
**Efe Kerem Kesgin**

**21902857**

**Cs 342**

**HomeWork 2**

# Q1

a) In a single disk block, we can store 4096 bytes / 128 bytes/inode = 32 inodes.

b) We need to store 1024 inodes, so we need 1024 inodes / 32 inodes/block = 32 blocks to store the inode table.

c) To store 1024 inodes, we need 1024 inodes / 8 bits/byte = 128 bytes. Therefore, we need 128 bytes * 8 bits/byte = 1024 bits in the inode map.

d) To store the inode map, we need 1024 bits / 4096 bits/block = 0.25 blocks. Rounded up to the nearest integer, we need 1 block to store the inode map.

e) To store the block bitmap for the 1000 blocks on the disk, we need 1000 blocks / 8 bits/byte = 125 bytes. Therefore, we need 125 bytes * 8 bits/byte = 1000 bits in the block bitmap.

f) To store the block bitmap, we need 1000 bits / 4096 bits/block = 0.244 blocks. Rounded up to the nearest integer, we need 1 block to store the block bitmap.

g) In a single block, we can store 4096 bytes / (50 bytes/entry + 4 bytes/entry) = 85 directory entries.

# Q2

The size of the disk is 64 GB, or 64 * 1024 MB = 65536 MB.

Each block is 4 KB, or 4 * 1024 bytes = 4096 bytes.

Therefore, there are 65536 MB / 4096 bytes/block = 16384 blocks on the disk.

The size of the FAT table in bytes is equal to the number of blocks on the disk, since each entry in the FAT table corresponds to a block on the disk and each entry is 8 bytes long. Therefore, the size of the FAT table is 16384 blocks * 8 bytes/block = 131072 bytes.

The FAT table will occupy 131072 bytes / 4096 bytes/block = 32 blocks.

# Q3

For the single level page table scheme, the memory space required to hold all page table information for the two processes can be calculated as follows:

The virtual address space for each process is 2^36 bytes, and the size of each page is 2^14 bytes. Therefore, there are 2^22 pages in the virtual address space for each process.

The size of each page table entry is 8 bytes, so the size of the page table for each process is 2^22 pages * 8 bytes/page = 2^24 bytes.

The total memory space required to hold all page table information for the two processes is therefore 2 * 2^24 bytes = 2^25 bytes.

For the two-level page table scheme, the memory space required to hold all page table information for the two processes can be calculated as follows:

The virtual address space for each process is divided into pages of size 2^14 bytes, and each page is mapped to a physical address using a page table entry (PTE). The size of a PTE is 8 bytes.

The virtual address space for each process is divided into 2^11 pages (2^11 bits for p1 and 2^11 bits for p2).

The size of the page table for each process is therefore 2^11 pages * 8 bytes/page = 2^14 bytes.

The total memory space required to hold all page table information for the two processes is therefore 2 * 2^14 bytes = 2^15 bytes.

For the inverted page table scheme, the memory space required to hold all page table information for the two processes can be calculated as follows:

The virtual address space for each process is divided into pages of size 2^14 bytes, and each page is mapped to a physical address using an inverted page table entry.

The virtual address space for each process is divided into 2^22 pages.

The size of the inverted page table for each process is therefore 2^22 pages * 8 bytes/page = 2^24 bytes.

The total memory space required to hold all page table information for the two processes is therefore 2 * 2^24 bytes = 2^25 bytes.

# Q4

At the start, all three frames are empty and no page faults have occurred yet.

Page reference: 3

Memory state: [3] [] []

No page fault

Page reference: 5

Memory state: [3, 5] [] []

No page fault

Page reference: 4

Memory state: [3, 5, 4] [] []

No page fault

Page reference: 3

Memory state: [3, 5, 4] [] []

No page fault (3 is already in memory)

Page reference: 5

Memory state: [3, 5, 4] [] []

No page fault (5 is already in memory)

Page reference: 6

Memory state: [3, 5, 6] [] []

Page fault (6 is not in memory)

Page reference: 2

Memory state: [3, 2, 6] [] []

Page fault (2 is not in memory)

Page reference: 5

Memory state: [3, 2, 5] [] []

No page fault (5 is already in memory)

Page reference: 2

Memory state: [3, 2, 5] [] []

No page fault (2 is already in memory)

Page reference: 3

Memory state: [3, 2, 5] [] []

No page fault (3 is already in memory)

Page reference: 4

Memory state: [3, 2, 4] [] []

No page fault (4 is already in memory)

Page reference: 2

Memory state: [3, 2, 4] [] []

No page fault (2 is already in memory)

Page reference: 5

Memory state: [3, 2, 5] [] []

No page fault (5 is already in memory)

Page reference: 4

Memory state: [3, 2, 4] [] []

No page fault (4 is already in memory)

Page reference: 2

Memory state: [3, 2, 4] [] []

No page fault (2 is already in memory)

Page reference: 7

Memory state: [3, 2, 7] [] []

Page fault (7 is not in memory)

Page reference: 4

Memory state: [3, 4, 7] [] []

No page fault (4 is already in memory)

Page reference: 7

Memory state: [3, 7, 4] [] []

No page fault (7 is already in memory)

Page reference: 3

Memory state: [3, 7, 4] [] []

No page fault (3 is already in memory)

# Q5

The maximum file size using indexed allocation with an inode that can store 10 direct pointers is 10 direct pointers * 4 bytes/pointer * 4 KB/block = 160 KB.

To access a byte at offset 0 of the file, 1 disk access would be required.

To access a byte at offset 1000000 of the file, 1 disk access would be required to read the block containing the byte.

To access a byte at offset 100000000 of the file, 10 disk accesses would be required to read the blocks containing the byte. This is because the file is larger than the direct pointers in the inode, so the indirect pointer must be used to access the blocks containing the byte. This requires 1 disk access to read the indirect pointer, and then 1 disk access for each of the 9 blocks that the indirect pointer points to.

# Q6

At the moment, there is no deadlock. This is because no process is blocked waiting for a resource that is currently held by another process.

A deadlock occurs when a process is blocked waiting for a resource that is held by another process, and the other process is blocked waiting for a resource held by the first process, forming a cycle. In this case, none of the processes are blocked waiting for a resource, so there is no deadlock.

It is possible that a deadlock may occur later if the resource requests are granted and the processes enter a state where they are blocked waiting for resources held by other processes, forming a cycle. However, at the moment there is no deadlock.