MidOS Semester Project

Overview

This project is meant for a first course in operating systems design.

Operating Systems classes often end up like survey courses of algorithms. Most assignments, such as building schedulers, don't give the student the feel of building an operating system.

Building an entire operating system from scratch isn't an option in a single semester class. Instead, this project is a simplified operating system running on a virtual machine that the student also builds. There are many differences between this project and an actual operating system. Still, it will give the student the feel of implementing a process scheduler, virtual memory, memory management, paging, and more.

The project is organized as a series of modules, which involves the student building different parts of a virtual machine and an operating system. Each module builds on the last. Each module contains ideas for implementation, allowing students to choose multiple approaches.

When implementing this project, I allow students to choose the programming language they wish to use but don't offer support for the language itself.

Finally, I am always asked about the project's name, MidOS. It's very simple. This operating system is rudimentary at best. Hence the name, MidOS, because it's mid af.

Happy programming.

Project Overview

This project aims 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 2

- Students may use C/C++, Java or C# to write the operating system. Other languages may be used with the 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 result in a 0 grade.
    - The students are responsible for providing the instructor with any documentation needed 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.

    - The members themselves must resolve issues or conflicts between team members (keep me out of it).
- DO NOT CHEAT! I cannot stress this enough. The last time this class ran, five students were caught cheating. They all failed.

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 until it exits or tries to acquire a resource 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. Input is used to receive input).
- 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 made available for other processes.
    - All internal OS structures used for the process (Process control block etc.) must also be released.
    - 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.
- Implement 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 a 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 input from the command line or a config file.

    - Normal data files may be used to store the unused state of a process.
    - Each process may have its distinct files for paging or may share files.
    - OS code must service page faults and handle them properly.

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 process 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 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 that 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.
        - It's 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 the 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 – r10 are undefined.
        - Register r11 contains the id of the process.
        - R12 contains the virtual address of the global data region (of length 1024 bytes).
        - r13 (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 accesses 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 from 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 process 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 of the occurrence of an event.
        - The OS provides up to ten built-in events, 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