Lab 6: Adding functionality to HERACLOS

In this lab, we will upgrade the low-level hardware and write new operating system components for the HERA processor. The files will be available in `Shared_Final_Project`.

For this lab, I expect the hardware to be much simpler than the software, and we will undertake this project if I can get the hardware modifications ready in time. You will then choose one software project; at the end of exam period, we will meet for each student to present their project. The projects have some degree of flexibility, and could be suitable for one person or a group of two.

HARDWARE. (I'll do these, unless someone else wants to do them too.)

- Modify the HERA processor so it can transfer data to and from the shadow PC.
- Add a clock line that interrupts HERA periodically (say, every 1000 instructions).
- Add a memory controller system to map logical addresses starting at 0000 into some contiguous region of physical addresses; the controller should have a base register indicating the physical location of logical address 0000, and a limit register that holds one more than the maximum legal logical address. When in "kernel mode" or when both registers are zero, all memory operations should involve physical, not logical, addresses. Address translation should be performed for `LOAD`, `STORE`, `CALL`, and `RETURN` (i.e., operations involving our data RAM), but not instruction addresses (involving ROM). To support shared memory access, addresses f000 through ffff should always be legal, and never be remapped.

SOFTWARE. (Choose **ONE** of the following and discuss it with me before you start.)

- Create a process table, **thread switch** interrupt routine, and thread start and end operations. On interrupt, you will need to store the current state information in the process table, select a new process to run, load the process number into the memory controller, and resume that process. You do not have to change process states (except between `RUNNING` and `READY`), but your table should be flexible enough to allow other states (see "semaphores" below) not run any process in any other state (so that someone could implement, say, semaphores).
- Create a simple **semaphore** data structure in the kernel and write semaphore acquire and release system calls. These will need to change process states in the process table, so you will need to coordinate with who ever is doing that. Modify the input and output routines so that processes use a semaphore to ensure exclusive access to the terminal, and ensure that a process waiting for terminal input is never selected by the scheduler.

- Implement a system for **address translation** and **memory protection**, using the `base` and `limit` registers described in the hardware modifications. The `base` and `limit` will need to be entered in the process table, so you will need to coordinate with anyone doing the thread switching project. You should also provide separate `smalloc` and `sfree` functions to allocated and free regions in the shared section of the address space (f000 ... ffff), but you need not do anything more sophisticated than my existing `malloc` and `free`. This change will keep user threads from accessing the input and output buffers in the kernel, so you should make sure that I/O still works after you make these changes.

- Create system calls for **memory management**. You will need to write functions or system calls that allocate and free regions of memory, replacing my library `malloc` and `free` functions and the `smalloc` and `sfree` from the project above. Note that my `free` function does nothing, and thus my `malloc` can simply return the next unused address. Thus, any program doing repeated `malloc` and `free` calls will always run out of memory with my functions, which should not be the case for a program that frees everything it allocates and never needs very much storage. Your task is, essentially, to fix this. You can undertake this as a project for a single-threaded program, or be more ambitious and connect this with someone doing the address translation project. For a challenge, to try to minimize the amount of work done in kernel mode while still ensuring that the timing of the thread switch interrupt can't cause problems.

- Write a multi-threaded HERA program that uses the features described above to do something that shows them off.

- **As an alterative**, you may ignore the HERA system altogether, and read about the implementation of one of the topics from the last weeks of class (process scheduling, virtual memory, file systems and disk access) or a topic we did not cover in detail (e.g., special concerns of operating systems running on multiprocessors). You should **write a description** contrasting the implementations on at least two real systems, and explaining what each does beyond the extremely simplified system described above, and explain these things to the class (one way to do this is via one or two detailed example programs that you can run on two systems to show their differing responses).