CIS 415 - Operating Systems
Assignment #1 – Answers
Fall 2015 - Prof. Malony

OSPP Questions – Chapter 1

Exercise 4. Suppose a computer system and all of its applications were completely bug free. Suppose further that everyone in the world were completely honest and trustworthy. In other words, we need not consider fault isolation.

a) How should an operating system allocate time on the processor? Should it give the entire processor to each application until it no longer needs it? If there were multiple tasks ready to go at the same time, should it schedule first the task with the least amount of work to do or the one with the most? Justify your answer.

Answer. The operating system should allocate time on the processor by using a scheduling scheme which allows for time-sharing among processes. The application of the machine decides whether its best to prioritize tasks or to maximize fairness among them. All of the processors resources should not be given to a single application until it finishes because if there is an lengthy application, no other work will get done until the application terminates and other processes would be starved. Furthermore, if the application performs significant I/O then the CPU will be idle at times when it could be doing work. While context switches are not free we still dont want to run applications uninterrupted. One solution is to use a queue and have each app running for a finite time, then repeatedly reload it into queue until its completed. If there are multiple tasks ready to run at the same time the scheduled task should be the one with the least amount of work because then throughput is increased and starvation and wait time are decreased.

b) How should the operating system allocate physical memory to applications? What should happen if the set of applications does not fit in memory at the same time?

Answer. The operating system should allocate physical memory to applications by dividing the space between all running applications. If the set of applications does not fit in memory at the same time then applications with high priority should be kept in memory and the unused applications should be moved to disk using virtualization.

c) How should the operating system allocate its disk space? Should the first user to ask acquire all of the free space? What would the likely outcome be for that policy?

Answer. The operating system should allocate its disk space by dividing it equally among users, where each users space stores their own applications. The first user to ask should not acquire all the free disk space. The result of this would be that the other users would be unable to acquire any additional disk space.

Exercise 5. Now suppose the computer system needs to support fault isolation. What hardware and/or operating support do you think would be needed to do the following?

a) Protect an applications data structures in memory from being corrupted by other applications.

Answer. The hardware and/or OS support that would be needed in a system that supports fault isolation in order to protect an applications data structures in memory from being corrupted by other applications would be a security policy that restricts access to the applications data structures in memory from the user. Applications should only have access to the minimum resources required for them to perform their task. There should also be support for bounds checking to circumvent buffer overflow. Additionally, supporting sandbox testing could be used to determine side-effects of a particular application without damaging existing data structures.

b) Protecting one users disk files from being accessed or corrupted by another user.

Answer. To protect one users disk files from being accessed or corrupted by another user would be to establish privileges for each user. Each user can then control permissions for reading and writing their own files. Passwords and file encryption are extra ways to control access of disk files that the OS should support.

c) Protecting the network from a virus trying to use your computer to send spam.

Answer. To protect the network from a virus trying to use a computer to send spam would be to prevent the spread of a virus through the network by restricting particular ports and addresses. Using a network firewall along with scanning ports and incoming files is also useful to detect the virus itself or at least its source.

Exercise 7. How would you design combined hardware and software support to provide the illusion of a nearly infinite virtual memory on a limited amount of physical memory?

Answer. One way to design combined hardware and software support to provide the illusion of a nearly infinite virtual memory on a limited amount of physical memory is to do something similar to demand paging of virtual addresses. This entails the address translation (via hardware) of processes virtual addresses to physical addresses of physical memory. During task execution, if the referenced address is not found in memory (by checking valid-invalid bit in page table) the hardware triggers a page fault. The kernel then loads the demanded page frame from disk into memory. Therefore recently active processes reside in memory and idle programs on the disk. The virtual memory enables efficient multitasking with limited physical memory. Credit based on clear discussion of one or more among: paging, address translation, concept that frequently used processes should reside on memory and less frequent ones on disk.

OSPP Questions – Chapter 2

Exercise 1. When a user process is interrupted or causes a processor exception, the x86 hardware switches the stack pointer to a kernel stack, before saving the current process state. Explain why.

Answer. It is necessary to save the program state before an interrupt handler is called because the interrupt handler may change the register contents. Additionally, if the stack pointer is stored on the process's stack in user mode instead of a a kernel stack, other user-mode programs can modify its contents if the process has multiple threads of execution or shared memory.

Exercise 7. Most hardware architectures provide an instruction to return from an interrupt, such as iret. This instruction switches the mode of operation from kernel-mode to user-mode.

a) Explain where in the operating system this instruction would be used.

Answer. The operating system can call iret from kernel mode to give control back to a user program after being interrupted by an exception, external interrupt, or software-generated interrupt.

b) Explain what happens if an application program executes this instruction.

Answer. Because iret is a priveleged command normally run in kernel mode, exectution of the iret instruction while in user-mode will cause an exception[1].

Exercise 10. Which of the following components is responsible for loading value in the program counter for an application program before it starts running: the compiler, the linker, the kernel, or the boot ROM?

Answer. The kernel is responsible for setting the initial PC for an application program. Because the kernel controls where in memory the program is loaded, it is the only component that knows which address the PC should point to initially.

Exercise 12. System calls vs. procedure calls: How much more expensive is a system call than a procedure call? Write a simple test program to compare the cost of a simple procedure call to a simple system call (getpid() is a good candidate on UNIX; see the man page). To prevent the optimizing compiler from optimizing out your procedure calls, do not compile with optimization on. You should use a system call such as the UNIX gettimeofday() for time measurements. Design your code so the measurement overhead is negligible. Also, be aware that timer values in some systems have limited resolution (e.g., millisecond resolution). Explain the difference (if any) between the time required by your simple procedure call and simple system call by discussing what work each call must do.

Answer. The system call should take slightly longer than a procedure. This is because more work need to be done to execute a system call, including an interrupt and a context switch both to and from the system. However, modern CPUs are highly optimized to perform system calls, so the difference is not usually significant.

```
// Sample test program
#include <sys/time.h>
#include <stdio.h>
#include <unistd.h>

int testfunc () {
  return 10;
}

int main(int argc, char *argv[]) {
  int iterations = 1000000;
  struct timeval start, stop;
```

[1] iret can be executed from Virtual 8086 Mode, but not in the context of this question

```
  // Run a system call 1000000 times
  gettimeofday(&start, NULL);
  for (int i = 0; i < iterations; ++i) {
    getpid();
  }
  gettimeofday(&stop, NULL);
  printf("System call time elapsed: %d microseconds\n", stop.tv_usec - start.tv_use
  // Run a procedure 1000000 times
  gettimeofday(&start, NULL);
  for (int i = 0; i < iterations; ++i) {
    testfunc ();
  }
  gettimeofday(&stop, NULL);
  printf("Procedure call time elapsed: %d microseconds\n", stop.tv_usec - start.tv_
}
```

Exercise 15. Explain the steps that an operating system goes through when the CPU receives an interrupt.

Answer. The following steps are performed:

1. *Mask Interrupts.* The hardware starts by preventing any interrupts from occurring while the processor is in the middle of switching from user mode to kernel mode.

2. *Save three key values.* The hardware saves the values of the stack pointer, the execution flags, and the instruction pointer to internal temporary hardware registers.

3. *Switch onto the kernel interrupt stack.* The hardware then switches the stack segment/stack pointer to the base of the kernel interrupt stack, as specified in a special hardware register.

4. *Push the three key values onto the new stack.* Next, the hardware stores the internally saved values onto the stack.

5. *Optionally save an error code.* Certain types of exceptions, such as page faults, generate an error code to provide more information about the event; for these exceptions, the hardware pushes this code, making it the top item on the stack. For other types of events, the software interrupt handler pushes a dummy value onto the stack so that the stack format is identical in both cases.

6. *Invoke the interrupt handler.* Finally, the hardware changes the code segment/program counter to the address of the interrupt handler procedure. A special register in the processor contains the location of the interrupt vector table in kernel memory. The kernel can only modify this register. The type of interrupt is mapped to an index in this array, and the code/program counter is set to the value at this index.

In handling the interrupt, there are operations that require atomicity (such as saving the CPU state) that call for interrupts being masked while these operations are performed. To avoid losing interrupts and for timeliness, many operating systems split the interrupt handler into two parts: a smaller top half that quickly performs atomic operations with interrupts disabled and a larger, reentrant bottom half for which interrupts can be enabled.

The top half is executed immediately and without interruption. The bottom half is treated more like a process and is enqueued in the scheduler. With that done, the operating system can resume normal operations, likely going to the scheduler. Eventually, the interrupted process will be restored, a procedure that always finishes with iret and a return to user mode.

Supplemental

Q1: Why is the separation of mechanism and policy important and desirable?

Answer. Mechanism and policy must be separate to ensure that systems are easy to modify. No two system installations are the same, so each installation may want to tune the operating system to suit its needs. With mechanism and policy separate, the policy may be changed at will while the mechanism stays unchanged. This arrangement provides a more flexible system.

Another way to say this is that proper separation of mechanism and policy allow the same set of mechanisms to be implemented in several different policies. Also, policies can be modified without reimplementing the underlying mechanisms. Conversely, this separation can allow the same policy to be enforced on systems with different mechanisms without major modification to the policy.

Q2: Describe the actions taken by a kernel to context-switch between processes.

Answer. The kernel must save the current context of the process (registers, process state, PCB) that is running, decide which process should be run next, and load the saved context (PCB)of that process into the CPU to run.

Q3: In what circumstances is the system-call sequence *fork(); exec()* most appropriate? When is *vfork()* preferable?

Answer. *fork(); exec()* is most appropriate when a situation needs the forked process to reuse any data or information in the PCB of its parent.

vfork(); exec() is preferable when the child needs no information from the parent process. The intent of vfork is to eliminate the overhead of copying the whole process image if you only want to do an *exec()* in the child. Because *exec()* replaces the whole image of the child process, there is no point in copying the image of the parent (hence *vfork()*).

Q4: Suppose that you are developing a new computer and OS with significant resource constraints. The hardware is only allowed to support either clock interrupts or I/O interrupts, but not both.

(a) Discuss how your system would support multiprogramming without clock interrupts, while preventing any process from monopolizing the CPU or going into an infinite loop. Your solution should support the ability to time share amongst processes by switching quickly between them. In this case, all processes are able to perform I/O.

Answer. The system must become completely I/O (or user) driven. One program will run (monopolizing the cpu) indefinitely until the user provides a hardware or software interrupt. In the meantime, processes that are being scheduled to run will need to remain in a queue. Once an I/O interrupt is received, the CPU

will stop the current process, do any work that it needs to do on the processes within the queue, and allow a new process to monopolize it.

(b) Do the same but for a system without I/O interrupts. In this case, you want to support multiprogramming as before, but also I/O for every process.

Answer. Here the idea is to create a queue to buffer the I/O actions. The clock interrupt is used to create an I/O polling operation every certain time period, say once every 100 ms. After each poll, any I/O operations that need servicing take place. Then the currently running process is interrupted and the next process in the queue is run.

(c) Argue which you would choose with the goal of providing the best user performance for a low price. What makes your solution more economical than the alternative? Are there circumstance under which your OS performs well versus the other approach? Under what conditions would it respond poorly? Note that your design is for a uniprocessor system and that user processes and the OS must take turns with the one CPU.

Answer. The case with no clock interrupts is the harder case, in the sense that it reduces to a I/O-driven system. However, in environments where this is the nature state of affairs, the machine can be very efficient. The prime example of this system is an ATM or vending machine. In the latter example, the machine loops the ”Make selection” print process until the user provides an I/O interrupt, which in this case would be making a selection. Once that input has been received, the machine is then interrupted and free to prompt the user with another message, such as, ”Enter amount: $1.00”. This will then loop forever until the user enters a dollar.

On the other hand, having a clock interrupt provides a periodic and regular way for the OS to gain control. I/O hardware actions can be service by polling the devices to see what state has changed. In environments where there are infrequent I/O events, this is fine. It allows for better sharing of the CPU among processes that do not require fast response or high frequency I/O operations.