# SYSTEMS II — PROJECT 1
# Multiprogramming and CPU scheduling

## 1 The goal of this project

Now that we have both a BIOS and a fledgling kernel (Project 0), we want to augment the kernel so that it can load and run multiple processes, using a simple round-robin CPU scheduler.

## 2 Strings and the `console` device

The `console` drive is really a RAM whose contents are continually displayed in a window as $80 \times 24$ character grid of byte-sized ASCII values. To change the appearance of the console, simply write values into the memory.[1]

The assembler also provides slight assistance in printing. The assembler provides a `.Text` mode, in which you may label *null-terminated strings*. That is, you may provide, as a constant, a sequence of characters, enclosed in double-quotes. Moreover, you may label that string constant, and thus refer to the addresss at which it will be loaded. The assembler, for each string constant, emit the sequence of byte values that represent the characters in the quotes, and then *append a null character* (that is, a character whose underlying numeric value is zero).

There is also a *console library*—a collection of procedures that help to manipulate the console. In particular, there is a `print()` procedure provided, which itself calls on a `scroll()` procdeure to scroll the console as more text is added. I recommend using these, although you are welcome to write your own console maniulation code.

## 3 Single-segment virtual addressing

Part of your implementation of a multiprogrammed system is the use of simple virtual addressing. Specifically, each process should be allocated a single, contiguous segment of main memory. The MMU should enforce the use of that memory segment. By setting the `BS` (*base*) and `LM` (*limit*) registers, and then by selecting the *virtual addressing* mode, the MMU should virtualize each process's addresses, allowing it use virtual locations between 0 and $limit - base$. Moreover, any attempt to use virtual locations outside that range will trigger an `INVALID_ADDRESS` interrupt.

### 3.1 Assembler changes

In order for the MMU to do its work, the kernel must be capable of setting the `BS` and `LM` registers as part of scheduling each process on the CPU. Therefore, the assembler now recognized two new instruction opcodes:

1. `0x0021:   SETBS` *[source A]*

[1] If you look at the physical address range to which the console responds, that RAM, known as a *display buffer*, is exactly $80 \times 24 = 1920$ bytes.

**Set** the `BS` (*base*) register. This one-word register stores the base physical address of a segment of the virtual memory used by a process. The MMU will add this register's value to each virtual address requested by the user-level process, making the virtual address space appear to begin at address zero. It with then check that the resulting physical address is not less than the base itself before sending that address to the memory bus.

2. `0x0022:   SETLM` *[source A]*

   **Set** the `LM` (*limit*) register. This one-word register stores the limit physical address of the segment of virtual memory used by a process. After adding the base to each virtual address requested by the user-level process, the MMU will check that the resulting physical address is less than the limit itself[2] before sending that address to the memory bus.

For the kernel to activate the translation capabilities of the MMU, it must put the processor into *virtual addressing mode*. Specifically, the `JUMPMD` process should set two flags:

- **User mode:** The most significant bit of the *mode* value, stored into the `MD` address, indicates whether the processor is in user or supervisor mode. Setting this bit to 1 activates user mode; 0 indicates supervisor mode.
- **Virtual addressing mode:** The next most significant bit (that is, bit 30, given that bit 0 is least significant and bit 31 is most) is a boolean flag indicating the addressing mode. When this bit is 0, the MMU is inactive, and all addresses are physical. Set this bit to 1 to activate the MMU to perform single-segment virtual addressing.

## 3.2 Simulator changes

You should be able to use the `setregister` and `showregister` commands to examine the `MD`, `BS`, and `LM` registers. Do so to ensure that the kernel is configuring and activating virtual addressing correctly. Additionally, you may set the `DB` (*debug*) register to 2 if you want to see more detailed messages about how the MMU is translating addresses.

Finally, note that you may use the following simulator commands to see **all** of the registers:

- `showregisters text`: This command will print, to your shell, the values of all of the named and unnamed registers.
- `showregisters graphic`: This command will create a new window, in which the values of all the named and unnamed registers are shown and updated continually.

## 4 Your assignment

Modify your kernel so that the simulator may be passed any number of additional ROM images (after the BIOS and kernel). For example, you should imagine invoking the simulator like so:

```
$ k-simulator bios.vmx kernel.vmx init.vmx add-two-numbers.vmx fibonacci.
```

[2]Note that the comparison is for strict inequality. The implication is that the *limit* is the physical address of the first byte **beyond** the segment—that is, the lowest **invalid** address beyond the *base*.

## 4.1 The `init` program

Notice that, after the BIOS and kernel images, the example above passes an image named `init.vmx`. This initialization program is one that you should write to perform a specific task: It should call on the kernel to create one new process for each of the remaining executable images (`.vmx` files). The implication is that the kernel should create a new process using the image that follow itself (here, `init`); in turn, the `init` program should request that the kernel create two new processes, one each for `add-two-numbers` and `fibonacci`.

## 4.2 System calls

The above description implies that `init` must perform *system calls* to request information about how many ROM's (executable images) there are beyond the first three (BIOS, kernel, init), and to request the creation of each new process. Additionally, all of the user-level processes should, when they have completed their work, be able to terminate themselves via a system call. Therefore, your kernel should implement (at least) three system calls:

1. `IMAGE_COUNT`: When a user-level process performs this system call, the kernel should return the number of user-level executable images available in the system (that is, the number passed when the simulation began).

2. `EXECUTE`: This system call should create a new user-level process. The caller must be able to specify which executable image (ROM, `.vmx`) should be used in the creation of this new process. The new process must be created and added to the CPU scheduler's collection of runnable processes, to be scheduled later.

3. `EXIT`: When a user-level process performs this system call, its process object should be removed from the CPU scheduler's set of runnable processes, and its main memory segment should be released.

## 4.3 Other kernel capabilities

In order to support the simulataneous execution of multiple processes, the kernel must also use and handle clock alarm interrupts. Specifically, when an alarm interrupt occurs, the CPU scheduler should be invoked. This scheduler should deschedule the current user-level process, recording its state into its process object. It should then select a new process object, and then schedule that process on the CPU by restoring its state to the CPU itself.

To perform these scheduling tasks, the kernel will need be maintain its own stack and heap. The kernel should be composed of a set of procedures, each of which properly maintains its own stack in the kernel's segment. Additionally, the kernel should contain a procedure that can allocate $b$ bytes from the the heap region of the kernel's space, returning a pointer to that space. This allocator should be used to create the process objects maintained by the CPU scheduler.

## 5 How to submit your work

Use the CS department submission system command to turn in your programs. You should submit all of your `.asm` files, BIOS, kernel, init program, and any user-level programs you write for testing.