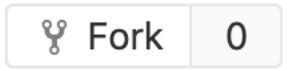
CS 444/544 OS II
Lab Session #8

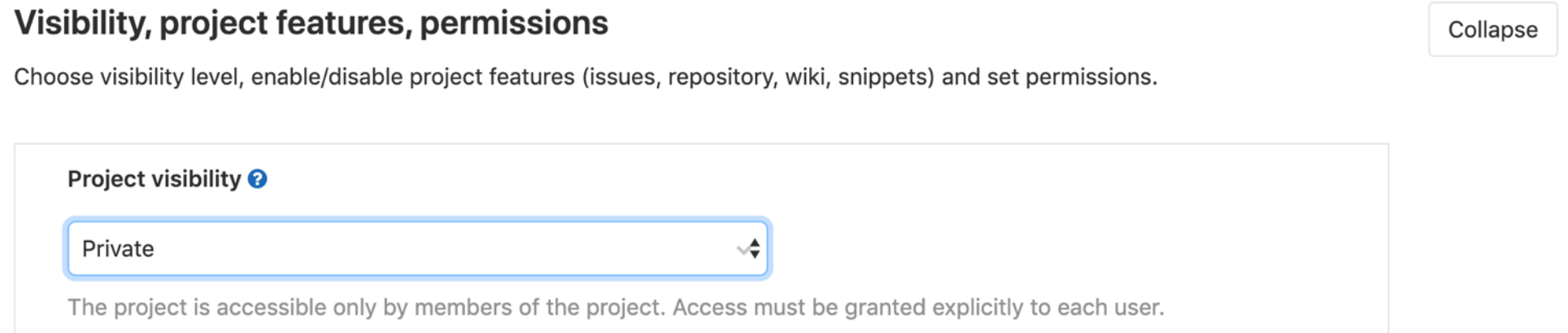
Concurrency Lab #2

Fork Concurrency Lab 2

- Go to
  - https://gitlab.unexploitable.systems/root/concurrency2

- Click fork 

- Set visibility: Private



Clone your lab

- On flip1,2,3 or os1 or 2
  - git clone git@gitlab.unexploitable.systems:[your-gitlab-id]/concurrency2.git
- This will generate a directory ‘concurrency2’
- Copy your c1.c to concurrency2 directory

Some commands

- make
  - Will build all test programs
- make grade
  - Will run all tests with grading script
- make clean
  - Remove all files. You can build all by running ‘make’ again

```
./grade-c2
Running deadlock_q1
deadlock_q1: OK (15/100)
Running deadlock_q2
deadlock_q2: OK (30/100)
Running deadlock_q3
deadlock_q3: OK (45/100)
Running deadlock_q4
deadlock_q4: OK (60/100)
Running array
array:
OK Test 1 (70/100)
OK Test 2 (80/100)
OK Test 3 (90/100)
OK Test 4 (100/100)
```

Check

```
[jangye@os2 (master) ~/concurrency2$] ls -l
total 20
-rw-r--r--. 1 jangye upg3275 6098 May 28 08:00 array.c
-rwxr-xr-x. 1 jangye upg3275  241 May 28 08:00 check_libc1.py
-rw-r--r--. 1 jangye upg3275 3314 May 28 08:00 deadlock_q1.c
-rw-r--r--. 1 jangye upg3275 3454 May 28 08:00 deadlock_q2.c
-rw-r--r--. 1 jangye upg3275 4244 May 28 08:00 deadlock_q3.c
-rw-r--r--. 1 jangye upg3275 3024 May 28 08:00 deadlock_q4.c
-rwxr-xr-x. 1 jangye upg3275 3634 May 28 08:00 grade-c2
-rw-r--r--. 1 jangye upg3275 1248 May 28 08:00 lock.h
-rw-r--r--. 1 jangye upg3275  846 May 28 08:00 Makefile
-rw-r--r--. 1 jangye upg3275  366 May 28 08:00 README.md
-rw-r--r--. 1 jangye upg3275  420 May 28 08:00 thread.c
-rw-r--r--. 1 jangye upg3275  279 May 28 08:00 thread.h
```

Check 2

Copy your c1.c to the repository

```
[jangye@os2 (master) ~/concurrency2$] make
c1.c does not exist! Please copy your c1.c to this directory!
make: *** No rule to make target `c1.c', needed by `libc1.so'.  Stop.
[jangye@os2 (master) ~/concurrency2$] cp ../concurrency1/c1.c .
[jangye@os2 (master) ~/concurrency2$] make
g++ -o libc1.so c1.c -shared -fPIC -lpthread -g
g++ -o thread.o thread.c -lpthread -g -c
g++ -o deadlock_q1 deadlock_q1.c *.o -g -lpthread -lc1 -L. -Wl,-rpath,.
g++ -o deadlock_q2 deadlock_q2.c *.o -g -lpthread -lc1 -L. -Wl,-rpath,.
g++ -o deadlock_q3 deadlock_q3.c *.o -g -lpthread -lc1 -L. -Wl,-rpath,.
g++ -o deadlock_q4 deadlock_q4.c *.o -g -lpthread -lc1 -L. -Wl,-rpath,.
g++ -o array array.c *.o -g -lpthread -lc1 -L. -Wl,-rpath,.
```

Check 3

```
total 56
-rwxr-xr-x. 1 jangye upg3275 25472 May 28 08:02 array
-rw-r--r--. 1 jangye upg3275  6098 May 28 08:00 array.c
-rw-r--r--. 1 jangye upg3275 12889 May 28 08:01 c1.c
-rwxr-xr-x. 1 jangye upg3275   241 May 28 08:00 check_libc1.py
-rwxr-xr-x. 1 jangye upg3275 24352 May 28 08:02 deadlock_q1
-rw-r--r--. 1 jangye upg3275  3314 May 28 08:00 deadlock_q1.c
-rwxr-xr-x. 1 jangye upg3275 24352 May 28 08:02 deadlock_q2
-rw-r--r--. 1 jangye upg3275  3454 May 28 08:00 deadlock_q2.c
-rwxr-xr-x. 1 jangye upg3275 24520 May 28 08:02 deadlock_q3
-rw-r--r--. 1 jangye upg3275  4244 May 28 08:00 deadlock_q3.c
-rwxr-xr-x. 1 jangye upg3275 24032 May 28 08:02 deadlock_q4
-rw-r--r--. 1 jangye upg3275  3024 May 28 08:00 deadlock_q4.c
-rwxr-xr-x. 1 jangye upg3275  3634 May 28 08:00 grade-c2
-rwxr-xr-x. 1 jangye upg3275 11896 May 28 08:02 libc1.so
-rw-r--r--. 1 jangye upg3275  1248 May 28 08:00 lock.h
-rw-r--r--. 1 jangye upg3275   846 May 28 08:00 Makefile
-rw-r--r--. 1 jangye upg3275   366 May 28 08:00 README.md
-rw-r--r--. 1 jangye upg3275   420 May 28 08:00 thread.c
-rw-r--r--. 1 jangye upg3275   279 May 28 08:00 thread.h
-rw-r--r--. 1 jangye upg3275  4440 May 28 08:01 thread.o
```

deadlock_q1

- Please resolve ‘deadlock’ in thread_func_0 and 1

```
        ########################################
        ## Welcome  to  Deadlock  Quiz  1  ##
        ########################################
This program spawns two threads, and both threads
will try acquire two locks competitively.
However, a novice programmer (possibly Prof. Jang)
wrote a poor code, so the program encounters 'deadlock'.
Can you resolve the deadlock in the program by
changing the 'order' of acquiring locks?
DEADLOCK, no points!
```

deadlock_q1

- Please resolve ‘deadlock’ in thread_func_0 and 1

```c
void
thread_func_0(struct thread_args *args) {
    /* Do not change the following line */
    usleep(10000);

    for (int i=0; i<N_TIMES; ++i) {
        /* Do not change the following line */
        usleep(10);

        /* QUIZ 1: Change from here */
        xchg_lock(&l0);
        printf("Thread 0 acquired l0\n");
        xchg_lock(&l1);
        printf("Thread 0 acquired l1\n");
        /* to here, if required */

        /* Do not change the following line */
        counter += args->args[0];

        /* Change from here */
        xchg_unlock(&l1);
        printf("Thread 0 released l1\n");
        xchg_unlock(&l0);
        printf("Thread 0 released l0\n");
        /* to here, if required.. */
    }
}
```

```c
void
thread_func_1(struct thread_args *args) {
    /* Do not change the following line */
    usleep(10000);

    for (int i=0; i<N_TIMES; ++i) {
        /* Do not change the following line */
        usleep(10);

        /* QUIZ 1: Change from here */
        xchg_lock(&l1);
        printf("Thread 1 acquired l1\n");
        xchg_lock(&l0);
        printf("Thread 1 acquired l0\n");
        /* to here, if required */

        /* Do not change the following line */
        counter += args->args[0];

        /* Change from here */
        xchg_unlock(&l0);
        printf("Thread 1 released l0\n");
        xchg_unlock(&l1);
        printf("Thread 1 released l1\n");
        /* to here, if required */
    }
}
```

deadlock_q2

- Please resolve ‘deadlock’ in thread_func_0 and 1

```
        ## Welcome  to  Deadlock  Quiz  2  ##
        ########################################
This program spawns two threads, and both threads
will try acquire two locks competitively.
However, a novice programmer (possibly Prof. Jang)
wrote a poor code, so the program encounters 'deadlock'.
Can you resolve the deadlock in the program by
creating a meta lock to make acquiring all locks
run atomically?
DEADLOCK, no points!
```

deadlock_q3

- Please resolve ‘deadlock’ in thread_func_0 and 1

```
        ########################################
        ## Welcome  to  Deadlock  Quiz  3  ##
        ########################################
This program spawns two threads, and both threads
will try acquire two locks competitively.
However, a novice programmer (possibly Prof. Jang)
wrote a poor code, so the program encounters 'deadlock'.
Can you resolve the deadlock in the program by
removing no-preemption?
In this case, you cannot change the order of lock,
i.e., Thread 0 acquires l0 and then l1,
Thread 1 acquires l1 and then l0,
and you should implement trylock/unlock,
which you can see at the page 42 of the slide,
https://os.unexploitable.systems/l/W7L2.pdf
```

How to Remove No Preemption

Release the lock if obtaining a resource fails…

```
top:
    lock(A);
    if (trylock(B) == -1) {
        unlock(A);
        goto top;
    }
    …
```

Can’t acquire B, then Release A!

deadlock_q3

- Please use _xchg() as trylock()
  - if (xchg() == 1), you failed to acquire the lock
  - if (xchg() == 0), you have acquired the lock

- Algorithm
  - Acquire l0
  - Try Acquire l1
    - If acquired, great!
    - If not acquired, release l0, and go back to the first step

deadlock_q4

- DO NOT use lock at this time..
- Please use _cmpxchg (lock cmpxchg, an atomic operation) to increment counter values…
- Thread 0 increments the counter by 1
- Thread 1 increments the counter by 2

How to Remove Mutual Exclusion

- Do not use lock
  - Use atomic operations instead

- Replace locks with atomic primitives
  - compare_and_swap(uint64_t *addr, uint64_t prev, uint64_t value);
  - if *addr == prev, then update *addr = value;
  - lock cmpxchg in x86..

```
void add (int *val, int amt) {
        Mutex_lock(&m);
        *val += amt;
        Mutex_unlock(&m);
}
```

```
void add (int *val, int amt) {
        do {
                int old = *value;
        } while(!CompAndSwap(val, old, old+amt);
}
```

A simple thread-safe array (array.c)

- Two or more threads will use an array (or more)

- Their use must be synchronized to get a consistent result

Test1

```
void
thread_func_0(struct thread_args *args) {
    Array *a = (Array *) args->args[0];
    for (int i=0; i<300000; ++i) {
        a->insert(i);
    }
}

Array *a = new Array();

args0.args[0] = (uint64_t) a;

thread_array[0] = fork_a_thread((void*) thread_func_0, &args0);
thread_array[1] = fork_a_thread((void*) thread_func_0, &args0);
```

- Two threads, each inserts 300,000 items to the array a.
- Expected result: 600,000 items

```
Test1 Size: 334413 (must be 600,000)
```

Test2

```
void
thread_func_0(struct thread_args *args) {
    Array *a = (Array *) args->args[0];
    for (int i=0; i<300000; ++i) {
        a->insert(i);
    }
}

void
thread_func_1(struct thread_args *args) {
    usleep(1000);
    Array *a = (Array *) args->args[0];
    for (int i=0; i<300000; ++i) {
        a->remove_last();
    }
}
```

```
Array *a = new Array();

args0.args[0] = (uint64_t) a;

thread_array[0] = fork_a_thread((void*) thread_func_0, &args0);
thread_array[1] = fork_a_thread((void*) thread_func_1, &args0);
```

- One thread adds 300,000 items
- The other thread removes 300,000 itmes

- Expected Result: 0 items…

```
Test2 Size: 49654 (must be 0)
```

Test3

```
void
thread_func_1(struct thread_args *args) {
    usleep(1000);
    Array *a = (Array *) args->args[0];
    for (int i=0; i<300000; ++i) {
        a->remove_last();
    }
}

void
thread_func_2(struct thread_args *args) {
    Array *full = (Array *) args->args[0];
    Array *half = (Array *) args->args[1];
    full->cut_in_half(half);
}
```

- One thread returns an array that contains the first half of the element in the array
- The other thread removes 300,000 itmes
- Expected Result: getting first 150,000 items

```
array: array.c:213: void test3(): Assertion `b->size == 150000' failed.
```

Test4

- One thread returns a concatenated array of two arrays..
- One other thread removes 300,000 items from a
- One other thread removes 300,000 items from b
- The other threads inserts 300,000 items to c
- Expected Result: a = 0, b = 0, c = 900,000

```
void
thread_func_1(struct thread_args *args) {
    usleep(1000);
    Array *a = (Array *) args->args[0];
    for (int i=0; i<300000; ++i) {
        a->remove_last();
    }
}

void
thread_func_3(struct thread_args *args) {
    usleep(100);
    Array *a = (Array *) args->args[0];
    for (int i=0; i<300000; ++i) {
        a->insert(i+1000000);
    }
}

void
thread_func_4(struct thread_args *args) {
    Array *a = (Array *) args->args[0];
    Array *b = (Array *) args->args[1];
    Array *concat = (Array *) args->args[2];
    a->concatenation(b, concat);
}
```

Array.c

- Change the implementation of struct Array
- Place locks if required
- Implement nolock functions if required
  - E.g., nolock_insert