Cory Agami, Brandon Hillan
10/30/2014

Kernel Shuttle Simulation Project Report

1 Overview

The goal of this project was to create a Linux kernel module that simulated a real-time airport shuttle, run to pick-up and drop-off passengers at respective initial and destination terminals. We needed to create Linux system calls, implement them within out module (alongside all the shuttles logic) and write a user-space program to test.

Our implementation utilizes the Linux Kernel's built-in linked list to store the queue of passengers waiting to be picked up, as well as to store the passengers currently on the shuttle. We stored the shuttle and all of its data into a struct for easy accessibility and modularity. We created many helper functions for updating nested/complex data structures for things list keeping a total of the types of passengers picked up and dropped off, as well as running totals of the types delivered to each terminal.

Additionally, mutex locks and kthreads were used to allow for parallelism between the kernel and user space (module and user-space program) and to account for data hazards when both access things like the passenger queue.

A comprehensive pseudocode diagram is provided at the end of this report.

2 Design

Designing the program took many iterations and a few overhauls, but resulted in a remarkably clean and compact codebase.

Our initial design and logic for the algorithm came from writing down pseudo-code in python. It was much simpler to conceive in python initially given the ability to use different data structures. Once we got the complete picture we were able to bring over our logic from python to C much more easily, and implement the kernel's list structure.

2.1 Dynamic Kernel Linked List

Writing our own dynamic array class seemed silly when the kernel had a well-designed and fast linked list implementations ready for us to use. Implementing this was a breeze once we got past the unusual syntax, and memory freeing was also made simple.

We have 2 kernel linked lists for both the shuttles passengers, and the queue of people waiting. We chose to have just one queue for the 5 terminals instead of 5 individual ones for simplicity, as searching through the list was an easy and quick task. With more time we may have considered creating more lists for efficiency, but we believe the tradeoff would have been minimal given our scheduling algorithm.

2.2 Memory Management

Managing our memory inside the module was a matter of simply allocating and freeing memory when it was created and deleted, namely using kmalloc and kfree at points where passengers were getting added or removed from either the shuttle or queues.

2.3 Assumptions

- It takes the shuttle (abs(T1 – T2) * 30 seconds) to travel from terminal T1 to terminal T2
- The shuttle waits 3 seconds more for each passenger if there are more than 4 passengers who wish to load or unload
- Once the shuttle parks at a terminal it waits at least 10 seconds
- We can pick up passengers in any order
- Every 1 second real-time is implemented as .1 seconds in the simulation
- A child needs only half of a seat during transit
- An adult without luggage needs only one seat during transit
- An adult with luggage needs two seats because they need an extra seat for their luggage
- The module will be loaded before calling any of its syscalls
- The modules and syscalls will be properly setup in the kernel before executing the module/userspace program.

3 Development Process

We met on a variety of occasions to divide work and code. We first developed a python implementation of the shuttle (code included at the bottom) which proved vital when developing the final C module. We took some time to lay out the basis for how the module/syscalls should get setup in the kernel and created a starting module to work with. It took some time to figure out how to implement the STUB's and syscall headers correctly, as we were developing on different platforms (32/64bit) but we quickly had it figured out and began working on the module.

Implementing the main shuttle code was easy thanks to the python code as reference. Once we had the shuttle logic working correctly we implemented mutex locks, the kernel linked lists, and proc file support. Thanks to the amount of research we did before beginning the project never got too difficult and we were never out of a solution for very long.

3.1 Additional Issues Faced

Initially in trying to convert the python dictionary and list structures over into C code, we did spend some time trying to create our own data structure, but after talking to our TA we decided to use our better judgment and go with the kernel linked-list.

We had some issues when first testing the module as our threads and locks weren't getting synchronized properly, but we quickly managed to get those under control. Also it took a bit to figure out how to implement the use-case of stopping the shuttle while it had passengers on it, but we realized that fixing the shuttles main run() logic to continue the thread until all passengers were off was the proper way of doing it.

3.2 JOURNAL

- 10/15/14- met and compiled the kernel
- 10/17/14- met in class and discussed how we should go about the project
- 10/18/14- continued research on necessary functions to use and their implementation along with building basic system calls
- 10/19/14- wrote and tested the logic in python
- 10/21/14- met in the lab and started building our module with the logic from the python that we created.
- 10/24/14- continued to build our module and implement stubs and wrappers
- 10/27/14- implemented our main shuttle.c file from our python logic so we could pick up, drop off, and run the shuttle.
- 10/28/14-developed the data structure needed to store shuttle information
- 10/29/14- implemented the proc creation and finished our user space program for testing.
- 10/30/14- Continued to test, optimized code, wrote report, and submitted.

3.3 DIVISION OF WORK

Cory Agami:

> Designed and implemented the program's logic
> Implemented the 3 system calls
> Implemented the kernel list data structure
> Improved code efficiency and logic to our existing code
> Optimized algorithm
> Testing and debugging
> Researched helpful information

Brandon Hillan:

> Created initial module and basic system calls
> Implemented proc creation
> Testing and debugging
> Authored read me and report
> Researched helpful information

4 FINAL THOUGHTS

We strongly believe the development process was not nearly as stressful as it could have been because of our preparedness and because we had an early working prototype in python. We will both take than note into consideration when developing projects in the future.

For future projects we would suggest having part 2 of the project including basic system calls and having part 1 be the module and proc creation. This would give us better background knowledge and remove a little bit of the learning curve with part 3. Overall this would give us a head start on part 3, which is a great degree of difficulty higher than the first two parts.

Cory Agami, Brandon Hillan
10/30/2014

5 PSEUDOCODE (IN PYTHON3.4)

```
list = []
shuttle = { "status":"OFFLINE" , "capacity":50, "load":0, "direction":"forward" ,
      "pass_now":{"C":0,"A":0,"L":0}, "curr_term":3, "dest_term":0 , "passengers":[],
      "pass_had":{"C":0,"A":0,"L":0} , "online":False}
weight = { "C":1, "A":2, "L":4 }
tcoeff = .1

def start_shuttle():
   if(shuttle["online"] == False):
      shuttle["online"] = True
      shuttle["status"] = "PARKED"
      moveto(3)
      return 0
   else:
      print("shuttle already started.\n")
      return 1

def stop_shuttle():
   if(shuttle["status"] != "DEACTIVATING" and shuttle["status"] != "OFFLINE"):
      shuttle["status"] = "DEACTIVATING"
      while(shuttle["load"] > 0):
         run()
      shuttle["status"] = "OFFLINE"
      moveto(3)
      return 0
   else:
      print("shuttle already deactivating.\n")
      return 1

def issue_request(pass_type, init_term, dest_term):
   if((pass_type != "C" and pass_type != "A" and pass_type != "L") or
      init_term > 5 or init_term < 1 or dest_term > 5 or dest_term < 1):
      print("invalid request for passenger(", pass_type, init_term, dest_term, ")\n")
      return 1
   else:
      p = (pass_type, init_term, dest_term)
      list.append(p)
      return 0
```

Cory Agami, Brandon Hillan
10/30/2014

```
def run():
    moveto(shuttle["dest_term"])
    people_getting_onoff = 0

    # removing people
    tmp_passengers = [person for person in shuttle["passengers"]]
    for person in tmp_passengers:
        print("person[2]", person[2])
        if(person[2] == shuttle["curr_term"]):
            people_getting_onoff += 1
            shuttle["load"] -= weight[person[0]]
            shuttle["passengers"].remove(person)
            shuttle["pass_now"][person[0]] -= 1

    # adding people
    tmp_waiters = [person for person in list]
    for person in tmp_waiters:
        if(person[1] == shuttle["curr_term"]):
            if(shuttle["load"] + weight[person[0]] <= shuttle["capacity"]):
                people_getting_onoff += 1
                shuttle["load"] += weight[person[0]]
                shuttle["pass_had"][person[0]] += 1
                shuttle["pass_now"][person[0]] += 1
                shuttle["passengers"].append(person)
                list.remove(person)

    if(people_getting_onoff > 4):
        time.sleep( (people_getting_onoff-4) * 3 * tcoeff )

def moveto(term):
    shuttle["status"] = "MOVING"
    time.sleep( abs(shuttle["curr_term"]-term) * 30 * tcoeff )
    shuttle["status"] = "PARKED"
    time.sleep( 10 * tcoeff )
    shuttle["curr_term"] = term

    # set direction. shuttle moves 1-2-3-4-5-4-3-2-1...
    #                              f-f-f-f-r-r-r-r-f...
    if(shuttle["curr_term"] == 5):
        shuttle["direction"] = "reverse"
    elif(shuttle["curr_term"] == 1):
        shuttle["direction"] = "forward"

    # update dest_term accordingly
    if(shuttle["direction"] == "forward"):
        shuttle["dest_term"] = shuttle["curr_term"]+1
    elif(shuttle["direction"] == "reverse"):
        shuttle["dest_term"] = shuttle["curr_term"]-1
```