CSE 312 /CSE 504 Spring 2022

Operating System

1901042262

HW1

1. Thread Structure
    1. Thread Class

```
class Thread{
    friend class ThreadManager;
    private:
        common::uint8_t stack[4096];
        CPUState * cpustate;
        int threadMode;
        Thread *joinThread;

    public:
        CPUState* getCPUState(){ return cpustate;}
        void setMode(int mode);
        void setjoinThread(Thread *joinThread);
        Thread* getjoinThread();
        int getMode();
        Thread();
        ~Thread();
};
```

Each thread keeps its own CPUState and stack. It also keeps 2 private variables as threadMode, id and joinIndex.

- threadMode:

```
#define TERMINATEMODE -1
#define NORMALMODE 0
#define YIELDMODE 1
#define JOINMODE 2
```

As seen above, there are 4 modes for threads. When the thread is created, the threadMode is assigned as NORMALMODE. threadMode is used when schudule threads in ThreadManager's Schudule function.

- joinThread:

```
void pthread_join(Thread *thread, Thread * other){
    if(thread->getMode()!=TERMINATEMODE){
        thread->setMode(JOINMODE);
        thread->setJoinIndex(other->getID());
    }
}
```

joinThread keeps the other thread. joinThread is used when schedule threads in ThreadManager's Schedule function.

- Thread Constructor:

```
Thread::Thread()
{
    threadMode=NORMALMODE;

    cpustate = (CPUState*)(stack + 4096 - sizeof(CPUState));

    cpustate -> eax = 0;
    cpustate -> ebx = 0;
    cpustate -> ecx = 0;
    cpustate -> edx = 0;

    cpustate -> esi = 0;
    cpustate -> edi = 0;
    cpustate -> ebp = 0;
    cpustate -> eflags = 0x202;

}
```

2. ThreadManager Class

```
class ThreadManager{
private:
    Thread * threads[256];
    int numThreads;
    int currThread;
    void setCurrentThread(CPUState *cpustate);
public:
    ThreadManager();
    ~ThreadManager();
    bool addThread(Thread* task);
    CPUState* Schedule(CPUState* cpustate);
};
```

ThreadManager class is used to control threads. It has functions of adding threads and scheduling threads.

- addThread:

```
bool ThreadManager::addThread(Thread* thread)
{
    if(numThreads >= 256)
        return false;
    threads[numThreads++] = thread;
    return true;
}
```

Adds a new thread to the thread.

- Schedule:

```
CPUState* ThreadManager::Schedule(CPUState* cpustate)
{
    int numOfTerminated=0;
    if(numThreads <= 0)
        return cpustate;
     takeMod(&currThread,numThreads);

    /*
    This loop will loop until it finds a thread that has been normal or joined and the other thread has terminated.
    */
    while(true){
        //If the thread is in yield mode, it switches to normal mode and moves to the next thread.
        if(threads[currThread]->getMode()==YIELDMODE){
            threads[currThread]->setMode(NORMALMODE);
            takeMod(&currThread,numThreads);
        }

        //If the thread is in join mode and the other thread is not terminated, it moves to the next thread.
        if(threads[currThread]->getMode()==JOINMODE && threads[currThread]->getjoinThread()->getMode() != TERMINATEMODE)
            takeMod(&currThread,numThreads);


        //If the thread is in terminate mode, it moves to the next thread.
        if(threads[currThread]->getMode()==TERMINATEMODE){
            takeMod(&currThread,numThreads);
            numOfTerminated++;
        }

        //If thread is normal or joined and the other thread has terminated
        if(threads[currThread]->getMode()==NORMALMODE || threads[currThread]->getMode()==JOINMODE)
            break;

        //if All threads are terminated
        if(numOfTerminated==numThreads){
            numThreads=0;
            return cpustate;
        }
    }
    return threads[currThread]->cpustate;
}
```

```
void takeMod(int *num1,int num2){
    if(++(*num1) >= num2)
        *num1 %= num2;
}
```

As seen above, this function checks the threads according to the modes and returns the CPUState of the appropriate thread. This function is called within the Schedule function of the TaskManager.

2. Task Structure

    1. Task Class

```
class Task
{
friend class TaskManager;
private:
    ThreadManager threadManager;
public:
    Task();
    Task(Thread * thread);
    ~Task();
    bool addThread(Thread * thread);
};
```

Tasks start with a single thread present. It has function of adding threads.

- Task(Thread * thread) Constructor:

```
Task::Task(Thread * thread)
{
    this->threadManager.addThread(thread);
}
```

- addThread:

```
// adds new thread to the task
bool Task::addThread(Thread * thread){
    return this->threadManager.addThread(thread);
}
```

2. TaskManager Class

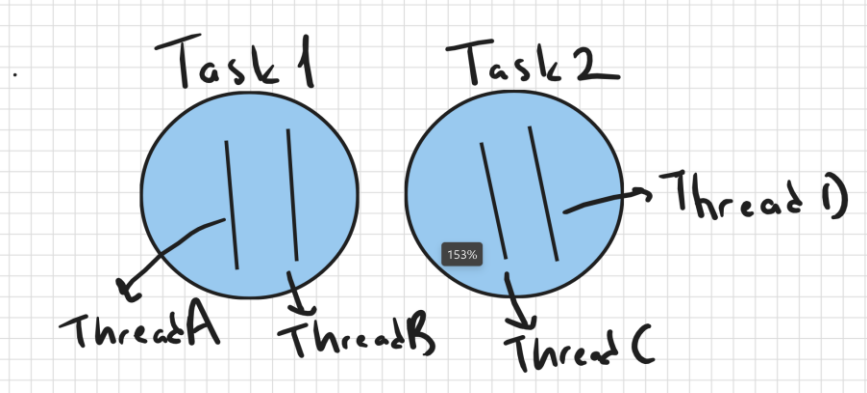
```
class TaskManager
{
private:
    Task* tasks[256];
    int numTasks;
    int currentTask;
public:
    TaskManager();
    ~TaskManager();
    bool AddTask(Task* task);
    CPUState* Schedule(CPUState* cpustate);
};
```

Only the Schedule function has changed in the TaskManager class.

- Schedule:

```
CPUState* TaskManager::Schedule(CPUState* cpustate)
{
    int numOfEnded=0;
    if(numTasks <= 0)
        return cpustate;

    if(currentTask >= 0)
        tasks[currentTask]->threadManager.setCurrentThread(cpustate);

    if(++currentTask >= numTasks)
        currentTask %= numTasks;
    while(true){
        if(numOfEnded==numTasks){
            numTasks=0;
            return cpustate;
        }
        if(tasks[currentTask]->threadManager.getNumThreads()==0){
            numOfEnded++;
            if(++currentTask >= numTasks)
                currentTask %= numTasks;
        }
        else
            break;

    }

    return tasks[currentTask]->threadManager.Schedule(cpustate);
}
```

- Scheduling Algorithm:
  In this algorithm, when the thread interrupts, the scheduler changes the process and runs the next thread in the process. When there is no task left to run, it return cpustate parameter.



Looking at the above example, runs respectively;
ThreadA,ThreadC,ThreadB,TreadD,ThreadA,ThreadC…

2. Producer-Consumer Fashion

- Petersons Algorithm

```
/*
    Petersons algorithm functions
*/
void enter_region(int process){         /* process is 0 or 1 */
    int other;                          /* number of the other process */
    other=1-process;                    /* the opposite of process */
    interested[process]=TRUE;           /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE);   /* null statement */
}
void leave_region(int process){         /* process: who is leaving */
    interested[process]=FALSE;          /* indicate departure from critical region */
}
```

- Producer and Consumer Functions

```
/*
    Producer Consumer Fashion.
    In these 2 thread functions, one function tries to increase the item value by 1,
    while the other function tries to decrease it.
    If 2 threads want to access the item value at the same time, critical action happens here.

*/
void producer(){
    while(true){
        enter_region(0);
        item++;                 // It tries to increase the global variable item by 1. !Critical action happen
        printf("Producer: ");
        printfHex(item);
        leave_region(0);
    }

}
void consumer(){
    while (true)
    {
        enter_region(1);
        item--;                 // It tries to decrease the global variable item by 1. !Critical action happen
        printf("Consumer: ");
        printfHex(item);
        leave_region(1);
    }

}
```

In these 2 thread functions, one function tries to increase the item value by 1, while the other function tries to decrease it. If 2 threads want to access the item value at the same time, critical action happens.

3. Functions for creating, terminating, yielding, and joining the threads.

```
void *pthread_create(Thread *thread, GlobalDescriptorTable *gdt, void entrypoint());
void pthread_terminate(Thread *restrict);
void pthread_yield(Thread *restrict);
void pthread_join(Thread *thread, Thread * other);
```

- Create Function:

```
void *pthread_create(Thread *thread, GlobalDescriptorTable *gdt, void entrypoint()){

    thread->getCPUState()->cs = gdt->CodeSegmentSelector();
    thread->getCPUState()->eip = (uint32_t)entrypoint;


}
```

This function creates a thread by assigning the entrypoint and gdt that it takes as parameters to the thread.

- Terminate Function:

```
void pthread_terminate(Thread *restrict){
    restrict->setMode(TERMINATEMODE);
}
```

Sets the thread's mode to TERMINATEMODE. So this terminated thread can never run anymore.

- Yield Function:

```
void pthread_yield(Thread *restrict){
    if(restrict->getMode()!=TERMINATEMODE)
        restrict->setMode(YIELDMODE);
}
```

If the thread is not terminated already, it makes the threading mode YIELDMODE.

- Join Function:

```
void pthread_join(Thread *thread, Thread * other){
    if(thread->getMode()!=TERMINATEMODE){
        thread->setMode(JOINMODE);
        thread->setjoinThread(other);
    }
}
```

If the thread is not terminated already, It sets the thread's mode to JOINMODE and assigns the other thread to the thread's joinThread. Since it is not known when the other thread will end, the thread runs when other thread terminates.

4. Tests

- Producer-Consumer Fashion Test:

```
/* 2 threads that communicate with each other in a producer consumer fashion. */
pthread_create(&t1,&gdt,producer);
pthread_create(&t2,&gdt,consumer);
Task task1 = Task(&t1);
task1.addThread(&t2);
taskManager.AddTask(&task1);
```

- Demonstrating the scheduling algorithm:

```
/* //4 thread execution demo in 2 processes with 2 threads
pthread_create(&t1,&gdt,threadA);
pthread_create(&t2,&gdt,threadB);
pthread_create(&t3,&gdt,threadC);
pthread_create(&t4,&gdt,threadD);
Task task1 = Task(&t1);
task1.addThread(&t2);
Task task2 = Task(&t3);
task2.addThread(&t4);
taskManager.AddTask(&task1);
taskManager.AddTask(&task2);*/
```

- Terminate Function Test:

```
/*  //terminate function demo.
pthread_create(&t1,&gdt,threadA);
pthread_create(&t2,&gdt,threadB);
Task task1 = Task(&t1);
Task task2 = Task(&t2);
taskManager.AddTask(&task1);
taskManager.AddTask(&task2);
pthread_terminate(&t1);

*/
```

- Join Function Test:

```
/*  // join function demo. when this block is executed, it will not print the value A to the screen.
    //If we terminate the t3 threat, the value A will be printed.
pthread_create(&t1,&gdt,threadA);
pthread_create(&t2,&gdt,threadB);
pthread_create(&t3,&gdt,threadC);
Task task1 = Task(&t1);
Task task2 = Task(&t2);
task1.addThread(&t3);
pthread_join(&t1,&t3);
taskManager.AddTask(&task1);
taskManager.AddTask(&task2);

//pthread_terminate(&t3);     //terminate t3

*/
```

- Yield Function Test:

```
/*  //yield function demo. At first it tries to run another thread, then it starts to run itself.
pthread_create(&t1,&gdt,threadA);
Task task1 = Task(&t1);
task1.addThread(&t2);
pthread_yield(&t1);
taskManager.AddTask(&task1);
*/
```

- Others:

```
/*   //if all threads in the task are terminated
pthread_create(&t1,&gdt,threadA);
pthread_create(&t2,&gdt,threadB);
pthread_create(&t3,&gdt,threadC);
Task task1 = Task(&t1);
Task task2 = Task(&t2);
task1.addThread(&t3);
pthread_terminate(&t2);
taskManager.AddTask(&task1);
taskManager.AddTask(&task2);
*/
/*   //If there is no task
pthread_create(&t1,&gdt,threadA);
pthread_create(&t2,&gdt,threadB);
Task task1 = Task(&t1);
Task task2 = Task(&t2);
taskManager.AddTask(&task1);
taskManager.AddTask(&task2);
pthread_terminate(&t1);
pthread_terminate(&t2);
*/
```