The Big Final Thing

Split OS-like program into two parts:

Loader: does almost everything that our VM program does. Set up virtual memory, interrupt vector, syscalls, etc. Ensure that its own memory pages are in the free list (at the end for safety). Find the other half, `system.exe`, on tape from its length add the correct number of pages to OS code area and one page to OS stack area. Read OS into those pages, set flags and jump to that OS code.

System: the real operating system. As a result of the loader's actions it will only ever run under virtual memory and with the interrupt vector and call gate vector set up already. It makes whatever the user wants to happen happen.

What system does, first step:

User enters basic commands.

If one is unrecognised (e.g. `thing`) search for search for `thing.exe` (search tapes first then disc, and search really just means attempt to open) if found, use its length to calculate the number of code pages needed, add those pages to user code space, read the file into them, add one user stack page, set registers correctly, run it in user mode.

Remember that a lot of things won't be doable in user mode. That will be fixed in step two.

Trap page faults (as in class 19) adding pages to user or system stack area rejecting any non-stack-area requests as fatal run-time errors. A fatal run-time error for a user program just means killing it off. Recycle its memory and return to the OS. There is no real point in trying to handle OS code errors.

Trap halts (as in class 19) in user mode, return to OS, all user memory pages back on free list.

Trap keyboard interrupts, putting the characters received into iosb's keyboard buffer just like readch_tty does when it finds the buffer is empty. Readch_tty should not do this itself any more. If readch_tty finds that the keyboard buffer is empty, it must just wait (little loop including a short pause) until the buffer is not empty (it will fill as if by magic as the keyboard interrupt handler does its job).

OS is using iosb stuff so user programs won't receive keyboard input, but can still use `io` for output.

Second step, making it usable.

Make syscalls for essential things:

Open, close, read, write, etc. (iosb stuff). Big things only, the things that can't be done in user mode. Readch, writech, write, etc can do their work in user mode, using syscalls for the major operations.

Request n pages of memory for (or to be added to) the heap. The OS must know how many code pages you have, so it can decide where your heap pages should be. Heap implementation all happens in user mode, just using this syscall for the one thing that can't be done in user mode.

exit(n) is cleaner than trapping halt instructions, allows for an exit code.

You will need to create a library (analogous to `<unistd.h>`) that user programs will import. It will define functions like exit, open, and so on, and include the correct syscalls.

Pick a control character for control-C's usual duties (control-c itself, along with control-d, control-h, control-i, control-q, control-s, and control-z would be bad choices) and use it to stop user programs (remember OS has control of the keyboard still). So control-C (equivalent) will be detected by the keyboard interrupt handler: you will already be in system mode as is needed.

Third step, multiple processes.

When a user program is running, the OS is still in control of the keyboard (and always will be), so for the time being user programs will still not receive user input. This means that every time the keyboard interrupt handler receives a newline `'\n'`, the OS will take the received line as a command to be obeyed.

Create a moderate array of Process Control Blocks in the OS attic area. If an interrupt or other event requires that a process should pause, that process' volatile state should be copied from the system stack into its PCB, another process is selected, its volatile state is copied from its PCB to the system stack, PDBR is set to the correct address for that process, and a return from interrupt resumes its run.

For this to work, there must be an "idle" process, it does nothing useful but means there will always be another process that can run. This should be the first process that runs, and is made before the OS starts accepting commands. Set aside the first PCB for the idle process, put a reasonable initial state for the registers in its PCB and let it run as a user program.

Create a timer interrupt handler, and set the timer so that it will produce regular interrupts at fairly short intervals. When timer interrupt happens, check all user processes. If there is another one that can run, let it.

Add a `sleep(t)` syscall so that processes can voluntarily be put to sleep for the period of time indicated by t. Don't worry about what that exact period of time is, you can experimentally find out how frequent the timer interrupts are, then sleep will convert the time given (maybe in units of milliseconds) into a number of timer interrupts. A sleeping process will not be selected for running until its sleep period has reduced to zero.

The PCB is the place to store a process' state (so far just runnable or sleeping) and its remaining sleep time.

Fourth step, user input.

Give each process its own keyboard buffer and select another unused control character (lets say control-P for Process). If the user types control-P followed by the digit N, then all subsequent key-presses go to process N's keyboard buffer. N = 0 sets it back to the OS keyboard buffer so more commands can be given.

Create another syscall to wait until keyboard input is ready. This would have the effect of putting the process to sleep until input is available, but don't use the existing sleep state, add a third one.