Implementation Project Report
Group: Chris, Omar, Chad, Joseph, Matthew

Our Goals / Initial Thoughts:

First and foremost we wanted to design an operating system that we could successfully implement in the time that we were given. We did not want to bite off more than we could chew. So this meant creating a list of features that we would definitely implement, and then a list features that we would implement if we had the time to do so. The two lists that we came up with were:

Critical path:
- User level processes
- Basic system calls (printf, sleep, malloc)
- Clock interrupt
- Simple scheduler
- Screen output
- User input (basic shell)

Things that would be nice to have:
- Memory protection
- Inter-process Communication (Send, Receive, Reply)
- A filesystem
- Complex scheduler

In order to achieve user level processes we knew that we would have to implement basic memory management functions (ie. kmalloc) so that we could give each process a space in memory. We also had to decide what information would be kept in our process structure, and how we would schedule and context switch between processes. We also had to think about the necessary system calls that we had to implement in order for these processes to do anything useful, and how these system calls would function internally.

NOTE: For technical details we have included a documentation directory with our OS that specifically details how we went about implementing all of this functionality. The files in this directory explain at a low level how the operating system runs, and what the code in each file does.

Trouble that we ran into:

The lack of readable documentation on the broadcom chip made it extremely hard to determine how the hardware functions by default, and what peripheral devices were included with it. After several hours of research we determined that the Raspberry Pi has multiple timers that can be used to generate an interrupt (for our clock interrupt), and the documentation for these timers directly contradicts itself in multiple places. This same situation occured over and over again throughout development, and made it extremely hard to determine what the correctly thing to do was.

Version control was another massive problem for our team. SVN worked fine, but it was our group members use of SVN that was completely flawed. Instead of consistently updating the repository with the latest version of our code, it seemed like passing around local copies of the code was much more popular. This meant that the repository was outdated, and many of the group members were completely left in the dark as to what the state of the operating system was.

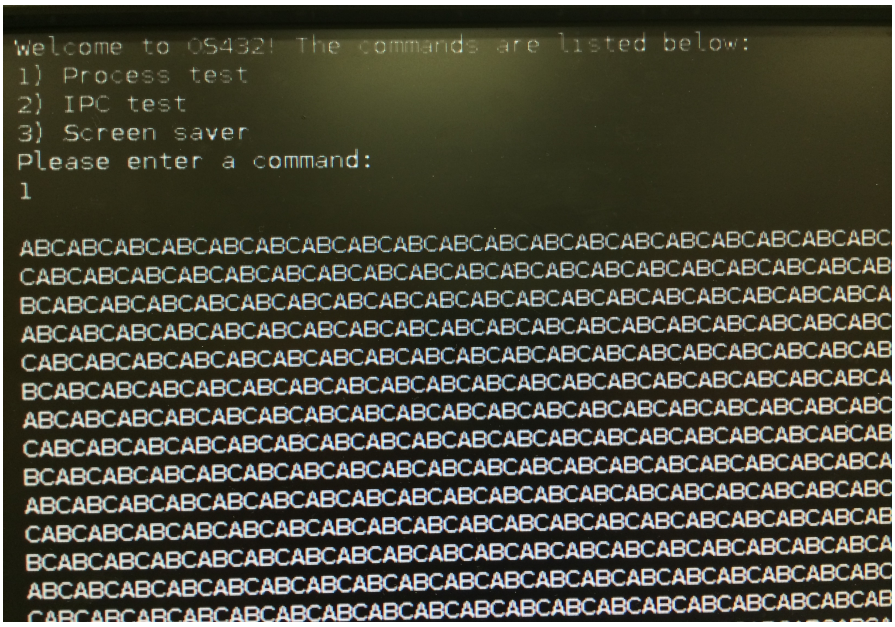
Allocating jobs within our project group was another large issue. Not because any one member was lazy, but because the jobs were not being assigned properly. We didn't spread tasks amongst the group members evenly, and we didn't enforce deadlines appropriately.

We determined that writing drivers for the USB and keyboard would take too much time, and that we would not have been able to finish the operating system if we had done them. So we compromised and decided to use Alex Chadwicks (Creator of the Baking Pi tutorials) drivers. Unfortunately, the drivers that he made were designed for the original version of the Raspberry Pi and do not work well with the version of the Pi that we were given for this class. This meant that our operating system did not boot reliably. Our code ran perfectly without crashing, but the code located inside these drivers crashed a large portion of the time.

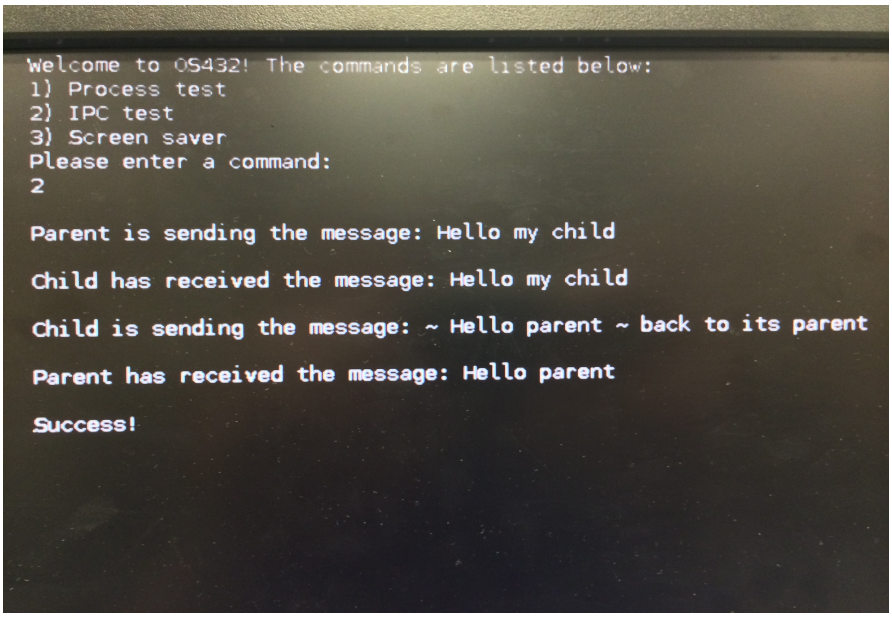
What we achieved:

We were successfully able to implement the critical path stated above, as well as some of the basic IPC primitives that we had on our nice things to have list. This means that we can create multiple user processes, schedule them, context switch between them, and make system calls successfully. In its current state, the operating system prints a shell and accepts one of three commands. The first command launches three user processes. The first of which prints a's, the second prints b's, and the third prints c's and then they all sleep for a small amount of time. This allows us to see the processes being scheduled in a round robin fashion. The second command is intended to show off our IPC system calls. It launches a process which launches a child process of its own, messages are then exchanged between the parent and child process and printed to the screen. The third command launches a process that we are calling the screen saver process. This process "randomly" draws lines across the screen with alternating line colours. Each one of these lines is draw by a system call, and the color is also changed via a system call.

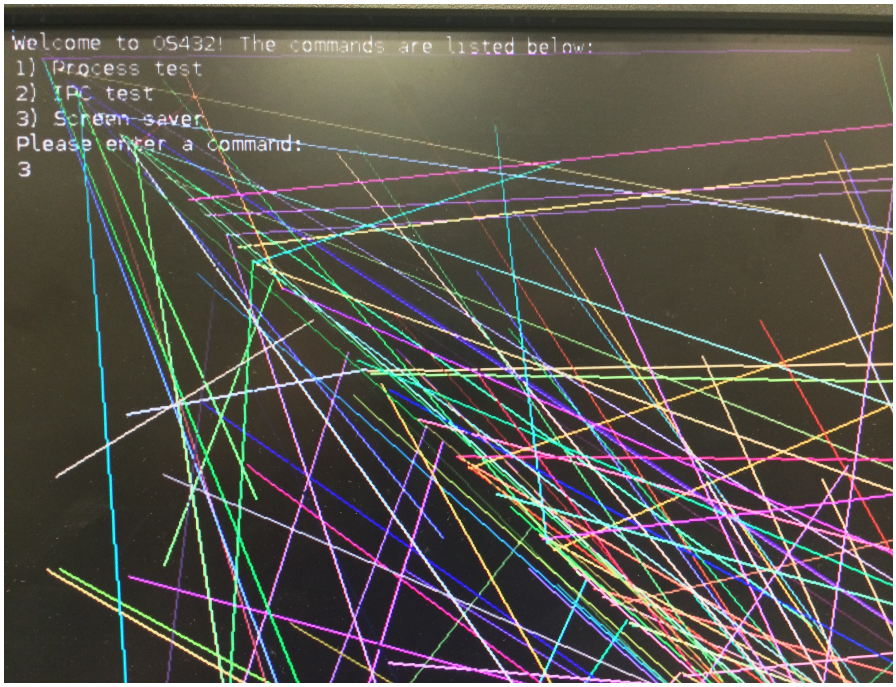
Process test example:



IPC test example:



Screen saver process example:



Experience / Recommendations for the future:

Overall our group agrees that this project was a very valuable learning experience, even with the difficulties that we faced. We've expanded our knowledge of operating systems greatly by doing this project, and we believe that it has given us practical experience for systems programming in the real world. Many job applications that we've see list experience with arm as a valuable trait to have, so we think that working with the Raspberry Pi has been very good experience. However, we have several recommendations that would make this project even more valuable for students in the future:

1) Providing students with a keyboard driver would be extremely valuable, since having to work around a bad keyboard driver has wasted a significant amount of our time.

2) It would be useful if future students had access to parts of our documentation that they could use as a reference, since a lot of the information they need isn't worth wasting huge amounts of time trying to find.

3) Assigning project leaders to manage and distribute the work load amongst the people in the group, and to enforce that deadlines are met.

4) Spend a bit of time reviewing basic version control principles with students in the class, and clearly explain to them that just because your code doesn't currently do anything doesn't mean that it shouldn't be pushed onto the repository. There is a difference between pushing broken code, and pushing incomplete code. Too often people refused to put their code on the repository because they claimed it was broken, but it was simply incomplete. They failed to realize that the code isn't going to "work" until nearly the entire operating system is implemented.

In conclusion:

Overall we consider our project a success, and we have implemented the things that we expected to implement during our design meetings at the beginning of the semester. We also managed to implement one of the things on our "nice to have" list, namely IPC. We believe that we have met the requirement of "designing the components of a simple, but complete operating system" quite well, and we are happy with the final result. As mentioned previously, if you wish to see technical details of the implementation they can be found in the documentation directory contained within our operating system tar ball.