Bryan Arnold

CSE 4300

4/5/2018

Homework 3

1.1) For this example, we need to determine which of the things in the figure are the processes, and which are the resources. When driving down the road and reaching an intersection, we must turn or go straight. While doing this, we must share all of these potential options with 3 other drivers. This makes the intersection a sort of resource for the drivers, so in this figure the intersections are the resources. As I just said as well, the drivers utilize these resources. So, the processes of this figure are the rows of cars on the streets. Now, to be considered deadlock, the following four things must occur: mutual exclusion, hold and wait, no preemption, and circular wait.

Mutual exclusion occurs when resources can only be used by one process at a time. In the figure, the intersection can only be used by one row of cars at a time, since the other cars cannot access the blocked intersection, thus mutual exclusion is present. Next, hold and wait occurs when a process holding a resource while waiting for another process to release their resource. In the figure, each line of cars is holding its own intersection (resource), while simultaneously trying to access another resource that another row of cars is holding, thus hold and wait occurs. Next, no preemption occurs due to a process voluntarily releasing a resource. No other rows of cars in this scenario can force another row of cars to release the intersection, as the entire row of cars must pass the intersection to voluntarily give it up, hence no preemption exists. Finally, circular wait occurs when consecutive processes wait on the next process, which loops back to the first. In the figure, one row of cars waits for the next row, that row waits for the next row, that row waits for the next row, and finally the last row waits for the original first row, thus circular wait exists here.

1.2) An easy way to avoid any deadlocks within the system is something implemented in traffic laws, a stop sign. By adding a stop sign at each corner of the intersection, the second condition of hold and wait would be broken. The line of cars would be forced to stop at the stop sign, leaving the intersection available to whichever process takes it first. Each car would have to wait at the intersection before taking it, leaving time for other cars to go through it, thus avoiding a deadlock.

2) For this problem, it is easy to start at the smallest value for x and build up. By doing this, we can check all possible values of x until all the processes have the resource and execute without holding up another resource in deadlock. First, let us try x = 0.

When x = 0, the available resource becomes 1 1 1 0 1. Looking at each process, we want to fulfill only one request matrix at a time. So, if we give this resource to process C, it can execute and increase the resources allocated to 2 1 1 1 2. Next, D only needs 1 on the last element of the matrix, so give the resources to D and increase the resources to 3 2 2 1 3. Continuing, A can be met, so give the resources to it and increase it to 4 2 4 1 4. Now, B is the only process remaining.

The issue with B is it requesting 2 0 0 3 0. This could be easily fulfilled by the current resource of 4 2 4 1 4, except the 4th element in the matrix. It needs to be 3, when it is only 1. Looking at the other requests from other processes, there are no other elements greater than 0 in the 4th element. This would create deadlock, so x cannot equal 0. Since no other process requests in A, C, and D add onto the 4th element of the resources, x must accommodate for that. So, lets set x = 2, since in our last process the 4th element was 1, this should get it up to 3. Let's check.

Give the resources, which equal 1 1 1 2 1, to C first. This increases the resources to 2 1 1 3 2. Next, give the resources to D, which increases the resources to 3 2 2 3 3 after it is done. Next, onto process A. This increases the resources to 4 2 4 3 4. Finally, when we reach B, no deadlock has occurred, and the required resources are available to B for what it has requested. The final resources values will be 6 2 4 6 4.

Answer: x = 2, and the schedule of events to be safe is C, D, A, then B in that order.

3.1) Contiguous memory allocation is a method in which to address into memory processes by their physical address. This includes any process that may run on the machine as well as the memory of the OS. First, the OS data is put into the highest possible memory slot. If the segment of OS data is a size of 200 in memory and the maximum physical address is 3000, then the OS would be put into 2800-3000 in memory by the contiguous memory method. Next, any process that comes into the memory comes one at a time. The first process to enter memory starts at the memory space 0. It is allocated into memory based on the size of its data, for instance 0-400. Any further processes that enter the memory space start at the new base of the memory, 400. They would allocate into memory just like the previous one, but with a new base. The memory of a process can never exceed the OS memory address, or the limit. Each process would be loaded by allocating the contiguous segment for the process. Whenever a process is no longer needed in the memory, it is removed, and newer processes shift down in memory address space to accommodate or shift up (depending how one implements this).

The hardware involved in contiguous memory include the memory and the memory controller. The memory is simply storage space for data that allocates processes in when an address is assigned. The memory controller consists of a few things. The CPU gets the logical address in memory with an adder of what is to be done by the memory controller. A relocation register (of the base of the memory) gets the virtual address in memory the CPU requests, which in turn grants the physical address. The address is then compared to the limit of the memory to see if a trap must be called on addressing errors. If the physical address exceeds the limit address in memory, call an error using trap. In contiguous memory allocation the processes are unaware of any sharing between them and in memory. Each memory reference is checked, which ensures complete safety. Lastly, memory checks are fast with the memory controller, but swapping addresses can be very slow as only one process can be managed at a time.

The advantages are very clear. The hardware is very quick and simple, which allows for quick allocation and speed is everything in computing. The registers of add and compare work quickly to do this. The disadvantages pose a problem though. The processes are limited by the physical memory size, when virtual memory sizes could help to expand this. Also, multiprogramming is limited in that all the memory of the processes must fit into the memory, and parallelism is extremely important in an OS. Lastly, processes don't usually use the entire space in memory always, fragmentation is an issue, as well as compaction being very expensive for resources.

3.2) Paging is an alternative to contiguous memory allocation that involves dividing the logical memory into fixed-sizes, called pages. Then, the physical memory is also divided into fixed-sizes, called frames. The frames and pages match each other in size, and the OS manages pages mapped onto the frames. An easy example of this would be dividing one process into 5 parts. Then, address them into physical memory into 6 different frames. By doing this, you can keep certain components of the process active in memory only when they need to be. This eliminates expensiveness of accessing memory. Where these pages are mapped to depends on a few things. They are mapped to be able to fix the hole-fitting problem contiguous memory has. So mapped memory is allocated in order to eliminate hole fitting.

The hardware of paging includes the following. The processes utilize virtual memory, starting at 0 or a known base address, and the OS lays the processes down into physical memory. The MMU (memory management unit) is the main hardware for doing all of this. It translates virtual memory into physical memory for pages by using what are called page tables. The CPU is of course the main thing requesting the MMU to functions by sending it requesting virtual memory. The page table dynamically reallocates memories by mapping whilst also providing protection. Address translation uses powers of two, with the powers not matching in virtual and physical memory, with virtual address bits and offset bits. To lookup certain addresses, a TLB (translation look side buffer) it utilized by the MMU to do so.

Paging has multiple benefits. First, it eliminates the hole-fitting issues with contiguous memory allocation, as well as the external fragmentation issue. By eliminating these, compaction is no longer necessary as well. Processes are also able to run only with partial portion of the process loaded into memory. Paging also allows for sharing between processes, as shared pages can have the same virtual memory but be allocated to the same physical address space. This can reduce memory requirements and allows the OS to keep track of segments like these. Finally, paging does have a few disadvantages. Translation using TLB can get very large and costly. Constantly translating virtual addresses into physical ones would take a lot of hardware to support it. The OS must also be much more sophisticated and would require large resources to enable context swapping.

4.1) The number of entries that should be put into this page table depends on the virtual address space as well as the size of the pages. For each page, there must be a corresponding entry in the page table, so the number of entries must match the number of pages. To find the number of pages, the virtual memory space must be divided by the size of each page. For this problem, virtual memory space = 24-bits (2^24 bits) and page size = 4 KB (2^12 bits). Now, the number of entries = virtual memory space / page size: 2^(24-12) bits = 2^12 entries.

4.2) For the number of bits to index for the page table we know that there are 2^12 entries in our page table. Each power of two of entries can each be indexed by a single bit. So, the final number of bits to index the number of entries in the page table is simply 12 bits.

4.3) For this problem, I will be doing one at a time. You first need to translate the decimal number into binary. Then, combine this binary with the offset of each one. The first component of the binary, excluding the offset, will determine the physical address of the number. Then, if the physical address is found, turn it into decimal and again into binary of 20 bits. The first 8 bits will be determining a hit or miss.

First address: 1536 (misses)

Binary = 11000000000, pairing on the offset and turning into 24 bits VM = 0000 0000 0000 0110 0000 0000. The last 12 bits are the offset, so looking at the first 12 we see that it is 0000 0000 0000, which is page 0. Looking at the table, there is no virtual memory address listed on the table, so this results in a page fault, or a miss.

Second address: 5102 (hits)

Binary = 1001111101110, pairing on the offset and turning into 24 bits VM = 0000 0000 0001 0011 1110 1110. The last 12 bits are the offset, so looking at the first 12 we see that it is 0000 0000 0001, which is page 1. This is one the virtual memory table, so we can continue to check the physical memory of the address. 17390 = 0000 0100 0011 1110 1110, as only 20 bits are used in the physical address. Excluding the last 12 bits for offset, the last 8 are 0000 0100, which is page 4 in decimal. Looking at the table, we have a section with physical address 4. Since the virtual and physical memory both match up, this address is a hit.

Third address: 10250 (misses)

Binary = 10100000001010, pairing on the offset and turning into 24 bits VM = 0000 0000 0010 1000 0000 1010. The last 12 bits are the offset, so looking at the first 12 we see that it is 0000 0000 0010, which is page 2 in decimal. Looking at the table, there isn’t a virtual address page of 2, so this address misses.