Optional-Assignmet 1

Design Document

Manavjeet Singh, 2018295

Paste your struct thread structure

```
struct thread {
        void *esp;
        void *stack_base;
        int clear;
        struct thread *next;
        struct thread *prev;
};
```

Paste any new global variables or struct that you have added to the existing code

```
int waiting_threads=0;
struct thread *to_clean;
```

Paste your code corresponding to sleep.

```
void sleep(struct lock *lock)
{
   // printf("Sleep called\n");
   struct thread *current_wait_list=(struct thread*)lock->wait_list;
        struct thread *temp=current_wait_list;
        if(current_wait_list==NULL){
                current_wait_list=cur_thread;
                current_wait_list->next=NULL;
                current_wait_list->prev=NULL;
                // printf("Added lock %p\n",cur_thread);
                lock->wait_list=(void*)current_wait_list;
                return;
        }
        while (temp->next!=NULL){
                temp=temp->next;
        }
        temp->next=cur_thread;
        cur_thread->prev=temp;
        cur_thread->next=NULL;
        // printf("Added lock %p\n",cur_thread);
        lock->wait_list=(void*)current_wait_list;
        waiting_threads+=1;
        schedule();
}
```

Paste your code corresponding to wakeup.

```
void wakeup(struct lock *lock)
{
   // printf("Wakeup called\n");
  struct thread *current_wait_list=(struct thread*)lock->wait_list;
   struct thread *toRet=current_wait_list;
   if(current_wait_list!=NULL)
        {
                current_wait_list=current_wait_list->next;
                if(current_wait_list!=NULL)
                        current_wait_list->prev=NULL;
                lock->wait_list=current_wait_list;
                push_back(toRet);
           waiting_threads-=1;
        }
}
```

Paste your code corresponding to the foo routine in race1.c.

```
void foo(void *ptr)
{
        struct lock *l = (struct lock*)ptr;
        int val;

        acquire((struct lock*) ptr);
        val = counter;
        // printf("aquire called\n");
        val++;
        thread_yield();
        counter = val;
        // printf("release called\n");
        release((struct lock*) ptr);
        thread_exit();

}
```

Dump the output of “make test2”

```
/usr/bin/time -v ./leak 1024000 2>&1 |egrep "kbytes|counter"
main thread exiting : counter:1024000
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 5360
        Average resident set size (kbytes): 0
```

Does running race2 cause deadlock in your submission?

Yes this is beacuse foo is calling thread_exit() without giving up the lock. This cause the deadlock since bar is waiting for the lock .

Does your strategy for eliminating memory leak is different from what you suggested in the assignment-2 design documentation. If yes, please highlight the changes.

Yes my strategy is different. In the assignment 2 I was freeing the memory in thread_exit() routine but that implemtation was dependent on the memory allocator. Since context_change is called in thread_exit(), that implemetation can lead to segmentation fault.

In my current implemetation, ‘to be freed momory’ is saved in a global variable.→context switch is called→ then the memory is freed.