**Problem Set 1 CS543 Mangesh Raut(mbr63)**

1. Why are special instructions used to implement system calls? Why not use normal subroutine calling instructions?

Ans: System calls are implemented using special instructions that allow the operating system kernel to distinguish between a normal function call and a request for a privileged service, such as the "syscall" instruction in the x86 architecture. This distinction would not be provided by using a regular subroutine call instruction, which would make it more challenging for the kernel to manage and secure system calls. Special instructions might also offer extra functionality, including facilitating easy access by the kernel to the system call number and arguments supplied by the user-level process.

In contrast to regular function calls, system calls do not often require any extra knowledge from the compiler. Since the libraries we link to have simple, tiny functions for each of the system calls, we can get away with it. These tiny stub functions typically only set up the system call arguments in a clear manner before handing control over to the operating system.

A software interrupt, also known as a trap command, is frequently used to transfer control. These unique instructions cause the CPU to record the machine's present status and hand off control to an operating system feature. They look like a cross between a hardware interrupt and a regular subroutine call instruction. The operating system typically stops the requesting procedure after accepting the transfer of control. When the request can be fulfilled, the requesting process is resumed, and the results of the system call are returned in a certain manner.

2. Is it possible that we could fail to open all three initial file descriptors in emuinit ( ) but not print the error message? How?

Ans: e⃗pgrp⃗dot = cclone(e⃗pgrp⃗slash);

poperror ( );

strcpy (up⃗ text , "main");

These few lines open the console for three file descriptors. As in POSIX environments, processes in Inferno generally start life with open file descriptors for the standard input, standard output, and standard error files. For our initial process, all three of these refer to the console.

The emuinit() function may fail to open the initial file descriptors for the standard input, standard output, and standard error files without printing an error message. This might occur if the emuinit() function's error handling code is constructed so that if the file descriptors fail to open, no error messages or information is output. Instead of displaying the error messages in the terminal, another alternative is to direct them to another file descriptor or log file.

The "poperror()" method, as seen in the given code, can be used to silence any error messages if the error handling code in the emuinit() function supports it. Furthermore, if the code redirects standard error output to a different file descriptor or log file, the error message is not displayed in the console.

As in emuinit ( ), we might generate errors in some of the functions we call, so we need a place to fall back. In this case, if we do get an error, then we call panic(), which causes the OS to halt.

if (waserror())

panic ("disinit␣error:␣%r");

For Linux systems, file descriptors can have values between 0 and 1023. (32-bit or 64-bit system). A file descriptor cannot be created with a value greater than 1023. If the file descriptor is 1024, it will return an EBADF error (bad file descriptor, error no-9).

An error has occurred when a file descriptor with a negative value is returned.

3. Sketch the pseudocode for an implementation of semaphores using a test-and-set instruction.

Ans: One of the most often studied and implemented models of mutual exclusion for user processes is the semaphore. Semaphores are best understood in terms of the opera- tions that are performed on them. We define the semaphore by the operations up and down. Originally, these operations were called V and P, respectively. (Because they are mnemonically meaningful in Dutch, but not English, we will use up and down.) As their names imply, up ( ) increments the semaphore and down ( ) decrements it. We define them more formally as follows:

• up() — Increase the value of the semaphore by one.

• down ( ) — If the semaphore is 0, block until it becomes > 0. Decrease the value of

the semaphore by one.

As an example, we show how we can implement a semaphore using a monitor. In doing so, we prove a theorem, which can be stated informally as, “Monitors are at least as powerful as semaphores in the sense that they can be used for all applications of semaphores.” Thus, in any application where we use semaphores, we can use our monitor- based implementation with the same functionality.

To suggest the ideas behind a proof, we present this implementation using a C-like syntax:

monitor semaphore { unsigned int sem;

void up(void) { ++sem;

signal;

}void down(void)

{ while (sem ≡ 0) wait;

−−sem; }}

Note that C itself does not provide a monitor mechanism, so this code is not valid C. It should be clear that this implementation works given that the monitor provides the necessary atomicity in the implementation of up() and down().

Semaphore sem

sem.value = 1

up(Semaphore sem):

sem.value = 1

down(Semaphore sem):

while(test\_and\_set(sem.value) == 0):

continue

sem.value = 0

With its initial value set to 1, the semaphore is formed.

The semaphore value is set to 1, which denotes the availability of the resource, by the up() method.

To examine the value of the semaphore, the down() operation employs a test-and-set instruction. If the value is 0, the operation decrements the semaphore and indicates that the resource is now being used if the value is 1, signaling that it is currently available. As long as the semaphore is set to 0, the operation will halt the process.

boolean TestAndset (boolean \*target) {

boolean rv = \*target;

\*target = TRUE;

return rv;

?

Figure 5.3 The definition of the TestAndSet () instruction.

do while (TestAndSetLock(&lock))

; // do nothing

// critical section

lock = FALSE;

// remainder section

while (TRUE);

Figure 5.4 Mutual-exclusion implementation with Test AndSet).

4. When dealing with detected deadlock, we kill one of the processes in the cyclic wait. Why can we not just detect that granting a request will create deadlock and block the requesting process?

Ans: The definition of circular wait gives us one approach to detecting deadlock. Periodically, we can build a dependency graph and test whether it is cyclic or acyclic. If it’s acyclic, then there is no deadlock and we continue. If there is a cycle in the graph, then we have deadlock and we must deal with it. Since we can’t do anything about the other conditions, we have no choice but to break the cycle which means we have to terminate one of the processes in it, releasing all of the resources that process has locked.

When a process is waiting for access to a resource and another one is waiting on it, it can be hard to tell which one is which. For systems like this, we can make a good guess regarding when deadlock has occurred by checking periodically how long each process has been blocked. If two threads are stuck in a deadlock, we must terminate one of them to allow the other to continue. One can respond by either killing the thread and releasing all its resources, or by forcing it to roll back. The latter strategy is usually the preferred method for dealing with deadlock.

If you are writing code that is expected to be run with deadlock detection, it needs to be able to handle thread death. A simple practical solution to detecting deadlock is to simply put a time limit on the acquisition of resources. This check can be done periodically or when a new resource is requested.

In order to detect deadlock, we can use the following algorithm:

Make a copy of the resource allocation graph

While there is a process or resource with no outgoing edge:

erase it (this is like running it to completion)

and all incoming edges to it (since it relinquishes all its resources when it finishes)

If you can remove all processes, there is no deadlock: you can run all processes to completion

The system is in a stalemate if you can’t because there must be a cycle.

Instead of ending one of the processes in the cyclic wait, it is feasible to predict when granting a request will result in a deadlock and block the requesting process, although this method has significant disadvantages. This strategy has several drawbacks, one of which is that it can be challenging to foresee when a deadlock will develop and to decide which process must be halted. To determine whether a stalemate would occur, the system would need to continuously assess the condition of all processes and resources, which can be costly computationally and time-consuming. Additionally, it may cause the system to repeatedly block and unblock programs, which could harm system performance and result in inefficient resource use.

Deadlock prevention is a different strategy that is centered on making sure that at least one of the four prerequisites for deadlock will never materialize. For instance, when resources are requested in a particular order using a technique called resource ordering, deadlock is avoided. Another approach is to use a technique called deadlock prevention, which is based on ensuring that at least one of the four conditions for deadlock will never occur. For example, by using a technique called resource ordering, resources are requested in a specific order, and this prevents deadlock from occurring.

A simpler method that can address the problem right away and doesn't require continuous system state analysis is to stop one of the processes in the cyclic wait after a stalemate has been found. Additionally, by enabling other processes to keep running, it promotes a more effective use of resources. It's important to note that each strategy has its own set of trade-offs, and the ideal approach depends on the requirements and limits of the system.

5. Why can the line in a trajectory such as the one exemplified in Figure 5-10 never move to the left or down?

Ans: A trajectory showing a computer system's running time is shown in Figure 5-10. Because it indicates the length of time the process has been operating, the line in the figure can never move to the left or down. If the process continues to use CPU time, it can never fall or advance. The allocation of resources in a system is shown by the line in Figure 5-10, where the x-axis shows the amount of CPU time allotted to Process A and the y-axis shows the amount of CPU time allotted to Process B. The graph's shaded portions denote those where mutual exclusion is broken, or when many processes can access the same resource.

Chart

Description automatically generated

The graph's line can only move up or to the right (A receives CPU time in this case), not down or to the left. This is because moving to the left or down indicates a process has freed resources, which only happens after a process has finished. The line cannot move to the left or down if a process has not finished since it cannot release any resources. Because it would imply that a process has released resources that it hasn't freed, entering the zone indicated by the letter "U" would be risky because it would eventually result in deadlock. By determining whether the system is likely to enter a dangerous area and rejecting the request if it is, the banker's algorithm can be utilized to prevent stalemate. The line represents the allocation of resources over time and moving the line to the left or down would indicate the release of resources, which can only occur after a process has finished. This would violate the principle of mutual exclusion and lead to deadlock.

6. In the per-instruction loop of xec(), why can’t we call the instruction execution function with the following line?

optab [R.PC⃗ op ]( );

Ans:

The line "optab[R.PC->op]" is incorrect because it is not referencing the correct value of the operation code. The correct line would be "optab[op]", This is because op is the variable that holds the value of the operation code of the current instruction, whereas R.PC->op is the operation code of the current instruction. which was saved before incrementing the program counter.

The use of "op" as the index into the "optab" array ensures that the correct function pointer, which corresponds to the current instruction, is called. By saving the operation code to op before incrementing the program counter, we ensure that we are executing the correct instruction, rather than the next instruction.

The optab array holds the function pointers for each Dis instruction and the function pointed to by optab[op] is called, executing the instruction with operation code op.

The value op we saved before incrementing the program counter is the current instruction’s operation code.

optab [op ]( );

7. Why do you suppose that the isched structure has a pointer to the tail of the vmq list and not to the tail of the idlevmq list?

Ans:

The isched structure has a pointer to the tail of the vmq list because the vmq list is a queue of procs (processes) that are waiting for virtual memory, and the tail of the list is the last element in the queue. The tail pointer allows for quick and efficient addition of new processes to the end of the queue, which is important for managing the state of processes that are waiting for virtual memory. On the other hand, the idlevmq list is a list of idle interpreter kprocs (kernel processes) that are not associated with any user process.

The idlevmq pointer is not stored in the isched structure because it is not used as frequently as the vmq pointer, and therefore, it is not necessary to have a quick access to the tail of the idlevmq list.

It is likely because the vmq list represents the interpreter kprocs that are in the Ready state and have a user process table entry attached to the iprog structure member, while the idlevmq list represents the interpreter kprocs that are idle and have no associated user process. The tail of the vmq list is needed because the list is dynamically updated as kprocs are added to or removed from the Ready state, and the tail pointer is used to efficiently add new kprocs to the end of the list. On the other hand, the idlevmq list may not change as frequently and does not require constant updating, so a pointer to the tail is not necessary.

The isched structure has a pointer to the tail of the vmq list because the vmq list represents the queue of processes that are waiting for their turn to run and are blocked but are now ready again. These processes need to be given the opportunity to run periodically, so the scheduler needs to keep track of the tail of this list to efficiently add and remove processes from it. On the other hand, the idlevmq list represents the queue of processes that are idle, meaning they have no associated user process. These processes are not actively waiting to be scheduled, so there is no need to keep track of the tail of this list in the isched structure.

8. Suppose we have a hardware segmentation system like that shown in Figure 9-3 where the base and limit tables have eight entries. Base register i (numbered from 0) contains the value i × 210 + 0x100 and each limit register contains the hexadecimal value 0x200. If the 16-bit virtual address is 0x605f, then what physical address is computed? Does this address generate a fault?

Diagram

Description automatically generated

Ans: If the 16-bit virtual address is 0x605f, the segment number is obtained by dividing the virtual address by the segment size, which is typically a power of 2. In this case, with a limit of 0x200, the segment number can be calculated as follows:

0x605f / 0x200 = 0x303

The segment number is used to index the base and limit tables. If the base register corresponding to the segment number 0x303 is b, and the limit register is l, the physical address is computed as follows:

physical address = b + (0x605f % 0x200) = b + (0x5f)

The value of b can be calculated as follows:

b = 0x303 \* 210 + 0x100 = 0x303 \* 1024 + 0x100 = 0x30700 + 0x100 = 0x30800

Thus, the physical address is 0x30800 + 0x5f = 0x3085f.

To determine if this address generates a fault, we need to check if it is within the limits of the segment. If the limit is l, the last valid physical address in the segment is b + l - 1. If the physical address is greater than this value, a fault is generated. In this case, the last valid physical address is 0x30800 + 0x200 - 1 = 0x30fff. Since 0x3085f < 0x30fff, this address does not generate a fault.

To find the physical address, we first need to find the segment number. The segment number is given by the most significant 4 bits of the virtual address, in this case 0x60. Dividing this by 16 gives us 0x06, which is the segment number.

Next, we need to find the base address of this segment. The base address is stored in base register 6, which has the value 6 × 210 + 0x100 = 0x700.

Finally, we need to add the offset to the base address to find the physical address. The offset is the least significant 12 bits of the virtual address, in this case 0x05f. The physical address is then equal to the base address plus the offset, which is 0x700 + 0x05f = 0x75f.

To determine whether this address generates a fault, we need to compare it with the limit stored in limit register 6. The limit is 0x200, so as long as the physical address is less than or equal to the limit, it is within bounds and no fault will be generated. In this case, the physical address of 0x75f is less than the limit of 0x200, so no fault is generated.

To determine whether the virtual address 0x605f generates a fault, we need to see if the virtual address falls within the limits of any of the segments.

First, we find the segment number by dividing the virtual address by the segment size (210 = 1024):

0x605f ÷ 1024 = 6

So, the virtual address is in segment 6.

Next, we check if the virtual address falls within the limits of this segment by comparing it with the base and limit values for this segment:

Base = i × 210 + 0x100 = 6 × 1024 + 0x100 = 6144 + 256 = 6400

Limit = 0x200 = 512

So, the segment starts at virtual address 6400 and goes up to virtual address 6400 + 512 = 6912.

The virtual address 0x605f falls within the limits of this segment, so it does not generate a fault.

Finally, we compute the physical address by adding the virtual address to the base:

Physical address = virtual address + base = 0x605f + 6400 = 12895.

9. If we have a machine with a 32-bit virtual address and a 1-KB page size, then how many pages are in the virtual address space? If each PTE takes 4 bytes, then how many pages are required to hold a complete page table? If the physical address space is 256 MB, then how many bits are needed for the page frame number (PFN) in the PTE?

Ans: A machine with a 32-bit virtual address and a 1-KB page size would have 2^32 / (1 KB) = 2^22 pages in the virtual address space.

With a 32-bit virtual address and a 1-KB page size, the virtual address space is 2^32 bytes. The number of pages in the virtual address space is then 2^32 / 2^10 = 2^22 pages.

If each PTE takes 4 bytes, then the page table requires 4 \* 2^22 bytes = 4 \* 4 MB = 16 MB.

So, the number of pages required to hold a complete page table is 16 MB / 1 KB = 16 \* 1024 = 16384 pages.

The physical address space is equal to 256 MB = 2^28 bytes. If the physical address space is 256 MB, then the number of bits needed for the PFN in the PTE is log2(256 MB / 1 KB) = log2(256 \* 2^20 / 2^10) = 24 bits.

Since the page size is 1 KB = 2^10 bytes, we can calculate the number of pages in the physical address space as follows:

2^28 bytes / 2^10 bytes/page = 2^18 pages

Thus, the PFN in the PTE must have at least 18 bits to address all the pages in the physical address space.

10. If memory access time is 70 nS and disk access time is 12 mS, then what is the maximum fraction of memory accesses that can generate page faults and maintain an expected memory access time of no more than 100 nS?

Ans: To calculate the maximum fraction of memory accesses that can generate page faults while maintaining an expected memory access time of no more than 100 nS, we need to determine the average time spent accessing memory when some of the memory accesses result in page faults.

Let X be the fraction of memory accesses that generate page faults, and let Tm be the average time spent accessing memory. Then:

Tm = X \* 12 mS + (1 - X) \* 70 nS

Setting Tm = 100 nS and solving for X gives us the maximum fraction of memory accesses that can generate page faults:

X = (100 nS - 70 nS) / (12 mS - 70 nS) = approximately 0.0067.

So, the maximum fraction of memory accesses that can generate page faults and maintain an expected memory access time of no more than 100 nS is approximately 0.0067.

To maintain an expected memory access time of no more than 100 nS, the average time for a memory access, including page faults, must be less than 100 nS. Let's call the fraction of memory accesses that result in page faults as p. Then, the average time for a memory access is given by:

average\_time = (1-p)70 nS + p12 mS

Since 1 mS = 10^6 nS, the above equation can be written as:

average\_time = (1-p)70 nS + p12 \* 10^6 nS

Setting average\_time = 100 nS, we get:

100 nS = (1-p)70 nS + p12 \* 10^6 nS

Solving for p, we get:

p = (100 nS - 70 nS)/(12 \* 10^6 nS - 70 nS)

p = (30 nS)/(12 \* 10^6 nS - 70 nS)

So, the maximum fraction of memory accesses that can generate page faults and maintain an expected memory access time of no more than 100 nS is equal to p.

To maintain an expected memory access time of no more than 100 nS, the total time taken by both memory accesses and page faults should not exceed 100 nS. If the memory access time is 70 nS, then the maximum time that can be taken by page faults is 30 nS.

Let X be the fraction of memory accesses that generate page faults. Then the time taken by page faults is X \* 12mS = X \* 12000 nS.

Since X \* 12000 nS <= 30 nS, we have X <= 30 / 12000 = 0.0025.

So, the maximum fraction of memory accesses that can generate page faults and maintain an expected memory access time of no more than 100 nS is 0.0025 or 0.25%.

REFERENCES:

Principles of Operating System Book and Internet (Google) – Geeksforgeeks.