CE2005/CZ2005 Operation Systems
Q & A

MEMORY ORGANIZATION & VIRTUAL MEMORY

If the lecture slide changes 2K to 4K, then the offset bits will be 12? Am I right to say so?

Correct. If page size is 4K bytes (that is, 2^12 bytes), the offset needs 12 bits.

Regarding the two-level paging, why are there 10 bits for outer level page? For 32-bit address 4k page size, there are 2^32/4*2^10 entries, which is understandable. Then we want to page 2^20 entries, which should result in 2^20/4*2^10 entries for outer level instead of 2^10.

It went wrong here “Then we want to page 2^20 entries, which should result in 2^20/4*2^10 entries”. You are correct that there 2^20 entries. But, each entry occupies 4 bytes. So, the size of the page table is 4*2^20. So, if the page table is paged, we have (4*2^20)/(4*2^10) = 2^10 pages (that is, 2^10 entries in the outer level page table).

Why does each entry occupy 4 bytes? Is this a value we assume? Also, for 32bit address, why do we calculate it by 2^32/... instead of 2^32*4/...

Yes, it’s an assumption that each entry occupies 4 bytes. In general, entry size may be larger than 4 bytes. Given 32 bit address, we can use it to address 2^32 bytes. That’s why the address space is 2^32 bytes and the logical address has 32 bits!

I have a couple of clarifications regarding Two-Level Paging. Kindly let me know if my understanding is incorrect.

I understand that it is normally a good idea to break up a page table into smaller parts when it becomes overly large. And when this happens, we will also be required to implement an "outer page table".

1) When paging the inner page table, we do the following:
- Determine size of page table (i.e. 4 bytes * 2^20 entries)
- Divide the total size by the page size to obtain the number of pages it which the original large page table can be broken up into

Suppose after this we calculate the number of pages to be 2^10 and therefore 10 bits is required for P1 in the logical address.

Correct.

2) To determine P2, which allows us to identify the actual entry in the individual pages we obtain in Step 1, can we do the following instead:
- (No. of entries in the original page table) / (No. of pages broken up from original page table) = (2^20) / (2^10) = 2^10
- I don't quite understand your working for arriving at P2 in the slide

The two methods are actually the same. Recall how the “No. of pages broken up from original page table” is calculated:

(No. of entries in the original page table * 4) / page size

Put the above into your formula, we have:

(No. of entries in the original page table) / (No. of pages broken up from original page table) =
(No. of entries in the original page table) / ((No. of entries in the original page table * 4) / page size) =
page size / 4

4 is the size per entry.

3) May I also know the purpose of offset in paging?

We need offset in order to address every byte within a page. For example, if page size is 4 bytes (i.e., 2^2), the number of bits required for offset will be 2 (that is, we can use 00, 01, 10, and 11 to address the four bytes within a page).

It was mentioned in our slides that the Inverted Page Table is a single table with one entry for each physical frame. It was also mentioned that the table holds the 2 value: process-id, page-no

Is the process-id here the id of the process which we're trying to access? Also, is the page-no referring to the frame number in the physical memory?

Correct. Process-id is the id of the process of concern. The page-no refers to the page that we would like to access in process’ logical address space. For example, if an entry i of the inverted page table contains (p, k), that means page k of process p is mapped to frame i in the physical memory.

When you say that the Inverted Page Table is sorted by the physical address, do you mean sorted by the page-no, but searching is performed on the process-id, that's why it increases search time?

I mean it is sorted according to the frame order. There is one entry per frame in the inverted page table. The first entry is for frame 0, and second entry for frame 1, and so on.

What do you mean "To access the memory, the pair <process-id, page-no> is presented to inverted page table to find a match. If match is found, say at entry i, then physical addr <i, offset> is obtained."?

The logical address format in this case is <process-id, page-no, offset>. To translate this logical address, the pair <process-id, page-no> will be used to search the inverted page table for a matching entry (recall, if frame i is allocated to page k of process p, the entry i will contain <p, k>). If a match is found, say at entry i, that means the page we are trying to access is mapped to frame i. So, the physical address will be <i, offset>.

Inverted Page Table: Why is it called inverted? Is it actually inverted?

It's called inverted page table since the entries are order according to frame numbers (rather than page numbers as in the page table).

In slide 6.17 regarding compaction to solve the issue of fragmentation, I did not quite understand what happens? Is it that the process images are reorganized to be grouped together so that the free memory blocks can be brought together? However, in the diagram in the slide there are still non contiguous free spaces left. So how is compaction exactly implemented?

You are correct. That slide is animated. So, the diagram is not very meaningful. I'll put a copy of slides in "PowerPoint Slide Show" format (with animation) in edveNTUre.

In slide 6.19, how is internal fragmentation still possible in paging? If the page size in the logical memory and frame size n the physical memory are equal, how is it that internal fragmentation takes place and on average half of a page is unused?

As I explained during the lecture, the logical address space may not be divided into exact number of pages. Internal fragmentation happens only at the last page since the last page may not occupy the entire frame. On average the last page occupies only half of the frame.

For Shared Pages, you had mentioned that the code must be read only for pages to be shared among multiple processes. Could you explain why?

The sharing is on the basis that the code must be common. If code is not reentrant, it is no longer common amongst processes. So, it can't be shared.

Why can't segmentation be a method of fixed partitioning memory allocation and consequently exhibit internal fragmentation also?

If fixed partitioning is used, the number of segments that can be loaded into memory would be fixed. To run a process, all its segments need to loaded into memory. So, fixed partitioning would limit the number of processes that can run concurrently. In general, memory allocation for segment uses similar strategies that are used under dynamic portioning (i.e., first fit, best fit, and worst fit).

Paging involves logical address to physical address(main memory). Is the physical address from demand paging referring to main memory as well? How is demand paging different from paging?

Yes, physical address in demand paging refers to the address to access physical memory.

Using demand paging, a process' logical address space needs not to loaded into physical memory entirely. A page in the logical address space is only loaded when it is referenced. Demand paging is used to illustrate how virtual memory can be implemented.

Paging scheme discussed under the memory organization assumes that the entire logical address space must be loaded into physical memory when a process is executed.

For the page replacement, you said that if the page in the physical memory is modified, then we have to page out to the backing store and update the page in the backing store. Does that mean there is 2 similar pages, 1 in the physical memory and 1 in the backing store? for example page A exist BOTH in the physical memory and backing store. If page A is modified, then that page is written to the backing store.

You are right. The entire process image can be always found in the backing store

I have a doubt regarding 'inclusion property' that was taught in the lecture last week, 13 Oct. Do you mind if you explain to me regarding this property?

Inclusion property is satisfied if given the same page reference string, pages loaded using n frames is always a subset of pages loaded using n+1 frames. So, if a page fault occurs when using n+1 frames, it must also happen when using n frames. Hence, number of page faults generated using n frames is always greater than that using n+1 frames.

For example, FIFO suffers from Belady's anomaly, because the inclusion property doesn't hold. You can check slides 7.20 and 7.21 – after page 5 is referenced, the following pages are in the memory when 3 frames are used: 5, 1, 2; whereas pages 5, 2, 3, 4 are in the memory when 4 frames are used. {5, 1, 2} is not a subset of {5, 2, 3, 4}.

FILE SYSTEMS

I did not understand the meaning of symbolic link in file sharing. What does it mean by "resolving a link by using the path name to locate the real file" ? Going by the acyclic graph directory structure given in the slide 8.17, does it mean that directory D contains the path name /X/Y, so user can use directory D to reach the file whose original path name is /X/Y.

Symbolic link is a special entry in the directory. It contains a file name and a path name of the original directory entry of the file. As in the example given in the lecture, the original entry is /A/B/C/D, which contains all the file attributes, including where the file data blocks are in the hard disk. /X/Y is the symbolic link, which contains the path name /A/B/C/D. Thus, user can use the path name to locate the original entry and then all the file attributes.

In problems in file sharing, what does it mean when you say: In traversing the the file system, the shared files may be visited more than once?

Since for one physical file, there are multiple directory entries that can be used to access the file (i.e.., the original directory entry of the file, and the symbolic or hard links). Traverse the file system means to access every entry in every directory in the file system. So, the same physical file will be accessed more than once if it is shared using either symbolic or hard link by multiple users.

What will happen in the absence of the per process open file table?

Without per process open file table, some additional information needs to be kept in the system wide open file table, for example, the position pointer fir each process. Recall a file can be opened by more than one process and processes may access different places in the file (and so they have different position pointers).

Does the wastage of space in case of contiguous allocation refer to when more physical blocks are being assigned to a file than needed? Are all physical blocks of same size and is this size the same as the size of the logical block?

If more physical blocks are assigned to a file than needed, internal fragmentation happens (since wastage happens within the file). Contiguous file allocation suffers external fragmentation (I.e., holes between files) anyway. Yes, physical blocks all have the same size. The size of physical block and logical block are the same.

In case of the bit map, when we want to deallocate a block, it's bit value is changed in the bit map from 1 to 0. But does the file get deleted from the secondary storage or not or is the block just freed from the file it was holding?

No, the data are not actually deleted. The file system just simply frees up the block.

Is it possible to know page size given the page table entry size and the logical address known?

No, you don't have sufficient information to determine the page size. If you also know the page table size, then you can determine the page size.

For example, what will be the page size if the page table entry size is 4 bytes and the logical address is 32 bit?

In this case, if you also know the page table size (e.g., 4x2^20), then you can determine the page size is 2^12 as follows:

Assuming page size is n, then we have: (2^32/n) x 4 = 4x2^20 => n = 2^12

Where is the file allocation table located? If it is located in the hard disk and not the memory, how is the access time optimized?

FAT is located in a fixed location on disc. To optimize it's performance, a copy of it is usually also cached in memory.

Referring to slide 8.38, in indexed allocation it is said that the max file size is 128K bytes and the block size is 512 bytes. I am confused as to what exactly is the block size here. Is it the size of each logical block or is it the size of each block in the Index table?

In normal index allocation scheme, there is no special type of blocks for index blocks. So, both index block and data block are physical blocks. They have the same size. We also assume that logical block and physical Block are of the same size.

Also, earlier in the indexed allocation it has been stated that each file has an index block allocated to it. Hence in the above example, why is it that two index blocks are allocated to a file and not one?

If the file size is big, it may need an array of index blocks. Recall each index block can only accommodate a fixed number of entries.

In the same slide, for single indirect access, how was the 2^12 obtained the calculation and what does it represent?

The block size is 2^12. The pointer (or entry) size is 2^2, so, the number of pointers an index Block contains is: 2^12/2^2 = 2^10. Each pointer pointing to a data block. So, the maximum file size that can be supported by using a single indirect pointer alone is 2^10 * 2^12 = 2^22.

In your lecture, you told us to think why is stripped mirror is better.

Is it because if a disk fails in a striped mirror layout, only the failing disk is detached, and only that portion of the volume loses redundancy. When the disk is replaced, only a portion of the volume needs to be recovered. Mirrored drives are independent units. When a disk is lost, it affects only one mirrored pair. Thus, compared to a mirrored-stripe, a striped-mirror offers more tolerance to disk failure. If a disk failure occurs, the recovery time is shorter for a striped-mirror layout?

And I would like to clarify with you: Striped mirrors is to do mirroring then stripe while Mirrored stripes is to stripe then do mirror right?

Striped mirror (i.e., 0+1) is to do the stripping first followed by mirroring. Mirrored strip (i.e., 1+0) is to do mirroring first followed by stripping.

For example, assume we have two sets of hard disks: {C11, C12, C13, C14} and {C21, C22, C23, C24}. For 0+1, given a block b with sub-blocks {b1, b2, b3, b4}, it will be first stripped to the first set of hard disks: {b1, b2, b3, b4} -> {C11, C12, C13, C14}. Then, the stripped block is mirrored to the second set of hard disks. For 1+0, the two sets of hard disks are arranged in mirrored pair first. That is, we have {(C11, C21), (C12, C22), (C13, C23), (C14, C24)}. Then, the block is stripped to the mirrored pairs of the hard disks: {b1, b2, b3, b4} -> {(C11, C21), (C12, C22), (C13, C23), (C14, C24)}.

So, in 0+1, if there is one hard disk failed in each set, we won't be able to retrieve the block. However, in 1+0, as long as both hard disks are not failed in a mirrored pair, we should be able to retrieve the block.

Hence, in terms of fault tolerance, 1+0 is better than 0+1.