OPERATING SYSTEMS PROJECT #2

CS 4328.004

ANDY HERR, jah534
AUSTIN FAULKNER, a_f408

1. A Brief Overview of the Design & Implementation of the 'Discrete-time Event Simulator'

The `program2.cpp` code is written in a set of largely dependent functions such that `main(int argc, char* argv[])` has a few simple, elegant calls to a minimum set of functions. We unpack the fundamentals here.[1]

Before establishing a set of functions, though, we have some constructs that will aid in structuring our approach to the the needed functions:

```
enum EventType {ARRIVAL, DEPARTURE, PREEMPT};
enum Algorithm {UNKNOWN, FCFS, STRF, RR};

typedef struct Process{
   s_t      pid;
   float timeArrived;
   float burstLength;
   float timeRemaining;
} Process;
using procPtr = std::shared_ptr<Process>;

//event structure
typedef struct Event{
   procPtr     relevantProcess;
   float       time;
   EventType   type;

   Event(procPtr p, float t, EventType et)
   : relevantProcess(p)
   , time(t)
   , type(et)
   {}
} Event;
using evePtr = std::shared_ptr<Event>;
```

The `enum` **EvenType** sets out the *events* we will be dealing with in our functions. The `enum` **Algorithm** sets out the *algorithms for handling the events* by means of different scheduling algorithms. The objective a **discrete-time event simulator**, embodied in `program2.cpp`, is to compare and contrast the performance of these algorithms in handling events based upon a few performance metrics (statistics): namely, average turnaround time, total throughput, CPU utilization, and average number of processes in the ready queue−or,

Date: November 8, 2021.

[1] LaTeX

ready queue length.

One can see from the code image on the first page that there is both a `using procPtr = std::shared_ptr<Process>;` for the `Process struct` and a `using evePtr = std::shared_ptr<Event>;` for the `Event struct`. Closer inspection of the two `struct`s reveals that `Process` and `Event` depend on on another in the following way. The `Process` shared pointer `procPtr` is an argument (the left-most one) to the constructor in the `Event struct`. And `procPtr` is the argument type of the function `SRTFEnqueProcess(procPtr process)`. Furthermore, `evePtr` is the argument type of `schedule_event(evePtr eve);`, `process_arrival(evePtr eve);`,`process_departure(evePtr eve);`, and `process_preempt(evePtr eve);`, four of the eleven functions. Notice as well the `eventList` is a linked list of type `evePtr` and the `readyQueue` is a linked list of type `pocPtr`. Thus, the observation can be made that these shared pointers form a *central* role in facilitating the queuing (list-handling) and proper semantics of the functions listed below (along with their behavior).

```
std::list<evePtr>    eventList{};
std::list<procPtr>      readyQueue{};
```

To implement the discrete-time simulator modeled by `program2.cpp`, we use the following functions:

`void init(int argc, char** argv);` // Initializes all CLI input,

`bool run_sim();` // Depends on all of the others in some way and runs the Simulation,

`void generate_report();` // Generates all the Simulation performance statistics,

`void schedule_event(evePtr eve);` // Schedules an event in the future in the `eventList` ,

`void process_arrival(evePtr eve);` // Determines which process remains in the single CPU and which returns to the `readyQueue`. Works on a `switch` among `Algorithm` cases,

`void process_departure(evePtr eve);` // Schedules the departure of a process in the CPU to the `readyQueue`. Works on a switch among `Algorithm` cases,

`void process_preempt(evePtr eve);` // Preempts a process in the CPU to the `readyQueue` and moves the process to the front of the `readyQueue`, then pops it. Works on a `switch` in which the only two relevant `Algorithm` schedulers are SRTF and RR,

`void schedule_new_arrival(bool isFirst);` // Generate the process after determining the process's arrival time. Then, generates the event in which the process arrives. Finally, adds the event to the event queue or `eventList`,

`float genexp(float rate);` // Returns a random decimal number in $[0, 1]$ that follows an exponential (i.e., a Poisson) distribution,

`void SRTFEnqueProcess(procPtr& process);` // Places the process in the CPU in the appropriate spot in `readyQueue`,

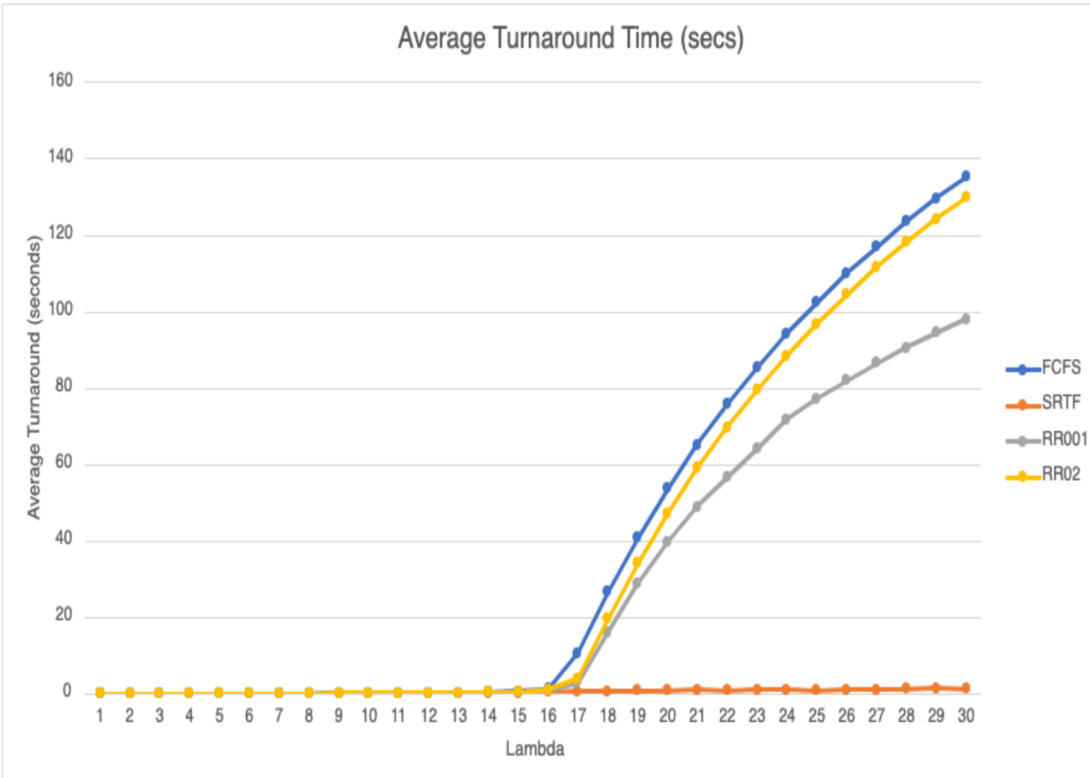
`void RRPreemptOrDepart();` // Schedules the new process's Preemption or Departure (from the CPU),

2. Instructions on How to Compile & Run on the CS Linux Servers

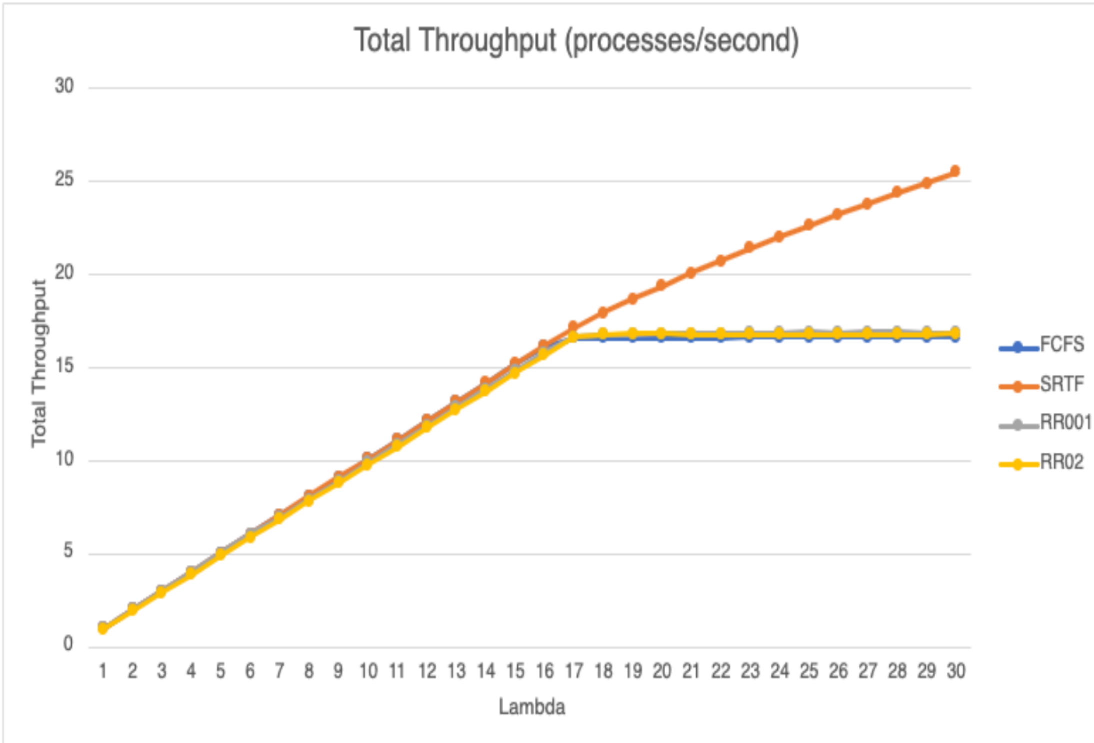
In the `program1_a_f408.zip` or `program1_jah534.zip` files, there is a `makefile`. When you issue the command `make runall` on TX State Linux Server both the program `program2.cpp` and all four the bash scripts will be executed. The output we are concerned with are the files `program2_1.csv` (the FCFS results), `program2_2.csv` (the SRTF results), `program2_3_001.csv` (the RR results with quantum set to 0.01), and `program2_3_02.csv` (the RR results with quantum set to 0.2).

In order to clean up the directory in which all the above files are held, simply issue the command `make clean`. This command like the one is designed for simplicity of use and will erase all `.csv` files and the `program2` executable. The directory will still contain `program2.cpp`, the `makefile`, all bash scripts , and two README files.

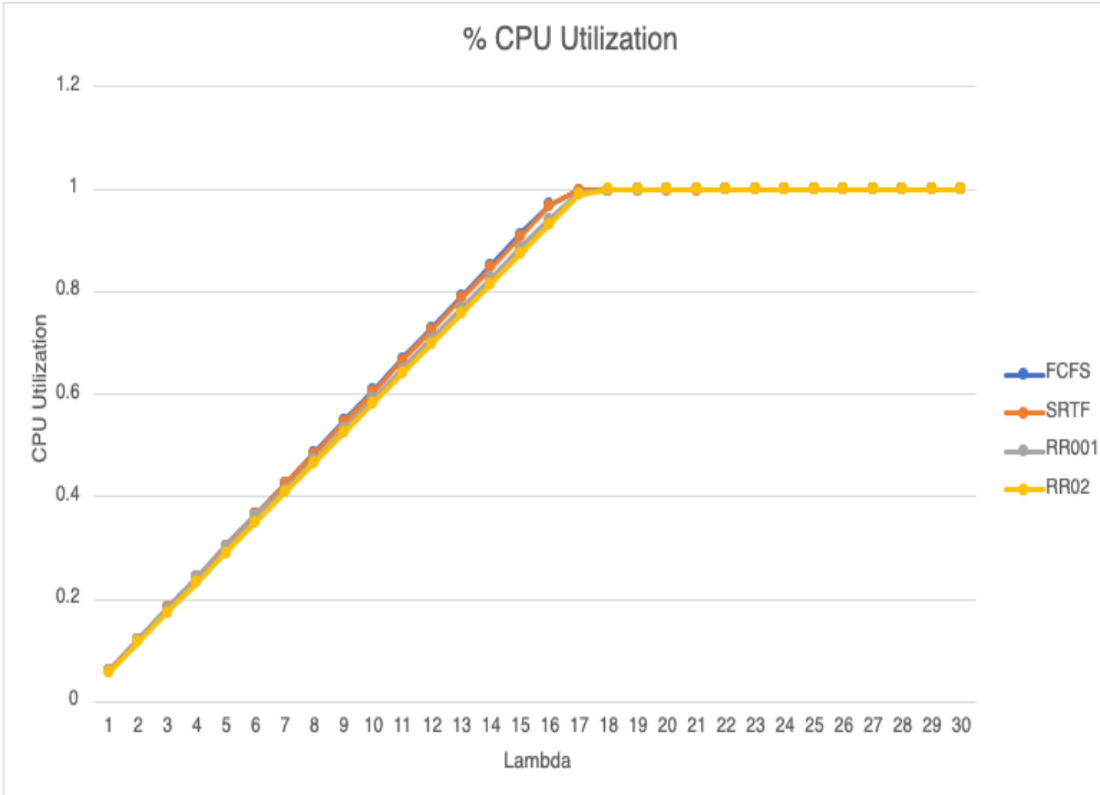
3. Results of the Experiments & and Their Interpretation



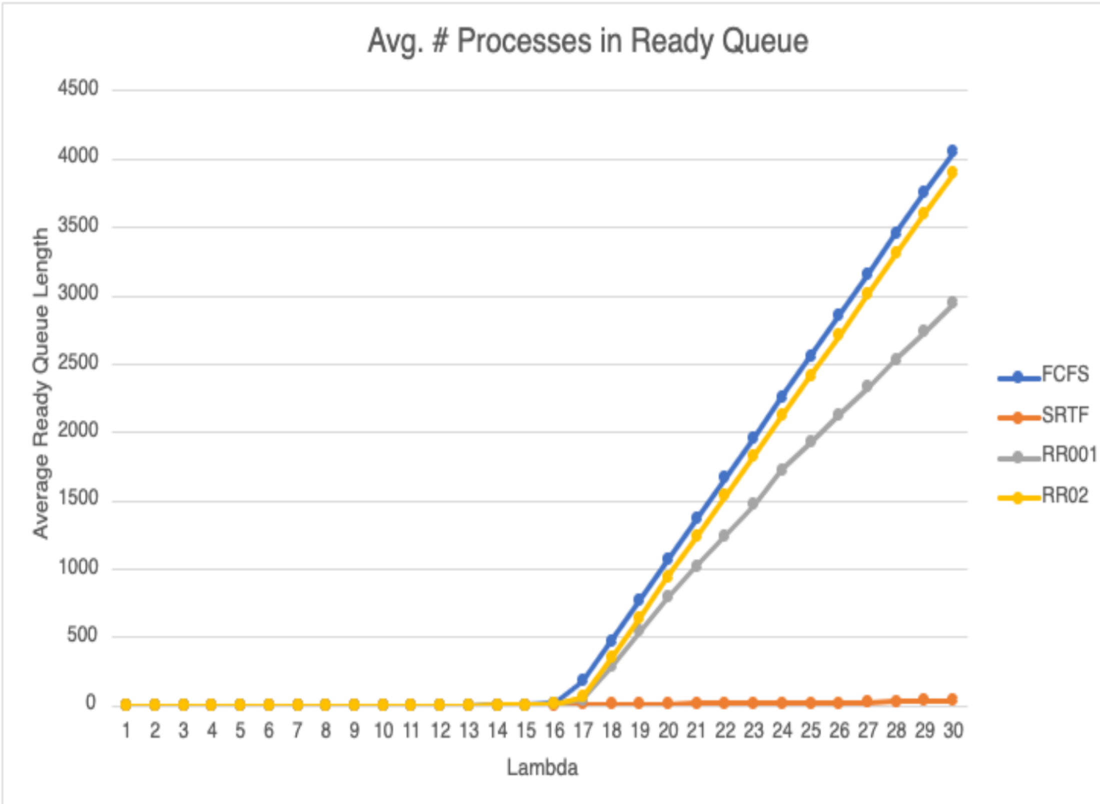
Observations: The Average Turnaround Time (seconds) for all scheduling algorithms are roughly level flat at 0 seconds (that is, $\frac{d(\text{FCFS})}{d\lambda} \approx \frac{d(\text{SRTF})}{d\lambda} \approx \frac{d(\text{RR001})}{d\lambda} \approx \frac{d(\text{RR02})}{d\lambda} \approx 0$) until a critical point at $\lambda = 16$ or $\lambda = 17$ after which the graph exhibits radically different behavior. FCFS, RR001, and RR02 rise dramatically. FCFS and RR02 are very close given that for most processes, the service times are < 0.2, and so processes can complete in one quantum. The result then is that FCFS and RR02 exhibit very similar behavior. On the other hand, smaller quanta, say, RR001's $\lambda = 0.01$, are shorter than than common service times and are helpful in reducing the Average Turnaround Time as shown in grey above. Note that efficiencies fall as we rise up the graph to the right of $\lambda = 17$. In fact, we notice that SRTF barely rises above 1, making it the *most efficient algorithm* when considering Average Turnaround Time. **The Avgerage Turnaround Time efficiency scheme (from most efficient to least) is SRTF $<$ RR001 $<$ RR02 $<$ FCFS**.



Observations: Total Throughput appears to roughly linear in lambda (that is, $\frac{d(\mathrm{FCFS})}{d\lambda} \approx \frac{d(\mathrm{SRTF})}{d\lambda} \approx \frac{d(\mathrm{RR001})}{d\lambda} \approx \frac{d(\mathrm{RR02})}{d\lambda} \approx \alpha\,\lambda$) until roughly the critical point $\lambda = 17$. Afterward FCFS, RR001, and RR02 level off to a constant (that is, $\frac{d(\mathrm{FCFS})}{d\lambda} \approx \frac{d(\mathrm{RR001})}{d\lambda} \approx \frac{d(\mathrm{RR02})}{d\lambda} \approx 0$), while SRTF continues a slower linear trajectory$-$say, $\frac{d(STRF)}{d\lambda} \approx \beta\,\lambda$ with $\beta < \alpha$. Thus, after $\lambda \approx 17$ the Total Throughput of FCFS, RR001, and RR02 roughly halts to about 16 to 17 processes per second while STRF maxes out at roughly 26 processes per second. Notice especially the behavior of SRTF past $\lambda = 17$. In comparison to the rest of the algorithms, which essentially flat line, SRTF sores to a Total Throughput $=$ 26 at $\lambda = 30$, making it far and away the most efficient scheduling algorithm.



Observations: All four scheduling algorithms are roughly linear under a similar trend of Percent CPU Utilization until about $\lambda = 17$. Then all halt to 100% CPU Utilization, exhibiting as the graph does a constant function at 1 (that is, $\frac{d(\text{FCFS})}{d\lambda} \approx \frac{d(\text{SRTF})}{d\lambda} \approx \frac{d(\text{RR001})}{d\lambda} \approx \frac{d(\text{RR02})}{d\lambda} \approx 0$). A closer look does reveal a slight deviation between STRF on the one hand and FCFS, RR001 RR02 on the other. The linear deviation to roughly the left of $\lambda = 17$ demonstrates a *steeper ascent* in Percent CPU utilization for STRF. FCFS, RR001, and RR02 indicate an approximately *equal, slower ascent. This means for STRF that it uses* ***more*** *of the CPU for smaller average arrival times* λ. Whereas FCFS, RR001 and RR02 are more slow to utilize the single CPU source for larger average arrival times λ. Again, this shows that SRTF is the most efficient algorithm of the set.



Observations: Before roughly $\lambda \approx 16$, FCFS is leveled to 0 along with the other scheduling algorithms (in the case of the others, this leveling to 0 occurs up to $\lambda \approx 17$ whereas FCFS's leveling to 0 occurs up to $\lambda = 16$). Then after their respective critical points FCFS and RR02 show an approximately linear trend and very similar behavior where FCFS's trend edges out RR02's by a slightly longer Average Ready Queue Length. As in the case of Average Turnaround time, however, SRTF is roughly leveled with 0, meaning the Average Number of Processes in the Ready Queue is everywhere much smaller than the Average Number Process in the Ready Queue for RR001, RR02, and FCFS, ordered from fewer to more and more. So, just as in the Average Turnaround Time (secs) graph, we again have an **ordering of efficiencies (from greater to lesser): SRTF $<$ RR001 $<$ RR02 $<$ FCFS**. This makes sense, since you would expect the time a process sits around waiting (Average Turnaround Time) to be related to the number of waiting processes (Average Ready Queue Length).