OSSIM – Objective 2

- Overview
  - 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.
  - Boot:
    - 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.

OSSIM – Objective 2

- Overview
  - 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.

OSSIM – Objective 2

- Important Variables
  - MEMMAP
    - A pointer to 2* MAXSEGMENTS of type struct segment_type
    - User memory
      - MEMMAP[0] … MEMMAP[MAXSEGMENTS - 1]
    - Kernel Memory
      - MEMMAP[MAXSEGMENTS] … MEMMAP[2 * MAXSEGMENTS - 1]
    - Each segment_type
      - segment length in instructions (seglen)
      - the base address (membase) in memory where the segment begins.

OSSIM – Objective 2

- Important Variables
  - MEM
    - A pointer to MEMSIZE array of data types of type struct instr_type (defined in osdefs.h).

OSSIM – Objective 2

- 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.

OSSIM – Objective 2

- **Directions:**
  - 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.
  - 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.
  - 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.
  - Call Display_pgm() to display each program.

OSSIM – Objective 2

- 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.

OSSIM – Objective 2

- **Directions:**
  - Read the instructions from boot.dat (PROGM_FILE[BOOT]).
  - 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.
  - 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.

OSSIM – Objective 1

- Data structures
  - struct instr_type {

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

  - union opernd_type {

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

OSSIM – Objective 1

- void Cpu(void)
  - This function simulates the basic functions of a CPU.
  - It fetches instructions from memory and handles them accordingly.
- Directions
  1. SetMAR(&CPU.pc)
  2. Fetch(&IREG);
     - If Fetch() returns negative, a FAULT has occurred. In this case, CPU() returns. Otherwise, continue.

OSSIM – Objective 1

- **Directions:**

3. Decode IREG and execute the instruction
    - SIO, WIO, and END instructions: compute the deltaT defined by the operand, add to CLOCK to get future event time, and add this new event to the event list. Increment CPU.pc.offset by 2 and return
    - FOR SKIP, evaluate the operand. If the operand of IREG changes, you must update MEM by a call to Write() with the modified IREG. Increment CPU.pc.offset by 2 if the next instruction is to be skipped and repeat from step (1).
    - Otherwise, Fetch() the JUMP instruction at CPU.pc.offset+1. Execute the JUMP by placing its operand in CPU.pc and repeat from step (1).

- NOTE: you will find the function Burst_time() in SIMLATOR.C of use when converting from CPU cycles to simulation time.
- NOTE: For OBJECTIVE 2 you should use a special agent code (0) to identify the BOOT program. For all other OBJECTIVES the agent code should be: CPU.actvpcb->termnl+1.

OSSIM – Objective 2

- void SetMar(struct addr_type *addr)
    - This function sets a global variable MAR representing the memory address register.
- **Directions:**
    - Set the MAR with the value of (addr) and return.
- This function must be called to define the logical MEM address of the next Fetch(), Read(), or Write() operation on memory.

OSSIM – Objective 2

- int Fetch(struct instr_type *instr)
  - This function will try and fetch an instruction from memory and store it in instr.
- **Directions:**
  - This function calls Mu() to validate and map the logical address in MAR to a physical address. Mu() will return a negative value if some kind of FAULT was generated. In this case, Fetch() returns -1.
  - If Mu() returns a non-negative value, say x, then Fetch sets *instr = MEM[x] and returns +1.

OSSIM – Objective 2

- int Mu(void)
  - This function simulates the address translation hardware of the memory unit.
  - It uses the contents of MAR = [s, d] as the logical address to be translated to a physical address, x.

OSSIM – Objective 2

- **Directions:**
  - First compute the effective entry in MEMMAP.
    - Set SEG = s + CPU.mode*MAXSEGMENTS.
    - This forces the upper half of the MEMMAP to be used if CPU.mode == 1 (priviledged mode) and the lower half if CPU.mode != 1 (user mode).
  - If MEMMAP[SEG].accbits == 0x00, then generate an SEGFAULT event at the current CLOCK time and add it to the event_list using Add_event(). return -1.
    - use Agent = CPU.actvpcb->termnl+1. (agent = 0 for objective 2)
  - If MEMMAP[SEG].seglen <= d, then generate an ADRFAULT event at the current CLOCK time and add it to the event_list. return -1.
    - use Agent = CPU.actvpcb->termnl+1. (agent = 0 for objective 2)
  - return x = MEMMAP[SEG].membase + d.

OSSIM – Objective 2

- void XPGM(struct state_type *state)
  - This function simulates a privileged instruction causing a context switch.
- Directions:
  - switch placing a user program in execution. It does this by copying (state->mode) into CPU.mode and (state->pc) into CPU.pc.
  - After the state of the CPU has been redefined, the CPU resumes execution at CPU.pc -- this is implemented by simply calling the function, Cpu().

OSSIM – Objective 2

- int Read(struct instr_type *instr)
  - This function is identical to Fetch()
- Directions
  - This function calls Mu() to validate and map the logical address in MAR to a physical address. Mu() will return a negative value if some kind of FAULT was generated. In this case, Read() returns -1.
  - If Mu() returns a non-negative value, say x, then Read sets *instr = MEM[x] and returns +1.
- int Write(struct instr_type *instr)
  - This function is similar to Fetch
- Directions
  - This function calls Mu() to validate and map the logical address in MAR to a physical address. Mu() will return a negative value if some kind of FAULT was generated. In this case, Write() returns -1
  - If Mu() returns a non-negative value, say x, then Write sets MEM[x] = *instr and returns +1.

OSSIM – Objective 2

- void Display_pgm(struct segment_type *segtab, int numseg, struct pcb_type *pcb)
  - This function outputs a program to simout.
  - Use this function after every program load. */
  - Use segtab, segment table * to locate each segment. Use the provided sample output in intro.doc as an example of the format of the dump and what information should be output.
- **Note:** Be sure to print the process and program names as "BOOT" in Objective 2 since pcb will always be null.