Concurrent Threads Report

Camille Williford - awillif4

December 4, 2021

The point of this lab was to implement treading functionality in the xv6 operating system. I implemented two system calls and 5 functions as part of the thread library for this lab. I am going to take a chronological order to this report detailing changes that I made to the xv6 OS in the order that I made them. Here is a link to the change log in GitHub: Threads Commit

1 System Calls

1.1 syscall.c

Modified: 4 Lines
Defined two new system calls $sys\_clone()$ and $sys\_join()$ and their respective system functions.

1.2 syscall.h

Modified: 2 Lines
Assigned numbers to the new system calls $sys\_clone()$ and $sys\_join()$. These numbers are used when a user initiates a system call to safely determine which system call the user requested be performed.

1.3 user.h

Modified: 2 Lines
I prototyped the two new system calls as callable functions. These are the user functions that can be used to to call into the operating system.

$$int\ clone(void\ (*start\_routine)(void*, void*),\ void*,\ void*,\ void*);$$

$$int\ join(void**);$$

1.4 usys.S

Modified: 2 Lines
Assembly code that loads register with syscall number, makes syscall, and makes transition to the kernel mode using int instruction. I added the two new system calls to this list.

1.5 sysproc.c

Modified: 18 Lines
I created the two new system calls $sys\_clone()$ and $sys\_join()$! These are a pass-through to the implementation in proc.c. Each parses the user input with $argint()$, and returns -1 if unable to parse arguments.

1.6 proc.c

Modified: 112 Lines
Here the *clone*() and *join*() system calls are implemented. *clone*() works by getting the current running process and allocating a new process copying the current process's data including page directory, and parent process. Then the new thread establishes it's own stack with the given user arguments, which is saved to the process's threadstack. Any open file descriptors are also passed the new thread. The new thread is then set as RUNNABLE, and will begin executing at the address the user has provided. *join*() simply searches for any zombie children. When zombie children are found they will be removed from the kernel stack, and all process information will be reset for the next process to utilize in the ptable.

1.7 proc.h

Modified: 1 Lines
Added an new address to the thread stack for each process, which will be allocated and freed as threads are created by that process.

2 Threads Library

2.1 user.h

Modified: 9 Lines
I added a new lock struct which contains an *int ticket* and an *int turn* to create a ticket lock implementation for our threads. The user functions *thread_create*(), *thread_join*(), *lock_init*(), *lock_acquire*(), and *lock_release*() are defined here. These are the functions required to implement a thread library.

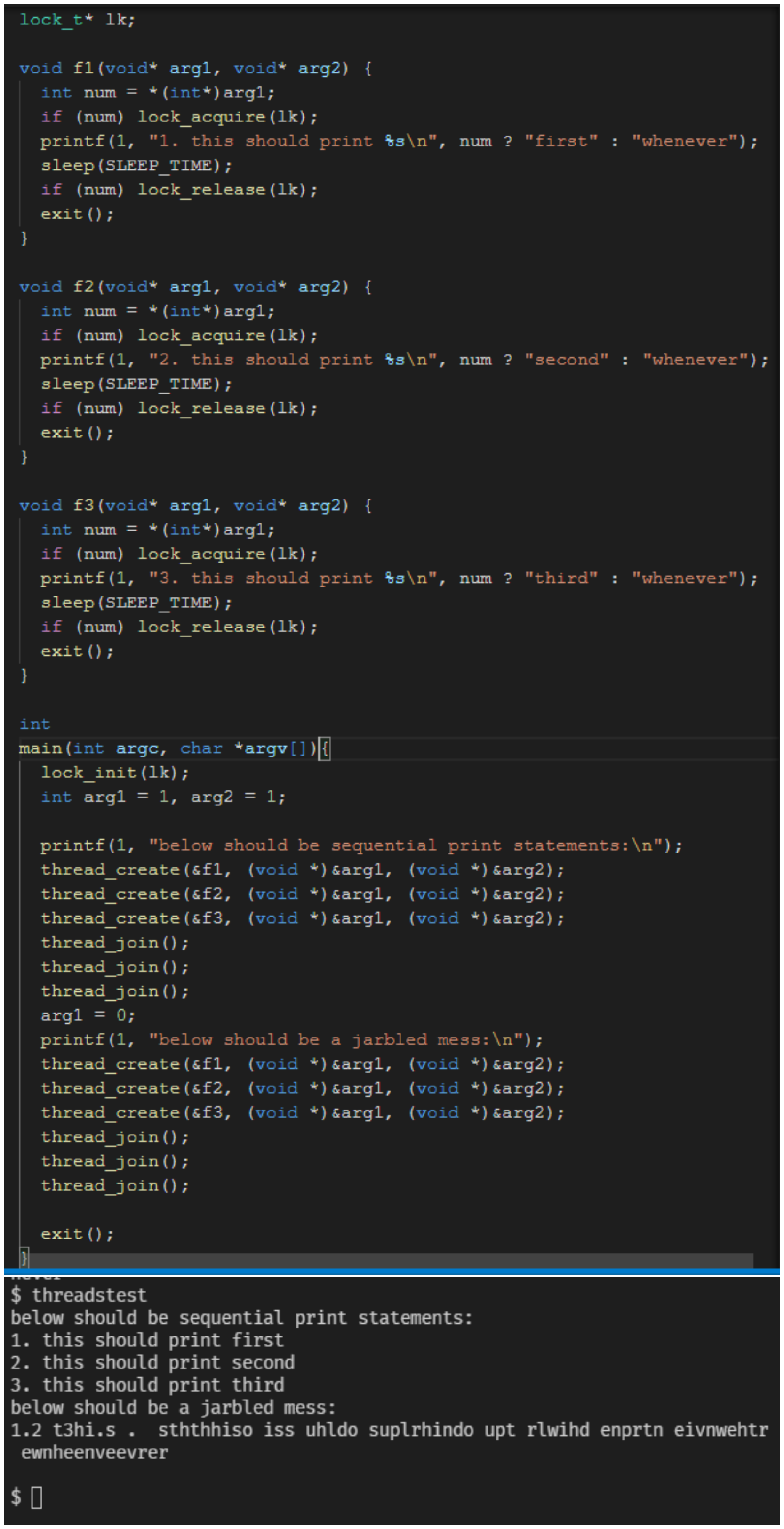
2.2 ulib.c

Modified: 40 Lines
Here the functions for the thread library are implemented as well as an atomic *fetch_and_add*(*int* ∗ *var*, *int val*) routine which returns the old value at a particular address. *thread_create*() allocates space for the stack of the new thread and then uses the *clone*() system call. *thread_join*() utilizes the system call *join*(). *lock_init*(*lock_t* ∗ *lock*) simply sets the given lock's tickets and turn to 0. *lock_acquire*(*lock_t* ∗*lock*) performs the *fetch_and_add*() routine, and if it returns that it is our lock's turn we weill acquire the lock. Otherwise if it is not our lock's turn we will spin lock to indefinitely wait until it is our turn. *lock_release*(*lock_t* ∗*lock*) will simply increment the given lock's turn variable to indicate it must move onto the next process.

3 Tests

3.1 threadstest.c

New File
This test simply initializes a thread lock, and creates 3 threads that run in the order that they are created, and waits for each thread to finish running and calls *thread_join*() to clean up after the threads. Then we create 3 more threads, but with an invalid user argument this time. Now when the threads run there is no way to determine which order they ran in or for how long. Then we clean up after these 3 treads before exiting.



```c
lock_t* lk;

void f1(void* arg1, void* arg2) {
  int num = *(int*)arg1;
  if (num) lock_acquire(lk);
  printf(1, "1. this should print %s\n", num ? "first" : "whenever");
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}

void f2(void* arg1, void* arg2) {
  int num = *(int*)arg1;
  if (num) lock_acquire(lk);
  printf(1, "2. this should print %s\n", num ? "second" : "whenever");
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}

void f3(void* arg1, void* arg2) {
  int num = *(int*)arg1;
  if (num) lock_acquire(lk);
  printf(1, "3. this should print %s\n", num ? "third" : "whenever");
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}

int
main(int argc, char *argv[]){
  lock_init(lk);
  int arg1 = 1, arg2 = 1;

  printf(1, "below should be sequential print statements:\n");
  thread_create(&f1, (void *)&arg1, (void *)&arg2);
  thread_create(&f2, (void *)&arg1, (void *)&arg2);
  thread_create(&f3, (void *)&arg1, (void *)&arg2);
  thread_join();
  thread_join();
  thread_join();
  arg1 = 0;
  printf(1, "below should be a jarbled mess:\n");
  thread_create(&f1, (void *)&arg1, (void *)&arg2);
  thread_create(&f2, (void *)&arg1, (void *)&arg2);
  thread_create(&f3, (void *)&arg1, (void *)&arg2);
  thread_join();
  thread_join();
  thread_join();

  exit();
}
```

```
$ threadstest
below should be sequential print statements:
1. this should print first
2. this should print second
3. this should print third
below should be a jarbled mess:
1.2 t3hi.s .  sththhiso iss uhldo suplrhindo upt rlwihd enprtn eivnwehtr
 ewnheenveevrer

$ 
```