I started off by writing process overview, which shows the list of running programming the winix. This is very useful for debugging purposes, and should be implemented to give me a rough idea of how it feels like to program in winix.

I used a for loop to iterate through all the priority levels, print out the name of the process if it is found in the list. I turned out to be really easy to me, I didn't spent too much time on it, so I moved on to the next task, fork.

Initially , I thought fork is simply creating another data structure, name it a different process, and pointing at the same text segment, with a separate stack, until I received GPF exception.

The idea was to pretty much copy everything, every data field in the proc_t from the original process, except name, proc_index. But Rex Simulator gives GPF everything I tried to do that, at address 0x0000000, which is obviously the default $ra value.

Aftering talking to paul, he suggested starting the system by dynamically writing the binary records values of init.c into the memory at the start, so that way we can know the length of the init.c, and the starting position of init.c (rbase), which facilitates the implementation of fork.

I spent about the next week working on reading srec files, and miscellaneous staff. I copied the load_srec() code from RexSimulator for reading srec file, and convert it from c# into C. It was interesting to see how C# differs from C, by actually translating the code. E.g. wcc compiler only allows me to declare variable at the top of the method, so I would have to move all the temp variables to the top, (about 10), and the top of the method look a bit messy even with commenting. Also, C doesn't have internally implemented String class, and it doesn't have toString(), so I would have to use a temporary buffer to hold the temporary substring value. Since buffer is a fixed array, it would be dangerous to step out of array and write something unsafe. A safer way is to have a big enough buffer, and checkout against the limit everytime buffer is written. I also wrote a hex2int() method that converts hex string into decimal int.

I also added strcat() and strcpy() to the string.c in lib. For copying names in the forked process, and append "fork_" to to the start. Again, both strcat() and strcpy() doesn't check the limit of buffer before copying the values into it, it is up to the programmer to declare a big enough buffer. But again, human can make mistakes, it's safer to have boundary checking before writing anything to the buffer. However, since this project doesn't require me to do strcat() and strcpy(), I didn't spend too much time on those two. I 'll probably improve it in the future.

Apart from load_srec() for reading the srec file, I also implemented reformat_srec.java. The problem with srec file is that they don't have length data, so the program reading it would either have to declare a dynamically linked list, or write record value to memory on the way it's reading it, which could be potentially unsafe. So I wrote a java program that reads the srec file, count the number of lines, and the number of record values in the srec file, and

wrap them up respectively in S5 and S6 data field, which is the header for storing data length the srec standard. In this way, I can read the first two line of the srec file, get the value for the linecount and recordCount, declared a fixed size array with a length of linecount, and then read the rest of the srec data into that array, load that array into load_srec() with the parameter specifying the number of recordData, so that a fixed size array of length of recordCount is declared at the top.

However, the reality is that I working with a rather limited verision of C compiler, which doesn't support variable length array, the data specifying the length of array has to be a constant, which is, by definition in C89, a data declared by #define, or a literal data, e.g. 2.

So that leads to the decision of developing malloc() and free(). I was bit hesistant first because this is not required by the project. But considering the fact that I can't declare variable size array, I would have to dynamically read one line of data from srec file, compile them and write it to memory, until S7 is read at the end. Since it's continuously reading the data from the srec,and writing data to memory, without a way of protecting it, it could be very dangerous. One way of preventing it is to set up a global variable, and no memory write operation is permitted until that variable is set to false. This is too much hard coding, and I decided to rather write malloc() and free() to elegantly solve the problem, rather than ugly hard coding.

malloc() and free() all reside in sys_memory.c. Basically we have a list of struct hole_t, which is struct representing the memory holes in the memory, just like what we did in assignment 1. The malloc first search up all the holes in the unused_holes, if it finds any hole that it fit, it will decrease the size of that hole by the required length (delete the hole if the hole size is equal to the required size). If it can't find any hole that fits, it will call sbrk(), which is increment FREE_MEM_BEGIN by the required size, and return the initial address as a void pointer. Then malloc() create a new hole_t in this case, indicate the start address, which is returned by sbrk, and the length, add it to the used_holes, which indicate all the holes that is allocated by malloc(). The rationale behind that is that we don't want to free() up any memory space that is not malloc(), so we keep track of all the memory we 've malloc, and only free up memory in the used_holes list.

free() takes in the void pointer, which points to the start address of the memory space to be deleted, free() then search up used_holes to find if there is a hole that matches, if it matches, then it's gonna delete that hole from used_space, rewrite all the data in that hole to default memory, value, and add it to the pending_holes. free() also merges adjacent holes together, to ensure that a contiguous block of free or used memory is represented only by one hole struct, rather a list of small hole.

However, it is bit like a chicken egg thing, I can't test my fork(), until exec() works!, so a working exec() would be compulsory for my project to work.

Then I started working exec, which reads a chunk of binary values, write it into the memory space of newly created process, or replace the text segment of existing process with the binary data. Then it reset appropriate register values, e.g. rbase to the top of the text

segment, sp to the virtual address of sp (aboslute address of sp - rbase), make sure OKU in CCTRL is set to 0, pc to the virtual address of the starting position.

BUT, it doesn't work, no matter I did, it exec doesn't work for some reason. I can see from the process overview, that the stack pointer and program counter of the exec loaded process is changing, so it means it is running, but it's not running correctly for some reason I don't know.

I tried to put a break point on the PC of the dynamically created process, but the simulator just crashes for some reason. It also crashes, or simply not responding when I try to read the memory values of a relatively large memory address e.g. 0x2950. Debugging seems impossible in the RexSimualtor.

I would say this is a fairly huge project, I've written about 1500 hundred lines of code so far, (comparing my latest commit to paul's original commit), and I still have finished yet. This is probably more complicated than I thought.