Project: 01 Report

By: Nicholas Rich

George Mason University
Prof. Aydin
CS 483 (MW 1:30-2:45pm)

October 6, 2013

HIGH LEVEL-DESIGN

There are five main simulator specific components.

VirtualCPU The core handler of process movement and decision making for the cpu scheduler simulation.

ProcessQueue This is a single process queue, either FCFS or Round Robin at initialization. Handles all process movement within the queue.

DeviceDescriptor This is a single device queue, always FCFS at initialization. Handles all process movement within the queue.

ProcessControlBlock This is a single process. Each of which tracks arrival time, waiting time, turnaround time, and the current state of the process.

BurstNode This is the representation of a single CPU or I/O burst. Each of which tracks the type of burst, duration of burst left, and if of type I/O, the device id.

There are also seven other non-simulator specific components to help with File I/O, settings, and logging.

Settings Handles the command line input and processes it accordingly.

FileReader Provides functions for reading files, uses a callback function to process individual lines using mmap.

FileWriter Provides functions for writing to files. Simply takes a char* and dumps it to file.

Logger Handles all output both console and file based. Determines what to print when based on user input from the Settings component.

LogEntry This is the representation of a single LogEntry.

SimulationState (located in LineInterpreter.c) Provides functions for interpreting file input on a line by line basis and helps the VirtualCPU read in process and device queues as well as processes that have yet to arrive.

Deque This is a macro for defining typed Double-ended queues. For each type that invokes it, it provides the deque functions, an iterator capability, and other utility functions.

All other items seen in the working directory are Deque declarations for the components listed above. For instance, BurstNodeList is a deque of BurstNode's

Deque

This macro ends up defining 3 struct's each time it is invoked. They are type##_dequeN,type##_deque, and type##_dequeI. The first is the deque node that contains pointers to the next and previous nodes and a pointer to data of type 'type'. The second is the deque data structure itself, containing a head and tail pointer to the corresponding type##_dequeN's. There are also two bool values 'trace' and 'debug' that were used to help debug the deque macro itself. The last is the data structure for the iterator. It has a pointer to the current node it is working with and a pointer to the deque it is working from. Creating this macro allowed me to focus on the business logic of the scheduler itself rather than worrying about the individual implementations of each of the queues required for this project.

Reference:

```
#define DEFINE_DEQUE(type) \
    typedef struct type##_dequeN { \
        struct type##_dequeN* next; \
        struct type##_dequeN* prev; \
        type* data; \
    } type##_dequeN ; \
    typedef struct type##_deque { \
        type##_dequeN* head; \
        type##_dequeN* tail; \
        bool debug; \
        bool trace; \
    } type##_deque; \
    typedef struct type##_dequeI { \
        type##_dequeN* current; \
        type##_deque* container; \
    } type##_dequeI;
```

VirtualCPU

This struct is rather straight-forward containing three deque's: One for the X process queues, One for the Y device queues, and One for processes that have completed. In addition, it holds a pointer to the settings struct that is initialized in main.c, the current clock time, and three function pointers to give a semi-object-oriented feel when using the struct. This practice of using function pointers was used in several places for the same reasons as above, purely vanity.

Reference:

```
typedef struct VirtualCPU {
    int clockTime;
    DeviceDescriptor_deque devices;
    ProcessQueue_deque queues;
    PCB_deque terminated;
    Settings* settings;
    bool (*doClockCycle)(struct VirtualCPU *, PCB_deque*);
    int (*getAvgTurnAroundTime)(struct VirtualCPU*);
    int (*getAvgWaitingTime)(struct VirtualCPU*);
} VirtualCPU;
```

ProcessQueue and DeviceDescriptor

These two are very similar and can be covered simultaneously. Both structures contain and id and a PCB_deque, which is a deque of ProcessControlBlocks mentioned above. Where these two differ in in their state data. The ProcessQueue contains two int's, 'quantum' and 'quantumCheck', which store the quantum of the queue and how much of the quantum has been used up by the current process, respectively. The DeviceDescriptor, on the other hand, stores its state as either IDLE or BUSY depending on if a process is in the queue and running or not. The VirtualCPU spends a lot of time manipulating both of these due to the nature of the project.

Reference:

```
typedef struct ProcessQueue {
    int id;
    int quantum;
    int quantumCheck;
    PCB_deque queue;
} ProcessQueue;

enum DDState {
    DD_IDLE,
    DD_BUSY,
};

typedef struct DeviceDescriptor {
    //>>The ID of the device
    int id;
    //>>The enum representation of the state of the device
    enum DDState state;
    //>>Queue of PCB's to be processed
    PCB_deque queue;
} DeviceDescriptor;
```

ProcessControlBlock

This struct stores process specific data. It contains an id for the process, arrival, waiting, and turnaround times, and the state of the process as represented in PCBState below. Additionally, a deque of BurstNodes is kept here to allow the process to communicate where it should go next and for how long.

Reference:

```
enum PCBState {
    PCB_NEW,
    PCB_RUNNING,
    PCB_WAITING,
    PCB_BURST_FINISHED,
    PCB_TERMINATED
};

typedef struct PCB {
    //>>The ID of the process
    int id;
```

```
    //>>The arrival time of the process
    int arrival_time;
    //>>The enum representation of the state of the process
    enum PCBState state;
    //>>The amount of time this process spends waiting until completion
    int waiting_time;
    //>>The internal time that this process completed at
    int turnaround_time;
    //>>A linked list of burstNodes
    BurstNode_deque schedule;
} PCB;
```

BurstNode

The burst node is like an event on a calendar that has no given start date, just a duration, location, and event type. BurstNodes are either CPU or I/O. If it is an I/O type, the device_id is stored so it can get to where it needs to go.

Reference:

```
enum burstType {
    BT_NONE = -1,
    BT_CPU,
    BT_IO,
};

// Structure of PCB (process control block)

typedef struct BurstNode {
    //>>The type of Burst this Node is representing
    enum burstType type;
    //>>The duration of the burst
    int duration;
    //>>The ID of the queue to start on for CPU type, device_id for IO type
    //>>0: pick best option
    int queue_id;
} BurstNode;
```

Other Structures

Settings

This structure holds the information provided at command line as well as a pointer to a Logger structure. There is an Input and Output File name stored, a flag for if the output file was provided and a flag to tell the CPU is the user wants only round robin queues or if a FCFS queue should be added to the end of the list of process ready queues.

Reference:

```
typedef struct Settings{
    char* jobInputName;
    char* jobOutputName;
    bool optfileProvided;
    bool roundRobinOnly;
    Logger* logger;
} Settings;
```

SimulationState

SimulationState is the heart of input processing and serves as a way to keep track of the many stages and flags that are needed to parse the project file format. Some of the data saved here is for bells and whistles such as bool in_comment which enables the files to contain any type of C compatible comment. The FormatStage tracks where we are in the file. The pointer to a BurstNode allows the simulator to accumulate information about a given burst. You will also see a similarity here with the VirtualCPU where there are 3 deque's of type PCB, DeviceDescriptor, and ProcessQueue. This is a temporary location for the loaded queues and processes which are loaded into the VCPU only after reading of the file has completed.

Reference:

```
enum FormatStage {
    FS_TQ,
    FS_PN_NEW,
    FS_PN_AT,
    FS_PN_SCHEDULE
};

typedef struct SimulationState {
    bool in_comment;
    enum FormatStage stage;
    BurstNode* bn;
    int seen_stage_req;
    bool error_thrown;
    bool addFCFSToEnd;
    PCB_deque notYetArrived;
    DeviceDescriptor_deque proto_devices;
    ProcessQueue_deque proto_queues;
    void (*call_back_read)(struct SimulationState*, const char*, const char*);
    void (*call_back_write)(struct SimulationState*, const char*, const char*);
} SimulationState;
```

Logger and LogEntry

Finally, the Logger holds a hodgepodge of data to aid in the formatting of output. The number of the file handle for both in and output files are stored. clockHasChanged, currentClock, and simulationEnded help track when to post a Time = XXX: on the left hand side. Every time Logger-¿log() is called Logger-¿records gets a new LogEntry pushed onto it and then the entire list get dumped and compressed into a single char array for file output. Each LogEntry is simply a char array pointer and a LogLevel. The LogLevel is simply a hat tip to the Java.util.logger.level values and allowed me to control what the user sees when and allows the user to select the level of information seen on the console and in the output file.

Reference:

```
enum LogLevel {
    LogLevel_ALL = INT_MIN,
    LogLevel_FINEST = 300,
    LogLevel_FINER = 400,
    LogLevel_FINE = 500,
    LogLevel_CONFIG = 700,
    LogLevel_INFO = 800,
    LogLevel_SEVERE = 1000,
    LogLevel_WARNING = 900,
    LogLevel_OFF = INT_MAX,
};

typedef struct LogEntry {
    char* entry;
    enum LogLevel level;
} LogEntry;

typedef struct Logger {
    int inputHandle;
    int outputHandle;
    int fileOnly;
    char* buffer;
    int currentClock;
    bool clockHasChanged;
    bool simulationCompleted;
    enum LogLevel LoggerLevel;
    LogEntry_deque records;
    void (*log)(struct Logger*, enum LogLevel, const char*, ...);
    bool (*canLog)(struct Logger*, int);
    void (*dumpToConsole)(struct Logger*);
    char* (*dumpToFile)(struct Logger*);
    char* (*doPrintResults)(struct Logger*);
    int filesize;
} Logger;
```

Thinking in objects, this project creates a Settings object that collects the command line arguments passed to the simulator and processes them, giving notice where needed. From the Settings object, the SimulationState is created and gets set to track all input arriving from a file, to be read line-by-line by the FileReader. The things tracked include new processes to be run, what process ready queues the CPU shall run with and the devices that are used by each process. Once read, the simulation state and settings are passed off to the VirutalCPU which, during initialization, transfers all queues to, and saves a pointer to the settings for future reference. Once initiated, the VirtualCPU begins to tick, each time being passed the list of processes that have yet to arrive. The CPU will continue ticking until all processes have been drained from the list of yet to arrive and while it determines that it still has work to do. For each tick, the CPU performs several sub-functions. It first checks to see if any processes have changed state since the last tick, for example if a burst has completed. This is done for both Device and Ready queues. Then it checks to see if a process should be preempted. Next it checks to see if it has a process that has arrived and treats it accordingly. Follow that, the dispatcher function goes to work seeing if any processes need to be started, preempting any lower running queues. Finally, a system wide tick occurs that updates all PCB's (ProcessControlBlock) in both the device and ready queues. This in turn updates either running or waiting times depending on current state and decrements the amount of time the top most burst has left before completion. While all this is happening, the Logger is receiving log calls and storing each as an entry. When the simulation completes the final wait and turnaround times are calculated and logged. The Logger is then called and it proceeds to iterate over the records to print to the console if the user hasn't disabled this. It then calls for the entire collection of records to be dumped into an appropriately sized char* array that is returned and passed to the FileWriter that then writes the output. Finally all objects malloc'ed are freed in a number of cascading calls. The simulation ends.