|  |  |  |
| --- | --- | --- |
|  | Homework 3 |  |
| Due: May 23, 2022 |  | Points: 100 |

When you do the homework, please put the answers to questions 1-4 on one page, and the answers to each of the others on separate pages. You can save this file and put your answers on it. This will make using Gradescope to grade the assignment much easier than if you submitted everything without regard to pages.

Remember, you must *justify all your answers*.

Short-answer

1. (6 points) What is “device independence”?

Device independence is the process of hiding the characteristics of I/O devices from higher level softwares. Hence, higher level softwares are able to access any I/O device without having to specify the device in advance. For example, a program that reads a file as input or executes a command should be able to read/write to a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.

1. (7 points) What is thrashing?

Thrashing occurs when a process spends more time paging than executing. This phenomenon occurs when the memory demands of the set of the running processes exceeds the available physical memory.

For instance, when there are only a few executing processes, the OS will bring in new processes as CPU utilization is relatively low. As new processes come in, more frames will be needed. These frames will be acquired from other processes, which in turn will begin page faulting and queueing for the paging device. This will lead to emptying the ready queue, which will drop CPU utilization, and hence more processes will be brought in again by the OS. Evidently, this cycle will continue, and processes will be stuck contending for the paging device.

1. (10 points) What is the difference between paging and segmentation?

Paging is the process of splitting program memory into fixed sized blocks called “pages” to prevent the contiguous allocation of physical memory. Page sizes are determined by available memory. However, this technique can cause internal fragmentation as unused allocated space within pages cannot be used by other processes. Paging is handled by the OS.

Segmentation, on the other hand, is the process of splitting program memory into a collection of variable sized chunks called “segments”. Segments are decided as a result of the structure of the program of a process. As processes are loaded and removed from the memory, the free memory space is broken chunks, causing external fragmentation. Segments are often determined by the compiler.

Long Answer Questions

1. (25 points) You are the president of Cheapo Computronics, Inc., and your star hardware designer has suggested a brilliant idea: Implement segmentation, but let the least significant m bits of a virtual address be used to select the segment, and let the other bits determine the offset. What is the problem with this idea?

This is not a good idea because if you access instructions sequentially, each access will map to a different segment.

Assume n is the number of instructions fetched sequentially. Then, the proposed strategy would be accessing n % 2^m segments which is far more than the n/2^m segments accessed in the normal strategy.

Thus, the program would need a lot more segments in memory in order to run. This creates more work for the CPU, as it will have to work more to swap segments in and out of memory.

1. (25 points) This question asks you to compare different disk scheduling policies.
2. Under very light loads, all the disk scheduling policies we have discussed degenerate into which policy? Why?

Under very light loads, there is only one disk request in the queues, so all the scheduling policies become FCFS.

1. Consider a system on which a seek takes 0.5+0.4T ms, where T is the number of cylinders moved. Then assume the arm is initially at cylinder 100, the disk has 200 cylinders, and the arm is moving inward. Will requests scheduled by a FCFS disk scheduling policy ever have a lower mean waiting time than those scheduled by a SCAN policy? Than those scheduled by a SSTF policy? Justify your answers.

If there is a situation where a new request arrives for a track that has just been passed by the arm, then FCFS is better, since SCAN will have to loop all the way to the edge of the disk and come back around to service that request.

Ex) Arm initially at track 100, moving inwards. Assume requests arrive as follows: **99, 100**

Using SCAN –

Request for 101 is handled first, then the arm wraps back around to 99, thus covering 201 tracks

mean wait time = [(0.5 + 0.4\*1) + (0.5 + 0.4\*201)]/2 = **40.9 ms**

Using FCFS –

Request for 99 is handled, then 101 is handled.

mean wait time = [(0.5 + 0.4\*1) + (0.5 + 0.4\*3)]/2 = **1.3 ms**

For SSTF, the closest track to current position is serviced first. The major problem with this policy is that requests with high variances can be starved.

Ex) Head initially at track 100, moving inwards. Assume requests in queue are as follows: **110, 95, 90**

Using SSTF –

Request for 95 is handled first, then 90, then 110.

mean wait time = [(0.5 + 0.4\*5) + (0.5 + 0.4\*5) + (0.5 + 0.4\*30) ]/3 = **5.8 ms**

Using FCFS –

Requests are handled in order of arrival

mean wait time = [(0.5 + 0.4\*10) + (0.5 + 0.4\*15) + (0.5 + 0.4\*5) ]/3 = **4.5 ms**

Thus, FCFS is better in this situation than SSTF.

1. (27 points) Consider a file currently consisting of 100 blocks of 512 words each. Ignoring the access to update the device directory, and assuming a disk block number fits into a single word, how many disk I/O operations are involved with contiguous, linked, and indexed allocation strategies, if one block:

Assume only reads and writes are I/O operations. Updating the file directory is not an I/O operation.

(a) is removed from the beginning?

1. Contiguous Allocation – **0 operations**

This is because we only need to update the position of the starting block in the file directory. We must change this from block 1 to block 2.

1. Linked Allocation – **1 operation**

Read the first block to get the address of the next block it points to. Then update the directory’s “first pointer” to point to the address of the second block.

1. Indexed Allocation – **2 operations**

Read the index table and then remove the data that the block points to.

(b) is added in the middle?

1. Contiguous Allocation – **201 operations**

Inserting the new block is a write and counts as 1 operation. Following this, we will need to perform two operations on each block after the inserted block; 1 read and 1 shift. However, assuming the worst case situation, we do not have sufficient space to store the data. As a result, we will have to delete and rebuild all the blocks in a new location.

The cost of this is 100 \* (1 read + 1 shift) + 1 write\_new\_data = 201

1. Linked Allocation – **52 Operations**

Since there are 100 blocks, 50 blocks must be read to reach the middle. Like a linked list insert, we must make the current 50th block point to the new block and then make the new block point to the previous 51st block. This amounts to 52 disk operations

1. Indexed Allocation – **3 operations**

Read the index table. Write the new block’s location in the table. Write the new data into the new block.

(c) is added at the end?

1. Contiguous Allocation– **101 operations**

Same as the previous case, we must assume worst case. Assume we do not have sufficient space to store the data. As a result, we will have to delete and rebuild all the blocks in a new location.

The cost of this is 100 + 1 write\_new\_data = 101

1. Linked Allocation **– 1 operation**

Use the “last pointer” to find the last block in the file. Change the next pointer of the current last block to point to the new block. Change the “last pointer” in the directory to point to the new block.

1. Indexed Allocation – **3 operations**

Read the index table. Write the new block’s location in the table. Write the new data into the new block.