# Chapter 9 Main Memory – Exercises

1. Name two difference between logical and physical addresses

A logical address does not refer to an actual existing address; rather, it refers to an abstract address in an abstract address space.

Contrast this with a physical address that refers to an actual physical address in memory. A logical address is generated by the CPU and is translated into a physical address by the memory management unit(MMU).

Therefore, physical addresses are generated by the MMU.

OR

Logical clusters are associated with the numbering of clusters on the disk or volume. Virtual clusters are associated with the offset within a file.

1. Why are page sizes always powers of 2?

Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

1. Consider a system in which a program can be separated into two parts: code and data. The CPU knows whether it wants an instruction (instruction fetch) or data (data fetch or store). Therefore, two base-limit register pairs are provided: one for instructions and one for data. The instruction base-limit register pair is automatically read-only, so programs can be shared among different users. Discuss the advantages and disadvantages of this scheme.

The major advantage of this scheme is that it is an effective mechanism for code and data sharing. For example, only one copy of an editor or a compiler needs to be kept in memory, and this code can be shared by all processes needing access to the editor or compiler code. Another advantage is protection of code against erroneous modification. The only disadvantage is that the code and data must be separated, which is usually adhered to in a compiler-generated code.

1. What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect could be used to decrease the amount of time needed to copy a large amount of memory from one place to another. What effect would updating some byte on one page have on the other page?

By allowing two entries in a page table to point to the same page frame in memory, users can share code and data. If the code is reentrant, much memory space can be saved through the shared use of large programs such as text editors, compilers, and database systems. “Copying” large amounts of memory could be effected by having different page tables point to the same memory location

However, sharing of nonreentrant code or data means that any user having access to the code can modify it and these modifications would be reflected in the other user’s “copy”

1. Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **First-Fit (KB)** | **300** | **600** | **350** | **200** | **750** | **125** |
| **115** | 175 | 600 | 350 | 200 | 750 | 125 |
| **500** | 175 | 100 | 350 | 200 | 750 | 125 |
| **358** | 175 | 100 | 350 | 200 | 392 | 125 |
| **200** | 175 | 100 | 150 | 200 | 392 | 125 |
| **375** | 175 | 100 | 150 | 200 | 17 | 125 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Best-Fit (KB)** | **300** | **600** | **350** | **200** | **750** | **125** |
| **115** | 300 | 600 | 350 | 200 | 750 | 10 |
| **500** | 300 | 100 | 350 | 200 | 750 | 10 |
| **358** | 300 | 100 | 350 | 200 | 392 | 10 |
| **200** | 300 | 100 | 350 | 0 | 392 | 10 |
| **375** | 300 | 100 | 350 | 0 | 17 | 10 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Worst-Fit (KB)** | **300** | **600** | **350** | **200** | **750** | **125** |
| **115** | 300 | 600 | 350 | 200 | 635 | 125 |
| **500** | 300 | 600 | 350 | 200 | 135 | 125 |
| **358** | 300 | 242 | 350 | 200 | 135 | 125 |
| **200** | 300 | 242 | 150 | 200 | 135 | 125 |
| **375** | N/A | N/A | N/A | N/A | N/A | N/A |

1. Given six memory partitions of 100 MB, 170 MB, 40 MB, 205 MB, 300 MB, and 185 MB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 200 MB, 15 MB, 185 MB, 75 MB, 175 MB, and 80 MB (in order)? Indicate which—if any—requests cannot be satisfied. Comment on how efficiently each of the algorithms manages memory.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **First-Fit (MB)** | **100** | **170** | **40** | **205** | **300** | **185** |
| **200** | 100 | 170 | 40 | 5 | 300 | 185 |
| **15** | 85 | 170 | 40 | 5 | 300 | 185 |
| **185** | 85 | 170 | 40 | 5 | 115 | 185 |
| **75** | 10 | 170 | 40 | 5 | 115 | 185 |
| **175** | 10 | 170 | 40 | 5 | 115 | 10 |
| **80** | 10 | 90 | 40 | 5 | 115 | 10 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Best-Fit (MB)** | **100** | **170** | **40** | **205** | **300** | **185** |
| **200** | 100 | 170 | 40 | 5 | 300 | 185 |
| **15** | 100 | 170 | 25 | 5 | 300 | 185 |
| **185** | 100 | 170 | 25 | 5 | 300 | 0 |
| **75** | 25 | 170 | 25 | 5 | 300 | 0 |
| **175** | 25 | 170 | 25 | 5 | 125 | 0 |
| **80** | 25 | 170 | 25 | 5 | 45 | 0 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Worst-Fit (MB)** | **100** | **170** | **40** | **205** | **300** | **185** |
| **200** | 100 | 170 | 40 | 205 | 100 | 185 |
| **15** | 100 | 170 | 40 | 190 | 100 | 185 |
| **185** | 100 | 170 | 40 | 5 | 100 | 185 |
| **75** | 100 | 170 | 40 | 5 | 100 | 110 |
| **175** | N/A | N/A | N/A | N/A | N/A | N/A |
| **80** | 100 | 90 | 40 | 5 | 100 | 110 |

1. Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):
   1. 3085
   2. 42095
   3. 215201
   4. 650000
   5. 2000001
2. page = 3; offset = 13
3. page = 41; offset = 111
4. page = 210; offset = 161
5. page = 634; offset = 784
6. page = 1953; offset = 129

# of bits in offset part   
Solution steps:  
1. Convert logical address: Decimal to Binary  
2. Split binary address to 2 parts (page # , Offset), offset : n digits

3. Convert offset & page# : Binary to Decimal

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Logical address (decimal)** | **Logical address (binary)** | **Page # (22 bits) (binary)** | **Offset (10 bits) (binary)** | **Page # decimal** | **Offset decimal** |
| **3085** | **00000000000000000000110000001101** | **0000000000000000000011** | **0000001101** | 3 | 13 |
| **42095** | **00000000000000001010010001101111** | **0000000000000000101001** | **0001101111** | 41 | 111 |
| **215201** | **00000000000000110100100010100001** | **0000000000000011010010** | **0010100001** | 210 | 161 |
| **650000** | **00000000000010011110101100010000** | **0000000000001001111010** | **1100010000** | 634 | 784 |
| **2000001** | **00000000000010000000000000000001** | **0000000000001000000000** | **0000000001** | 512 | 1 |

1. Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers)?
2. 21205
3. 164250
4. 121357
5. 16479315
6. 27253187
7. The BTV operating system has a 21-bit virtual address, yet on certain embedded devices, it has only a 16-bit physical address. It also has a 2-KB page size. How many entries are there in each of the following?
8. A conventional, single-level page table
9. An inverted page table

What is the maximum amount of physical memory in the BTV operating system?

1. **# of pages in conventional, single-level page table**

= # of pages in virtual address space

=

1. **# of entries in inverted page table**

= # of pages in physical address space

=

1. Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
   1. How many bits are there in the logical address?

Logical address: 16 bits

* 1. How many bits are there in the physical address?

Physical address: 15 bits

# of pages

Page size

Frame size = 32 =

1. Logical address space size =
2. Physical address space size =
3. Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
4. How many bits are required in the logical address?

Logical address space (size)

=

= = =

=

🡺 bits

1. How many bits are required in the physical address?

Let x be the number of physical addresses, then

Physical address space (size)

=

= = = =

=

🡺 bits

1. Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?
2. A conventional, single-level page table
3. An inverted page table

logical memory space = 32-bit = bytes

page size = 4-KB = bytes

physical memory size = 512-MB = bytes

1. # of entries in a conventional single-level page table =
2. # of entries in an inverted page table =
3. Explain the difference between internal and external fragmentation.

**External fragmentation**

* + Total memory space exists to satisfy a request, but it is not contiguous

**Internal fragmentation**

* Allocated memory may be slightly larger than requested memory
* This size difference is memory internal to a partition, but not being used

Internal fragmentation is the wasted space within each allocated block because of rounding up from the actual requested allocation to the allocation granularity.

External fragmentation is the various free spaced holes that are generated in either your memory or disk space. External fragmented blocks are available for allocation, but may be too small to be of any use.

1. Consider the following process for generating binaries. A compiler is used to generate the object code for individual modules, and a linker is used to combine multiple object modules into a single program binary. How does the linker change the binding of instructions and data to memory addresses? What information needs to be passed from the compiler to the linker to facilitate the memory-binding tasks of the linker?

The linkage editor has to exchange unresolved symbolic addresses with the actual addresses associated with the variables in the binary code.

To do this, the modules should save instructions that refer to unresolved symbols.

During linking, each module is assigned a sequence of addresses in the overall program binary and when this has been performed, unresolved references to symbols exported by this binary could be patched in other modules since every other module would contain the list of instructions that need to be connected.

OR

The linkage editor has to replace unresolved symbolic addresses with the actual addresses associated with the variables in the final program binary. In order to perform this, the modules should keep track of instructions that refer to unresolved symbols. During linking, each module is assigned a sequence of addresses in the overall program binary and when this has been performed, unresolved references to symbols exported by this binary could be patched in other modules since every other module would contain the list of instructions that need to be patched.

1. Most systems allow a program to allocate more memory to its address space during execution. Allocation of data in the heap segments of programs is an example of such allocated memory. What is required to support dynamic memory allocation in the following schemes?
2. Contiguous memory allocation

Due to the absence of enough space for the program to grow its allocated memory space then relocation of the entire program might be required

OR

Might require relocation of the entire program since there is not enough space for the program to grow its allocated memory space

1. Paging

Incremental allocation of new pages is potential without requiring relocation of the program’s address space

1. Compare the memory organization schemes of contiguous memory allocation and paging with respect to the following issues:
2. External fragmentation
3. Internal fragmentation
4. Ability to share code across processes

Contiguous memory allocation scheme suffers from external fragmentation as address spaces are allocated contiguously and holes develop as old processes die and new processes are initiated. It also does not allow processes to share code, since a process's virtual memory segment is not broken into non-contiguous fine-grained segments.

Pure segmentation also suffers from external fragmentation as a segment of a process is laid out contiguously in physical memory and fragmentation would occur as segments of dead processes are replaced by segments of new processes. Segmentation, however, enables processes to share code; for instance, two different processes could share a code segment but have distinct date segments.

Pure paging does not suffer from external fragmentation, but instead suffers from internal fragmentations. Processes are allocated in page granularity and if a page is not completely utilized, it results in internal fragmentation and a corresponding wastage of space. Paging also enables processes to share code at the granularity of pages.

1. On a system with paging, a process cannot access memory that it does not own. Why? How could the operating system allow access to additional memory? Why should it or should it not?

An address on a paging system is a logical page number and an offset. The physical page is found by searching a table based on the logical page number to produce a physical page number. Because the operating system controls the contents of this table, it can limit a process to accessing only those physical pages allocated to the process. There is no way for a process to refer to a page it does not own because the page will not be in the page table. To allow such access, an operating system simply needs to allow entries for non-process memory to be added to the process's page table. This is useful when two or more processes need to exchange data—they just read and write to the same physical addresses (which may be at varying logical addresses). This makes for very efficient inter process communication

1. Explain why mobile operating systems such as iOS and Android do not support swapping.

There are three reasons:

First is that these mobile devices typically use flash memory with limited capacity and swapping is avoided because of this space constraint.

Second, flash memory can support a limited number of write operations before it becomes less reliable.

Lastly, there is typically poor throughput between main memory and flash memory.

1. Although Android does not support swapping on its boot disk, it is possible to set up a swap space using a separate SD nonvolatile memory card. Why would Android disallow swapping on its boot disk yet allow it on a secondary disk?

Android doesn’t support swapping on its boot disk but allow it on a secondary disk because of boot limited storage capacity.

Users must provide their own separate SDcard for swap space to allow Android support swapping.

OR

Primarily because Android does not wish for its boot disk to be used as swap space for the reasons outlined in the previous question – the boot disk has limited storage capacity. However, Android does support swapping, it is just that users must provide their own separate SD card for swap space.

1. Explain why address-space identifiers (ASIDs) are used in TLBs.

ASIDs provide address space protection in the TLB as well as supporting TLB entries for several different processes at the same time.

1. Program binaries in many systems are typically structured as follows. Code is stored starting with a small, fixed virtual address, such as 0. The code segment is followed by the data segment, which is used for storing the program variables. When the program starts executing, the stack is allocated at the other end of the virtual address space and is allowed to grow toward lower virtual addresses. What is the significance of this structure for the following schemes?
2. Contiguous memory allocation
3. Paging
4. Contiguous-memory allocation requires the operating system to allocate the entire extent of the virtual address space to the program when it starts executing. This could be much larger than the actual memory requirements of the process.
5. Pure paging does not require the operating system to allocate the maximum extent of the virtual address space to a process at startup time, but it still requires the operating system to allocate a large page table spanning all of the program’s virtual address space. When a program needs to extend the stack or the heap, it needs to allocate a new page but the corresponding page table entry is preallocated.
6. The MPV operating system is designed for embedded systems and has a 24-bit virtual address, a 20-bit physical address, and a 4-KB page size. How many entries are there in each of the following?
7. A conventional, single-level page table
8. An inverted page table

What is the maximum amount of physical memory in the MPV operating system?

Virtual address = 24-bit

Physical address = 20-bit

Page size = 4KB =

Page offset = 12-bit

# of pages =

* 1. # of entries in single-level page table = # of pages in page table

= entries

* 1. # of entries in inverted page table = # of frames in physical memory

= frames

Maximum amount of physical memory = bytes = 1MB

1. Consider a logical address space of 2,048 pages with a 4-KB page size, mapped onto a physical memory of 512 frames.
2. How many bits are required in the logical address?
3. How many bits are required in the physical address?

# of pages in logical address space =

Page size = 4-KB =

# of frames in physical memory = 512 =

* 1. 🡺 23 bits
  2. 🡺 21 bits

1. Consider a computer system with a 32-bit logical address and 8-KB page size. The system supports up to 1 GB of physical memory. How many entries are there in each of the following?
2. A conventional, single-level page table
3. An inverted page table
   1. Total virtual memory size =

Size of a single page = 8KB =

Total # of pages of virtual memory = = 524K entries

* 1. Total physical memory = 1GB =

Frame size = page size =

Total # of frames in physical memory =

So inverted page table will be having = 128K entries

A close up of a mans face

Description automatically generated

1. Consider a logical address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.
   1. How many bits are there in the logical address?
   2. How many bits are there in the physical address?

Addressing within a 1024-word page requires 10 bits because 1024 = 210.  Since the logical address space consists of 8 = 23pages, the logical addresses must be 10+3 = 13 bits.  Similarly, since there are 32 = 25 physical pages, phyiscal addresses are 5 + 10 = 15 bits long.

A screenshot of a cell phone

Description automatically generated

1. Consider a paging system with the page table stored in memory.
2. If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

400 nanoseconds: 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.

1. If we add TLBs, and if 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

Effective access time = 0.75 × (200 nanoseconds) + 0.25 × (400 nanoseconds) = 250 nanoseconds.

1. What is the purpose of paging the page tables?

In certain situations the page tables could become large enough that by paging the page tables, one could simplify the memory allocation problem (by ensuring that everything is allocated as fixed-size pages as opposed to variable-sized chunks) and also enable the swapping of portions of page table that are not currently used.

1. Consider the IA-32 address-translation scheme shown in Figure 9.22.
2. Describe all the steps taken by the IA-32 in translating a logical address into a physical address.
3. What are the advantages to the operating system of hardware that provides such complicated memory translation?
4. Are there any disadvantages to this address-translation system? If so, what are they? If not, why is this scheme not used by every manufacturer?
5. The selector is an index into the segment descriptor table. The segment descriptor result plus the original offset is used to produce a linear address with a dir, page, and offset. The dir is an index into a page directory. The entry from the page directory selects the page table, and the page field is an index into the page table. The entry from the page table, plus the offset, is the physical address.
6. Such a page-translation mechanism offers the flexibility to allow most operating systems to implement their memory scheme in hardware, instead of having to implement some parts in hardware and some in software. Because it can be done in hardware, it is more efficient (and the kernel is simpler).
7. Address translation can take longer due to the multiple table lookups it can invoke. Caches help, but there will still be cache misses.

# Chapter 10 Virtual Memory – Exercises

1. Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

1. Assume that you have a page-reference string for a process with *m* frames (initially all empty). The page-reference string has length *p*, and *n* distinct page numbers occur in it. Answer these questions for any page-replacement algorithms:
   1. What is a lower bound on the number of page faults?
   2. What is an upper bound on the number of page faults?
2. *n*
3. *p*

|  |  |
| --- | --- |
| p 🡪 page-reference string length  n 🡪 number of different frames in P  m 🡪 number of frames |  |
| a. lower bound on the number of page faults is (n) | what is lower bound on the number of page faults  lower limit --> best case  we assume that m > n  thus, the best case i |
| b. upper bound on the number of page faults is (p) |  |

1. Consider the following page-replacement algorithms. Rank these algorithms on a five-point scale from “bad” to “perfect” according to their page-fault rate. Separate those algorithms that suffer from Belady’s anomaly from those that do not.
   1. LRU replacement
   2. FIFO replacement
   3. Optimal replacement
   4. Second-chance replacement

|  |  |  |
| --- | --- | --- |
| **Rank** | **Algorithm** | **Suffer from Belady’s anomaly** |
| 1 | Optimal | no |
| 2 | LRU | no |
| 3 | Second-chance | yes |
| 4 | FIFO | yes |

1. An OS supports a paged virtual memory. The central processor has a cycle time of 1 microsecond. It costs an additional 1 microsecond to access a page other than the current one. Pages have 1,000 words, and the paging device is a drum that rotates at 3,000 revolutions per minute and transfer 1 million words per second. The following statistical measurements were obtained from the system:

* One percent of all instructions executed accessed a page other than the current page.
* Of the instructions that accessed another page, 80 percent accessed a page already in memory
* When a new page was required, the replaced page was modified 50 percent of a time

Calculate the effective instruction time on this system, assuming that the system is running one process only and that processor is idle during drum transfers.

1. Consider the page table for a system with 12-bit virtual and physical addresses and 256-byte pages.

|  |  |
| --- | --- |
| Page | Page Frame |
| 0 | - |
| 1 | 2 |
| 2 | C |
| 3 | A |
| 4 | - |
| 5 | 4 |
| 6 | 3 |
| 7 | - |
| 8 | B |
| 9 | 0 |

The list of free page frames is *D, E, F* (that is, *D* is at the head of the list, *E* is second, and *F* is last). A dash for a page frame indicates that the page is not in memory

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal.

* 9EF
* 111
* 700
* 0FF

9EF - 0EF

111 - 211

700 - D00

0FF - EFF

1. Discuss the hardware functions required to support demand paging.

For every memory-access operation, the page table needs to be consulted to check whether the corresponding page is resident or not and whether the program has read or write privileges for accessing the page. These checks have to be performed in hardware. A TLB could serve as a cache and improve the performance of the lookup operation.

1. Consider the two-dimensional array A:

int A[][] = new int[100][100];

where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0.

For three page frames, how many page faults are generated by the following array-initialization loops? Use LRU replacement, and assume that page frame 1 contains the process and the other two are initially empty.

* 1. for (int j = 0; j < 100; j++)

for (int i = 0; i < 100; i++)

A[i][j] = 0;

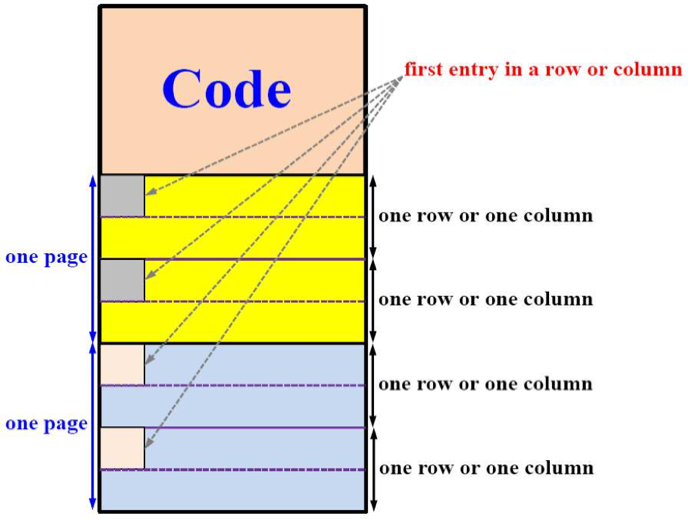
* 1. for (int i = 0; i < 100; i++)

for (int j = 0; j < 100; j++)

A[i][j] = 0;

1. 5000
2. 50

The system has three-page frames for this process. The first is for code and the second and third are for the array. Note that these three-page frames do not have to be consecutive, even though the diagram below shows they are. The problem statement does not state the page size unit (i.e., byte or something else) and the size of an int. For simplicity, we just assume each unit of the size 200 page fits an int. In this case, each page can fit two rows or two columns, depending on how the compiler stores a 2-dimensional array. If the compiler produces all entries of a row consecutively (i.e., row-major), then each page contains two rows. Otherwise, the compiler stores all entries of a column consecutively (i.e., column-major), then each page contains two columns. Because the code is in C style, the array is stored row-by-row (i.e., row-major).



Because each entry in the array is only visited once, FIFO and LRU have no difference. In Part (a), the initial- ization goes column-by-column. Therefore, when a page is loaded into memory, it is only access twice, once per row. Refer to the diagram above for the details. Because the two available page frames are initially empty, once the initialization procedure starts, the first page is loaded in and accessed twice for row 0 and row 1. When the initialization goes to row 2, a page fault occurs and the second page is loaded, which is accessed twice for row 2 and row 3. In this way, just for column 0 for every two rows there is a page fault. To complete initial- ization for column 0, there will be 50 page faults. Because we have 100 columns, the number of page faults is 5,000= 50 × 100.

|  |  |  |
| --- | --- | --- |
| Problem | (a) | (b) |
| Answer | 5,000 | 50 |

For Part (b), the initialization is done row-by-row. Consequently, we have one page faults for every two rows. Because we have 100 rows, the total number of page faults is 50 = 100/2.

If you interpret the size of 200 as 200 bytes and each int requires 4 bytes, then each row has 400 bytes (i.e., 2 pages) and there are 200 = 2 × 100 pages for matrix A[ ][ ]. For Part (b), A[ ][ ] is initialized row-by-row and, as a result, initializing each row generates 2 page faults and the total number of page faults is 200 = 2 × 100. For Part (a), A[ ][ ] is initialized column-by-column, and, each access to a row will cause a page fault because two page frames can only hold one row. Consequently, for each column there will be 100 page faults. Because there are 100 columns, the total number of page faults is 10,000= 100 × 100.

|  |  |  |
| --- | --- | --- |
| Problem | (a) | (b) |
| Answer | 10,000 | 200 |

Instructions all found in page 0, 3 page frames, frame 1 contains the process, others are initially empty. Mainly looking at how the array

Consider you have three frames, execute code, it needs to be in memory just like data in the memory, your data and your code are all using the memory. Page 0 in logical add space has all your code, you execute the code, you fetch the instruction, which is in the page 0.

Memory system holds both instruction and data, as you execute code, you’re fetching you instruction from your mem, you’re referencing your data from mem

LRU replacement is a clue, we are dealing with pages and frames, they have sizes

After the 200 page,

Leftover in frame 1 (page 0)

Use LRU, when I go to next 200, have to find another page, need to find a victim

|  |
| --- |
| **LRU**  A screenshot of a cell phone  Description automatically generated |
| **FIFO**  A close up of a keyboard  Description automatically generated |
| **Optimal**  A picture containing clock  Description automatically generated |

1. Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

* LRU replacement
* FIFO replacement
* Optimal replacement

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of frames** | **LRU** | **FIFO** | **Optimal** |
| 1 | 20 | 20 | 20 |
| 2 | 18 | 18 | 15 |
| 3 | 15 | 16 | 11 |
| 4 | 10 | 14 | 8 |
| 5 | 8 | 10 | 7 |
| 6 | 7 | 10 | 7 |
| 7 | 7 | 7 | 7 |

1. Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1

Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?

* LRU replacement
* FIFO replacement
* Optimal replacement
* LRU

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 7 | 7 | 1 | 1 | 1 | 3 | 3 | 3 | 7 | 7 | 7 | 7 | 5 | 5 | 5 | 2 | 2 | 2 | 1 |
|  | 2 | 2 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 4 | 1 | 1 | 1 | 4 | 4 | 4 | 3 | 3 | 3 |
|  |  | 3 | 3 | 3 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 0 | 0 | 0 | 6 | 6 | 6 | 0 | 0 |
| PF | PF | PF | PF |  | PF | PF | PF | PF | PF |  | PF | PF | PF | PF | PF | PF | PF | PF | PF |

Total Page Faults = 18

* FIFO

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 7 | 7 | 1 | 1 | 1 | 1 | 1 | 6 | 6 | 6 | 6 | 0 | 0 | 0 | 6 | 6 | 6 | 0 | 0 |
|  | 2 | 2 | 2 | 2 | 5 | 5 | 5 | 5 | 7 | 7 | 7 | 7 | 5 | 5 | 5 | 2 | 2 | 2 | 1 |
|  |  | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 1 | 1 | 1 | 4 | 4 | 4 | 3 | 3 | 3 |
| PF | PF | PF | PF |  | PF |  | PF | PF | PF |  | PF | PF | PF | PF | PF | PF | PF | PF | PF |

Total Page Faults = 17

* Optimal

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 7 | 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|  | 2 | 2 | 2 | 2 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 4 | 6 | 2 | 3 | 3 | 3 |
|  |  | 3 | 3 | 3 | 3 | 3 | 4 | 6 | 7 | 7 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PF | PF | PF | PF |  | PF |  | PF | PF | PF |  |  | PF |  | PF | PF | PF | PF |  |  |

Total Page Faults = 13

1. Suppose that you want to use a paging algorithm that requires a reference bit (such as second-chance replacement or working-set model), but the hardware does not provide one. Sketch how you could simulate a reference bit even if one were not provided by the hardware, or explain why it is not possible to do so. If it is possible, calculate what the cost would be.

You can use the valid/invalid bit supported in hardware to simulate the reference bit. Initially set the bit to invalid. On first reference a trap to the operating system is generated. The operating system will set a software bit to 1 and reset the valid/invalid bit to valid.

1. You have devised a new page-replacement algorithm that you think may be optimal. In some contorted test cases, Belady’s anomaly occurs. Is the new algorithm optimal? Explain your answer.

No. An optimal algorithm will not suffer from Belady’s anomaly because —by definition—an optimal algorithm replaces the page that will not be used for the longest time. Belady’s anomaly occurs when a page- replacement algorithm evicts a page that will be needed in the immediate future. An optimal algorithm would not have selected such a page.

1. Segmentation is similar to paging but uses variable-sized “pages.” Define two segment-replacement algorithms, one based on the FIFO page-replacement scheme and the other on the LRU page-replacement scheme. Remember that since segments are not the same size, the segment that is chosen for replacement may be too small to leave enough consecutive locations for the needed segment. Consider strategies for systems where segments cannot be relocated and strategies for systems where they can.
2. **FIFO**. Find the first segment large enough to accommodate the incoming segment. If relocation is not possible and no one segment is large enough, select a combination of segments whose memories are contiguous, which are “closest to the first of the list” and which can accommodate the new segment. If relocation is possible, rearrange the memory so that the first *N* segments large enough for the incoming segment are contiguous in memory. Add any leftover space to the free-space list in both cases.
3. **LRU**. Select the segment that has not been used for the longest period of time and that is large enough, adding any leftover space to the free space list. If no one segment is large enough, select a combination of the “oldest” segments that are contiguous in memory (if relocation is not available) and that are large enough. If relocation is available, rearrange the oldest *N* segments to be contiguous in memory and replace those with the new segment.
4. Consider a demand-paged computer system where the degree of multi- programming is currently fixed at four. The system was recently measured to determine utilization of the CPU and the paging disk. Three alternative results are shown below. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?
5. CPU utilization 13 percent; disk utilization 97 percent
6. CPU utilization 87 percent; disk utilization 3 percent
7. CPU utilization 13 percent; disk utilization 3 percent
8. Thrashing is occurring.
9. Given the disk utilization is very high(i.e, 97%) than the CPU Utilization.
10. The CPU Utilization is less (i.e.,13%)it will make system performance down which leads to trashing.
11. So, trashing occurs
12. The degree of multiprogramming cannot be increased.
13. Here, one or more processes(degree of multiprogramming) should be suspended to allow increase in the utilization of the CPU.
14. Paging is not helping because processes are spending most of their time for paging.
15. CPU utilization is sufficiently high to leave things alone, and increase degree of multiprogramming.
16. Given CPU utilization 87% which is sufficiently high and the system is well utilized, CPU is being kept busy most of the time.
17. The degree of multiprogramming probably should stay the same,
18. Any increase in the degree of multiprogramming may decrease the CPU utilization and could lead to thrashing.
19. Paging is helping. As the CPU is kept busy, the number of processes increases in a system that supports paging.
20. Increase the degree of multiprogramming
21. Given CPU Utilization is less (i.e.,13%)it will make system performance down. The CPU is not getting enough work.
22. The degree of multiprogramming should be increased. The CPU is available for executing additional processes.
23. Paging is not really helping
24. We have an operating system for a machine that uses base and limit registers, but we have modified the machine to provide a page table. Can the page table be set up to simulate base and limit registers? How can it be, or why can it not be?

The page table can be set up to simulate base and limit registers provided that the memory is allocated in fixed-size segments. In this way, the base of a segment can be entered into the page table and the valid/invalid bit used to indicate that portion of the segment as resident in the memory. There will be some problem with internal fragmentation.

1. Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

* TLB miss with no page fault
* TLB miss with page fault
* TLB hit with no page fault
* TLB hit with page fault

TLB miss with no page fault page has been brought into memory, but has been removed from the TLB

TLB miss and page fault page fault has occurred

TLB hit and no page fault page is in memory and in the TLB. Most likely a recent reference.  
TLB hit and page fault cannot occur. The TLB is a cache of the page table. If an entry is not in the page table, it will not be in the TLB.

1. A simplified view of thread states is ***ready***, ***running***, and ***blocked***, where a thread is either ready and waiting to be scheduled, is running on the processor, or is blocked (for example, waiting for I/O).

A picture containing drawing, clock

Description automatically generated

Assuming a thread is in the ***running*** state, answer the following questions, and explain your answers:

* 1. Will the thread change state if it incurs a page fault? If so, to what state will it change?
  2. Will the thread change state if it generates a TLB miss that is resolved in the page table? If so, to what state will it change?
  3. Will the thread change state if an address reference is resolved in the page table? If so, to what state will it change?
* On a page fault the thread state is set to blocked as an I/O operation is required to bring the new page into memory.
* On a TLB-miss, the thread continues running if the address is resolved in the page table.
* The thread will continue running if the address is resolved in the page table.

1. Consider a system that uses pure demand paging.
   1. When a process first starts execution, how would you characterize the page-fault rate?
   2. Once the working set for a process is loaded into memory, how would you characterize the page-fault rate?
   3. Assume that a process changes its locality and the size of the new working set is too large to be stored in available free memory. Identify some options system designers could choose from to handle this situation.
2. Initially quite high as needed pages are not yet loaded into memory.
3. It should be quite low as all necessary pages are loaded into memory.
4. (1) Ignore it;

(2) get more physical memory; (3) reclaim pages more aggressively due to the high page fault rate.

1. The following is a page table for a system with 12-bit virtual and physical addresses and 256-byte pages. Free page frames are to be allocated in the order 9, F, D. A dash for a page frame indicates that the page is not in memory.

|  |  |
| --- | --- |
| Page | Page Frame |
| 0 | 0x4 |
| 1 | 0xB |
| 2 | 0xA |
| 3 | - |
| 4 | - |
| 5 | 0x2 |
| 6 | - |
| 7 | 0x0 |
| 8 | 0xC |
| 9 | 0x1 |

Convert the following virtual addresses to their equivalent physical addresses in hexadecimal. All numbers are given in hexadecimal. In the case of a page fault, you must use one of the free frames to update the page table and resolve the logical address to its corresponding physical address.

* 0x2A1
* 0x4E6
* 0x94A
* 0x316

Virtual size: 12 bits

Page size: 256 bytes = 28 bytes

|  |  |
| --- | --- |
| 4 bit | page offset |
| page # |  |

| <------------- 12 -------------> |

* 1. 0x2A1

|  |  |
| --- | --- |
| 0010 | 1010 0001 |

Page number = 2

Frame number = 0xA

Frame number = 10

Offset = A1 = 160+1 = 161

Physical address = 161+10=170

A1+A = AB

* 1. 0x4E6

|  |  |
| --- | --- |
| 0100 | 1110 0110 |

Page number = 4 (miss)

Frame number = 0xF

Physical address = 0xF + 0xE6 = 0x0F5

|  |  |
| --- | --- |
| Page | Page Frame |
| 0 | 0x4 |
| 1 | 0xB |
| 2 | 0xA |
| 3 | - |
| 4 | 0xF |
| 5 | 0x2 |
| 6 | - |
| 7 | 0x0 |
| 8 | 0xC |
| 9 | 0x1 |

* 1. 0x94A

|  |  |
| --- | --- |
| 1001 | 0100 1010 |

Page number = 9 (hit)

Frame number = 0x1

Physical address = 0x1 + 0x4A = 0x4B

* 1. 0x316

|  |  |
| --- | --- |
| 0011 | 0001 0110 |

Page number = 3 (miss)

Frame number = 0x9

Physical address = 0x9 + 0x16 = 0x1F

|  |  |
| --- | --- |
| Page | Page Frame |
| 0 | 0x4 |
| 1 | 0xB |
| 2 | 0xA |
| 3 | 0x9 |
| 4 | 0xF |
| 5 | 0x2 |
| 6 | - |
| 7 | 0x0 |
| 8 | 0xC |
| 9 | 0x1 |

1. What is the copy-on-write feature, and under what circumstances is its use beneficial? What hardware support is required to implement this feature?

When two processes are accessing the same set of program values (for instance, the code segment of the source binary), then it is useful to map the corresponding pages into the virtual address spaces of the two programs in a write-protected manner. When a write does indeed take place, then a copy must be made to allow the two programs to individually access the different copies without interfering with each other. The hardware support required to implement is simply the following: on each memory access, the page table needs to be consulted to check whether the page is write protected. If it is indeed write protected, a trap would occur and the operating system could resolve the issue.

1. A certain computer provides its users with a virtual memory space of bytes. The computer has bytes of physical memory. The virtual memory is implemented by paging, and the page size is 4,096 bytes. A user process generates the virtual address 11123456. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

The virtual address in binary form is

0001 0001 0001 0010 0011 0100 0101 0110

Since the page size is , the page table size is . Therefore the low order 12 bits “0100 0101 0110” are used as the displacement into the page, while the remaining 20 bits “0001 0001 0001 0010 0011” are used as the displacement in the page table.

1. Assume that we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty frame is available or if the replaced page is not modified and 20 milliseconds if the replaced page is modified. Memory-access time is 100 nanoseconds.

Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?

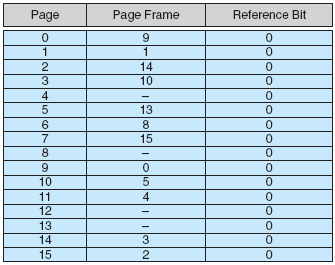
1. Consider the page table for a system with 16-bit virtual and physical addresses and 4,096-byte pages.

|  |  |  |
| --- | --- | --- |
| Page | Page Frame | Reference Bit |
| 0 | 9 | 0 |
| 1 | - | 0 |
| 2 | 10 | 0 |
| 3 | 15 | 0 |
| 4 | 6 | 0 |
| 5 | 13 | 0 |
| 6 | 8 | 0 |
| 7 | 12 | 0 |
| 8 | 7 | 0 |
| 9 | - | 0 |
| 10 | 5 | 0 |
| 11 | 4 | 0 |
| 12 | 1 | 0 |
| 13 | 0 | 0 |
| 14 | - | 0 |
| 15 | 2 | 0 |

The reference bit for a page is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates that the page is not in memory. The page-replacement algorithm is localized LRU, and all numbers are provided in decimal.

* 1. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.
  + 0x621C
  + 0xF0A3
  + 0xBC1A
  + 0x5BAA
  + 0x0BA1
  1. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault
  2. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?

1. The following page table is for a system with 16-bit virtual and physical addresses and with 4,096-byte pages. The reference bit is set to 1 when the page has been referenced. Periodically, a thread zeroes out all values of the reference bit. A dash for a page frame indicates the page is not in memory. The page- replacement algorithm is localized LRU, and all numbers are provided in decimal.



1. Convert the following virtual addresses (in hexadecimal) to the equivalent physical addresses. You may provide answers in either hexadecimal or decimal. Also set the reference bit for the appropriate entry in the page table.
   * 0xE12C
   * 0x3A9D
   * 0xA9D9
   * 0x7001
   * 0xACA1
2. Using the above addresses as a guide, provide an example of a logical address (in hexadecimal) that results in a page fault.
3. From what set of page frames will the LRU page-replacement algorithm choose in resolving a page fault?
   1. 0xE12C → 0x312C

0x3A9D → 0xAA9D

0xA9D9 → 0x59D9

0x7001 → 0xF001

0xACA1 → 0x5CA1

* 1. The only choices are pages 4, 8, 12, and 13. Thus, example addresses include anything that begins with the hexadecimal sequence 0x4..., 0x8..., 0xC..., and 0xD....
  2. Any page table entries that have a reference bit of zero. This includes the following frames {9, 1, 14, 13, 8, 0, 4}

1. When a page fault occurs, the process requesting the page must block while waiting for the page to be brought from disk into physical memory. Assume that there exists a process with five user-level threads and that the mapping of user threads to kernel threads is many to one. If one user thread incurs a page fault while accessing its stack, would the other user threads belonging to the same process also be affected by the page fault—that is, would they also have to wait for the faulting page to be brought into memory? Explain.

Yes, because there is only one kernel thread for all user threads, that kernel thread blocks while waiting for the page fault to be resolved. Since there are no other kernel threads for available user threads, all other user threads in the process are thus affected by the page fault.

**FIFO**: first in first out

**Optimal**: replace the page that will not be used for the longest period of time

**LRU**: replace the page that has not been used for the longest period of time

1. Apply the (1) FIFO, (2) LRU, and (3) optimal (OPT) replacement algorithms for the following page-reference strings:

* 2,6,9,2,4,2,1,7,3,0,5,2,1,2,9,5,7,3,8,5
* 0,6,3,0,2,6,3,5,2,4,1,3,0,6,1,4,2,3,5,7
* 3,1,4,2,5,4,1,3,5,2,0,1,1,0,2,3,4,5,0,1
* 4,2,1,7,9,8,3,5,2,6,8,1,0,7,2,4,1,3,5,8
* 0,1,2,3,4,4,3,2,1,0,0,1,2,3,4,4,3,2,1,0

Indicate the number of page faults for each algorithm assuming demand paging with **three** **frames**.

* 1. FIFO = 18 page faults

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** |
| 2 | 2 | 2 | 4 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 9 | 9 | 9 | 3 | 3 | 3 |
|  | 6 | 6 | 6 | 2 | 2 | 2 | 3 | 3 | 3 | 2 | 2 | 2 | 5 | 5 | 5 | 8 | 8 |
|  |  | 9 | 9 | 9 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 7 | 7 | 7 | 5 |

LRU = 17 page faults

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
| 2 | 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 7 | 7 | 7 | 5 |
|  | 6 | 6 | 4 | 4 | 7 | 7 | 7 | 5 | 5 | 5 | 9 | 9 | 9 | 3 | 3 | 3 |
|  |  | 9 | 9 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 5 | 5 | 5 | 8 | 8 |

OPT = 13 page faults

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 9 | 9 | 3 | 3 |
|  | 6 | 6 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 7 | 7 | 8 |
|  |  | 9 | 9 | 9 | 7 | 3 | 0 | 5 | 5 | 5 | 5 | 5 |

**1  4  2  3  3  2 | 0  6  4  5 | 3  1  2  4 | 2  6  7  3 | 4  2  6  3  5  6**

FIFO = 18 page faults

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 6 | 6 | 6 | 6 | 5 | 5 |
|  | 4 | 4 | 4 | 4 | 6 | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 7 | 7 | 7 | 7 | 6 |
|  |  | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 4 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
|  |  |  | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 5 | 4 | 4 | 4 | 4 | 2 | 2 | 2 |

LRU = 19 page faults

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 3 | 6 | 6 | 6 | 6 | 2 | 2 | 2 |
|  | 4 | 4 | 4 | 4 | 6 | 6 | 6 | 6 | 1 | 1 | 1 | 1 | 7 | 7 | 7 | 7 | 6 | 6 |
|  |  | 2 | 2 | 2 | 2 | 2 | 5 | 5 | 5 | 5 | 4 | 4 | 4 | 3 | 3 | 3 | 3 | 3 |
|  |  |  | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 2 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 5 |

OPT = 12 page faults

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 6 | 7 | 6 | 6 |
|  | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 |
|  |  | 2 | 2 | 0 | 6 | 5 | 2 | 2 | 2 | 2 | 2 |
|  |  |  | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

1. Consider the following page reference string:

7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1

Using the FIFO, LRU, and Optimal page replacement algorithms with 3 frames, show the page load order and number of page faults that would occur.

FIFO = 17 page faults

7;72;723;123;153;154;654;674;671;071;051;054;654;624;623;023;013

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** |
| 7 | 7 | 7 | 1 | 1 | 1 | 6 | 6 | 6 | 0 | 0 | 0 | 6 | 6 | 6 | 0 | 0 |
|  | 2 | 2 | 2 | 5 | 5 | 5 | 7 | 7 | 7 | 5 | 5 | 5 | 2 | 2 | 2 | 1 |
|  |  | 3 | 3 | 3 | 4 | 4 | 4 | 1 | 1 | 1 | 4 | 4 | 4 | 3 | 3 | 3 |

LRU = 18 page faults

7;72;723;123;125;325;345;346;746;716;710;510;540;546;246;236;230;130

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** |
| 7 | 7 | 7 | 1 | 1 | 3 | 3 | 3 | 7 | 7 | 7 | 5 | 5 | 5 | 2 | 2 | 2 | 1 |
|  | 2 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 1 | 1 | 1 | 4 | 4 | 4 | 3 | 3 | 3 |
|  |  | 3 | 3 | 5 | 5 | 5 | 6 | 6 | 6 | 0 | 0 | 0 | 6 | 6 | 6 | 0 | 0 |

OPT = 13 page faults

7;72;723;123;153;154;156;157;150;140;160;120;130

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| 7 | 7 | 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
|  | 2 | 2 | 2 | 5 | 5 | 5 | 5 | 5 | 4 | 6 | 2 | 3 |
|  |  | 3 | 3 | 3 | 4 | 6 | 7 | 0 | 0 | 0 | 0 | 0 |

1. A page-replacement algorithm should minimize the number of page faults. We can achieve this minimization by distributing heavily used pages evenly over all of memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages associated with that frame. Then, to replace a page, we can search for the page frame with the smallest counter.
   1. Define a page-replacement algorithm using this basic idea. Specifically address these problems:
   * What is the initial value of the counters?
   * When are counters increased?
   * When are counters decreased?
   * How is the page to be replaced selected?
   1. How many page faults occur for your algorithm for the following reference string with four page frames?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2.

* 1. What is the minimum number of page faults for an optimal page- replacement strategy for the reference string in part b with four page frames?

1. Define a page-replacement algorithm addressing the problems of”
   * 1. Initial value of the counters – 0
     2. Counters are increased – whenever a new page is associated with that frame
     3. Counters are decreased – whenever one of the pages associated with that frame is no longer required
     4. How the page to be replaced is selected – find a frame with the smallest counter. Use FIFO for breaking ties.
2. 14 page faults
3. 11 page faults
4. Assume that you are monitoring the rate at which the pointer in the clock algorithm moves. (The pointer indicates the candidate page for replacement.) What can you say about the system if you notice the following behavior:
   1. Pointer is moving fast.
   2. Pointer is moving slow.

If the pointer is moving fast, then the program is accessing a large number of pages simultaneously. It is most likely that during the period between the point at which the bit corresponding to a page is cleared and it is checked again, the page is accessed again and therefore cannot be replaced. This results in more scanning of the pages before a victim page is found. If the pointer is moving slow, then the virtual memory system is finding candidate pages for replacement extremely efficiently, indicating that many of the resident pages are not being accessed.

1. Discuss situations in which the least frequently used (LFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

Consider the following sequence of memory accesses in a system that can hold four pages in memory: 1 1 2 3 4 5 1. When page 5 is accessed, the least frequently used page-replacement algorithm would replace a page other than 1, and therefore would not incur a page fault when page 1 is accessed again. On the other hand, for the sequence “1 2 3 4 5 2,” the least recently used algorithm performs better.

1. Discuss situations in which the most frequently used (MFU) page-replacement algorithm generates fewer page faults than the least recently used (LRU) page-replacement algorithm. Also discuss under what circumstances the opposite holds.

Consider the sequence in a system that holds four pages in memory: 1 2 3 4 4 4 5 1. The most frequently used page replacement algorithm evicts page 4while fetching page 5, while the LRU algorithm evicts page 1. This is unlikely to happen much in practice. For the sequence “1 2 3 4 4 4 5 1,” the LRU algorithm makes the right decision.

1. The KHIE (pronounced “k-hi”) operating system uses a FIFO replacement algorithm for resident pages and a free-frame pool of recently used pages. Assume that the free-frame pool is managed using the LRU replacement policy. Answer the following questions:
   1. If a page fault occurs and the page does not exist in the free-frame pool, how is free space generated for the newly requested page?
   2. If a page fault occurs and the page exists in the free-frame pool, how are the resident page set and the free-frame pool managed to make space for the requested page?
   3. To what does the system degenerate if the number of resident pages is set to one?
   4. To what does the system degenerate if the number of pages in the free-frame pool is zero?
2. Consider a demand-paging system with the following time-measured utilizations:

|  |  |
| --- | --- |
| CPU utilization  Paging disk  Other I/O devices | 20%  97.7%  5% |

For each of the following, indicate whether it will (or is likely to) improve CPU utilization. Explain your answers.

* 1. Install a faster CPU.
  2. Install a bigger paging disk.
  3. Increase the degree of multiprogramming.
  4. Decrease the degree of multiprogramming.
  5. Install more main memory.
  6. Install a faster hard disk or multiple controllers with multiple hard disks.
  7. Add prepaging to the page-fetch algorithms.
  8. Increase the page size.

The system obviously is spending most of its time paging, indicating over-allocation of memory. If the level of multiprogramming is reduced resident processes would page fault less frequently and the CPU utilization would improve. Another way to improve performance would be to get more physical memory or a faster paging drum.

1. Install a faster CPU—No.
2. Install a bigger paging disk—No.
3. Increase the degree of multiprogramming—No.
4. Decrease the degree of multiprogramming—Yes.
5. Install more main memory—Likely to improve CPU utilization as more pages can remain resident and not require paging to or from the disks.
6. Install a faster hard disk or multiple controllers with multiple hard disks—Also an improvement, for as the disk bottleneck is removed by faster response and more throughput to the disks, the CPU will get more data more quickly.
7. Add prepaging to the page fetch algorithms—Again, the CPU will get more data faster, so it will be more in use. This is only the case if the paging action is amenable to prefetching (i.e., some of the access is sequential).
8. Increase the page size—Increasing the page size will result in fewer page faults if data is being accessed sequentially. If data access is more or less random, more paging action could ensue because fewer pages can be kept in memory and more data is transferred per page fault. So this change is as likely to decrease utilization as it is to increase it.
9. Explain why minor page faults take less time to resolve than major page faults.

**Minor page faults**

If the page fault handler running within the kernel is able to successfully map a physical page for the user-space virtual address that triggered the page-fault, this is known as minor page fault. Assuming that there is an abundant amount of physical memory freely available, accessing buffers allocated by **malloc()** would be satisfied by the Linux kernel via minor page faults.

**Major page faults**

Major faults occur on one of the following possible occasions:

* The page fault handler could be trying to reclaim physical pages that were swapped out earlier due to memory pressure (shortage of physical memory), thus incurring disk I/O in order to read from previously *swapped-out* page from the configured swap-space.
* The page fault handler could be trying to read from an open file that was just **mmap()**'ed**.**This would also incur disk I/O if the file contents were already not within the page cache.

In **summary**, for a page request triggered by page fault from user-space, if the page fault handler would satisfy this request without incurring disk I/O, it is treated as a minor page fault. But if the page fault handler has to incur disk I/O to satisfy the page request - it is treated as a major page fault.

**What a Minor page faults?**

There is also a special case scenario called a minor page fault which occurs when the code (or data) needed is actually already in memory, but it isn't allocated to that process. For example, if a user is running a web browser then the memory pages with the browser executable code can be shared across multiple users (since the binary is read-only and can't change). If a second user starts the same web browser then Linux won't load all the binary again from disk, it will map the shareable pages from the first user and give the second process access to them. In other words, a minor page fault occurs only when the page list is updated (and the MMU configured) without actually needing to access the disk.

minor page fault, 指的就是CPU要执行的指令实际上已经在物理内存page中了， 只是这个page没有被分配给当前进程, 这时CPU就会raise一个minor page fault, 让MMU把这个page分配给当前进程使用, 因此minor page fault并不需要去访问磁盘.

1. Explain why compressed memory is used in operating systems for mobile devices.

mobile systems generally do not support either standard swapping or swapping pages.

Memory compression is a memory management technique that reduces the size of inactive data in the random access memory (RAM) to free up unused space and allow more programs to run at once. ... Memory compression can be applied to computers, smartphones and embedded systems.

1. Suppose that a machine provides instructions that can access memory locations using the one-level indirect addressing scheme. What sequence of page faults is incurred when all of the pages of a program are currently nonresident and the first instruction of the program is an indirect memory-load operation? What happens when the operating system is using a per-process frame allocation technique and only two pages are allocated to this process?

The following page faults take place: page fault to access the instruction, a page fault to access the memory location that contains a pointer to the target memory location, and a page fault when the target memory location is accessed. The operating system will generate three page faults with the third page replacing the page containing the instruction. If the instruction needs to be fetched again to repeat the trapped instruction, then the sequence of page faults will continue indefinitely. If the instruction is cached in a register, then it

will be able to execute completely after the third page fault.

1. Consider the page references:

A close up of a piece of paper

Description automatically generated

What pages represent the locality at time (X)?

1. Suppose that your replacement policy (in a paged system) is to examine each page regularly and to discard that page if it has not been used since the last examination. What would you gain and what would you lose by using this policy rather than LRU or second-chance replacement?

Such an algorithm could be implemented with the use of a reference bit. After every examination, the bit is set to zero; set back to one if the page is referenced. The algorithm would then select an arbitrary page for replacement from the set of unused pages since the last examination.

The advantage of this algorithm is its simplicity—nothing other than a reference bit need be maintained. The disadvantage of this algorithm is that it ignores locality by using only a short time frame for determining whether to evict a page or not. For example, a page may be part of the working set of a process, but may be evicted because it was not referenced since the last examination (that is, not all pages in the working set may be referenced between examinations).

1. Consider a demand-paging system with a paging disk that has an average access and transfer time of 20 milliseconds. Addresses are translated through a page table in main memory, with an access time of 1 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference if the page-table entry is in the associative memory.

Assume that 80 percent of the accesses are in the associative memory and that, of those remaining, 10 percent (or 2 percent of the total) cause page faults. What is the effective memory access time?

effective access time

1. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

Thrashing is caused by underallocation of the minimum number of pages required by a process, forcing it to continuously page fault. The system can detect thrashing by evaluating the level of CPU utilization as compared to the level of multiprogramming. It can be eliminated by reducing the level of multiprogramming.

1. Is it possible for a process to have two working sets, one representing data and another representing code? Explain.

Yes, in fact many processors provide two TLBs for this very reason. As an example, the code being accessed by a process may retain the same working set for a long period of time. However, the data the code accesses may change, thus reflecting a change in the working set for data accesses.

1. Consider the parameter Δ used to define the working-set window in the working-set model. When Δ is set to a low value, what is the effect on the page-fault frequency and the number of active (non-suspended) processes currently executing in the system? What is the effect when Δ is set to a very high value?

When Δ is set to a small value, then the set of resident pages for a process might be underestimated, allowing a process to be scheduled even though all of its required pages are not resident. This could result in a large number of page faults. When Δ is set to a large value, then a process’s resident set is overestimated and this might prevent many processes from being scheduled even though their required pages are resident. However, once a process is scheduled, it is unlikely to generate page faults since its resident set has been overestimated.

1. In a 1,024-KB segment, memory is allocated using the buddy system. Using Figure 10.26 as a guide, draw a tree illustrating how the following memory requests are allocated:

* Request 5-KB
* Request 135 KB
* Request 14 KB
* Request 3 KB
* Request 12 KB

Next, modify the tree for the following releases of memory. Perform coalescing whenever possible:

* Request 3 KB
* Request 5 KB
* Request 14 KB
* Request 12 KB

The following allocation is made by the Buddy system: The 240-byte request is assigned a 256-byte segment. The 120-byte request is assigned a 128-byte segment, the 60-byte request is assigned a 64-byte segment and the 130-byte request is assigned a 256-byte segment. After the allocation, the following segment sizes are available: 64-bytes, 256-bytes, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, and 512K.

After the releases of memory, the only segment in use would be a 256-byte segment containing 130 bytes of data. The following segments will be free: 256 bytes, 512 bytes, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, and 512K.

1. A system provides support for user-level and kernel-level threads. The mapping in this system is one to one (there is a corresponding kernel thread for each user thread). Does a multithreaded process consist of (a) a working set for the entire process or (b) a working set for each thread? Explain

A working set for each thread. This is because each kernel thread has its own execution sequence, thus generating its unique sequence of addresses.

1. The slab-allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn’t scale well with multiple CPUs. What could be done to address this scalability issue?

This has long been a problem with the slab allocator—poor scalability with multiple CPUs. The issue comes from having to lock the global cache when it is being access. This has the effect of serializing cache accesses on multiprocessor systems. Solaris has addressed this by introducing a per-CPU cache, rather than a single global cache.

1. Consider a system that allocates pages of different sizes to its processes. What are the advantages of such a paging scheme? What modifications to the virtual memory system would be needed to provide this functionality?

The program could have a large code segment or use large-sized arrays as data. These portions of the program could be allocated to larger pages, thereby decreasing the memory overheads associated with a page table. The virtual memory system would then have to maintain multiple free lists of pages for the different sizes and also needs to have more complex code for address translation to take into account different page sizes.

s

# Chapter 12 Main Memory – Exercises

1. State three advantages of placing functionality in a device controller, rather than in the kernel. State three disadvantages.
2. Bugs are harder to fix — a new firmware version or new hardware is needed
3. Improving algorithms likewise require a hardware update rather than just a kernel or device-driver update
4. Embedded algorithms could conflict with application’s use of the device, causing decreased performance.

Advantages:

* Bugs are less likely to cause an operating system crash.
* Performance can be improved by utilizing dedicated hardware and hard-coded algorithms.
* The kernel is simplified by moving algorithms out of it.

Disadvantages:

* Bugs are harder to fix - a new firmware version or new hardware is needed.
* Improving algorithms likewise require a hardware update rather than just a kernel or device driver update.
* Embedded algorithms could conflict with application’s use of the device, causing de- creased performance.

1. The example of handshaking in Section 12.2 used two bits: a busy bit and a command-ready bit. Is it possible to implement this handshaking with only one bit? If it is, describe the protocol. If it is not, explain why one bit is insufficient.

It is possible, using the following algorithm. Let’s assume we simply use the busy-bit (or the command-ready bit; this answer is the same regardless). When the bit is off, the controller is idle. The host writes to data-out and sets the bit to signal that an operation is ready (the equivalent of setting the command-ready bit). When the controller is finished, it clears the busy bit. The host then initiates the next operation. This solution requires that both the host and the controller have read and write access to the same bit, which can complicate circuitry and increase the cost of the controller.

1. Why might a system use interrupt-driven I/O to manage a single serial port and polling I/O to manage a front-end processor, such as a terminal concentrator?

Polling can be more efficient than interrupt-driven I/O. This is the case when the I/O is frequent and of short duration. Even though a single serial port will perform I/O relatively infrequently and should thus use interrupts, a collection of serial ports such as those in a terminal concentrator can produce a lot of short I/O operations, and interrupting for each one could create a heavy load on the system. A well-timed polling loop could alleviate that load without wasting many resources through looping with no I/O needed.

1. Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping, and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a computing environment in which that strategy is more efficient than either of the others.

A hybrid approach could switch between polling and interrupts depending on the length of the I/O operation wait. For example, we could poll and loop N times, and if the device is still busy at N+1, we could set an interrupt and sleep. This approach would avoid long busy-waiting cycles. This method would be best for very long or very short busy times. It would be inefficient it the I/O completes at N+T (where T is a small number of cycles) due to the overhead of polling plus setting up and catching interrupts.

Pure polling is best with very short wait times. Interrupts are best with known long wait times.

1. How does DMA increase system concurrency? How does it complicate hardware design?

DMA increases system concurrency by allowing the CPU to perform tasks while the DMA system transfers data via the system and memory buses. Hardware design is complicated because the DMA controller must be integrated into the system, and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and DMA controller to share use of the memory bus.

1. Why is it important to scale up system-bus and device speeds as CPU speed increases?

Consider a system which performs 50% I/O and 50% computes. Doubling the CPU performance on this system would increase total system performance by only 50%. Doubling both system aspects would increase performance by 100%. Generally, it is important to remove the current system bottleneck, and to increase overall system performance, rather than blindly increasing the performance of individual system components.

1. Distinguish between a driver end and a stream module in a STREAMS operation.

The STREAMS driver controls a physical device that could be involved in a STREAMS operation. The STREAMS module modifies the flow of data between the head (the user interface) and the driver.

1. When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts.

A number of issues need to be considered in order to deter- mine the priority scheme to be used to determine the order in which the interrupts need to be serviced. First, interrupts raised by devices should be given higher priority than traps generated by the user pro- gram; a device interrupt can therefore interrupt code used for handling system calls. Second, interrupts that control devices might be given higher priority than interrupts that simply perform tasks such as copying data served up a device to user/kernel buffers, since such tasks can always be delayed. Third, devices that have real-time constraints on when their data is handled should be given higher priority than other devices. Also, devices that do not have any form of buffering for its data would have to be assigned higher priority since the data could be available only for a short period of time.

1. What are the advantages and disadvantages of supporting memory-mapped I/O to device-control registers?

The advantage of supporting memory-mapped I/O to device-control registers is that it eliminates the need for special I/O instructions from the instruction set and therefore also does not require the enforcement of protection rules that prevent user programs from executing these I/O instructions. The disadvantage is that the resulting flexibility needs to be handled with care; the memory translation units need to ensure that the memory addresses associated with the device control registers are not accessible by user programs in order to ensure protection.

1. Consider the following I/O scenarios on a single-user PC:
   1. A mouse used with a graphical user interface
   2. A tape drive on a multitasking operating system (with no device preallocation available)
   3. A disk drive containing user files
   4. A graphics card with direct bus connection, accessible through memory-mapped I/O

For each of these scenarios, would you design the operating system to use buffering, spooling, caching, or a combination? Would you use polled I/O or interrupt-driven I/O? Give reasons for your choices.

1. A mouse used with a graphical user interface  
   Buffering may be needed to record mouse movement during times when higher-priority operations are taking place. Spooling and caching are inappropriate. Interrupt-driven I/O is most appropriate.
2. A tape drive on a multitasking operating system (assume no device preallocation is available)  
   Buffering may be needed to manage throughput difference be- tween the tape drive and the source or destination of the I/O. Caching can be used to hold copies of data that resides on the tape, for faster access. Spooling could be used to stage data to the device when multiple users desire to read from or write to it. Interrupt-driven I/O is likely to allow the best performance.
3. A disk drive containing user files  
   Buffering can be used to hold data while in transit from user space to the disk, and vice versa. Caching can be used to hold disk-resident data for improved performance. Spooling is not necessary because disks are shared-access devices. Interrupt-driven I/O is best for devices such as disks that transfer data at slow rates.
4. A graphics card with direct bus connection, accessible through memory-mapped I/O  
   Buffering may be needed to control multiple access and for performance (double-buffering can be used to hold the next screen image while displaying the current one). Caching and spooling are not necessary due to the fast and shared-access natures of the device. Polling and interrupts are useful only for input and for I/O completion detection, neither of which is needed for a memory-mapped device.
5. In most multiprogrammed systems, user programs access memory through virtual addresses, while the operating system uses raw physical addresses to access memory. What are the implications of this design for the initiation of I/O operations by the user program and their execution by the operating system?

The user program typically specifies a buffer for data to be transmitted to or from a device. This buffer exists in user space and is specified by a virtual address. The kernel needs to issue the I/O operation and needs to copy data between the user buffer and its own kernel buffer before or after the I/O operation. In order to access the user buffer, the kernel needs to translate the virtual address provided by the user program to the corresponding physical address within the context of the user program’s virtual address space. This translation is typically performed in software and therefore incurs overhead. Also, if the user buffer is not currently present in physical memory, the corresponding page(s) need to obtained from the swap space. This operation might require careful handling and might delay the data copy operation.

1. What are the various kinds of performance overhead associated with servicing an interrupt?

When an interrupt occurs the currently executing process is interrupted and its state is stored in the appropriate process control block. The interrupt service routine is then dispatched in order to deal with the interrupt. On completion of handling of the interrupt, the state of the process is restored and the process is resumed. Therefore, the performance overheads include the cost of saving and restoring process state and the cost of flushing the instruction pipeline and restoring the instructions into the pipeline when the process is restarted.

1. Describe three circumstances under which blocking I/O should be used. Describe three circumstances under which nonblocking I/O should be used. Why not just implement nonblocking I/O and have processes busy-wait until their devices are ready?

Generally, blocking I/O is appropriate when the process will be waiting only for one specific event. Examples include a disk, tape, or keyboard read by an application program. Non-blocking I/O is useful when I/O may come from more than one source and the order of the I/O arrival is not predetermined. Examples include network daemons listening to more than one network socket, window managers that accept mouse movement as well as keyboard input, and I/O-management programs, such as a copy command that copies data between I/O de- vices. In the last case, the program could optimize its performance by buffering the input and output and using non-blocking I/O to keep both devices fully occupied.

Non-blocking I/O is more complicated for programmers, because of the asynchronous rendezvous that is needed when an I/O occurs. Also, busy waiting is less efficient than interrupt-driven I/O so the overall system performance would decrease.

1. Typically, at the completion of a device I/O, a single interrupt is raised and appropriately handled by the host processor. In certain settings, however, the code that is to be executed at the completion of the I/O can be broken into two separate pieces. The first piece executes immediately after the I/O completes and schedules a second interrupt for the remaining piece of code to be executed at a later time. What is the purpose of using this strategy in the design of interrupt handlers?

The purpose of this strategy is to ensure that the most critical aspect of the interrupt handling code is performed first and the less critical portions of the code are delayed for the future. For instance, when a device finishes an I/O operation, the device-control operations corresponding to declaring the device as no longer being busy are more important in order to issue future operations. However, the task of copying the data provided by the device to the appropriate user or kernel memory regions can be delayed for a future point when the CPU is idle. In such a scenario, a lower-priority interrupt handler is used to perform the copy operation.

1. Some DMA controllers support direct virtual memory access, where the targets of I/O operations are specified as virtual addresses and a translation from virtual to physical address is performed during the DMA. How does this design complicate the design of the DMA controller? What are the advantages of providing such functionality?

Direct virtual memory access allows a device to perform a transfer from two memory-mapped devices without the intervention of the CPU or the use of main memory as a staging ground; the device simply issues memory operations to the memory-mapped addresses of a target device and the ensuing virtual address translation guarantees that the data is transferred to the appropriate device. This functionality, however, comes at the cost of having to support virtual address translation on addresses accessed by a DMA controller and requires the addition of an address-translation unit to the DMA controller. The address translation results in both hardware and software costs and might also result in coherence problems between the data structures maintained by the CPU for address translation and corresponding structures used by the DMA controller. These coherence issues would also need to be dealt with and results in further increase in system complexity.

1. UNIX coordinates the activities of the kernel I/O components by manipulating shared in-kernel data structures, whereas Windows uses object-oriented message passing between kernel I/O components. Discuss three pros and three cons of each approach.

3 pros of the UNIX method:

Very efficient, low overhead and low amount of data movement. Fast implementation—no coordination needed with other kernel components. Simple, so less chance of data loss

3 cons:

No data protection, and more possible side effects from changes so more difficult to debug. Difficult to implement new I/O methods: new data structures needed rather than just new objects. Complicated kernel I/O subsystem, full of data structures, access routines, and locking mechanisms

1. Write (in pseudocode) an implementation of virtual clocks, including the queueing and management of timer requests for the kernel and applications. Assume that the hardware provides three timer channels.

A screenshot of text

Description automatically generated

1. Discuss the advantages and disadvantages of guaranteeing reliable transfer of data between modules in the STREAMS abstraction.

Reliable transfer of data requires modules to check whether space is available on the target module and to block the sending module if space is not available. This check requires extra communication between the modules, but the overhead enables the system to pro- vide a stronger abstraction than one which does not guarantee reliable transfer. The stronger abstraction typically reduces the complexity of the code in the modules. In the STREAMS abstraction, however, there is unreliability introduced by the driver end, which is allowed to drop messages if the corresponding device cannot handle the incoming data. Consequently, even if there is reliable transfer of data between the modules, messages could be dropped at the device if the corresponding buffer fills up. This requires retransmission of data and special code for handling such retransmissions, thereby somewhat limiting the advantages that are associated with reliable transfer between the modules.

### ETH Assignment10 – I/O Systems

The Linux operating system differentiates between character and block devices. What is the difference between them?

**Answer**:

Accesses to block devices are cached and buffered, while character device accesses are not. Block devices must allow random access.

**1.2 DMA**

How does DMA increase system concurrency? How does it complicate hardware design?

**Answer**:

DMA increases system concurrency by allowing the CPU to perform tasks while the DMA system transfers data via the system and memory buses. Hardware design is complicated because the DMA controller must be integrated into the system and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and DMA controller to share use of the memory bus.

Although DMA does not use the CPU, the maximum transfer rate is still limited. Consider reading a block from disk. Name three factors that might ultimately limit the file transfer

**Answer**:

There are four ways that the maximum transfer rate can be limited:

* Limiting speed of the I/O device - in our case, the disk read throughput.
* Limiting speed of the bus. In this case the bus itself is the bottleneck.
* A disk controller with no internal buffers or too small buffer sizes could also limit the performance of the read file operation.
* Erroneous disk or block read (i.e, if the checksum is incorrect). In this case, an error is signaled and no transfer of the block happens. The block has to be retransmitted.

A DMA controller has four channels. The controller is capable of requesting a 32-bit word every 100 nsec. A response takes equally long. How fast does the bus have to be to avoid being a bottleneck?

**Answer**:

Each bus transaction has a request and a response each taking 100 nsec, or 200 nsec per bus transaction. This gives 5 million bus transactions / sec. If each one is four bytes, the bus should be able to handle 20 MB/sec. The fact that these transactions may be distributed over four I/O devices (four channels) in round-robin fashion is irrelevant. A bus transaction takes 200 nsec, regardless of whether consecutive requests are to the same device or different device, so the number of channels the DMA controller has does not matter.

# Chapter 13 File System Interface – Exercises

1. **13.1**Some systems automatically delete all user files when a user logs off or a job terminates, unless the user explicitly requests that they be kept. Other systems keep all files unless the user explicitly deletes them. Discuss the relative merits of each approach.

Deleting all files not specifically saved by the user has the advantage of minimizing the file space needed for each user by not saving unwanted or unnecessary files. Saving all files unless specifically deleted is more secure for the user in that it is not possible to lose files inadvertently by forgetting to save them.

1. **13.2**Why do some systems keep track of the type of a file, while still others leave it to the user and others simply do not implement multiple file types? Which system is “better”?

Some systems allow different file operations based on the type of the file (for instance, an ascii file can be read as a stream while a database file can be read via an index to a block). Other systems leave such interpretation of a file’s data to the process and provide no help in accessing the data. The method that is “better” depends on the needs of the processes on the system, and the demands the users place on the operating system. If a system runs mostly database applications, it may be more efficient for the operating system to implement a database- type file and provide operations, rather than making each program implement the same thing (possibly in different ways). For general- purpose systems it may be better to only implement basic file types to keep the operating system size smaller and allow maximum freedom to the processes on the system.

1. **13.3**Similarly, some systems support many types of structures for a file’s data, while others simply support a stream of bytes. What are the advantages and disadvantages of each approach?

An **advantage** of having the system support different file structures is that the support comes from the system; individual applications are not required to provide the support. In addition, if the system provides the support for different file structures, it can implement the support presumably more efficiently than an application.

The **disadvantage** of having the system provide support for defined file types is that it increases the size of the system. In addition, applications that may require different file types other than what is provided by the system may not be able to run on such systems.

An alternative strategy is for the operating system to define no support for file structures and instead treat all files as a series of bytes. This is the approach taken by UNIX systems. The advantage of this approach is that it simplifies the operating system support for file systems, as the system no longer has to provide the structure for different file types. Furthermore, it allows applications to define file structures, thereby alleviating the situation where a system may not provide a file definition required for a specific application.

1. **13.4**Could you simulate a multilevel directory structure with a single-level directory structure in which arbitrarily long names can be used? If your answer is yes, explain how you can do so, and contrast this scheme with the multilevel directory scheme. If your answer is no, explain what prevents your simulation’s success. How would your answer change if file names were limited to seven characters?

If arbitrarily long names can be used then it is possible to simulate a multilevel directory structure. This can be done, for example, by using the character “.” to indicate the end of a subdirectory. Thus, for example, the name *jim.java.F1* specifies that *F1* is a file in subdirectory *java* which in turn is in the root directory *jim*.

If file names were limited to seven characters, then the above scheme could not be utilized and thus, in general, the answer is *no*. The next best approach in this situation would be to use a specific file as a symbol table (directory) to map arbitrarily long names (such as *jim.java.F1*) into shorter arbitrary names (such as *XX00743*), which are then used for actual file access.

1. **13.5**Explain the purpose of the open() and close() operations.

The purpose of the open() and close() operations is:

* The open() operation informs the system that the named file is about to become active.
* The close() operation informs the system that the named file is no longer in active use by the user who issued the close operation.

1. **13.6**In some systems, a subdirectory can be read and written by an authorized user, just as ordinary files can be.
   1. Describe the protection problems that could arise.
   2. Suggest a scheme for dealing with each of these protection problems.
2. One piece of information kept in a directory entry is file location. If a user could modify this location, then he could access other files defeating the access-protection scheme.
3. Do not allow the user to directly write onto the subdirectory. Rather, provide system operations to do so.
4. **13.7**Consider a system that supports 5,000 users. Suppose that you want to allow 4,990 of these users to be able to access one file.
   1. How would you specify this protection scheme in UNIX?
   2. Can you suggest another protection scheme that can be used more effectively for this purpose than the scheme provided by UNIX?
5. There are two methods for achieving this:
   1. Create an access control list with the names of all 4990 users.
   2. Put these 4990 users in one group and set the group access accordingly. This scheme cannot always be implemented since user groups are restricted by the system.
6. The universal access to files applies to all users unless their name appears in the access-control list with different access permission. With this scheme you simply put the names of the remaining ten users in the access control list but with no access privileges allowed.
7. **13.8**Researchers have suggested that, instead of having an access-control list associated with each file (specifying which users can access the file, and how), we should have a **user control list** associated with each user (specifying which files a user can access, and how). Discuss the relative merits of these two schemes.

* *File Control List.*

Since the access control information is concentrated in one single place, it is easier to change access control information and this requires less space.

* *User Control List.*

This requires less overhead when opening a file.

1. **13.9**Consider a file system in which a file can be deleted and its disk space reclaimed while links to that file still exist. What problems may occur if a new file is created in the same storage area or with the same absolute path name? How can these problems be avoided?

Let F1 be the old file and F2 be the new file. A user wishing to access F1 through an existing link will actually access F2. Note that the access protection for file F1 is used rather than the one associated with F2.

This problem can be avoided by insuring that all links to a deleted

file are deleted also. This can be accomplished in several ways:

1. maintain a list of all links to a file, removing each of them when the file is deleted
2. retain the links, removing them when an attempt is made to access a deleted file
3. maintain a file reference list (or counter), deleting the file only after all links or references to that file have been deleted
4. **13.10**The open-file table is used to maintain information about files that are currently open. Should the operating system maintain a separate table for each user or maintain just one table that contains references to files that are currently being accessed by all users? If the same file is being accessed by two different programs or users, should there be separate entries in the open-file table? Explain.

The solution would be keeping a central open-file table, the OS can perform the following operation that would not be possible otherwise. Consider a file that is currently being accessed by one or more processes. If the file is deleted, then it should not be removed from the disk until all processes accessing the file have closed it. This check can be performed only if there is a centralized accounting of number of processes accessing the file. On the other hand, if two processes are accessing the same file, then two separate states need to be maintained to keep track of the current location of which parts of the file are being accessed by the two processes. This requires the operating system to maintain separate entries for the two processes.

1. **13.11**What are the advantages and disadvantages of providing mandatory locks instead of advisory locks whose use is left to users’ discretion?

Advantages**:**

Different programs working concurrently and trying to access same file at such situation  result in mutual exclusion, mandatory locking will guarantee memory locks to provide synchronization.

Files would limit the flexibility for files to be accessed.

Disadvantages**:**

Increase in overhead associated with files.

1. **13.12**Provide examples of applications that typically access files according to the following methods:

* Sequential
* Random

Sequential: Application that access documents successively incorporate word processors, video players, music players and web administrations.  
Random: Application that access documents arbitrarily incorporate databases, video and sound editors.

1. **13.13**Some systems automatically open a file when it is referenced for the first time and close the file when the job terminates. Discuss the advantages and disadvantages of this scheme compared with the more traditional one, where the user has to open and close the file explicitly.

Programmed opening and shutting of documents remembers the client from the innovation of these function, and in this manner makes it more helpful to the client, however, it requires more overhead than the situation where unequivocal opening and closing is required.

1. **13.14**If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?

At the point when block is accessed, file system could prefetch the consequent blocks in suspicion of future requests to these blocks. This prefetching advancement word elicit the holding up time experienced by the process for future solicitations.

1. **13.15**Give an example of an application that could benefit from operating-system support for random access to indexed files.

An application that keep up a database of sections could profit by such support. Case in point, if a project is keep up an Employee database, then access to the database can't be displayed by any foreordained access design. The access to records are random and finding the records would be more proficient if the working framework were to give some type of tree base index

1. **13.16**Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

With a single copy, several concurrent updates to a file may result in user

obtaining incorrect information, and the file being left in an incorrect state.

With multiple copies, there is storage waste and the various copies may not be

consistent with respect to each other.

# Chapter 14 File System Implementation – Exercises

1. **14.1**Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grow at the end. Also assume that the block information to be added is stored in memory.
2. The block is added at the beginning.
3. The block is added in the middle.
4. The block is added at the end.
5. The block is removed from the beginning.
6. The block is removed from the middle.
7. The block is removed from the end.

The results are:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Contiguous | Linked | Indexed |
|  | 201 | 1 | 1 |
|  | 101 | 52 | 1 |
|  | 1 | 3 | 1 |
|  | 198 | 1 | 0 |
|  | 98 | 52 | 0 |
|  | 0 | 100 | 0 |

1. **14.2**Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

In case of system crash (memory failure) the free-space list would not be lost as it would be if the bit map had been stored in main memory.

1. **14.3**Consider a system that supports the strategies of contiguous, linked, and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?
   * **Contiguous**—if file is usually accessed sequentially, if file is relatively small.
   * **Linked**—if file is large and usually accessed sequentially.
   * **Indexed**—if file is large and usually accessed randomly.
2. **14.4**One problem with contiguous allocation is that the user must preallocate enough space for each file. If the file grows to be larger than the space allocated for it, special actions must be taken. One solution to this problem is to define a file structure consisting of an initial contiguous area of a specified size. If this area is filled, the operating system automatically defines an overflow area that is linked to the initial contiguous area. If the overflow area is filled, another overflow area is allocated. Compare this implementation of a file with the standard contiguous and linked implementations.

This method requires more overhead then the standard contiguous allocation. It requires less overhead than the standard linked allocation.

1. **14.5**How do caches help improve performance? Why do systems not use more or larger caches if they are so useful?

Caches allow components of differing speeds to communicate more efficiently by storing data from the slower device, temporarily, in a faster device (the cache). Caches are, almost by definition, more expensive than the device they are caching for, so increasing the number or size of caches would increase system cost.

1. **14.6**Why is it advantageous to the user for an operating system to dynamically allocate its internal tables? What are the penalties to the operating system for doing so?

**Answer:**

Dynamic tables allow more flexibility in system use growth — tables are never exceeded, avoiding artificial use limits. Unfortunately, kernel structures and code are more complicated, so there is more potential for bugs. The use of one resource can take away more system resources (by growing to accommodate the requests) than with static tables.

1. **14.7**Consider a file system that uses a modified contiguous-allocation scheme with support for extents. A file is a collection of extents, with each extent corresponding to a contiguous set of blocks. A key issue in such systems is the degree of variability in the size of the extents. What are the advantages and disadvantages of the following schemes?
2. All extents are of the same size, and the size is predetermined.
3. Extents can be of any size and are allocated dynamically.
4. Extents can be of a few fixed sizes, and these sizes are predetermined.

**Answer:**

If all extents are of the same size, and the size is predetermined, then it simplifies the block allocation scheme. A simple bit map or free list for extents would suffice. If the extents can be of any size and are allocated dynamically, then more complex allocation schemes are required. It might be difficult to find an extent of the appropriate size and there might be external fragmentation. One could use the Buddy system allocator discussed in the previous chapters to design an appropriate allocator. When the extents can be of a few fixed sizes, and these sizes are predetermined, one would have to maintain a separate bitmap or free list for each possible size. This scheme is of intermediate complexity and of intermediate flexibility in comparison to the earlier schemes.

OR

1. Advantages: No external fragmentation, since we can always reuse the space occupied by a deleted extent. With this scheme, the number of extents is a function of the number of blocks in the file, rather than a product of the current fragmentation state of the disk.  
   Disadvantages: Internal fragmentation, which will be severe if extents are large. If extents are the same size as a disk block, this system degenerates into an indexed scheme.
2. Advantages: Little internal fragmentation, as only the last extent of a file will contain a block that is not completely filled. Some files (specifically, the first ones created on the disk) will be stored contiguously, so access to them will be fast.

Disadvantages: After some time, the free space on the disk will become fragmented. In the worst case, each free block will be in its own extent, and reading a file will require a seek for each block, making access slow.

1. This is similar to the “buddy system” of dynamic memory allocation. (See section 9.8.1.)

Advantages: As long as files rarely exceed the size of the largest extent, this allocation scheme is fast because the file system will attempt to place files into a single extent that is just large enough to hold it.

Disadvantages: Internal fragmentation will occur, since files will rarely fit exactly into an extent. Since the number and sizes of extents are fixed, the system may eventually run short of smaller extents, resulting in lots of internal fragmentation when large extents are used for small files. On the other hand, the system could run short of large extents, reducing internal fragmentation, but causing files to be scattered across the disk.

1. **14.8**Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.

**Answer:**

|  |  |  |
| --- | --- | --- |
|  | **Sequential** | **Random** |
| **Contiguous** | Works very well as the file is stored contiguously. Sequential access will simply involves traversing the contiguous disk blocks. | Works very well as you can easily determine the adjacent disk block containing the position you wish to seek to. |
| **Linked** | Satisfactory as you are simply following the links from one block to the next. | Poor as it may require following the links to several disk blocks until you arrive at the intended seek point of the file. |
| **Indexed** | Works well as sequential access simply involves sequentially accessing each index. | Works well as it is easy to determine the index associated with the disk block containing the position you wish to seek to. |

1. **14.9**What are the advantages of the variant of linked allocation that uses a FAT to chain together the blocks of a file?

* A section of disk at the beginning of each volume is set a side to contain the table. The table has one entry for each disk block and is indexed by block number.
* The directory entry contains the block number of the first block of file. The table entry indexed by that block number contains the block number of the next block in file. This chain continues until the last block, which has a special-end-of file value as the table entry. Unused blocks are indicated by a 0 table value. Advantage is that random-access time is improved, because the disk head can find the location of any block by reading the information in FAT
* The advantage is that while accessing a block that is stored at the middle of a file, its location can be determined by chasing the pointers stored in the FAT as opposed to accessing all of the individual blocks of the file in a sequential manner to find the pointer to the target block.

OR

While accessing a block that is stored at the middle of a file, a location can be found by chasing the pointer stored in the FAT as opposed to accessing all of the file blocks separately in a sequential manner to find the pointer to the target block.

1. **14.10**Consider a system where free space is kept in a free-space list.
2. Suppose that the pointer to the free-space list is lost. Can the system reconstruct the free-space list? Explain your answer.
3. Consider a file system similar to the one used by UNIX with indexed allocation. How many disk I/O operations might be required to read the contents of a small local file at /a/b/c? Assume that none of the disk blocks is currently being cached.
4. Suggest a scheme to ensure that the pointer is never lost as a result of memory failure.
5. Yes, but the amount of time required will grow linearly with the number of files in the file system. To reconstruct the free-space list, the OS must scan the file system beginning at the root. As each directory and file is scanned, the OS marks the associated disk blocks as used. (Notice that this includes not only file data, but any file-control blocks and index blocks associated with the file, as well as blocks (both data and meta-data) used by directories. In addition, blocks occupied by volume- and boot-control blocks must be accounted for.)

One problem with this scheme is that some OS’s deal with damaged (unreadable and/or unwritable) disk blocks simply by removing them from the free list. The scanning technique described above would identify these blocks as free, since they are not part of the file system structure. The danger here is that these blocks might be allocated to a file, even though we know they are bad. Thus, the OS would need to keep a separate list of these blocks on disk. Early versions of Unix solved this problem by allocating these bad blocks to a special file whose only purpose was to be composed of bad blocks, and placing this file in a special directory. This directory (“/lost+found”) still exists on some systems.

1. Remember that in Unix, directories are just regular files that contain a list of file/directory names and pointers to FCB’s (inodes). In the best case, reading each directory or file requires three disk block accesses: one to read the inode for the file or directory, one to read the index block pointed to by the inode, and then one to read the first data block of file or directory itself.

If we assume we know the address of the root inode when we begin, then 12 reads are required:

1. read the root inode,  
2. read the index block of the root directory, 3. read the root directory,  
4. read a’s inode,  
5. read a’s index block,  
6. read directory a,  
7. read b’s inode,

8. read b’s index block,

9. read directory b,  
10. read c’s inode,  
11. read c’s index block, 12. read first block of file c.

If we assume a double-indirect block scheme, than in the worse case, four reads are required to get the appropriate block of a directory or file into memory: read the inode, read the double indirect block, read the indirect block, read the data block. This gives us a total of 16 reads.

1. Store the pointer to the free space list in several different locations on disk.
2. **14.11** Some file systems allow disk storage to be allocated at different levels of granularity. For instance, a file system could allocate 4 KB of disk space as a single 4-KB block or as eight 512-byte blocks. How could we take advantage of this flexibility to improve performance? What modifications would have to be made to the free-space management scheme in order to support this feature?

Such a scheme would decrease internal fragmentation. If a file is 5KB, then it could be allocated a 4KB block and two contiguous 512-byte blocks. In addition to maintaining a bitmap of free blocks, one would also have to maintain extra state

regarding which of the subblocks are currently being used inside a block. The allocator would then have to examine this extra state to allocate subblocks and coalesce the subblocks to obtain the larger block when all of the subblocks become free.

1. **14.12** Discuss how performance optimizations for file systems might result in difficulties in maintaining the consistency of the systems in the event of computer crashes.

When parts of the file system are cached, blocks that are in the cache but which have not yet written to disk would be lost in a crash. Imagine that a new file is created consisting of one block. If the file-control block is written to disk, but the data block is not, the file-control block will point to a disk block of garbage after a crash.

Suppose that free space is managed using a bitmap. The system could allocate a new block to a file, write the file to disk, update the bitmap in memory, but then crash before the bitmap can be written to the disk. This scenario requires some type of consistency checker (e.g. “fsck” in Unix and “chkdsk” in NT.)

OR

One primary difficulty that might arise is due to delayed updates of data and metadata. Updates could be delayed hoping that the same data might be updated in the future or that the updated data might be temporary and might be deleted in the near future. However, if the system crashes without committing the delayed updates, the consistency of the file system is destroyed.

1. **14.13**Discuss the advantages and disadvantages of supporting links to files that cross mount points (that is, the file link refers to a file that is stored in a different volume).

Advantages and Disadvantages of supporting Links to Files that Cross Mount Points:

         From the perspective of the user, they do not need to be alluded to the fact that the file exists on a different volume. The disadvantage is that when the related volume is not mounted, opening the file would raise an error to the user making it apparent that it is a dead link and sses file system boundaries. Inter-volume linking is only a convenience if the host volume is always available to provide the linked file.

OR

the main advantage is there may be more disk space on the other mount point. This is usually done with symbolic links, which has the disadvantage that renaming the file will not have the effect you expect or when creating a file with a similar name in the same location, the file will be created on the first file system. Also, renaming files across the different mount points may fail, e.g. if the file is renamed from the symlinked file to the local dir.

1. **14.14**Consider a file system on a disk that has both logical and physical block sizes of 512 bytes. Assume that the information about each file is already in memory. For each of the three allocation strategies (contiguous, linked, and indexed), answer these questions:
2. How is the logical-to-physical address mapping accomplished in this system? (For the indexed allocation, assume that a file is always less than 512 blocks long.)
3. If we are currently at logical block 10 (the last block accessed was block 10) and want to access logical block 4, how many physical blocks must be read from the disk?

**Answer:**

1. Let Z be the starting file address (block number).

**–**Contiguous. Divide the logical address by 512 with X and Y the resulting quotient and remainder respectively. Add X to Z to obtain the physical block number. Y is the displacement into that block.

**–**Linked. Divide the logical physical address by 511 with X and Y the resulting quotient and remainder respectively. Chase down the linked list (getting X + 1 blocks). Y + 1 is the displacement into the last physical block.

**–**Indexed. Divide the logical address by 512 with X and Y the resulting quotient and remainder respectively. Get the index block into memory. Physical block address is contained in the index block at location X. Y is the displacement into the desired physical block.

1. **–**Contiguous. 1

**–**Linked. 4

**–**Indexed. 2

1. **14.15**Consider a file system that uses inodes to represent files. Disk blocks are 8 KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

Disk block size = 8KB = 213 bytes

Pointer size = 4 bytes

# of direct disk blocks = 12

# of pointers/blocks = 8K / 4 = 2048

1. **14.16**Fragmentation on a storage device can be eliminated through compaction. Typical disk devices do not have relocation or base registers (such as those used when memory is to be compacted), so how can we relocate files? Give three reasons why compacting and relocating files are often avoided.

To compact disk blocks, we need to read each file and its associated FCB into memory, free up the blocks that the file occupies on disk, locate a contiguous section of free blocks large enough to hold the file data, rewrite the data blocks, and then update the FCB.

* 1. It’s time consuming, since each change requires two block accesses – a read of the original block followed by a write to its new location. Suppose transferring a 1K block requires 0.1 microsecond. (That is, the disk has a transfer rate of 10 MB / sec) Then processing a disk with a 100 GB file system would require

This assumes that the compaction algorithm would only have to move a file once.

* 1. If a disk is nearly full, we may have to read many files into memory before we can free up enough contiguous space to rewrite even one of them. These files may not fit in memory, or they may have to be read and re-written several times.
  2. If a crash occurs while the disk is being compacted, some data may be lost, since the file system is in an inconsistent state while a file is being moved. What would happen if the system crashes while the root directory of the file system or the file that contains the kernel is being relocated?

1. **14.17**Explain why logging metadata updates ensures recovery of a file system after a file-system crash.

In the case of a system crash, the system will be able to determine all metadata updates that did not finish. Using this information, it will be able to redo those updates to keep the file system in a consistent state.

1. **14.18**Consider the following backup scheme:  
   • **Day1**. Copy to a backup medium all files from the disk.

• **Day2**. Copy to another medium all files changed since day1.

• **Day3**. Copy to another medium all files changed since day1.

This differs from the schedule given in Section 14.7.4 by having all subsequent backups copy all files modified since the first full backup. What are the benefits of this system over the one in Section 14.7.4? What are the drawbacks? Are restore operations made easier or more difficult? Explain your answer.

Section 14.7.4:

Benefits: less storage space needed, since less changes to files

Drawbacks: when we need to restore, it’s more time consuming. We need to restore all levels up to full backup.

Given backup scheme:

This is incremental backup (it is a backup of all changes since the last backup)

Benefits of this system are:

If we need a full recovery , we can restore easily in less time .

Less number of storage devices or media are required.

Almost two backup media are required to restore all the data. It simplifies restoration process.

***OR***

Several different kinds of backup methods can be used, often in concert with each other:

**Full**: Backup all files on the system. (This is what you are doing on day 1, in both the methods.)

**Incremental**: Backup all files that have changed since the last incremental or full backup. ( This is what you are talking in Section 11.7.4)

**Differential**: Backup all files that have changed since the last full backup. (This is what you are talking in your recent method.)

**Multiple level incremental**: Backup all files that have changed since the previous backup at the same or a previous level.

**User**: Backup only files in a specific user's directory.

Advantages and disadvantages of the technique you tried:

**DisAdv:** As you move for the subsequent days, the amount of data that is backed up by the differential backup will increase because the changes accumulates, whereas, in incremental backups, the increase may not be to that extent, as you backup only the data that differs from that of the previous incremental backup.

**DisAdv:** If your data is very volatile, your differential backup might back as much data as the full backup. Here, it is really time consuming, when compared to that of incremental backup.

**Adv:** The important advantage of this technique is, you only need 2 disks, the most recent differential backup disc, and the full backup disc, to restore fully. Whereas, in incremental backups, you need all the intermediate discs, from the most recent incremental backup, to the most recent full backup. Its like stepping back, each and every step carefully.

Restore operations are made easier or difficult with this technique: Definitely we can say that restore operations, are made easier, as you need only 2 discs, to restore, and there are no intermediate stages, to restore, as opposed to that of incremental backups. In incremental backups, you have to step back every step, whereas, here, it is like you are moving back to a safe position of the beginning.

1. **14.19**Discuss the advantages and disadvantages of associating with remote file systems (stored on file servers) a set of failure semantics different from those associated with local file systems.

Advantages and Disadvantages of associating with Remote File Systems:

The advantage is that there is greater transparency in the sense that the user does not need to be aware of mount points and create links in all scenarios. The disadvantage however is that the file system containing the link might be mounted while the file system containing the target file might not be, and therefore one cannot provide transparent access to the file in such a scenario; the error condition would expose to the user that a link is a dead link and that the link does indeed cross file system boundaries.

1. **14.20**What are the implications of supporting UNIX consistency semantics for shared access to files stored on remote file systems?

Implications of supporting UNIX Consistency semantics:

Consistency semantics requires updates to a file to be immediately available to other processes. Supporting such a semantics for shared files on remote file systems could result in the following inefficiencies: all updates by a client have to be immediately reported to the file server instead of being batched (or even ignored if the updates are to a temporary file), and updates have to be communicated by the file server to clients caching the data immediately, again resulting in more communication.

What problems could occur if a system allowed a file system to be mounted simultaneously at more than one location?

**Answer:**

There would be multiple paths to the same file, which could confuse users or encourage mistakes (deleting a file with one path deletes the file in all the other paths).

Some systems provide file sharing by maintaining a single copy of a file. Other systems maintain several copies, one for each of the users sharing the file. Discuss the relative merits of each approach.

Relative Merits of given each Approach :

   With a single shared copy, several concurrent updates to the file may have an unintended effect but the file will not be left in an incoherent state – which can happen if multiple unresolvable copies of the same file were to exist. Furthermore, maintaining independent mirrors of a file would result in a waste of storage.

# Berkeley 2019 Fall Final

1. A process may request the operating system to perform a privileged action on its behalf by issuing a/an system call .
2. Stride Scheduling (or CFS) is a scheduling algorithm that is based on a similar principle as Lottery Scheduling, but is not based on randomness.
3. The subset of its virtual address space that a process is actively using over a given time interval is called a/an working set .
4. To evict a page to disk, the operating system finds which page table entries map to pages that are not recently used .
5. A transaction is a/an atomic sequence of reads/writes.
6. A group of n machines can maintain safety and liveness in the face of fail-stop failures as long as at least a majority of (or ) node(s) is/are not faulty.
7. Components of a file system
   1. What are the four components of a file system?

**Solution**: Directory Structure, Index, Storage Blocks, and Free Blocks

* 1. Which component(s) of a file system might the kernel access to handle an open system call? Assume the file path corresponds to a regular file that already exists (not an I/O device, directory, or soft link).

**Solution**: Directory Structure and Index

1. In the Redo Logging scheme discussed in class, explain why it is not necessary to undo any operations in the log when recovering after a crash.

**Solution: All operations written to the log are idempotent, so it is safe to redo them even if they were executed before the crash or in a previous recovery attempt.**

1. The concept of *dispatch* – passing a request to the appropriate request handler for processing – arises in a variety of systems. Give two examples of dispatch *in software systems* that we studied in class.

**Solution**: Any two of the following would receive full credit: interrupt vector table, system call handler, RPC/HTTP server, and Linux VFS. Other examples are vtables (polymorphism), opcode dispatch in interpreters, and protocol dispatch in networks.

1. Before you build a buffer cache, you run measurements and find that on average it takes 10 ms to perform each file operation. With the buffer cache, you can perform an operation in 1 ms when it does not have to go to disk. What fraction of the operations must be serviced by the buffer cache for the average operation time to be below 2 ms?

**Solution:** . We also accepted , as the problem can also be read as including the 1 ms on cache misses.

1. How will your buffer cache influence the I/O queuing delays experienced by applications?

**Solution**: It will decrease I/O queueing delays because the bandwidth is higher and the latency lower for the buffer cache.

1. In distributed file systems, like AFS and NFS, which component is responsible for providing consistency?

**Solution**: The server is responsible for notifying clients when data they may have cached has been overwritten by another client.

1. When either the client or server closes a connection socket, how is the other party notified? How do they reach agreement to close?

**Solution**: One notifies the other with a FIN message. They agree by each sending a FIN-ACK.

1. In order for complex data objects to be passed as arguments or results in an RPC over a socket to a possibly remote machine, what needs to be done to the data?

**Solution**: It needs to be serialized into a canonical form that can be reconstructed by the other participant.

1. What should a client do after establishing a TCP connection with a server, but before sending user identity and password over that connection?

**Solution**: It should (1) verify the identity of the server it is connecting to, and (2) establish a secure channel over the TCP connection.

---

1. False. Accesses to kernel memory (PHYS\_BASE and above) in the system call handler and interrupt handlers operate directly on physical addresses, bypassing all page tables.

# Chapter 15 File System Internals – Exercises

1. **15.1**Explain how the VFS layer allows an operating system to support multiple types of file systems easily.

**Answer:**VFS introduces a layer of indirection in the file system implementation. In many ways, it is similar to object-oriented programming techniques. System calls can be made generically (independent of file system type). Each file system type provides its function calls and data structures to the VFS layer. A system call is translated into the proper specific functions for the target file system at the VFS layer. The calling program has no file-system-specific code, and the upper levels of the system call structures likewise are file system-independent. The translation at the VFS layer turns these generic calls into file-system-specific operations.

1. **15.2**Why have more than one file system type on a given system?
2. **15.3**On a Unix or Linux system that implements the procfs file system, determine how to use the procfs interface to explore the process name space. What aspects of processes can be viewed via this interface? How would the same information be gathered on a system lacking the procfs file system?
3. **15.4**Why do some systems integrate mounted file systems into the root file system naming structure, while others use a separate naming method for mounted file systems?
4. **15.5**Given a remote file access facility such as ftp, why were remote file systems like NFS created?
5. **15.6**Assume that in a particular augmentation of a remote-file-access pro- tocol, each client maintains a name cache that caches translations from file names to corresponding file handles. What issues should we take into account in implementing the name cache?
6. **15.7**Given a mounted file system with write operations underway, and a system crash or power loss, what must be done before the file system is remounted if: (a) The file system is not log-structured? (b) The file system is log-structured?
7. **15.8**Why do operating systems mount the root file system automatically at boot time?
8. **15.9**Why do operating systems require file systems other than root to be mounted?