# Objective 2 Directions

### Objective 2 Overview

In this objective you will develop functions which initialize the memory and memory management data structures of the simulator, simulate the basic functions of the CPU, as well as provide more simulation output.

The simulator will initally call your `Boot()` function that will load programs from `boot.dat` that are stored in the format described in intro.doc. `Boot()` will also initialize the data structures responsible for managing the simulated memory and will call `Get_Instr()` repeatedly to read instructions from `boot.dat` and will store them in the simulated memory. Finally, `Boot()` will call `Display_pgm()` for each program in `boot.dat` to output it to `simout`.

After `Boot()` has completed `XPGM()` will be called which simulates a context switch and then calls `Cpu()`. `Cpu()` sets the memory address register (MAR) to the value passed to it by `XPGM()`, this will be 0 initially. `Cpu()` then calls `Fetch()` to get the next instruction to execute from memory. `Fetch()` calls `Mu()` to determine the physical location in memory of the requested instruction and uses the result to return the instruction to `Cpu()`. `Cpu()` then handles the instruction accordingly depending on the operation. This entire cycle then repeats until there are no more simulation events.

### Important variables and data types

`MEMMAP`:

defined in `simulator.c` is a pointer to `2 * MAXSEGMENTS` variables of type `struct segment_type` (defined in `osdefs.h`). User memory is `MEMAP[0] ... MEMMAP[MAXSEGMENTS – 1]`, the rest is reserved for the OS. Each segment_type has fields for the segment length in instructions (`seglen`) and the base address (`membase`) in memory where the segment begins.

`MEM`:

defined in `simulator.c` is a pointer to `MEMSIZE` variables of type `struct instr_type` (defined in `osdefs.h`).

### Boot

```
void Boot(void)
```

*This function is called from* `simulator.c` *and reads from* `boot.dat` *and initializes the memory and memory management data structures. The programs from* `boot.dat` *represent the OS and are loaded into the upper half of*

*`MEMMAP`*.

**Directions:**

1. Read the file `boot.dat` whose file pointer is `PROGM_FILE[BOOT]` and whose format is given in intro.doc. You will have to check for `PROGRAM` on the first line and read in the number of programs in the file. Then read in each segment and store the access bits and number of instructions.
2. With the program and segment data initialize `MEMMAP` starting at segment `MAXSEGMENTS`. The size of `MEMMAP` is `2 * MAXSEGMENTS`. The first half is reserved for user memory, while the upper half is reserved for the OS.
3. Call `Get_Instr()` repeatedly to read instructions from `boot.dat` and update `TotalFree` and `FreeMem` based on the number of instructions read from `boot.dat`.
4. Call `Display_pgm()` to display each program.

## Get_Instr

```
void Get_Instr(int pgmid, struct instr_type *instr)
```

*This function reads the next instruction from* `file (fp)` *into* *`instr`*. *The external* *`file (fp)`* *is* *`PROGM_FILE[pgmid]`*. *The format of the file is a series of statements of the form:* *`OPCODE   x   y   z`* *where the form and type of the operands (x,y,z) depend on* *`OPCODE`*. *Each instruction starts on a new line. There is more information in intro.doc on the format of boot.dat.*

**Directions:**

1. Read the instructions from `boot.dat` (`PROGM_FILE[BOOT]`).
2. Convert the instruction to its opcode by using the lookup table `opidtab` which is defined in `simulator.c` if the instruction is not a device. If it is a device look up its opcode in the `devid` field of the `devtable`.
3. After determining the opcode set the operand as described in intro.doc. Each instruction has a field for the opcode and operand. The operand field is a C union and depending on the opcode, only certain fields will be used in the operand. The `address` field is used for `REQ` and `JUMP` instructions, the `count` field is used for `SKIP` instructions, the `burst` field is used for `SIO`, `WIO`, and `END` instructions, and the `bytes` field is used for device instructions. The data structures are show below:

```
 struct instr_type {
   unsigned   char     opcode;
   union opernd_type operand;
 };

union opernd_type {
      struct addr_type address;
      unsigned int      count;
      unsigned long     burst;
      unsigned long     bytes;
};
```

## Cpu

```
void Cpu(void)
```

*This function simulates the basic functions of a CPU. It fetches instructions from emory and handles them accordingly.*

**Directions:**

1. Follow the instructions given in `obj2.c` in the comments surrounding the `Cpu()` function.

## XPGM

```
void XPGM(struct state_type *state)
```

*This function simulates a priveleged instruction causing a context switch.*

**Directions:**

1. Follow the instructions given in `obj2.c` in the comments surrounding `XPGM()` function.

## Mu

```
int Mu(void)
```

*This function simulates the address translation hardware of the memory unit.*

**Directions:**

1. Follow the instructions given in `obj2.c` in the comments surrounding the `Mu()` function.

## SetMAR

```
void SetMar(struct addr_type *addr)
```

*This function sets a global variable MAR representing the memory address register.*

**Directions:**

1. Follow the instructions given in `obj2.c` in the comments surrounding `SetMar()` function.

### Fetch

```
int Fetch(struct instr_type *instr)
```

*This function will try and fetch an instruction from memory and store it in `instr`.*

**Directions:**

1. Follow the instructions given in `obj2.c` in the comments surrounding `Fetch()` function

### Read

```
int Read(struct instr_type *instr)
```

*This function is identical to* Fetch()

### Display_pgm

```
void Display_pgm(struct segment_type *segtab, int numseg,
      struct pcb_type *pcb)
```

*This function outputs a program to `simout`.*

**Directions:**

1. Follow the instructions given in `obj2.c` in the comments surrounding `Display_pgm()` function.
2. Be sure to print the process and program names as "`BOOT`" in Objective 2 since `pcb` will always be null.