

University of Central Florida  
School of Electrical Engineering and Computer Science  
COT4600 - Operating Systems.  
Fall 2009 - dcm

**Class Project:** simulate the operation of a simple kernel for a computer system.

The simulator should be written in Java; the students who are not familiar with Java could use C or C++; the code should be amply commented. Keep in mind that the simulator becomes increasingly more complex as you add more functions to it and a good design is one that has the ability to accommodate changes.

The detailed design and the functions supported by the simulator are left to the student, we only provide some hints for minimal functionality. The more functions your simulator supports, the more points the project deserves. Do not procrastinate, start working on the project immediately; think carefully before starting the implementation. It takes more time than you think to design the system and even longer to implement and test it.

For each phase you are expected to submit a two to four page design outline. Email the code, the instruction how to test it, and the design outline to the TA only. Make sure that you get an acknowledgment from the TA that your submission has been received.

The project involves multiple phases:

1. Simulate a processor with a minimal instruction set operating in kernel and user mode. Due week 4.
2. Virtualize the memory. Design and implement a paging system and a virtual memory manager. Due week 8.
3. Virtualize the processor. Add a thread management system. Due week 10.
4. Add a virtual communication channel allowing threads to communicate using a bounded buffer and *send* and *receive* primitives. Due week 14.

The analysis of resource virtualization (memory, processor, communication channels) is covered in Chapter 5 and will be covered in class starting with Lecture 17. To have sufficient time to complete the project the students are encouraged to read ahead the topics regarding virtual memory and threads and to ask questions regarding the implementation during the class or at the office hours.

The Processor The processor operates either in *user mode* when it uses the user registers, or in *kernel mode* when it uses the kernel registers.

The processor has several registers:

1. Mode Register: set to 1 in user mode and to 0 in kernel mode:

$$MR \leftarrow 1 \text{ for user mode} \quad \text{and} \quad MR \leftarrow 0 \text{ for kernel mode}$$

2. User mode

- Program Counter- UPC

- Stack Pointer - USP
- Page Map Address Register - UPMAR
- 16 General Purpose Registers:  $R0-R15$

### 3. Kernel mode

- Program Counter- KPC
- Stack Pointer - KSP
- Page Map Address Register - KPMAR
- 16 General Purpose Registers:  $KR0 - KR15$

The processor operates on 32 bit words (or 4 byte words) and has 32 bit registers. The maximum size of the address space of a process is  $Mem_{max} = 2^{32} - 1$ . Addresses and the contents of a memory address are expressed as hexadecimal integers. The 16 hexadecimal symbols are digits are:  $0 - 9, A(10), B(11), C(12), D(13), E(14), F(15)$ . We use *little endian* representation (big-endian and little-endian refer to which bytes are most significant in multi-byte data types and describe the order in which a sequence of bytes is stored in memory). For example the integer  $1025 = 1024 + 1 = 2^{10} + 2^0$  is stored in memory on four bytes with addresses 00, 01, 10, 11 as follows:

Byte address (decimal)	Contents (in binary)	Contents (in hex)
00	00000001	01
01	00000100	04
02	00000000	00
03	00000000	00

The processor has the following instruction set:

- **L** - load the contents of memory location  $addr$  into a general-purpose register  $R_i$ ,  $0 \leq i \leq 15$ :  $R_i \leftarrow Mem(addr)$  and  $KR_i \leftarrow Mem(addr)$ . For example:

$L \quad R1, X'0010'$       Load in R1 the contents of address X'0010'

- **ST** - store the contents of the general-purpose register  $R_i$  to memory location  $addr$ :  $Mem(addr) \leftarrow R_i$  and  $Mem(addr) \leftarrow KR_i$ . For example:

$ST \quad R1, X'0010'$       Store the contents of R1 at address X'0010'

- **STS** - Save the state on the stack. Store the contents of the Program Counter and of the registers  $R_i$  to the memory address pointed at by the stack pointer:  $Mem[USP] \leftarrow UPC, R0, R1, \dots, R14, R15$  and  $Mem[KSP] \leftarrow KPC, KR0, KR1, \dots, KR14, KR15$ .
- **RTS** - Restore the state. Load the Program Counter and the registers  $R_i$  from the memory address pointed at by the stack pointer:  $UPC, R0, R1, \dots, R14, R15 \leftarrow Mem[USP]$  and  $KPC, KR0, KR1, \dots, KR14, KR15 \leftarrow Mem[KSP]$ .
- **AR** - add the contents of the register  $R_i$  to the contents of the register  $R_j$  and store the result in register  $R_i$ :  $R_i \leftarrow R_i + R_j$  and  $KR_i \leftarrow KR_i + KR_j$ . For example:

$AR \quad R1, R2$       Add the contents of R1 and R2 and store the result at address X'0010'

- **LA** - load address: load the hexadecimal constant  $X'xxxx'$  into register  $Ri$ :  $Ri \leftarrow Mem(addr)$  and  $KRi \leftarrow Mem(addr)$ . For example:

$LA \quad R1, X'0010' \quad \text{Load in R1 } X'0010'$

- **RTI** - return from kernel mode to user-mode. The instruction carries out the following actions:
  1. Restore the state of the user process including the stack pointer (see the description of the memory map in Figure 1 and then the program counter, and the the general-purpose registers.
  2. Set the mode bit:  $MR \leftarrow 1$ ;
- **SVC k** - Switch from user to kernel mode - execute an SVC (supervisor call). The instruction carries out the following actions:
  1. Save the state of the user program (the state consists of program counter and general purpose registers);
  2. Load the state of the kernel from the kernel stack, including the kernel program counter.
  3. Switch the mode bit:  $MR \leftarrow 0$ .

The following SVC types are supported:

- $k = 5 \mapsto$  print the contents of register  $R5$  as an integer.
- $k = 10 \mapsto$  successful completion of user program.

Memory The byte-oriented memory is organized as follows:

- Byte addresses  $X'0000'$  to  $X'0FFF'$  are reserved for the kernel;
- Byte addresses  $X'1000'$  to  $'FFFF'$  are for user programs.

The kernel and each user program operate in their own address spaces. The memory map of an address space consists of three sections, as shown in Figure 1:

- **Text** - the executable code; occupies the first  $X'0100'$  at the low end of the address space.
- **Data** - the data section starts immediately after the text section and grows towards higher addresses. -
- **Stack** - grows from the end of the address space

Phase 1. Write a simulator of the processor and memory and very simple kernel and user code; simulate switching from kernel to user mode and back to kernel mode.

*Hints:* The simulator should at least carry out the following actions:

- The kernel should first print the integer 11 stored in the first word of the kernel data section (10 is a coded message meaning “Kernel started”); then save the kernel state and transfer control to the user code. at address  $X'1000'$ .

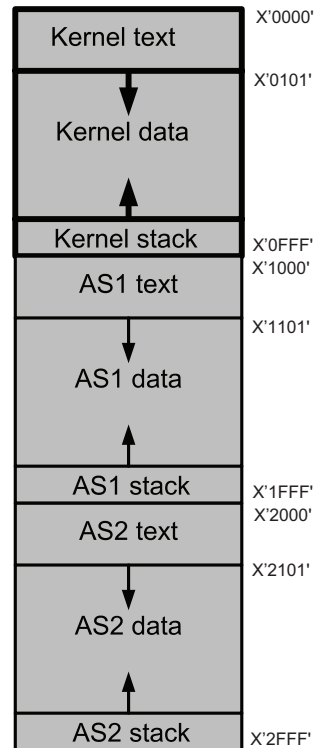


Figure 1: The memory map of a system with a kernel and two address spaces, AS1 and AS2. An address space consists of a “text” section starting at the lower end of the address space, a stack growing from the highest end of the address space, and “variables” section starting at the end of the “text” section and growing towards higher addresses

- The user program should load in register  $R5$  the integer 22 (a coded message meaning “User started” and issues an **SVC 5** requesting the kernel to print the integer. Then, after the kernel returns control to the user code, it should add integers 765 and 2009 and store the result in the second word of its data section and then request the kernel to print the result. Finally, it should issue an **SVC 10** requesting the kernel to print the integer 12 (a coded message meaning “User terminated successfully.”

Assume that:

- The kernel the user code are loaded in memory at addresses  $X'0000'$  and  $X'1000'$ , respectively;

- The kernel is given control during the initialization phase:

$KPC \leftarrow X'0000'$       Load the starting address of the kernel in the program counter  
 $KSP \rightarrow X'0FFF'$       Load the kernel stack pointer

**Phase 2 - Memory Virtualization.** Rewrite your simulator to support a partial implementation of virtual memory. Design an application to test memory virtualization. Read carefully Section 5.4 (pages 242 - 254) in the textbook before starting this phase.

*Hints:* Assume that the page size is 2 K (512 words) and the size of the address space is  $Max_{mem} = 2^{32} - 1$ . The kernel should maintain an AST (Address Space Table) pointing to individual ASCBs (Address Space Control Block). The ASCB describes the context of the address space including the ASID (the Address Space ID) and the address of the page

table. Write two procedures `CreateAddressSpace` and `RemoveAddressSpace` to carry out these functions.

Implement the DAT (Dynamic Address Translation) mechanism illustrated in Figure 2. For simplicity assume that the kernel uses physical addresses, in other words apply the DAT only to addresses starting with  $X'1000'$ . The physical memory is allocated on a FIFO basis in blocks of 2,024 bytes. Test your VM manager with only one address space running the application program in Step1. Extra credit: implement a page replacement algorithm. The procedure `TRANSLATE` on page 250 sketches the functionality of the DAT.

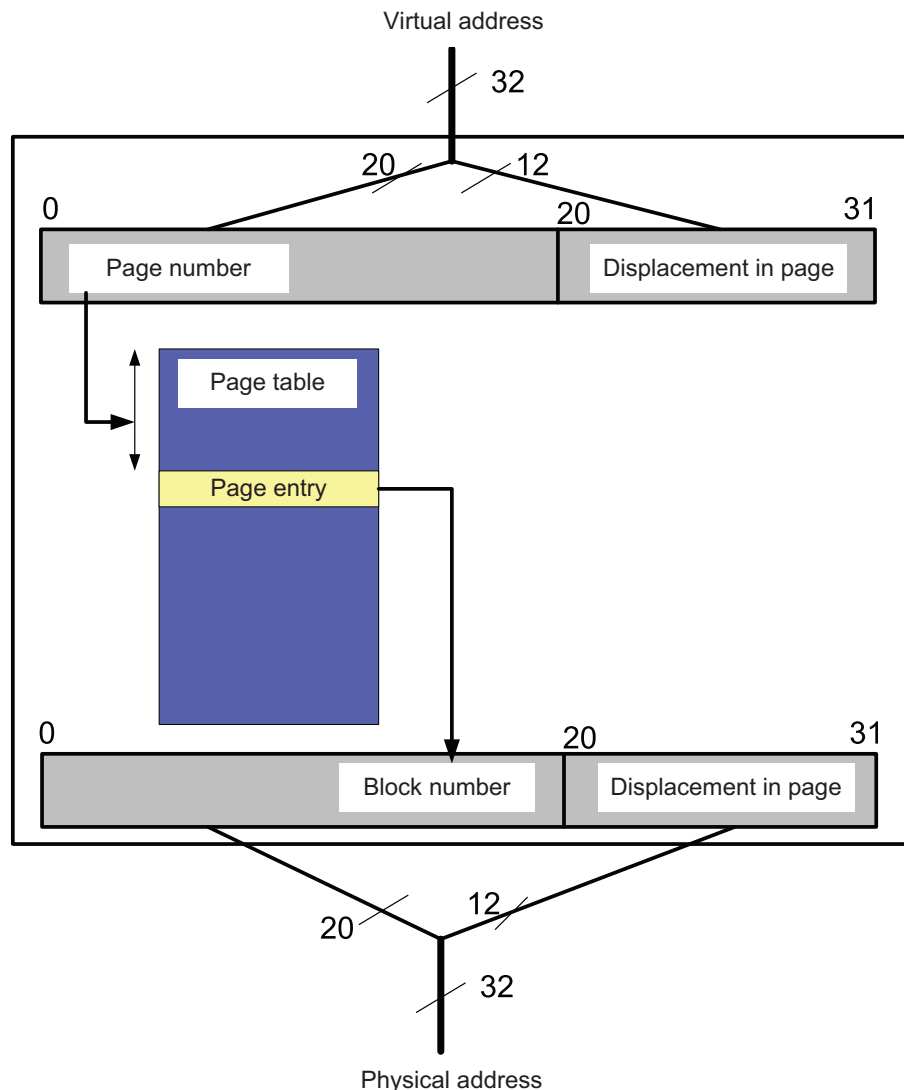


Figure 2: The dynamic address translation.

Phase 3 - Processor virtualization. Rewrite your simulator to support multiple threads. Design an application to test thread scheduling. Implement the FLIH (First Level Interrupt Handler); start with different types of program checks and timer interrupts. Read carefully Section 5.1.1.1 (pages 204 - 206) and the Section 5.5 (pages 255 - 272) in the textbook before starting this phase.

*Hints:* The cause of an interrupt is identified by the hardware and the control is passed to the corresponding interrupt handling routine. For example, when a timer interrupt occurs the timer interrupt handling routine is activated.

Write procedures to create and terminate threads: `AllocateThread`, `Exit Thread()`, and `DestroyThread(threadId)`. When a thread is done with its work it invokes `Exit Thread`. When a thread needs to terminate another thread it invokes `DestroyThread(threadId)`, see discussion on page 264 and Figure 5.25 on page 266. Implement the equivalent of procedure `Yield` discussed in on page 260 and in Figure 5.24.

Phase 4 - Virtualization of the communication channel. Rewrite your simulator and allow threads to communicate using a bounded buffer and *send* and *receive* primitives. Read carefully Section 5.6 (pages 273 -284). For simplicity assume that all your threads share the same address space. Design an application to test thread synchronization. Extra credit: simulate thread synchronization using locks; add a memory bus with an arbiter and implement a Test and Set instruction.

*Hints:* Define the set of states of a thread (see Figure 5.32 on page 279) and the events triggering state transitions. Design the set of primitives to wait and notify and the `send` and `receive` primitives.