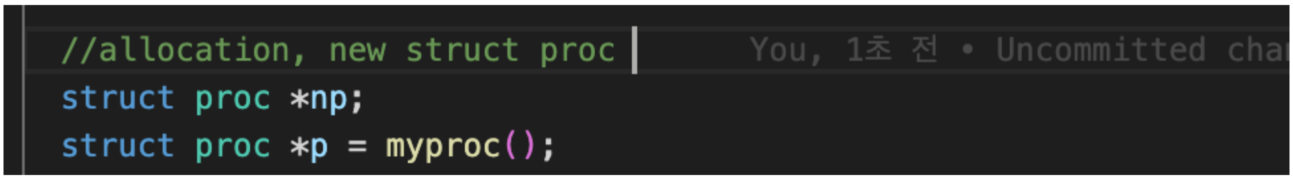
stack: is the address of the stack allocated for the new thread, which must be page-aligned and large enough for the new thread's stack



```c
uint64
sys_clone(void){
    void (*fcn)(void *, void *);
    void *arg1, *arg2, *stack;

    // Fetch the function pointer, arguments, and stack address from user space
    argaddr(0, (uint64*)&fcn); // Fetch the function pointer
    argaddr(1, (uint64*)&arg1);// Fetch arg1
    argaddr(2, (uint64*)&arg2);// Fetch arg2
    argaddr(3, (uint64*)&stack); // Fetch stack address

    return clone(fcn, arg1, arg2, stack);  // Call real clone
}
```

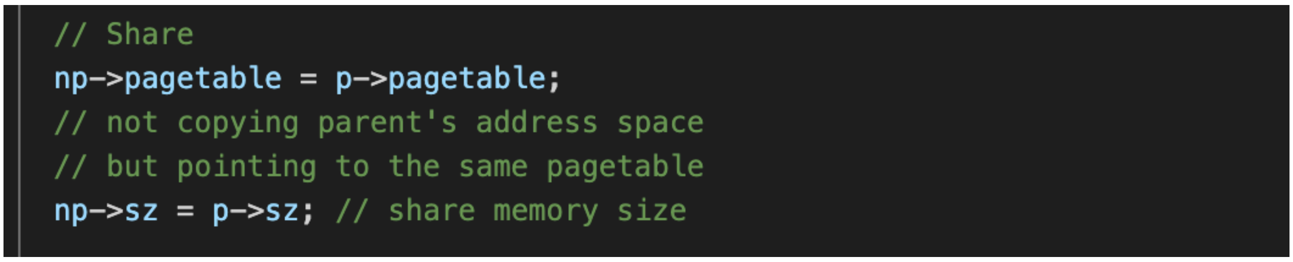
Below is the implementation of int clone() function in proc.c.

This is to allocate a new struct proc.



```c
//allocation, new struct proc        You, 1초 전 • Uncommitted cha
struct proc *np;
struct proc *p = myproc();
```

Threads share the same virtual memory. So instead of copying parent's address space, I am going to point to the same pagetable.



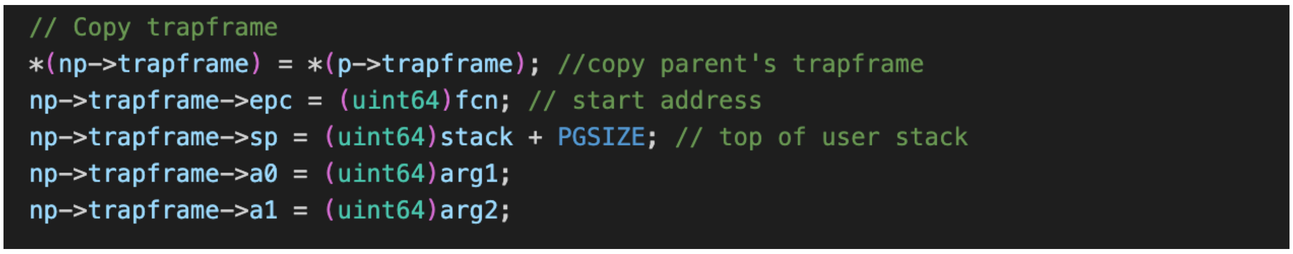
```c
// Share
np->pagetable = p->pagetable;
// not copying parent's address space
// but pointing to the same pagetable
np->sz = p->sz; // share memory size
```

epc: exception program counter. The thread will be start running.

sp: set to the top of the user stack

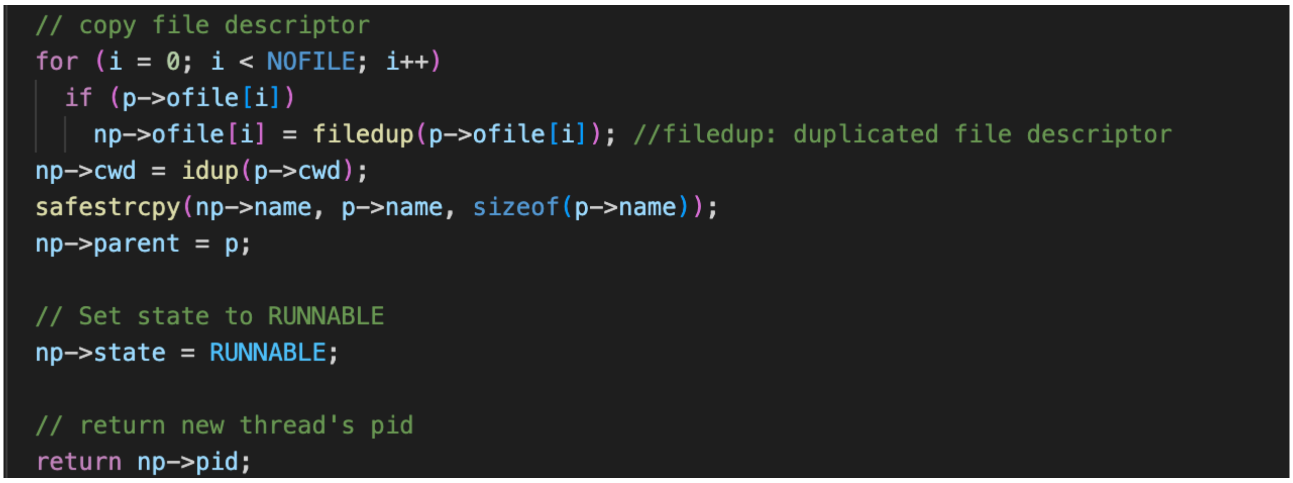
a0 and a1 are the first two arguments passed by registers.

This sets up the thread to run fcn(arg1, arg2) with its own stack.



```
// Copy trapframe
*(np->trapframe) = *(p->trapframe); //copy parent's trapframe
np->trapframe->epc = (uint64)fcn; // start address
np->trapframe->sp = (uint64)stack + PGSIZE; // top of user stack
np->trapframe->a0 = (uint64)arg1;
np->trapframe->a1 = (uint64)arg2;
```

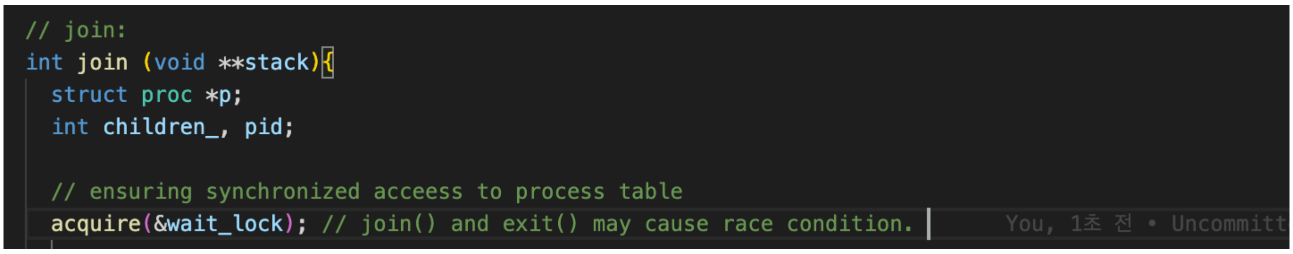
Then copy the file descriptors, and the thread's parent is set to the creating thread.
And then I set the state to RUNNABLE, and return the calling thread, the new thread's PID.



```
// copy file descriptor
for (i = 0; i < NOFILE; i++)
  if (p->ofile[i])
    np->ofile[i] = filedup(p->ofile[i]); //filedup: duplicated file descriptor
np->cwd = idup(p->cwd);
safestrcpy(np->name, p->name, sizeof(p->name));
np->parent = p;

// Set state to RUNNABLE
np->state = RUNNABLE;

// return new thread's pid
return np->pid;
```

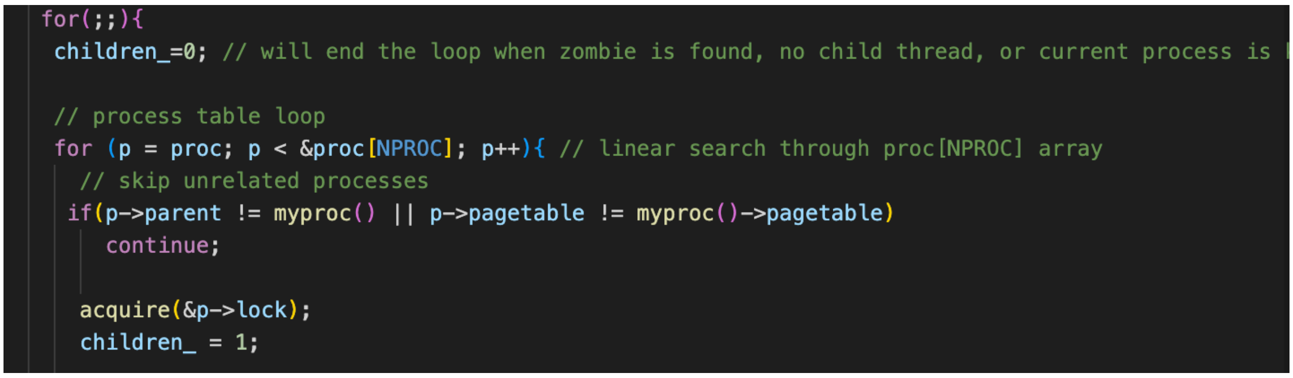
2.) join()

Starting the join() function with acquiring lock. This lock ensures that only one process at a time will traverse proc[]. Because there would be a race confition between join() and exit() so it is necessary to use lock.



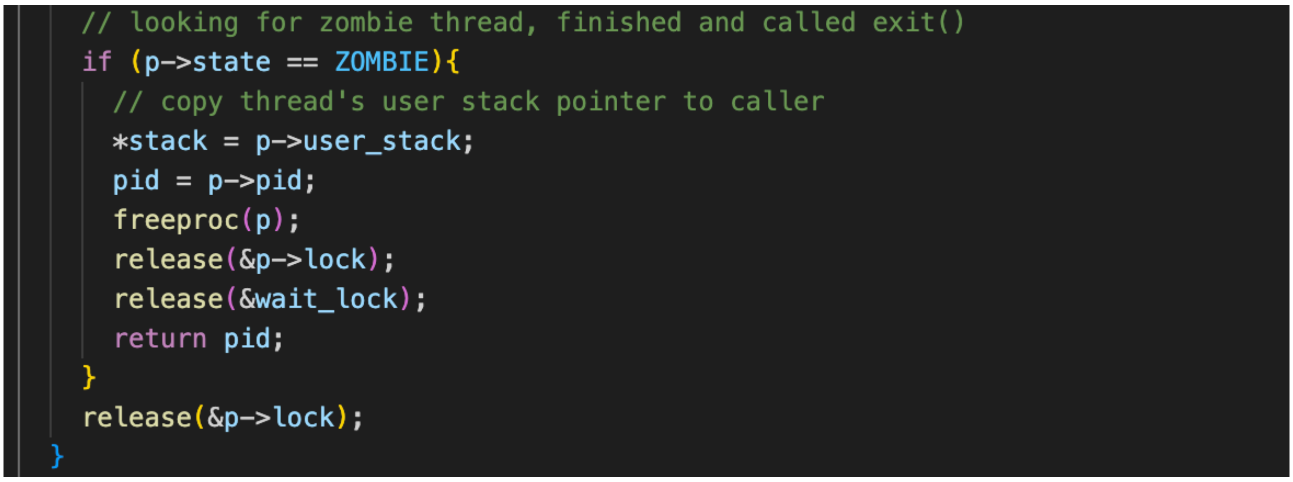
```
// join:
int join (void **stack){
  struct proc *p;
  int children_, pid;

  // ensuring synchronized acceess to process table
  acquire(&wait_lock); // join() and exit() may cause race condition.     You, 1초 전 • Uncommitt
```

The join function uses a for loop that ends when a zombie thread is found, or there is no child thread, or the process is killed. And inside the loop, I create another loop that searches through proc[NPROC] array and filter the necessary processes.
children_ variable indicates number of child threads.



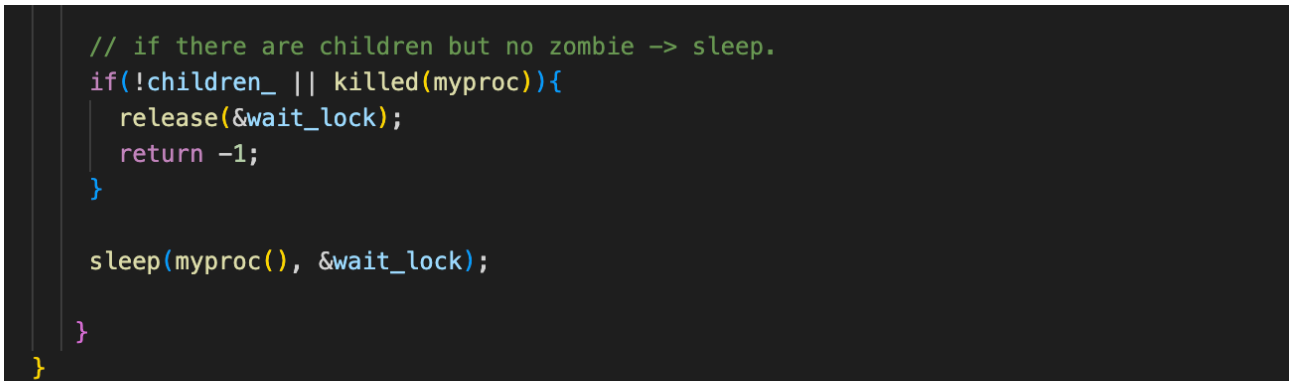
```
for(;;){
 children_=0; // will end the loop when zombie is found, no child thread, or current process is

 // process table loop
 for (p = proc; p < &proc[NPROC]; p++){ // linear search through proc[NPROC] array
   // skip unrelated processes
  if(p->parent != myproc() || p->pagetable != myproc()->pagetable)
    continue;

   acquire(&p->lock);
   children_ = 1;
```

If we find a zombie thread, and if then copy the value of p->user_stack into the user-space pointer that was passed to join() (so that in thread_join() we can call free(stack)), then release the lock and return failure signal. Otherwise, I free the trapframe and allocated struct proc, and return pid.



```
   // looking for zombie thread, finished and called exit()
   if (p->state == ZOMBIE){
     // copy thread's user stack pointer to caller
     *stack = p->user_stack;
     pid = p->pid;
     freeproc(p);
     release(&p->lock);
     release(&wait_lock);
     return pid;
   }
   release(&p->lock);
 }
```

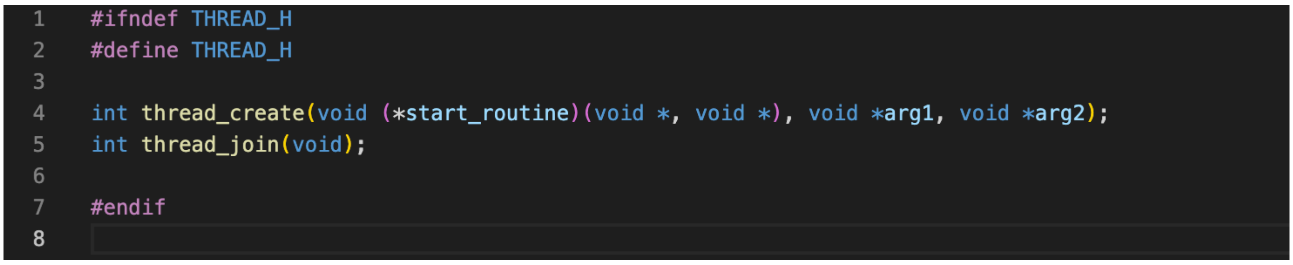
And this is to handle the case if there’s no zombie thread. I set it to sleep until a child changes state to zombie.



```
   // if there are children but no zombie -> sleep.
   if(!children_ || killed(myproc)){
     release(&wait_lock);
     return -1;
   }

   sleep(myproc(), &wait_lock);

  }
}
```

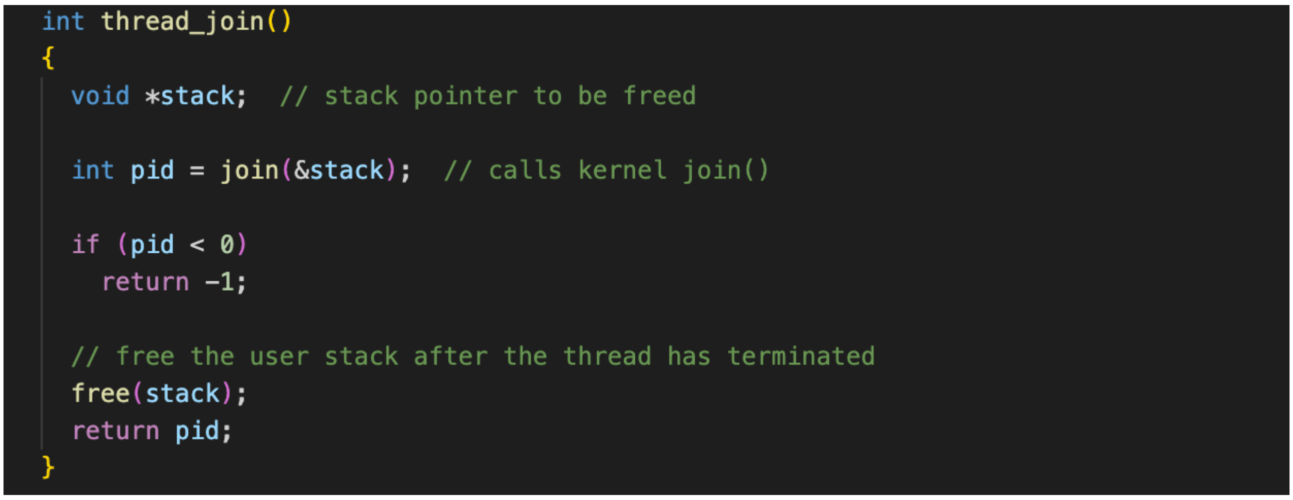
3.) thread_create() and thread_join()

I’ve added a new header and c file of thread in user space.



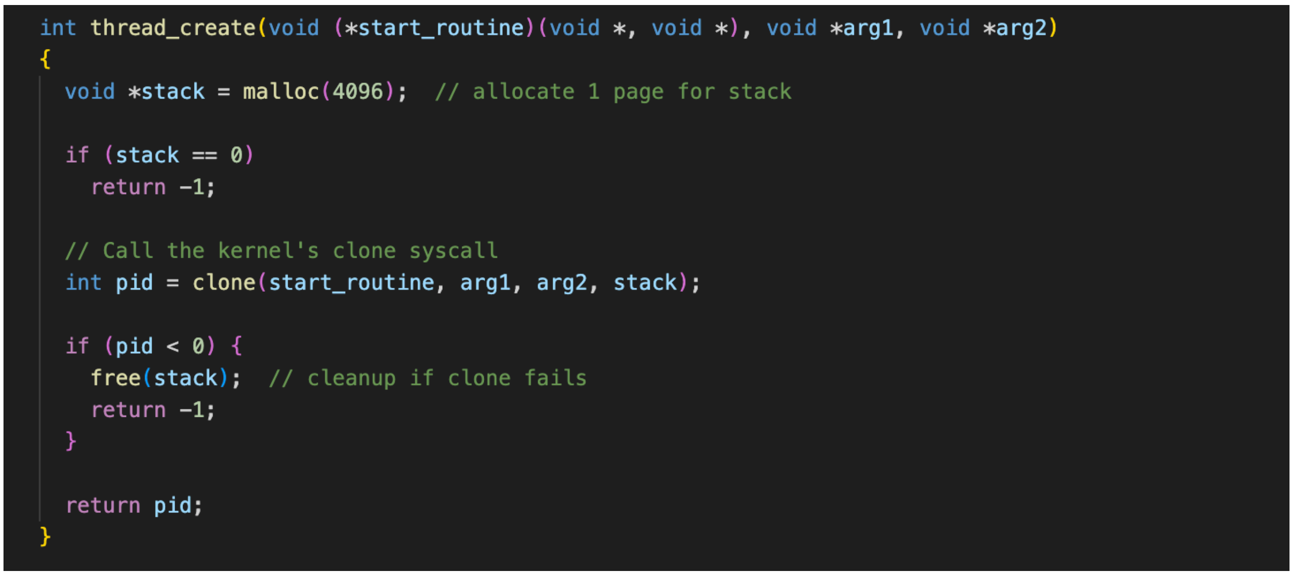
```c
#ifndef THREAD_H
#define THREAD_H

int thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2);
int thread_join(void);

#endif
```

It calls the kernel join() system call, and then frees the memory for the stack that has received from join(). And then it returns the pid of the joined thread, or -1 if error case.



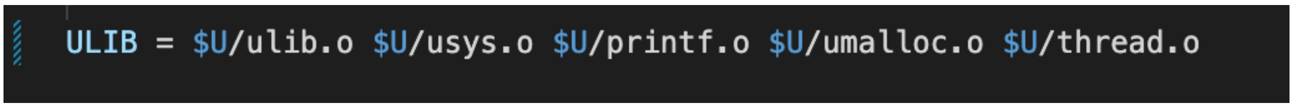
```c
int thread_join()
{
  void *stack;  // stack pointer to be freed

  int pid = join(&stack);  // calls kernel join()

  if (pid < 0)
    return -1;

  // free the user stack after the thread has terminated
  free(stack);
  return pid;
}
```

Because the caller doesn't supply a stack so I've allocated a stack here. Stack is page-aligned because malloc() in xv6 returns page-aligned blocks by default. So if clone() fails then return -1, if it successes then returns the new thread's pid.



```c
int thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2)
{
  void *stack = malloc(4096);  // allocate 1 page for stack

  if (stack == 0)
    return -1;

  // Call the kernel's clone syscall
  int pid = clone(start_routine, arg1, arg2, stack);

  if (pid < 0) {
    free(stack);  // cleanup if clone fails
    return -1;
  }

  return pid;
}
```

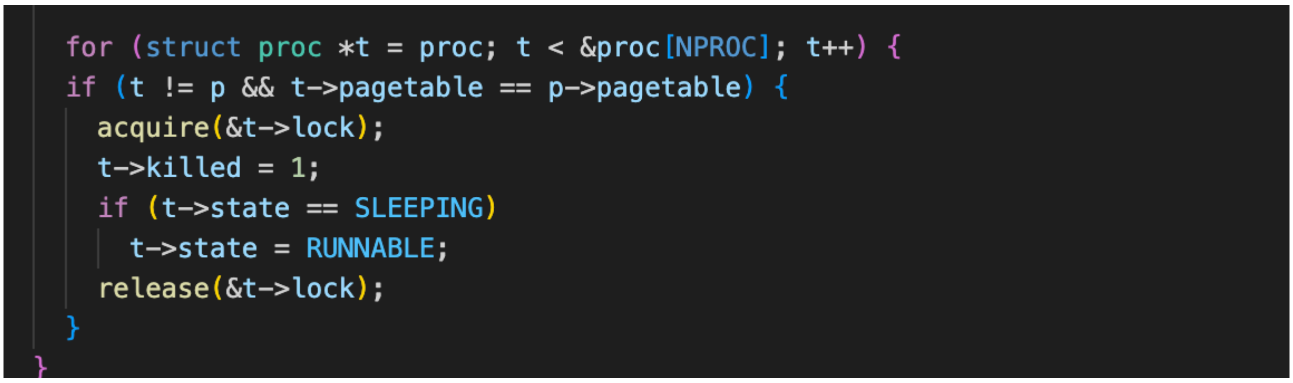
Modifications in Makefile:



```makefile
ULIB = $U/ulib.o $U/usys.o $U/printf.o $U/umalloc.o $U/thread.o
```

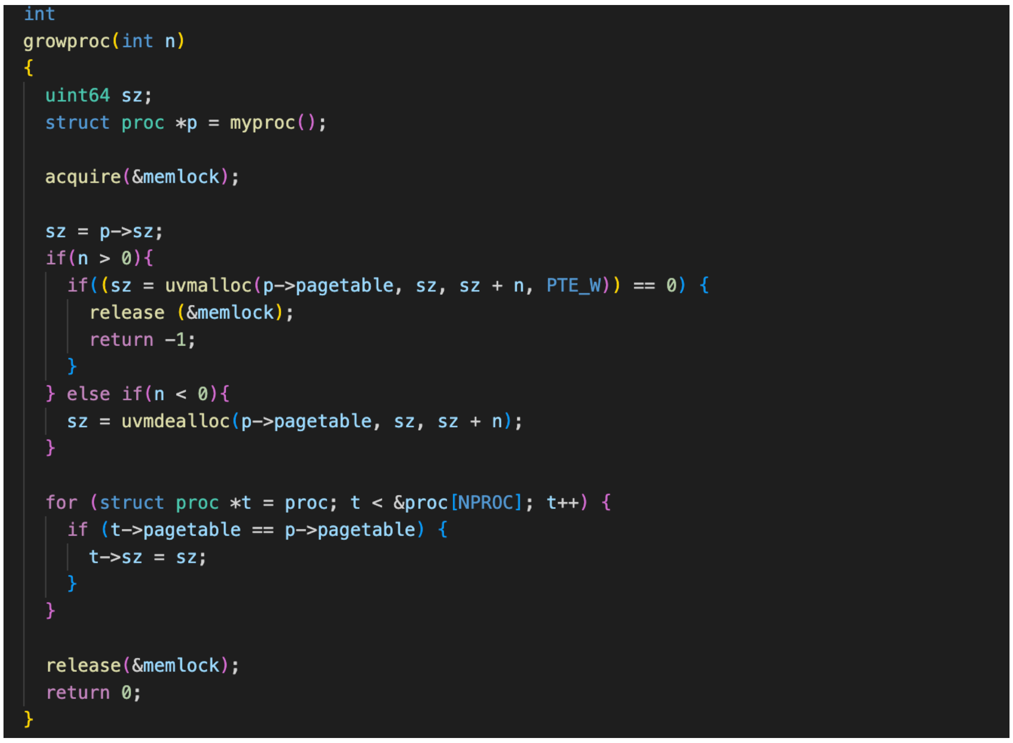
4.) System Call Modifications

exec() replaces the current process's address space, but if this process is a thread thenthose sibling threads still exist and will now be running on a memory that's either freed proc_freepagetable() or replaced with a new layout. So the solution: before replacing the address space and assigninb the new pagetable, find and kill all sibling threads except the calling thread. It ensures only the calling thread survives and enters the new program. Also, sibling threads get marked killed = -1 and exit as soon as scheduled.



```
  for (struct proc *t = proc; t < &proc[NPROC]; t++) {
  if (t != p && t->pagetable == p->pagetable) {
    acquire(&t->lock);
    t->killed = 1;
    if (t->state == SLEEPING)
      t->state = RUNNABLE;
    release(&t->lock);
  }
}
```

The problem of sbrk() is because threads created by clone() share the same pagetable and heap size sz. myproc()->sz, the default usage might cause race condition. Thus, I modify growproc() to use global lock memlock for synchronization, and after resizing memory, update heap size sz for all threads sharing the same pagetable.



```
int
growproc(int n)
{
  uint64 sz;
  struct proc *p = myproc();

  acquire(&memlock);

  sz = p->sz;
  if(n > 0){
    if((sz = uvmalloc(p->pagetable, sz, sz + n, PTE_W)) == 0) {
      release (&memlock);
      return -1;
    }
  } else if(n < 0){
    sz = uvmdealloc(p->pagetable, sz, sz + n);
  }

  for (struct proc *t = proc; t < &proc[NPROC]; t++) {
    if (t->pagetable == p->pagetable) {
      t->sz = sz;
    }
  }

  release(&memlock);
  return 0;
}
```

I modify kill() in proc.c so that it kills all threads sharing the same pagetable.

```c
int
kill(int pid)
{
  int found = 0;

  for (struct proc *p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->pid == pid) {
      // Kill all threads sharing the same address space
      pagetable_t target_pagetable = p->pagetable;
      release(&p->lock);

      for (struct proc *q = proc; q < &proc[NPROC]; q++) {
        acquire(&q->lock);
        if (q->pagetable == target_pagetable) {
          q->killed = 1;
          if (q->state == SLEEPING)
            q->state = RUNNABLE;
        }
        release(&q->lock);
      }

      return 0;
    }
    release(&p->lock);
  }
  return -1;
}
```

III. Result & Troubleshooting

```
xv6 kernel is booting

init: starting sh
$ thread_test

[TEST#1]
Thread 0 start
Thread 1 start
Thread 1 end
Thread 0 join failed
$ Thread 2 start
Thread 2 end
Thread 3 start
Thread 3 end
Thread 4 start
Thread 4 end
Thread 0 end
```

I've failed from the first test, which I think is the problem occuring in system calls clone() or join(). Either of them (or probably both of them) are returning -1 before the thread becomes a zombie. So my thread0 join is keep failing.

Right now I also am confused by the boundary. (the thread to block and not to block.) Thus it issued me a contiguous, non-stop of cloning.

Also the problems I met were locking problems. For a very long time, I've met "panic: acquire/release" issues. This is because I was confused by the timing of acquiring the lock, releasing the lock and the number of locks that I've used. One of the problems was that I didn't know allocproc() already contained acquiring the lock so I had to remove the line and manually acquire locks in my system calls join() and clone().