GEBZE TECHNICAL UNIVERSITY
COMPUTER ENGINEERING

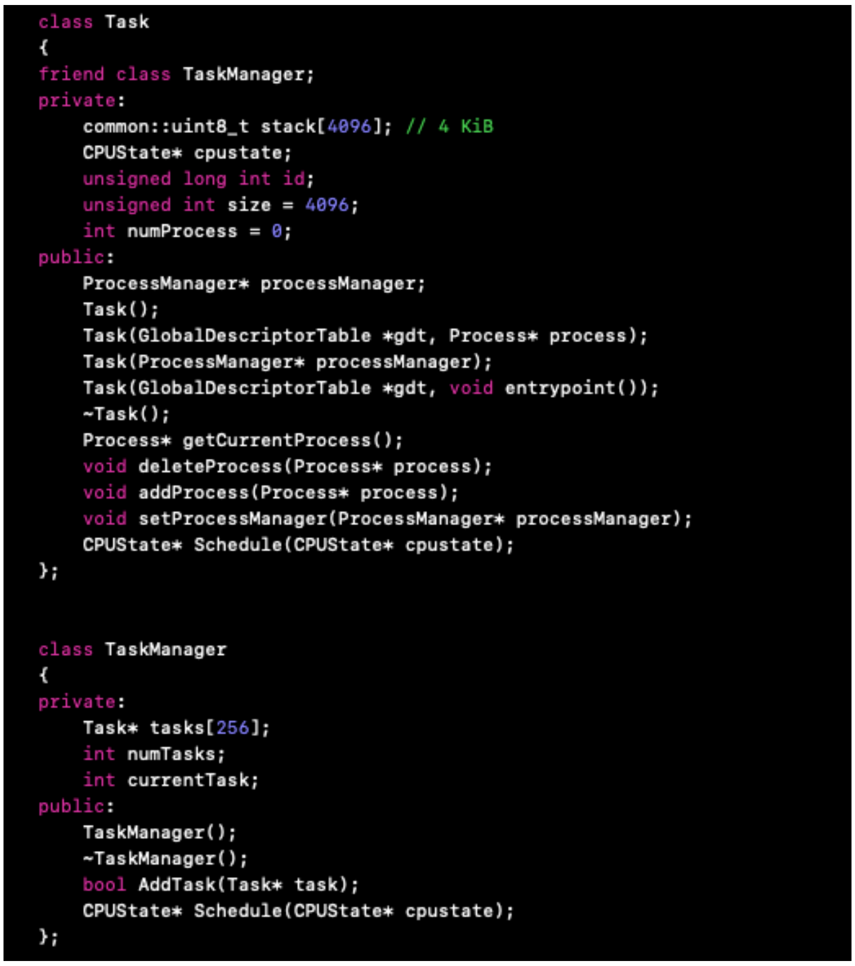
CSE 312 - 2023 SPRING
OPERATING SYSTEMS

HOMEWORK 1
REPORT

Dilara KARAKAŞ 171044010

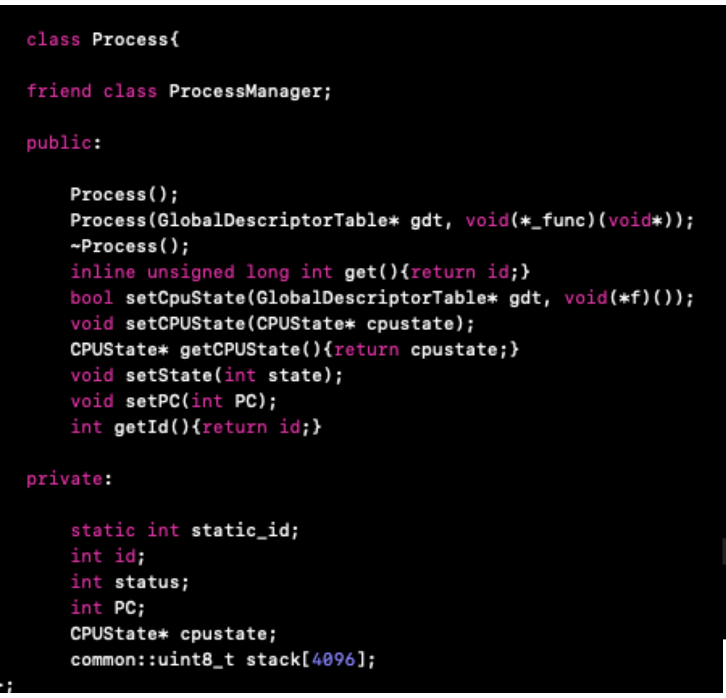
Structure

Task - Task Manager - Process - Process Manager

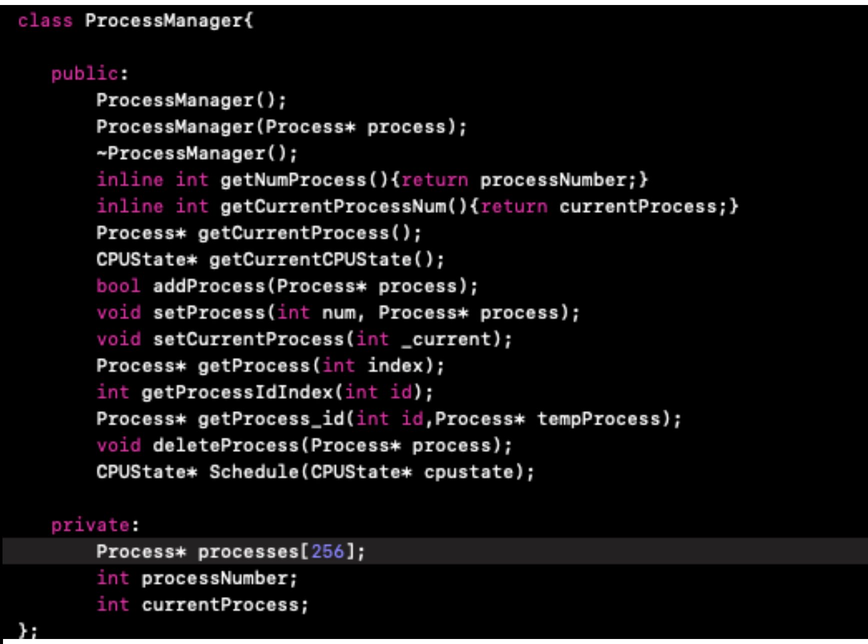


```
class Task
{
friend class TaskManager;
private:
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;
    unsigned long int id;
    unsigned int size = 4096;
    int numProcess = 0;
public:
    ProcessManager* processManager;
    Task();
    Task(GlobalDescriptorTable *gdt, Process* process);
    Task(ProcessManager* processManager);
    Task(GlobalDescriptorTable *gdt, void entrypoint());
    ~Task();
    Process* getCurrentProcess();
    void deleteProcess(Process* process);
    void addProcess(Process* process);
    void setProcessManager(ProcessManager* processManager);
    CPUState* Schedule(CPUState* cpustate);
};


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

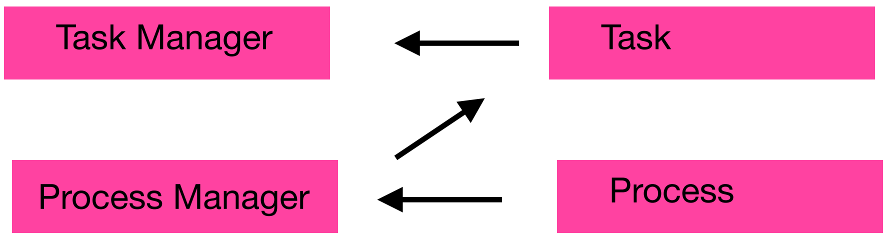
Task Manager has tasks to schedule and select the next task to run. The task has the process manager to get the next process from the process manager program. Process Manager has the process of programming them and choosing the next process to run. The process has its own processes and stack space to store data and start of the assigned process location.



```
class Process{

friend class ProcessManager;

public:

    Process();
    Process(GlobalDescriptorTable* gdt, void(*_func)(void*));
    ~Process();
    inline unsigned long int get(){return id;}
    bool setCpuState(GlobalDescriptorTable* gdt, void(*f)());
    void setCPUState(CPUState* cpustate);
    CPUState* getCPUState(){return cpustate;}
    void setState(int state);
    void setPC(int PC);
    int getId(){return id;}

private:

    static int static_id;
    int id;
    int status;
    int PC;
    CPUState* cpustate;
    common::uint8_t stack[4096];
};
```

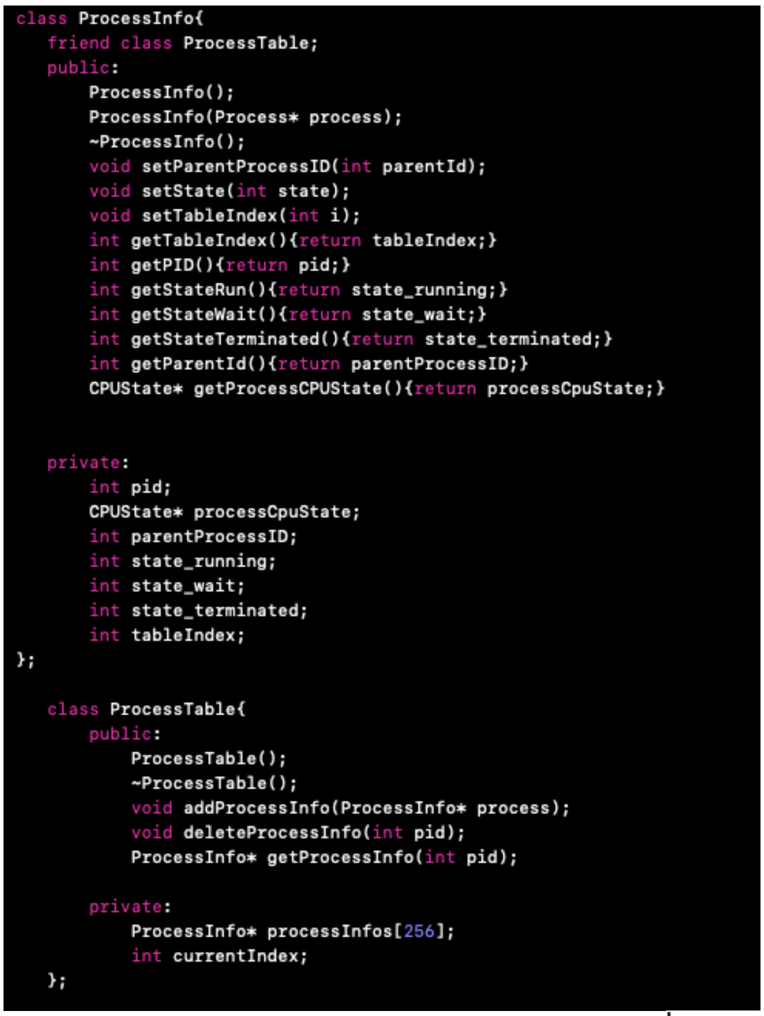


```
class ProcessManager{

  public:
      ProcessManager();
      ProcessManager(Process* process);
      ~ProcessManager();
      inline int getNumProcess(){return processNumber;}
      inline int getCurrentProcessNum(){return currentProcess;}
      Process* getCurrentProcess();
      CPUState* getCurrentCPUState();
      bool addProcess(Process* process);
      void setProcess(int num, Process* process);
      void setCurrentProcess(int _current);
      Process* getProcess(int index);
      int getProcessIdIndex(int id);
      Process* getProcess_id(int id,Process* tempProcess);
      void deleteProcess(Process* process);
      CPUState* Schedule(CPUState* cpustate);

  private:
      Process* processes[256];
      int processNumber;
      int currentProcess;
};
```



This is how the plot happens. One process is created and kept in the process manager. Process manager has processes and current process in an array. Every time an interrupt comes, it is expected to go to the next process. Each task contains a process manager, which is kept in the task manager.

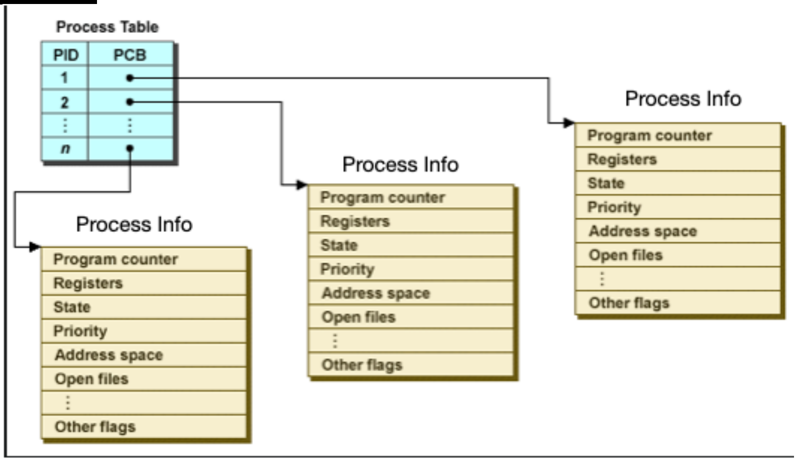
Process Table and Process Info



```
class ProcessInfo{
   friend class ProcessTable;
   public:
       ProcessInfo();
       ProcessInfo(Process* process);
       ~ProcessInfo();
       void setParentProcessID(int parentId);
       void setState(int state);
       void setTableIndex(int i);
       int getTableIndex(){return tableIndex;}
       int getPID(){return pid;}
       int getStateRun(){return state_running;}
       int getStateWait(){return state_wait;}
       int getStateTerminated(){return state_terminated;}
       int getParentId(){return parentProcessID;}
       CPUState* getProcessCPUState(){return processCpuState;}


   private:
       int pid;
       CPUState* processCpuState;
       int parentProcessID;
       int state_running;
       int state_wait;
       int state_terminated;
       int tableIndex;
};

   class ProcessTable{
       public:
           ProcessTable();
           ~ProcessTable();
           void addProcessInfo(ProcessInfo* process);
           void deleteProcessInfo(int pid);
           ProcessInfo* getProcessInfo(int pid);

       private:
           ProcessInfo* processInfos[256];
           int currentIndex;
   };
```

While designing the process table structure, I first decided to define a class called a process info. Process info object is intended to be a class where each process will keep its own information. This process info is also kept in a process table class. In this way, the process table was simulated.

The pid that appears in the table is actually the current index in the class. In this way, when the desired process is retrieved, the value we keep both as pid and index provides convenience.



Scheduler Management

Round Robin Scheduler

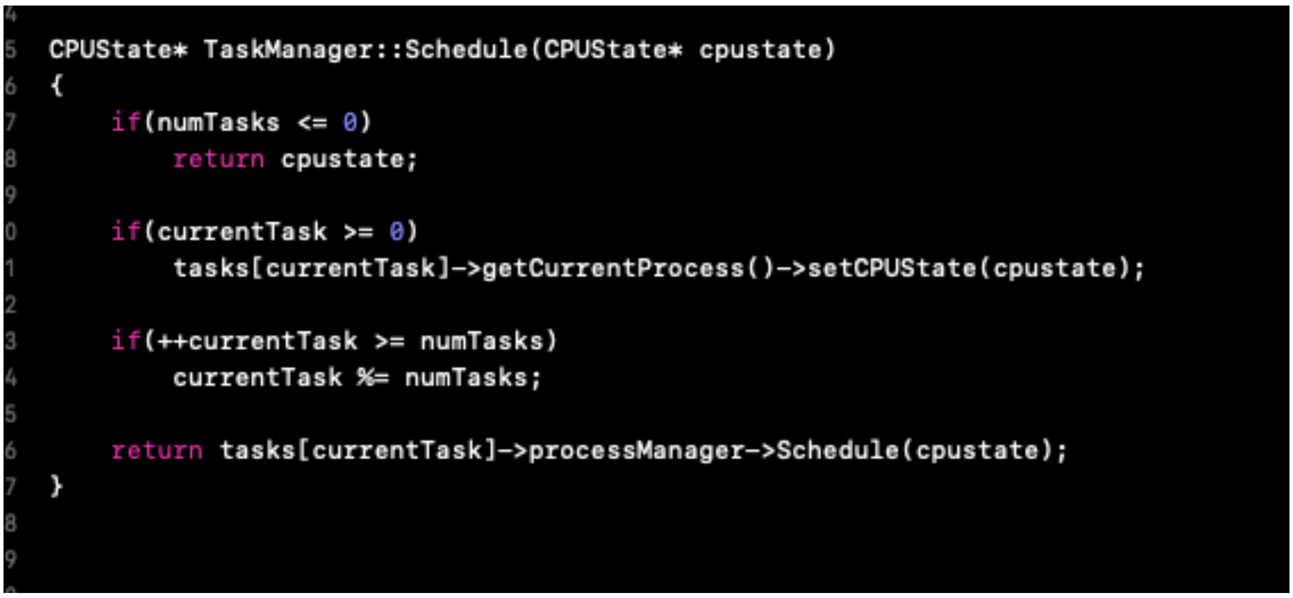
It is a scheduling algorithm. It is one of the famous algorithms used in the timing of the processor (CPU), especially in operating system design. According to this algorithm, the next process has to leave the processor after a certain time unit (time quadrant), even if its job is not finished in the processor.

In this way, there is no possibility of starvation in the operating system. Because it is never possible for a process to take the CPU and prevent other processes from taking their turn.

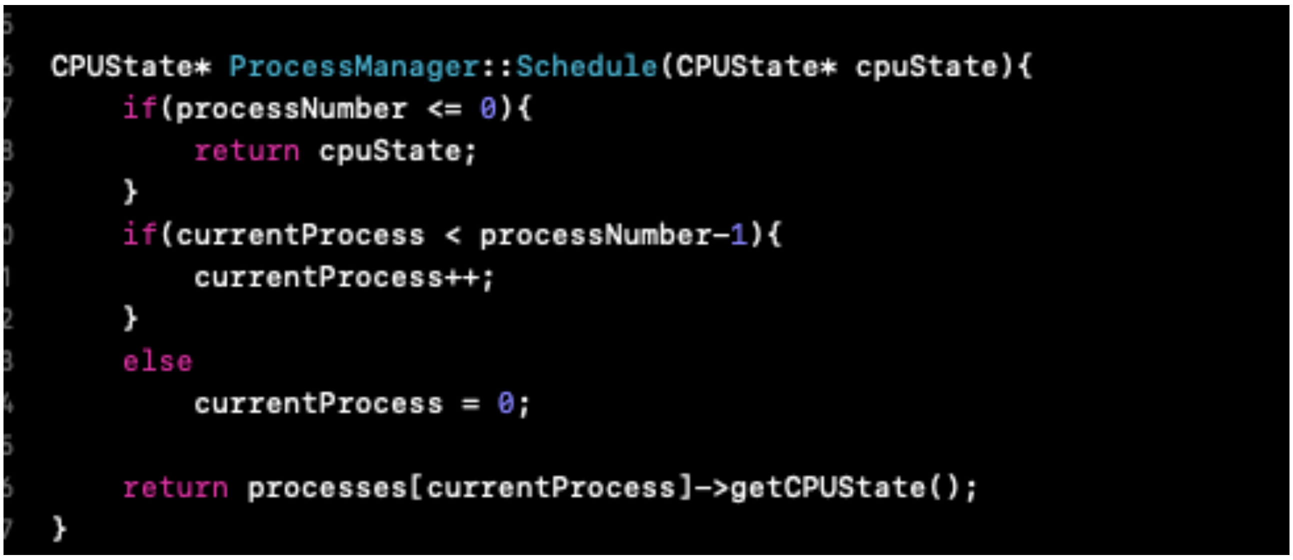
What we need to do is firstly managing the task using task manager then manage the process in the task with process manager with in order. Each process has its own CPU state and stack area but the executing thing in the operating system is tasks CPU state and stack are. So what we need to do is moving the process CPU state and stack area to Task's CPU state and stack area. So in the task class I keep these variable as pointer for easily pointing the processes special areas to not copy the whole structure each time.

First what we did is creating one or more processes and giving them a function for the execution register, then creating a process manager to manage the created these processes. After that to execute these process we are creating a Task object and adding the process manager to Task. Then to manage these created tasks we are creating a task manager object and adding these tasks to task manager. And creating an interrupt handler for the task manager to switch between processes and tasks.

1. When an interrupt occurred process switching algorithm :
2. Running process
3. Interrupt occurred
4. Task manager schedule decide which task's turn
5. Task schedule will be bridge between task manager schedule and process manager schedule
6. Process manager will decided it which process going to execute
7. Chosen process will return its CPU state
8. Process manager scheduler gets this CPU state
9. Process manager and making decision about schedule algorithm according to these (round robin)
10. Task gets CPU state and assign own CPU state pointer so task have a executable CPU state from process
11. Task manager get this CPU state and forwarding to the assembly code
12. Assembly code runs this CPU state on vm
13. Back to first step



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

    if(currentTask >= 0)
        tasks[currentTask]->getCurrentProcess()->setCPUState(cpustate);

    if(++currentTask >= numTasks)
        currentTask %= numTasks;

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



```cpp
CPUState* ProcessManager::Schedule(CPUState* cpuState){
    if(processNumber <= 0){
        return cpuState;
    }
    if(currentProcess < processNumber-1){
        currentProcess++;
    }
    else
        currentProcess = 0;

    return processes[currentProcess]->getCPUState();
}
```

This is how it was done. But the tests were not successful. It will be explained and examined in the demo with its justifications.