Assignment 2

OPERATING SYSTEMS

Introduction

In the given assignment we are to implement a non-preemptive threading library NPTlib for application threads. The OS view of an application is a sequential program with one private stack and registers. NPTlib allows an application thread to create non-preemptive threads without changing the OS view of the application. We have file thread.c that implements fAPIs for the above task.

1) Paste your code corresponding to push back.

1. My page metadata structure is :

```
static void push_back(struct thread *t)
{
        t->next = NULL;
        t->prev = NULL;
        struct thread * temp = ready_list;
        if(temp!=NULL){
                while(temp->next!=NULL){
                        temp=temp->next;
                }
                temp->next=t;
                t->prev=temp;
        }
        else{
                ready_list=t;
```

```
                ready_list->next = NULL;
            }
        }
```

---

2) Paste your code corresponding to pop front.

```
static struct thread *pop_front()

{
        struct thread * temp=ready_list;
        ready_list=ready_list->next;
        return temp;
}
```

3) Paste your code corresponding to create thread

```
void create_thread(func_t func, void *param)
{
unsigned *stack = malloc(4096);
stack += 1024;
struct thread *t = malloc(sizeof(struct thread));
stack-=1;
*(void * *)stack =param;
stack-=1;
*stack=0;
```

```
stack-=1;
*(func_t* )stack=func;
stack-=1;
*stack=0;
stack-=1;
*stack=0;
stack-=1;
*stack=0;
stack-=1;
*stack=0;

t->esp=stack;
push_back(t);

}
```

4) Dump the output of the ‘‘make test’’.

```
root@3cd80ad33d87:/app/NPTlib-master# make test
gcc -ggdb -Werror -m32 -O3 -c thread.c
gcc -ggdb -O3 -m32 thread.o context.o -o app app.c
./app 1024
starting main thread: num_threads: 1024
main thread exiting: counter:30768239300147200
./app 1024 1
starting main thread: num_threads: 1024
bar1: (nil)
bar2: (nil)
bar1: 0x1
main thread exiting: counter:0
root@3cd80ad33d87:/app/NPTlib-master#
```

5) A strategy to free struct thread and the stack corresponding to the thread that has exited (using thread exit API).

If we want to free the thread and as well as its stack , so , first of all we need the very first thing required to free the stack, that is a pointer that points towards the address in the stack where the allocation of that particular begins from in the starting itself. So, for that we can do is while we are allocating a thread we store a pointer of that allocation to a attribute of our structure thread.So everytime we allocate a new thread we allocate a pointer to a pointer defined in our struct thread like we are *esp in our present code. So, now we can have a function that can be called by thread_exit so that the thread can be scheduled and we can free the thread and its stack.

Efforts by:
Himanshi Mathur
2018037