CSCI 480/514 Operating Systems Semester Project

Overview

The goal of this project is to write a virtual operating system for an abstract machine (detailed below). It will provide the following services:

- Load and unload programs.
- Allocate and deallocate virtual memory.
- Provide I/O services.
    - These will just be standard Console Input/Output.
- Provide services for scheduling processes.
    - Based on priority.
    - Based on time cycles expiring.
    - Based on resources.
- Provide services for mutual exclusion and synchronization.
    - Semaphores
    - Locks (mutexes)
    - Events

Project requirements

Students will work in teams of 3 or 4.

- Students may use C/C++, Java or C# to write the operating system. Other languages may be used with consent of the instructor. It should be obvious that all students on a team must use the same programming language.
- The OS **MUST** compile without any errors.
    - Any errors in compilation results in 0 grade.
    - The students are responsible for providing any readme or any other documentation to the instructor to evaluate the project.
    - The program must compile on Windows 10 or later.
    - Warnings must be minimized.
    - Code should be well documented and written in a self-documenting fashion.

    - If the operating system code crashes, the grade provided is based on the instructor's subjective decisions.
    - The instructor will evaluate the project using sample files that he will provide later in the class.

- The team is graded as a unit for the project.

    - Any differences between the team members (if any) must be resolved by the members themselves (keep me out of it).

BASE FEATURE SET

The following features are mandatory. These features must be implemented and working for full credit. If some or all the features are not working as specified, the grade is left to the instructor's subjective decision.

- Process scheduling based on priority.

    - This implies that the current running process continues to run until it exits or tries to acquire a resource that is currently held by another process.
    - The functionality for Sleep is mandatory.
    - Process scheduling based on time quantum expiration.

- Provide output services. (Print. This is used to verify the functionality of the program).

- Provide mutual exclusion facilities. Minimally, locks must be implemented.

- An idle process is required. This process continues to run until time elapses for another process to wake up.

- On process exit:

    - All the memory that still belongs to the process MUST be deallocated and MUST be made available for other processes.

    - All internal OS structures used for the process (Process control block etc.) must be released as well.

    - If the process holds locks when it exits, the lock MUST be marked as not acquired (i.e. released)

        - This implies any process waiting for this lock MUST now be made eligible to run.

- These features, if implemented completely, account for 75% of the total grade for the semester.

The following features may be implemented, in addition to the above, for an additional 20% grade. However, it will be entirely up to the instructor's discretion which additional credits will be allowed if the base features are not done.

The final exam is optional. On the last regular day of class, each team must present their operating system and explain their design decisions. This is worth 5% of your grade.

While there will not be assignments due until the end of the semester, there will be monthly check ins with each team.

Example:

Bob 68% on the mandatory project and 20% on the enhanced feature set giving him 88%. If he takes the final, he will most likely get an A for the semester, otherwise a B would be the final grade

Students are strongly advised to get their mandatory features working before even venturing into the extended features. Students must clearly indicate at the time of submission what features they implemented. This feature set is known as the *Extended Feature Set* in this document.

Extended Feature Set

- Implement a dynamic memory allocation features.
    - Alloc and FreeMemory both must be implemented.
- Implement virtual memory using dynamic paging.
    - Each process's memory is now classified as virtual memory.
    - Each process has its own set of page tables that map its virtual memory to physical memory.
    - A process may incur page faults when it tries to access memory it currently does not own or does not have in memory.
    - Page size for the page tables will be taken as an input to the OS command line or via a config file
    - Normal data files may be used to store the unused state of a process.
    - Each process may have its own files for paging or may share files.
    - OS code must service page faults and handle them properly.
    - Page tables can either grow or be restricted to a fixed size.
        - The former is preferable although both are acceptable.

The following are the list of opcodes that are minimally needed for the above features. The instructor will use only these opcodes in his sample programs when testing students' projects.

Project Breakdown

Your assignment is to implement a virtual operating system. The term virtual in this context implies an operating system that will run on top of a standard platform.

- A standard PC with Windows or Linux
- Any standard C/C++/Java/C# compiler may be used
- Since there is no real hardware here, the operating system provides the above services and will drive execution
- In this sense, the operating system is both the system and the hardware, although encapsulation and abstraction of functionality will be expected.
- It interprets the programs by itself (like the Java virtual machine).

Definition of the abstract machine.

We will build an abstract machine for this project (Referred to as MID). There is no real hardware (CPU+ chipset + hardwired logic) needed to implement this. Instead, it will be emulated in software. The machine has the following specifications:

Processor

- 32-bit
- All addresses and registers are 32 bits
- 10 general purpose registers.
    - Each register is 32 bits wide and can store any information.
    - They are addressed as 1 through 10 in the opcodes for this machine.
    - Register 10 is also the stack pointer register.
        - Always points to the top of the stack.
        - The stack grows downward towards lower addresses.
        - Pushing decrements stack pointer. Popping increments it.
        - Only 32-bit quantities may be pushed or popped.

Registers and Flags

- Register access is atomic.
    - This is unlike x86 where a 32-bit register may be addressed by its full 32-bit , 16-bit or 8-bit portions.

- MID has two bit flag registers.
    - SF is the sign flag. This is set when comparison of 2 quantities results in a non-zero result.   X < y can be implemented in terms of x-y >0 and x-y <0. SF is set as a result.
    - ZF is the zero flag. This is set when any comparison of 2 quantities results in a zero (both quantities are equal).
    - These flags may be implemented as Booleans. You don't need to use bit manipulation.
- The instruction pointer register is special and is 32-bits wide.
    - It is not accessible to the programmer.
    - It is modified when the program executes.
    - It must be saved by the operating system on function calls and context switches.
    - Also known as the eip or ip in this document.

Opcodes

Since the intent of this project is to teach Operating system design and implementation, the machine is not complete in the sense that it does not provide a full repertoire of instructions Each opcode is exactly 1 byte long and is of the following format:

Opcode Format

- <opcode> <argument 1> <argument 2>   ; comment
    - Example: 2 r1 $1 where 2 is the opcode for add.
- All instructions are multiple bytes.
- All instructions take exactly one clock cycle to execute, independent of the actual operation of the instruction.
- Context-switches only happen on instruction boundaries.
- Constants are indicated by the operator $ in front of them.
- Registers are indicated by r1 … r10.
- The stack pointer (register 10) is often called SP

Mid will support the following opcodes.

Opcode	Value(integer)	Meaning
Incr	1	incr r1(increment value of register 1 by 1).
Addi	2	addi r1,$1 is the same as incr r1
Addr	3	Addr r1, r2(r1 <= r1 + r2).
Pushr	4	Pushr rx (pushes contents of register x onto stack. Decrements sp by 4).
Pushi	5	Pushi $x . pushes the constant x onto stack. Sp is decremented by 4 after push.
Movi	6	Movi rx, $y; rx ⇐ y
Movr	7	Movr rx, ry ; rx ⇐ ry
Movmr	8	Movmr rx, ry ; rx ⇐ [ry]
Movrm	9	Movrm rx,ry; [rx] ⇐ ry
Movmm	10	Movmm rx, ry [rx] ⇐ [ry]
Printr	11	Printr r1 ; displays contents of register 1
Printm	12	Printm r1; displays contents of memory whose address is in register 1.
Jmp	13	Jmp r1; control transfers to the instruction whose address is r1 bytes relative to the current instruction. R1 may be negative.
Cmpi	14	Cmpi rx, $y; subtract y from register rx. If rx < y, set sign flag. If rx > y, clear sign flag. If rx == y , set zero flag.
Cmpr	15	The same as cmpi except now both operands are registers.
Jlt	16	Jlt rx; if the sign flag is set, jump to the instruction whose offset is rx bytes from the current instruction.
Jgt	17	Jgt rx; if the sign flag is clear, jump to the instruction whose offset is rx bytes from the current instruction
Je	18	Je rx; if the zero flag is clear, jump to the instruction whose offset is rx bytes from the current instruction.
Call	19	Call r1; call the procedure at offset r1 bytes from the current instruction. The address of the next instruction to execute after a return is pushed on the stack.
Callm	20	Callm r1; call the procedure at offset [r1] bytes from the current instruction. The address of the next instruction to execute after a return is pushed on the stack.

Opcode	Value(integer)	Meaning
Ret	21	Pop the return address from the stack and transfer control to this instruction.
Alloc	22	Alloc r1, r2; allocate memory of size equal to r1 bytes and return the address of the new memory in r2. If failed, r2 is cleared to 0.
AcquireLock	23	AcquireLock r1; Acquire the operating system lock whose # is provided in the register r1. If the lock is invalid, the instruction is a no-op.
ReleaseLock	24	Releaselock r1; release the operating system lock whose # is provided in the register r1; if the lock is not held by the current process, the instruction is a no-op.
Sleep	25	Sleep r1; Sleep the # of clock cycles as indicated in r1. Another process or the idle process must be scheduled at this point. If the time to sleep is 0, the process sleeps infinitely.
SetPriority	26	SetPriority r1; Set the priority of the current process to the value in register r1; See priorities discussion in Operating system design
Exit	27	Exit. This opcode is executed by a process to exit and be unloaded. Another process or the idle process must now be scheduled.
FreeMemory	28	FreeMemory r1; Free the memory allocated whose address is in r1.
MapSharedMem	29	MapSharedMem r1, r2; Map the shared memory region identified by r1 and return the start address in r2.
SignalEvent	30	SignalEvent r1; Signal the event indicated by the value in register r1.
WaitEvent	31	WaitEvent r1; Wait for the event in register r1 to be triggered. This results in context-switches happening.
Input	32	Input r1; read the next 32-bit value into register r1.
MemoryClear	33	MemoryClear r1, r2; set the bytes starting at address r1 of length r2 bytes to zero.
TerminateProcess	34	TerminateProcess r1; terminate the process whose id is in the register r1.
Popr	35	pop the contents at the top of the stack into register rx which is the operand. Stack pointer is decremented by 4.

Opcode	Value(integer)	Meaning
popm	36	pop the contents at the top of the stack into the memory operand whose address is in the register which is the operand. Stack pointer is decremented by 4.

Operating system design goals

MidOS must provide the following services to programs written for this operating system:

- Provide virtual memory services.
    - All processes will have page tables allocated by the operating system.
    - OS will detect invalid memory accesses and terminate the processes.
- Terminating processes means the operating system will display an error message on the console and choose the next process to execute.
- A processes can allocate any amount of memory subject to the resources available.
- Processes only manipulate virtual addresses.
- Provide inter-process synchronization mechanisms.
    - MidOS provides mutexes for a process to lock out other processes while manipulating critical shared structures.
    - MidOS comes with (or is preconfigured with) 10 locks, each identified by the numbers 1 through 10.
- The OS does not provide any services for processes to allocate their own critical sections.
    - The instructions AcquireLock and ReleaseLock allow a process to acquire and release the OS provided locks.
    - The locks provided by the OS are meant for use by application processes to protect themselves while manipulating shared-critical structures.

- Provide scheduling services.
    - Each process is scheduled independently.
    - Processes are single threaded.
    - Only one process is running at a time.
    - Once a process is scheduled (starts running) it continues to run until the following happens.
        - It exits (by executing opcode Exit).

    - It sleeps for some time (by executing opcode Sleep).
    - It attempts to acquire a lock already acquired by someone else.
    - It accesses an address which requires a page fault and there is not enough memory available. Once another process frees up some memory, this process is scheduled again.
    - Waits on an event to be signaled by another process.
    - Its time quantum runs out.
        - Each process has a time quantum. A time quantum is the # of clock cycles it is allowed to run before it is scheduled out.
- Provide memory-mapping services.
    - This OS provides built-in or preconfigured shared memory regions of size 1000 bytes each for a total of 10 memory regions, each indexed by the number 1 thru 10 respectively.
    - Processes may map any of them into their address spaces by using the opcode MapSharedMem and passing the # of the shared memory region.
    - Processes may use this facility to share memory and perform inter-process communication.
- Provide I/O services.
    - Processes may use these I/O services to print out values to the console.
    - Processes may use input services to read values from the console.

Project features

- The OS must implement a hardware simulator that understands the above opcodes.
- The OS must be invoked as follows.
    - OS <size of virtual memory in bytes> <program1.txt> <program2.txt> .....
    - Each program file contains code to be loaded and executed.
    - The virtual memory size is the total size of memory available for all programs that can be loaded (including code, data, heap and stack
    - This size does not include the number of bytes the OS will use internally to keep track of information about each process such as process blocks.
- The OS will provide the following services.

- Loader to load programs of the above type.
    - The students may assume that the instructor will provide sample programs with correct code.
- Scheduler for scheduling programs.
    - Maintain process context (using process context blocks or otherwise).
    - The time quantum a process is limited to running before it is swapped out is 10 clock cycles.
    - All process context (including all registers, page tables, etc.) MUST be stored in the process control block (PCB).
    - An idle process must be provided that runs if no other process is eligible to run.
        - The idle process will be provided in a separate file called idle.txt.
        - The idle process just sits in a tight loop printing out the value 20.
        - The idle process only has a quantum of 5 clock cycles.
        - The idle process never exits.
        - The idle process always has the lowest priority
        - The scheduler always schedules the highest-priority process eligible to run.
        - There are 32 priorities in the OS.
        - Processes with higher priorities run until they give up control, exit or are terminated.
        - The bigger the priority number, the higher the priority for the process.
        - A process may adjust its own priority by invoking the opcode SetPriority.
        - In addition to these queues, the scheduler MUST implement queues for blocking & other types of processes.
- Virtual memory manager.
    - Each process MUST only access virtual addresses.

    - When a process is loaded, the memory manager MUST construct its page tables in the appropriate fashion.
        - Every memory access by each process is done through page tables to accesses the real memory.
        - This is true for all addresses (code, data, stack and heap).
        - The details of breaking down a virtual address to a physical address using page tables will be provided in the class.
        - If a process accesses an address it does not own, the OS must print an error message with the values of the registers of the process at the time the problem occurred, terminate the process and schedule another one.
- Process shared memory.
    - A process may map the OS provided shared memory region into its own address space using the MapSharedMem opcode.
    - This allows 2 or more processes to share memory.
    - The exact usage of these memory regions is left to the processes themselves.
- Process virtual memory.
    - A process's memory map is divided into 4 regions.
        - Code. This is the region containing the instructions with some combination of the above opcodes.
        - Stack. This is the region where the program may store temporary variables. When a function is called, the interpreter stores the return address here as well.
            - When a program is loaded, it is provided a stack of 4 bytes.
            - The stack grows and shrinks depending on usage. If it collides with any of the other sections, you have a stack overflow error.
        - Global data.
            - When a program is loaded, it is provided a memory region called global data of size 512 bytes. The contents are initialized to 0. Please see "process initial state" for more information.

        - Heap.
            - Heap is the portion of the address space that the process can allocate dynamically using the Alloc and FreeMemory opcodes. This should be 512 bytes
- Process initial state.
    - A program becomes a process when it is loaded by the operating system.
    - When a process is created, its registers have the following state, configured by the operating system:
        - r1 – r7 are undefined.
        - Register r8 contains the id of the process.
        - r9 contains the virtual address of the global data region (of length 1024 bytes).
        - r10 (sp) is set to the top of the stack.
            - Growing the stack implies it grows towards lower addresses.
        - The instruction pointer (ip) contains the value 0. It translates (via page tables) to the first instruction in the program to be executed.
- Process exiting.
    - A process may exit due to any of the following reasons.
        - It invokes the Exit opcode.
        - It performs an illegal operation(it access a memory location it does not own).
        - Another process kills this process using the TerminateProcess opcode.
            - It is left to the individual processes to find the id of the other processes.
            - Admittedly, the security is very weak (allowing any process to terminate any other process).
    - When a process exits for whatever reason, your OS must perform the following.

        - Display:
            - # of page faults
            - # of context switches
            - # of clock cycles it consumed.
- Process synchronization.
    - The OS provides 10 mutexes or locks.
    - The locks are numbered through 1 to 10.
    - A process acquires the lock using AcquireLock.
        - If another process already owns the lock, the current process is put to sleep on the wait queue.
        - If a process X tries to acquire the same lock Y two or more times, it is treated as a no-op operation. In short, the process MUST not deadlock against itself.
    - A processes releases a lock it is currently holding via ReleaseLock.
        - This MUST make only one process(the highest priority process currently waiting) eligible to be scheduled.
        - Other processes also blocking for this lock will not be scheduled. Only the highest priority process will be scheduled (if any).
        - The process releasing the lock might be swapped out because an eligible process may have a higher priority than the one releasing the lock.
    - The OS must take into account priority inversion.
        - It is possible that a higher-priority process is blocked for a resource that is currently owned by a lower-priority process.
        - In this case, the OS must bump up the priority of the lower priority process to that of the blocked process and keep it this way until the lower-priority process releases the resources.
    - If a process holding a lock exits or is terminated for whatever reason,
        - The lock MUST be freed and another waiting process (if any) should be made eligible for scheduling.
- Events.

- Events, unlike locks, are provided for process synchronization.
- They allow one process to notify another process the occurrence of an event.
    - The OS provides built-in events for a total of 10. They are numbered 1 to 10.
    - Any event is in one of 2 states.
        - Signaled. This implies any process waiting on it is eligible to run now.
            - An event is set to signaled when a process invokes the SignalEvent opcode.
        - Non-signaled.
            - This is the initial state of all events when the OS boots up (so to speak).
            - When a process that is waiting on it becomes eligible to run, the event becomes non-signaled allowing another process to wait on it.
            - There is no concept of ownership of an event

Helpful Tips and Advice

1. Make sure everyone on the team is comfortable with the language and development environment you choose.
2. It MUST run on a pc. Keep this in mind.
3. Don't procrastinate. If you don't start this project immediately, the chances of it getting done are minimal.

Some Hints for objects

Here are descriptions of a few objects you MAY need as part of the software, depending on your implementation approach. You are not required to, but you might want to consider using them.

- *CPU Object* – responsible for **physical memory** (which may be smaller than addressable memory) holding CPU registers and fetching program opcodes and executing them. A related object to consider is:
  - Memory Management Unit object responsible for translations between Addressable (Virtual) memory, Process memory and Physical Memory. All Processes access memory from *their* point of view…they never talk to "physical memory" (Which is just an array of bytes in the CPU object)
  - Be sure to organize the implementation of the opcodes well or you will have a mess.
- *OS Object* – The heart of the OS. It will probably have related objects:
  - The scheduler
  - Locks
  - Events,
- *Program Object* - represents a collection of instructions. Responsible for parsing the files off disk and holding them.
- *Process Object* – different than the Program object. It's an actual running process in the MidOS. There can be more than one instance of a Process for each Program. (You can run the same program twice, etc.).
  - Each Process has a Process Control Block (PCB) containing Process specific registers, etc.

- *EntryPoint Class* – Contains entry point and "bootstraps" the whole OS

What does a Program look like?

Here's an example program, specifically the "idle" loop. The idle loop is a process that never ends, it just prints out "20" over and over. Use it to keep the clock running when all the other processes are sleeping.

```
6        r4, $0                ;move 0 into r4
26       r4                    ;lower our priority TO 0
6        r1, $20               ;move 20 into r1
11       r1                    ;print the number 20
6        r2, $-19              ;back up the ip 19 bytes
13       r2                    ;loop forever (jump back 19)
```

Programs consist of:

```
Opcode, [optional param], [other optional param] ;optionalcomment
```

So, if we look at:

```
6        r4, $0                ; move 0 into r4
```

We see that it's operation 6, which is Movi or "Move Immediate." It moves a constant value into one of our 10 registers. The first param is r4, indicating register 4. The second param is $0. The "$" indicates a constant. So we are loading the value 0 into register #4. Just like x86 assembly – only not at all.

So if you look at the comments for this app you can see that it loops forever.

How do run Program(s)

Usage:

```
OS membytes [files]
```

For example:

```
OS 1568 prog1.txt prog2.txt prog3.txt prog1.txt idle.txt
```

This command line would run the OS with 1568 bytes of virtual (addressable) memory and start up 5 processes. Note that Prog1.txt is specified twice. That means it will have two separate and independent running processes. We also specified the forever running idle.txt. Use CTRL-C to break. The OS has operations for shared memory, locks, and events, so you can setup rudimentary inter-module communication.

External Configuration Files

You might want to store configurable items in a separate file so you don't have to recompile the OS when changing parameters:

Here's a sample OS config file represented as json.

```
“options”: {

          "PhysicalMemory":"384",
          "ProcessMemory":"384",
          "DumpPhysicalMemory":"true",
          "DumpInstruction":"true",
          "DumpRegisters":"true",
          "DumpProgram":"true",
          "DumpContextSwitch":"true",
          "PauseOnExit":"false",
          "SharedMemoryRegionSize":"16",
          "NumOfSharedMemoryRegions":"4",
          "MemoryPageSize":"16",
          "StackSize":"16",
          "DataSize":"16",
    }
```