# Virtual Memory

1. Consider a processor that uses segmentation and paging (i.e., this is not a MIPS processor). Below is the segment table being used for the currently executing process and below that are pages tables for several processes in the system. Note that some processes may not use all of the available segments. Recall that VPN is the virtual page number, PFN is the physical frame number, V is the valid bit and D is the dirty bit (i.e., the page can be dirtied/modified).

| Segment | PT base addr | Max VPN Value |
|---|---|---|
| 4 | 70700000 | 3 |
| 3 | 70200000 | 3 |
| 2 | 70500000 | 3 |
| 1 | 70300000 | 3 |
| 0 | 70100000 | 3 |

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 5177 | 1 | 0 |
| 2 | 20 | 0 | 0 |
| 1 | 77 | 1 | 0 |
| 0 | 4251 | 0 | 0 |

Base addr: 70000000

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 1311 | 1 | 0 |
| 2 | 12 | 1 | 0 |
| 1 | 711 | 0 | 0 |
| 0 | 23 | 1 | 0 |

Base addr: 70700000

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 52 | 1 | 1 |
| 2 | 41 | 1 | 1 |
| 1 | 30 | 1 | 1 |
| 0 | 5177 | 0 | 1 |

Base addr: 70400000

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 65 | 1 | 1 |
| 2 | 77 | 1 | 1 |
| 1 | 567 | 1 | 1 |
| 0 | 672 | 1 | 1 |

Base addr: 70300000

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 641 | 0 | 1 |
| 2 | 753 | 1 | 1 |
| 1 | 2577 | 1 | 1 |
| 0 | 517 | 1 | 1 |

Base addr: 70200000

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 5532 | 0 | 1 |
| 2 | 5177 | 1 | 1 |
| 1 | 336 | 0 | 1 |
| 0 | 77 | 1 | 1 |

Base addr: 70600000

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 5177 | 1 | 1 |
| 2 | 34 | 1 | 1 |
| 1 | 563 | 1 | 1 |
| 0 | 1641 | 1 | 1 |

Base addr: 70500000

| VPN | PFN | V | D |
|---|---|---|---|
| 3 | 516 | 1 | 1 |
| 2 | 37 | 0 | 1 |
| 1 | 7731 | 1 | 1 |
| 0 | 6341 | 1 | 1 |

Base addr: 70100000

Assume that the processor is using 32-bits for virtual and physical addresses, that the page size is 64 KB, that all addresses (virtual and physical) and values shown in the segment table and page tables are expressed in hexadecimal, and that the system uses 4 bits for segments.

a. Explain how many bits of the virtual address will be used to represent the offset?

$2^{16}$ = 64 KB, so 16 bits for the offset

b. What is the maximum possible size of a segment in this system in bytes (expressed as an equation).

32 virtual address - 4 for segment - 16 bits for page size/offset = 12 bits for a virtual page number and each page is $2^{16}$ bytes so $2^{12} * 2^{16} = 2^{28}$ .

Or more simply

32 bit virtual address - 4 bits for the segment = 28 bits. So $2^{28}$

c. Convert the virtual address 0x20043751 into a 32-bit physical addresses (also expressed in hexadecimal). Show your work and if the address can not be translated, explain why.

The first 4 bits are the segment so the segment is 2 and the page table address is 0x70500000 (this is really irrelevant). 12 bits are used for the VPN so the VPN is 4 but the maximum VPN for that segment is 3 so this does not result in a translation because the program is accessing memory outside the range of that segment.

d. Convert the virtual address 0x30022267 into a 32-bit physical addresses (also expressed in hexadecimal). Show your work and if the address can not be translated, explain why.

The first 4 bits are 0 so the segment is 3. and the page table address is 0x70200000. 12 bits are used for the VPN so the VPN is 2. The PFN for that VPN is 753 The last 16 bits are the offset which is 2267. So the physical address is 0x0753 2267.

e. Convert the physical address 0x51773721 into a 32-bit virtual addresses (also expressed in hexadecimal). Show your work and if the address can not be translated, explain why.

16 bits are used for the offset so 16 bits are used for the frame number. So the PFN = 5177. Now find an entry in one of the page tables for the executing process with PFN = 5177. The page table with base address 70500000 (segment 2) has PFN = 5177 in VPN 3. So the VPN is 3 and the segment is 2. So the virtual address is 0x20033721.
Note that other Page tables that contain 5177 in the PFN are not part of the executing process and therefore those translations are not possible.

2. Complete the table using the CLOCK page replacement algorithm:

| Num | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Refs | a | b | c | d | a | b | e | a | c | b | e | a |
| Frame 1 | a 1 | a 1 | **a 1** | d 1 | d 1 | **d 1** | e 1 | e 1 | **e 1** | e 0 | e 1 | **e 0** |
| Frame 2 | **- 0** | b 1 | b 1 | **b 0** | a 1 | a 1 | **a 0** | **a 1** | a 0 | b 1 | b 1 | b 0 |
| Frame 3 | - 0 | **- 0** | c 1 | c 0 | **c 0** | b 1 | b 0 | b 0 | c 1 | **c 1** | **c 1** | a 1 |
| Fault? | x | x | x 1 | x | x | x | x |  | x | x |  | x |

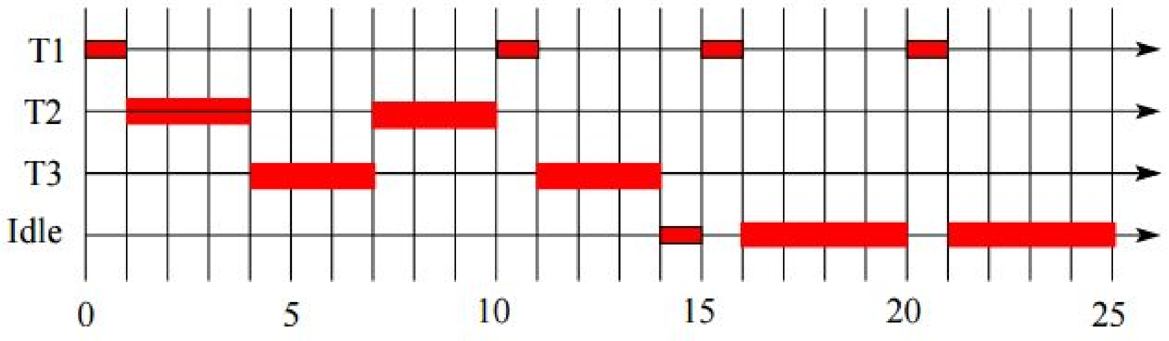
The victim is in bold and the number is the use bit after the reference has been resolved.

## Scheduling

1. Three threads are in the ready to run queue, in the order, T1, T2, and T3.
   - Thread T1 runs a loop that executes 4 times. During each loop it runs for 1 unit of time and then makes a system call that blocks for 4 units of time. After looping the program exits.
   - Threads T2, and T3 run a loop that executes 2 times. During each loop they run for 3 units of time and then call thread yield. After looping the program exits.

On a timeline, show when each thread runs and when the processor is idle by shading the appropriate thread or idle line. If two events happen at the same time, assume that T3 has the highest priority (i.e., its event occurs first) and T1 has the lowest priority.

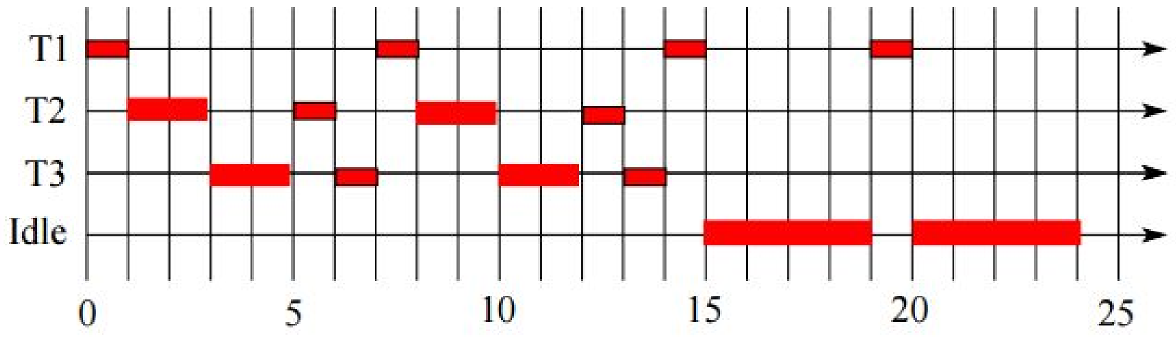
a. Assume a non preemptive FIFO scheduler and the starting point is as described at the top of this question.



One way is to track the queue at each point in time (below to simplify, we are running the thread at the front of the queue)

| Time | Queue |
|---|---|
| t0 | T1, T2, T3 |
| t1 | T2, T3 (T1 blocks) |
| t4 | T3, T2 (T2 yields, T1 is blocked) |
| t5 | T3, T2, T1 (T1 is unblocked) |
| t7 | T2, T1, T3 (T3 yields) |
| t10 | T1, T3 (T2 is finished) |
| t11 | T3 (T1 is blocked, T2 is finished) |
| t14 | Idle (T1 is blocked, T2 is finished, T3 is finished) |
| t15 | T1 (T1 is unblocked, T2 is finished, T3 is finished) |
| t16 | Idle (T1 is blocked, T2 is finished, T3 is finished) |
| t20 | T1 (T1 is unblocked, T2 is finished, T3 is finished) |
| t21 | Idle (T1 is blocked, T2 is finished, T3 is finished) |
| t25 | Idle (T1 is unblocked, T2 is finished, T3 is finished) |

b. Assume a preemptive round-robin scheduler with a quantum of 2 and that the running time for the thread is reset to zero every time the thread executes (i.e., unused quantums do not carry over to the next time the thread runs). Use the starting point as described at the top of the question.



| Time | Queue |
|---|---|
| t0 | T1, T2, T3 |
| t1 | T2, T3 (T1 is blocked) |
| t3 | T3, T2 (T2 is preempted, T1 is blocked) |
| t5 | T2, T3, T1 (T3 is preempted, T1 is unblocked) |
| t6 | T3, T1, T2 (T2 yields) |
| t7 | T1, T2, T3 (T3 yields) |
| t8 | T2, T3 (T1 is blocked) |
| t10 | T3, T2 (T2 yields, T1 is blocked) |
| t12 | T2, T3, T1 (T3 is preempted, T1 is unblocked) |
| t13 | T3, T1 (T2 is finished) |
| t14 | T1 (T2 is finished, T3 is finished) |
| t15 | Idle (T1 is blocked, T2 is finished, T3 is finished) |
| t19 | T1 (T1 is unblocked, T2 is finished, T3 is finished) |
| t20 | Idle (T1 is blocked, T2 is finished, T3 is finished) |
| t24 | Idle (T1 is finished, T2 is finished, T3 is finished) |

2. There are 3 threads in the system: T1, T2, and T3.
   - T1 has a priority of 1 and has been running for 3 time units.
   - T2 has a priority of 2 and has been running for 7 time units.
   - T3 has a priority of 3 and has been running for 6 time units.

   Which thread is scheduled to run next by the Linux Completely Fair Scheduler?

CFS selects the thread with the lowest virtual runtime. The virtual runtime for thread i is $\frac{c_i \sum_j p_j}{p_i}$. In this question, $\sum_j p_j = 1 + 2 + 3 = 6$. We can then find the virtual runtime of each thread:

T1: $\frac{3*6}{1} = 18$

T2: $\frac{7*6}{2} = 21$

T3: $\frac{6*6}{3} = 12$

Therefore, T3 will be scheduled next because it has the lowest virtual runtime.

# I/O

1. A disk drive has C cylinders, T tracks per cylinder, S sectors per track, and B bytes per sector. The rotational velocity of the platters is ω rotations per second.
    a. Consider s1 and s2, consecutive sectors on the same track of the disk. (Sector s2 will pass under the read/write head immediately after s1.) A read request for s1 arrives at the disk and is serviced. Exactly d seconds ($0 < d < 1/\omega$) after that request is completed by the disk, a read request for sector s2 arrives at the disk. (There are no intervening requests.) How long will it take the disk to service the request for sector s2?

The three parts to calculating the time to service a read request are: seek time, transfer time, and rotational latency. In this question the disk head does not change tracks, so:

seek time: 0

The next easiest part is the transfer time. We are given that the disk rotates $\omega$ times per second, so the inverse ($\frac{1}{\omega}$) is the number of seconds it takes to perform a single rotation. We are also told that there are $S$ sectors per track, each of which takes up an equal fraction of the track, so:

transfer time: $\frac{1}{S}\frac{1}{\omega}$

The last piece of the problem is to calculate the rotational latency. Because the question says that $d$ is greater than 0, we know that the disk will have to complete a full rotation to get back to the start of sector s2. As previously stated, a full rotation is $\frac{1}{\omega}$, but we do not count the portion of time before the request was actually issued, so:

rotational latency: $\frac{1}{\omega} - d$

Putting it all together:

Total service time: $\frac{1}{S}\frac{1}{\omega} + \frac{1}{\omega} - d$

    b. Suppose that s1 and s2 are not laid out consecutively on the disk. Instead, there are k sectors between s1and s2. For which value(s) of k will the time to service the read request for s2 be minimized? (Only consider $0 \le k \le S - 2$.)

The minimum time to service a read request occurs when the seek time and rotational latency are both 0. Notice that we can not make the transfer time 0, because you actually have to read the requested sector(s). Since the seek time is already 0, we only need to think about the transfer time. Idealy the request for s2 would arrive exactly as the desk head arrives at the start of s2. To make this happen, the time $d$ between finishing the read of s1 and the arrival of the new request for s2 would be exactly equal to the time it takes for the disk head to traverse the $k$ intermediate sections. Traversing $k$ sectors takes $\frac{k}{S}\frac{1}{\omega}$ seconds, and if you equate that to $d$ and solve for $k$ you get:

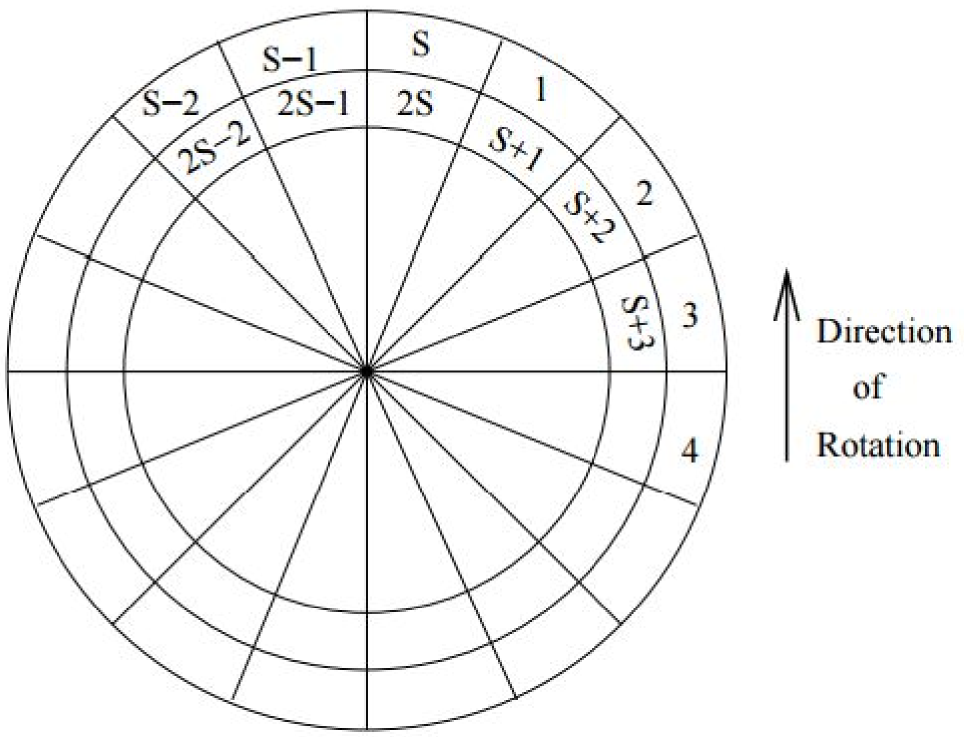
$k = dS\omega$

2. A disk drive has $S$ sectors per track and $C$ cylinders. For simplicity, we will assume that the disk has only one, single-sided platter, i.e., the number of tracks per cylinder is one. The platter spins at ω rotations per millisecond. The following function gives

the relationship between seek distance $d$, in cylinders, and seek time, $t_{seek}$, in milliseconds:

$$t_{seek} = 0 \qquad d = 0$$
$$t_{seek} = 5+0.05d \qquad 0 < d \le C$$

The sectors are laid out and numbered sequentially, starting with the outer cylinder (cylinder 0), as shown in the diagram below.



a. Suppose the disk read/write head is located over cylinder 10. The disk receives a request to read sector $S$. What is the expected service time for this request?

We are told that the disk head starts on cylinder 10 and the sector for reading is on cylinder 0, so we must seek a distance of 10:

seek time: $5 + 0.05 * 10 \ = \ 5.5$

The transfer is for a single sector, and the time to perform the transfer is equal to the fraction of a single rotation that 1 sector consumes multiplied by the time for a single rotation:

transfer time: $\frac{1}{S}\frac{1}{\omega}$

Because we are not told which sector the desk head started at on cylinder 10, it is impossible to tell which sector the desk head will be positioned over top of when the seek completes. If we assume that all of the sectors on the track are equally likely, then the fraction of a rotation needed for rotational delay is in the range from 0 to $\frac{S-1}{S}$, with all $S$

possibilities have a probability of $\frac{1}{S}$. Taking the expected value we get $\sum_{0}^{S-1} \frac{1}{S}\frac{i}{S} = \frac{S-1}{2S} \approx \frac{1}{2}$ as the expected fraction of a rotation that will be needed before sector $S$ is reached.

(expected) rotational latency: $\frac{1}{\omega}\frac{1}{2}$

Putting it all together:

Total (expected) service time: $5.5 + \frac{1}{S}\frac{1}{\omega} + \frac{1}{S}\frac{1}{2}$

b. Exactly d milliseconds after completing the request for $S$, the disk receives a request for sector $S$ + 1. What is the expected service time for this request?

Unlike the previous part of this question, we can calculate the exact time (instead of expected) because we know where the disk head started.

Both the seek and transfer times are not very exciting. The only difference is that we are only seeking by a single cylinder this time:

seek time: $5 + 0.05 * 1 \ = \ 5.05$

transfer time: $\frac{1}{S}\frac{1}{\omega}$

The tricky part of this question is calculating the rotational latency. During the seek operation, the platter is still rotating underneath of the disk head. We therefore need to determine which sector the desk head is positioned over top of when the seek is complete. The simple case to think about is when the time between finishing the read of sector $S$ and arriving at the new cylinder (which is $d + t_{seek}$) is less than one full rotation. In this case, we simply need to finish the rotation in order to arrive as section $S + 1$. In the more generally (and likely) case, the disk will have performed one or more full rotations by the time the disk head reaches the new cylinder and so we need to modulo time taken in getting to the new cylinder by $\frac{1}{\omega}$ (the time for one rotation):

rotational latency: $\frac{1}{\omega} - (d + t_{seek}) mod(\frac{1}{\omega}) = \frac{1}{\omega} - (d + 5.05) mod(\frac{1}{\omega})$

Putting it all together:

Total service time: $5.05 + \frac{1}{S}\frac{1}{\omega} + \frac{1}{\omega} - (d + 5.05) mod(\frac{1}{\omega})$

# File Systems

1. Assume that a file F has been opened, and that F is 200,000 bytes long. Assume that the block size of the file system is 1,000 bytes, and that the size of a block pointer is 10 bytes. Assume that the file system uses UNIX-like index structures (i-nodes), each having ten direct data block pointers, one single indirect pointer, and one double indirect pointer. Assume that the i-node for F is in memory but none of F's data blocks are.

   a. How many data blocks does F occupy?

F has 200,000 bytes, which are organized into blocks of 1,000 bytes each. Therefore F occupies a total of 200 blocks.

   b. How many blocks in total (not including the i-node) does F occupy?

The first 10 data blocks are referenced directly by the i-node, so there are no indirection blocks needed for them. The question is how many blocks of pointers are needed for the remaining 190 data blocks.
Because a pointer takes up 10 bytes, and a block is 1,000 bytes, we know that one block can hold 100 pointers. Since the i-node has one single indirect pointer, the pointer block referenced by the single indirect pointer takes care of the next 100 data blocks, leaving us with 90 data blocks.
The remaining blocks are referenced through the double indirect pointer. With 90 data blocks remaining, we only need 1 single indirect block to reference the remaining data blocks.
Putting it all together, we have 200 data blocks, 1 double indirect block of pointer, and 2 single indirect blocks of pointers (one pointed to by the i-node, and the other pointed to by the first entry in the double indirect block) for a total of 203 blocks.

c. The process that opened F issues a read() request for bytes 9500-10300 from the file. How many blocks must the file system retrieve from the disk to satisfy this request?

Converting a request for a particular byte of the file to its corresponding data block in the file is very similar to determining the virtual page on which a given virtual address resides. Each data block is like a page of virtual memory and the requested byte is the virtual address.
Given a particular byte in the file, we first need to remove the offset. In this case, the blocks are 1000 bytes each, which are addressable with 3 base 10 digits. Starting with the first byte requested, we see that it resides on page 9 by removing the bottom 3 digits. Similarly, the last byte requested is on page 10.
The next step is to identify the region (direct, single indirect or double indirect) in which each data block fits in order to determine how many pointer blocks the file system will need to traverse when servicing the request. Data blocks 0 through 9 are referenced by direct pointers in the i-node, so data block 9 is in the direct region. Data block 10 is in the next 100 data blocks (10-109), which are accessed via the single indirect pointer.
Add up the necessary pointer blocks we get a total of 1 for the pointer block referenced by the single indirect pointer.
Putting it all together, the file system must retrieve 3 blocks (2 data blocks and 1 pointer block) from the disk to satisfy a request for bytes 9500-10300.

d. The process that opened F issues a read() request for bytes 10500-11300 from the file. How many blocks must the file system retrieve from the disk to satisfy this request? Answer this question independently from part (c).

We already did most of the work in part (c) for this question. Instead of requesting blocks 9 and 10, this time the read is requesting blocks 10 and 11, and we already know both of those blocks are accessed through the single indirect pointer. The only trick to remember is that the file system does not need to access the pointer block twice, so the answer is still 3.

2. Assume that an i-node has 12 direct pointers, 1 single indirect pointer and 1 double indirect pointer. Assume that the block size of the file system is 4 KB and that the size of a block pointer is 32 bits.
   a. How many pointers fit in a single block?

32 bits = 4 bytes = $2^2$ bytes per pointer

4 KB = $2^{12}$ bytes per block
$2^{12}/2^{2} = 2^{10}$ pointers per block.

b. What is the largest file size that an i-node can support?

12 data blocks through the direct pointers
$2^{10}$ data blocks referenced through the single indirect pointer
$2^{10}2^{10}$ data blocks referenced by the double indirect pointer
The total size is then $(12 + 2^{10} + 2^{10}2^{10})2^{12}$ bytes.

c. How many blocks must the file system retrieve from the disk (not including the i-node) to access the byte at offset $2^{23}$?

The real *work* in solving a question like this is determining if the byte is accessed through a direct, single indirect or double indirect pointer. If the data can be accessed through a direct pointer, then only 1 block needs to be retrieved: the block that has the data we want. If the data is accessed through a single indirect pointer, then 2 blocks need to be retrieved: the data block, and the pointer block referenced by the single indirect pointer. Finally, if the data is accessed through a double indirect pointer, then 3 blocks need to be retrieved: the data block and two pointers blocks.
It is much easier to work with blocks then bytes, so the first thing we want to do is convert the requested byte to a requested block. Converting a byte in the file to its corresponding block is done by discarding the offset into the block (just like virtual memory). In this question, the blocks are 4 KB = $2^{12}$ bytes in size, so we want to remove the bottom most 12 bits to get the block number. Conveniently, the question has asked for the byte $2^{23}$, which is evenly divisible by the block size. In other words, the byte requested has an offset of 0 into the block in which it resides. We can then simply divide the requested byte by the size of a block to get the block number: $2^{23}/2^{12} = 2^{11}$. For completeness, we could also do this the longer way by converting $2^{23}$ to hex (0x800000), removing the bottom 3 hex digits (0x800) and then converting back to base 2 ($2^{11}$).
Once you have the block number, it is a matter of deciding which region it belongs to. In this case, $2^{11} > 2^{10} + 12$, so the block is accessed through the double indirect pointer. Therefore, the file system must retrieve 3 blocks.