OS

OS

Processes

- duplicate of parent, but with own address space (changes are not reflected)
- Child processes must be reaped, adopted but init process if termination of parent.
- Processes that never terminates are “zombie children”
- program is the “dead” code
- process is the “live” instance running the program
- Processes are a purely virtual concept, CPU has no idea what they are.
- Processes are isolated from each other.
- Processes can only directly interact with the outside world through system calls, mediated by the kernel.
  Unix process can be in the following states:
- running
- sleeping (waiting for an event or signal)
- stopped (paused)
- zombie (terminated but waiting for parent process to acknowledge)
- uninterruptible sleep (waiting for a hardware event)

Threads

- run in same address as the calling process (changes are reflected)
- have own thread context: ID, SP, PC, general purpose registers, condition codes

- can access “critical memory” must be handled with semaphores (mutexes) and / or condition variables
- dies when the process containing the thread dies

Kernel

- “Aways resident code that services request from the hardware and manages processes”
- Unprivileged, must make calls to kernel to switch to privileged state (interrupts)
- The kernel handles: hardware, system memory, File I/0 and context switching
- Handles the following
  - Memory allocation and deallocation
  - Interrupt handling
  - Task scheduling
  - File system management
  - Network stack management
  - Access to hardware devices
  - Creation and deletion of processes
  - Control and manipulation of system resources such as CPU, memory, and I/O devices.
- Allowing user-level programs to perform these operations could potentially lead to systeminstability or security vulnerabilities.
- Process Management: The kernel manages the creation, execution and termination of processes. It also manages the scheduling of processes and assigns the CPU time to different processes.
- Memory Management: The kernel manages the physical and virtual memory of the system and provides memory allocation and deallocation services to the processes.
- File System: The kernel provides an interface for the file system, it manages the file system, and it controls the access to the file system by the processes.

- Network Stack: The kernel provides the network stack, which is responsible for managing the network communication and protocols.
- Device Drivers: The kernel provides the device drivers, which are responsible for managing the communication between the hardware and the kernel.
- Security: The kernel provides security mechanisms such as access control and authentication to protect the system and user's data.

  Syscalls

  A request by a process that the kernel carries out some operation on its behalf.
- Only the kernel has direct access to hardware and system memory.
- Whenever we want to do IO we have to perform a system call
- Much like a function call, but implemented very di[ESC]erently.

See man syscalls to which functions perform syscall

How to write to memory

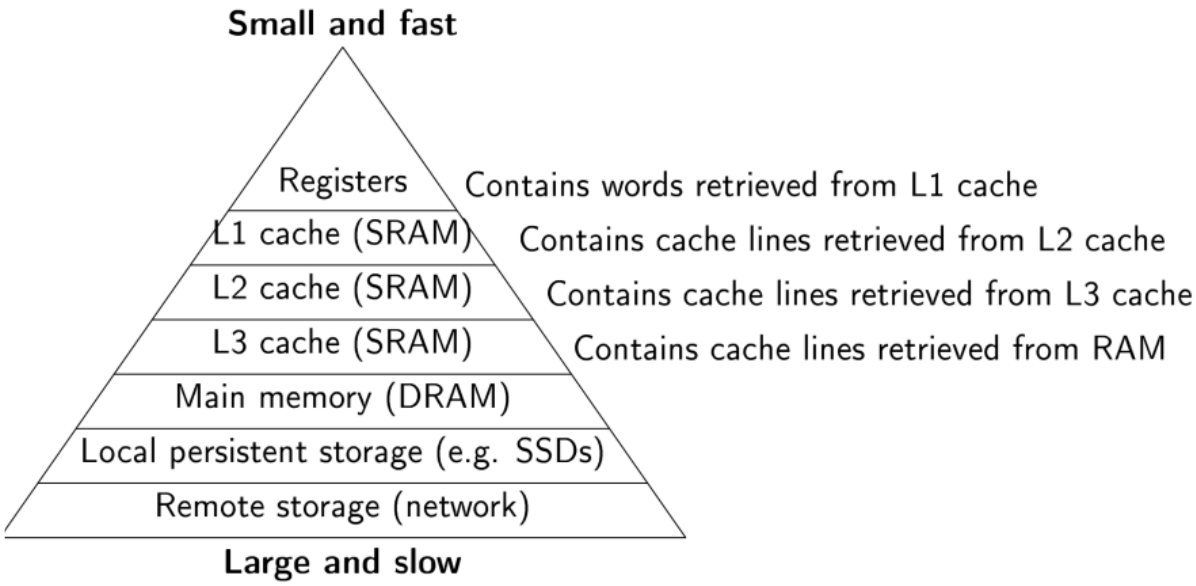
what happens on a write depends on the method used:

- Write-through: The more simple solution is to update the cache, and main memory each time we write. Data is written into the cache and the corresponding main memory location at the same time. The cached data allows for fast re-trieval on demand, while the same data in main memory ensures that nothing will get lost if a crash, power failure, or other system disruption occurs.
- Write-back: Another solution is that we only write to the cache so that the block is written to the main memory at a later time (when it is replaced by another block). Data is written into the cache every time a change occurs,but is written into the corresponding location in main memory only at specified intervals or under certain conditions.

- Write allocate: With a write miss, the address the data is to be written to is loaded into the cache, and then the data is written to the cache. Less consistency but faster.
- No-write allocate: On a write miss, you write the data directly to the main memory and do not add the address to the cache. Full consistency, but slower.

For more OS theory check out: exam notes detailed on DIKUnotes.

Locality



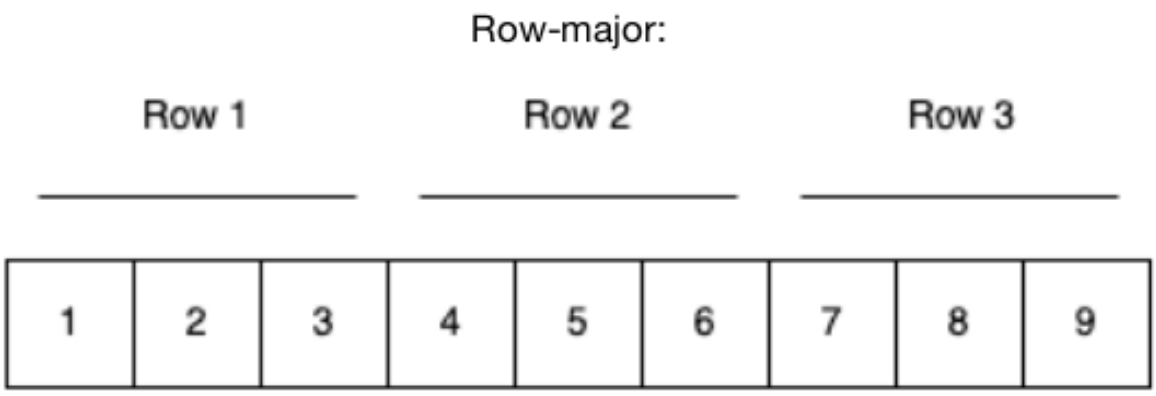
Borrowed from slides: Memory Hierarchy and Caching

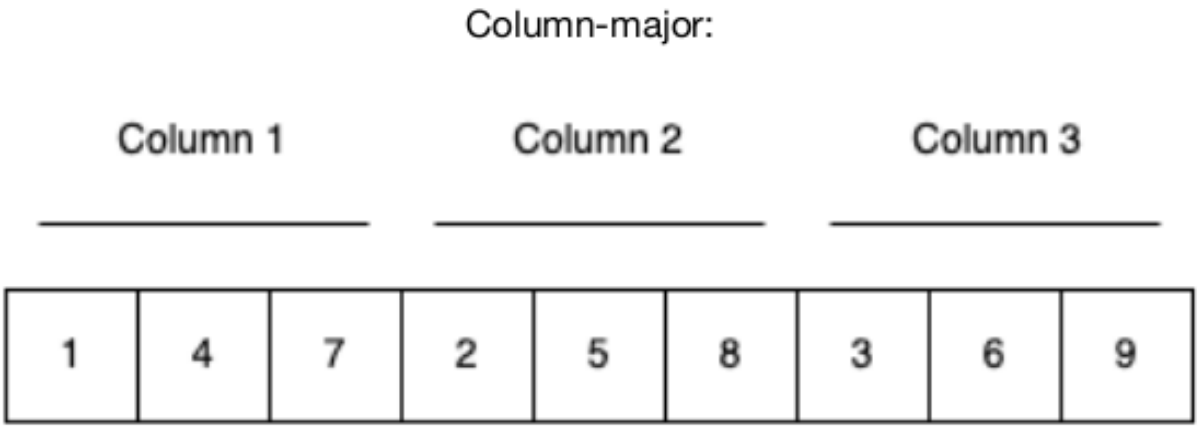
- **Spatial locality:** Where data is located in terms of each other, good spacial locality is accessing things close in memory.
- **Temporal locality:** Data that has recently been accessed, is accessed again soon. Which is good since it's in the cache making it quick to access.

• Stride: How large are the jumps between memory accesses?

row-major and column-major: How mulit dimensional arrays are stored in memory:

```
A[3][3]=
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]
```





If we try to access the 1 in row 1, then the cache in this example will contain {1,2,3}. When we try to iterate through rows first then we have good spacial locality since we only have to jump little in memory to get the next element 2, futhermore it's in the cache making it quick to get.

However when using column major the cache will contain {1,4,7} which will be discarded since we are trying to access 2, which is futher away than when using row major.

Locality: Example 1

```c
int A[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {6, 7, 8}
};
int sum = 0;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        sum += A[i][j];
    }
}
printf("%d", sum);
```

- Stride:
  - row-major = 1
  - column-major = 3
- Spatial locality:
  - row-major = Better (we make jumps of 1 elements)
  - column-major = Worse (we make jumps of 3 elements)
- Temporal locality: `sum`, `i` and `j`

Stride in this example:

row major: on average we have to jump 1 block in memory to get the next element, therefore stride is 1.

column: on average we have to jump 3 blocks in memory to get the next element, therefore stride is 3. from 1 to 2 the distance is 3. Even though we have to jump from 3 to 4 which is 5 blocks, most of the time we have to jump 3 blocks therefore stride becomes 3 regardless.

Locality: Example 2

```c
int A[3][2][3] = {
    {
        {1, 2, 3}, {4, 5, 6}
    },
    {
        {7, 8, 9}, {10, 11, 12}
    },
    {
        {13, 14, 15}, {16, 17, 18}
    }
};
int sum = 0;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        for (int z = 0; z < 2; z++) {
            sum += A[j][z][i];
        }
    }
}
printf("%d", sum);
```

- Is this the most optimal access order for either row-major or column-major?
  - Row-major: No, `i` that is used to access elements in the first dimension is incremented last
  - Column-major: No, `j`, that accesses 2-D arrays in the third dimension is not incremented first.
- What is the most optimal arrangement of the for-loops (from outer to inner)?
  - Row-major: `j, z, i`
  - Column-major: `i, z, j`

C is row major

R is column major

j has better temporal locality than i because it's accessed more often.

Semaphore

- Combination of a mutex and a counter
- Locked if 0 (memory is inaccessible)
- if the semaphore is 5, it would need 5 calls of P(s) to become locked.
- A mutex is a type of semaphore that is only 0 (locked) or 1 (unlocked).

```
wait (S){ // originally called P(S)
        while ( S < 0 ) do no-op;
        S--;
\}
```

```
Signal (S){ // originally called V(S)
        S++;
\}
```

That means when you see p(a) it means that a is being decremented, if a=1 (unlocked), p(a) will make a = 0 (locked). When you see V(a) it means a is being incremented so if a=0 (unlocked)then V(a) wil make a 1 (unlocked)

Drawing a process graph:

Question 2.2.1: Consider two threads performing the following semaphore operations.

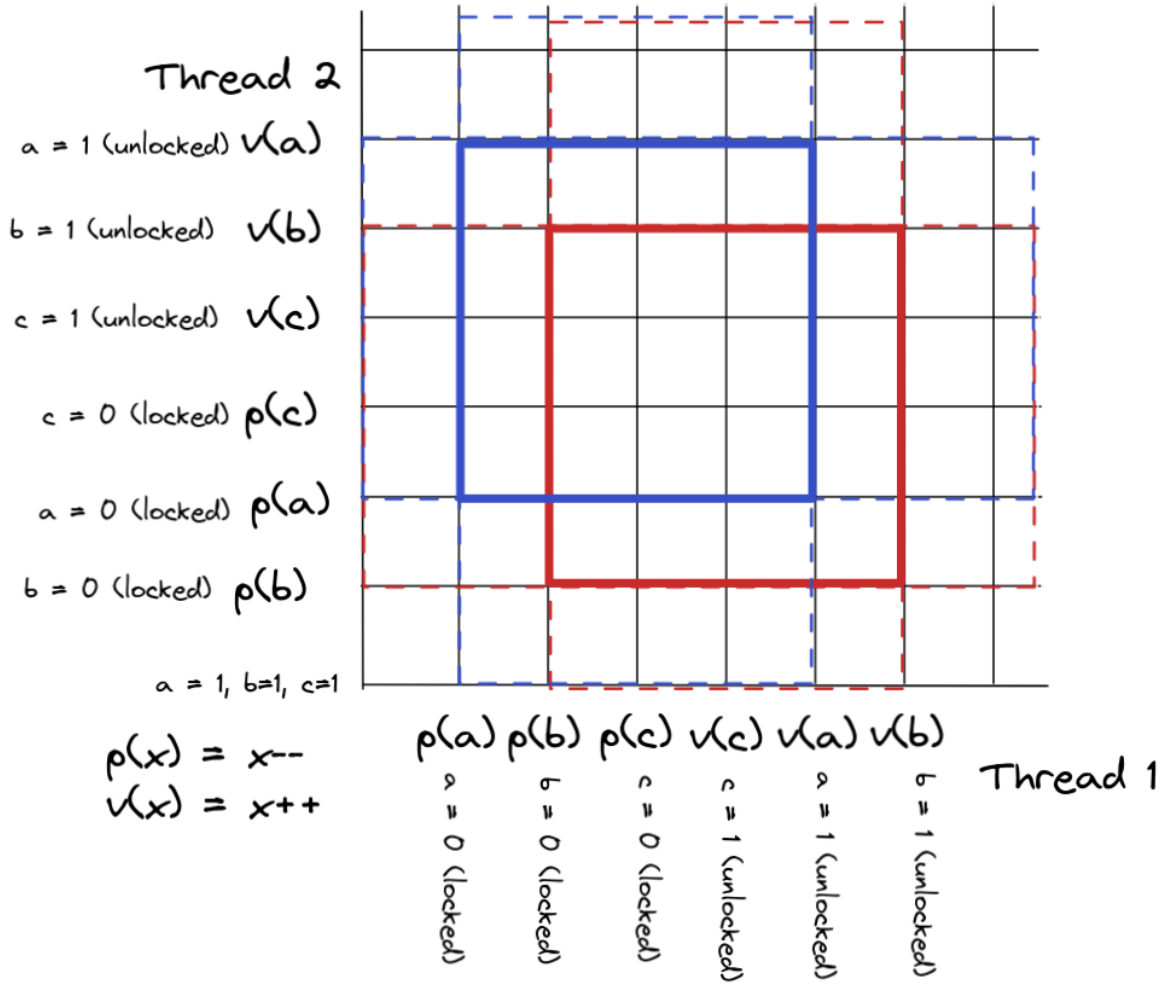
```
Initially: a = 1; b = 1; c = 1;

Thread 1:       Thread 2:
P(a);           P(b);
P(b);           P(a);
P(c);           P(c);
V(c);           V(c);
V(a);           V(b);
V(b);           V(a);
```

Draw a process graph with thread 1 along the horizontal axis and thread 2 along the vertical axis, show the forbidden regions, and argue whether this means deadlocks are possible or not. (You are welcome to attached the figure as an image in the handin.)

Make a graph with thread 1 and 2 along the axes as specified by the

text, along the each axis put the semaphore operation starting from 0,0.

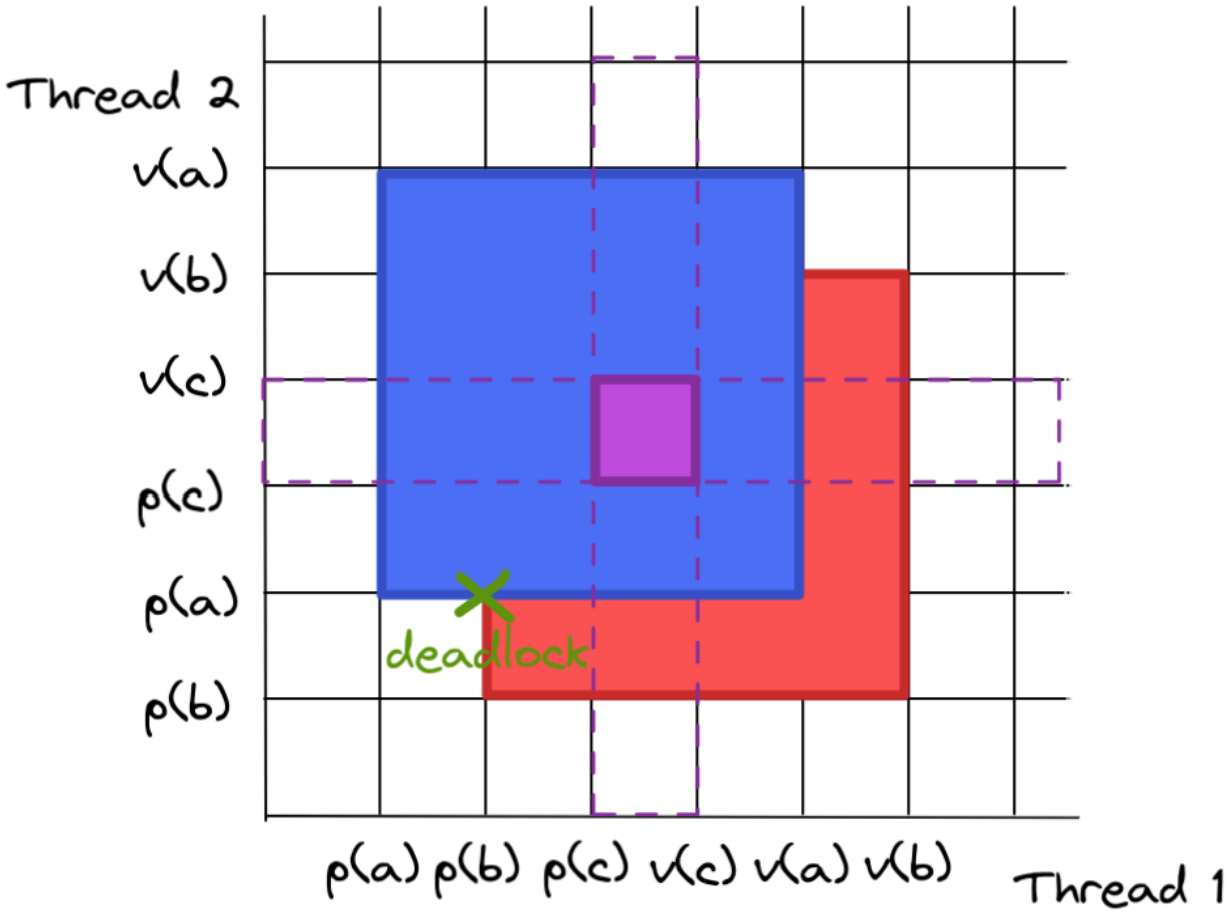
now make a square indicating where each variable is locked, in thread 1 a is locked from p(a) to v(a), same for thread 2. Blue square is where a is locked, red is where b is locked.



when moving right, one thread is progressing, when moving up, the other thread is progressing. we can only go right or up.

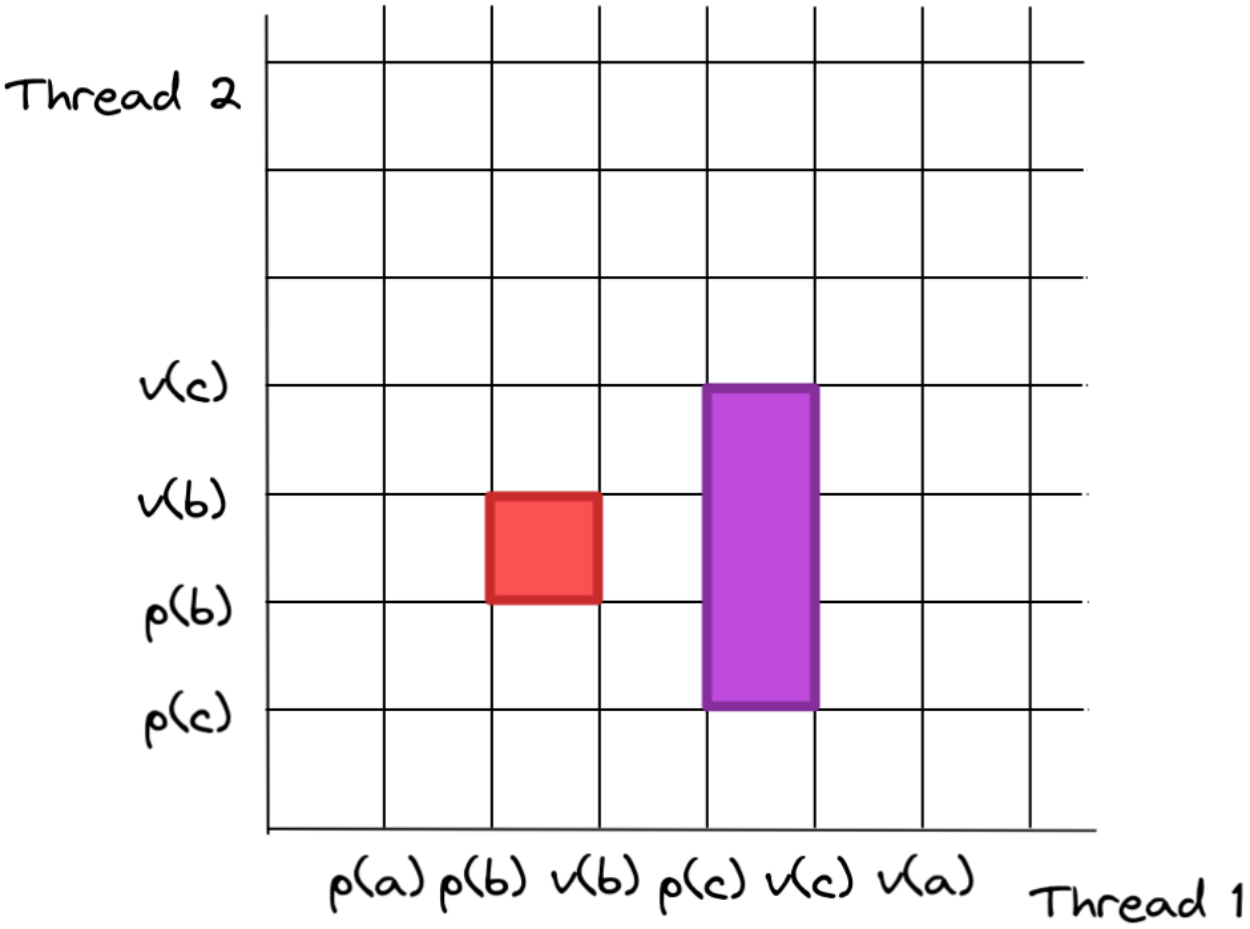
The place where both of threads lock a is forbidden. Same goes for other variables. that means if we move into the square we have a deadlock

Make a square indicating where each variable is locked, here we see it's not necessary to show where c is locked (purple square) because it's within the area where a and b is locked.



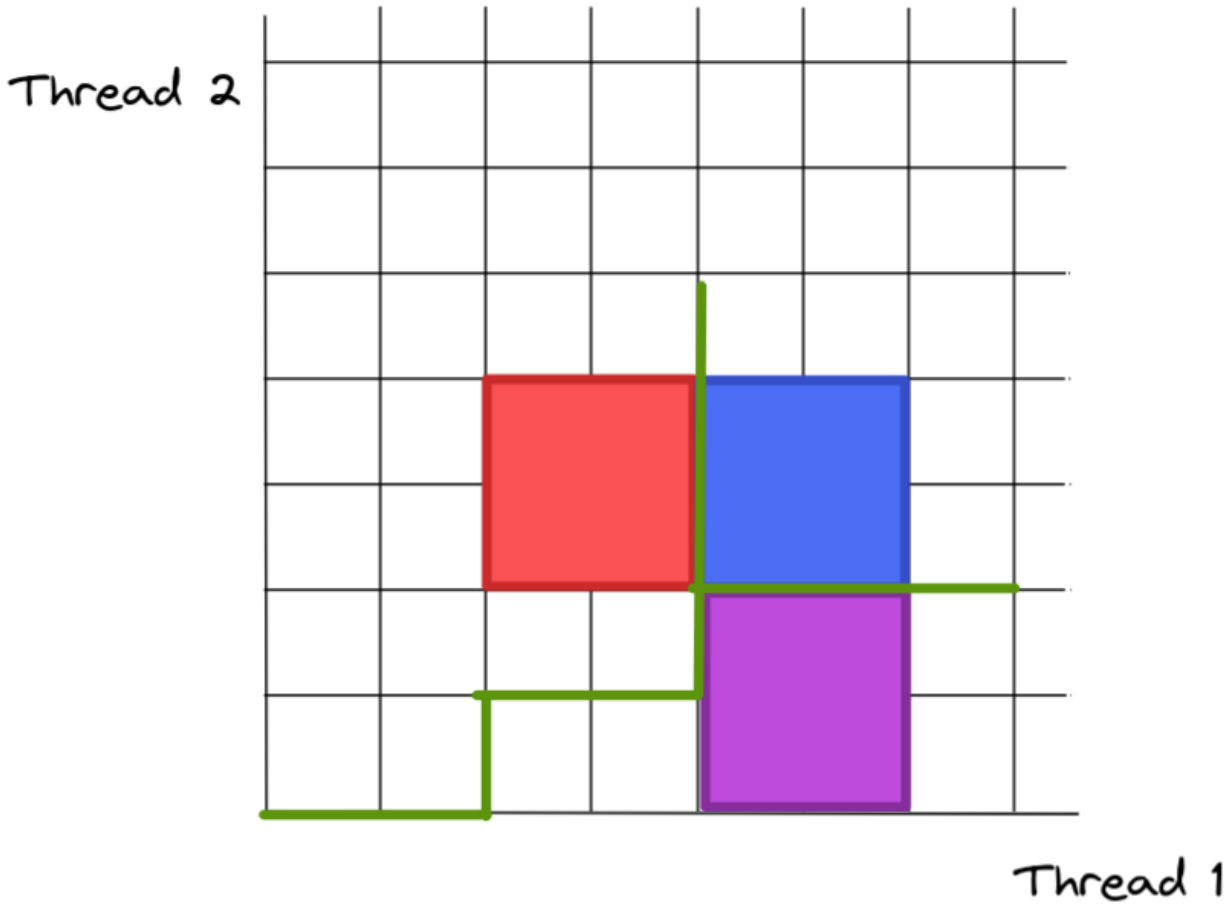
We see a deadlock is possible since when at (p(b),p(a)) we can go neither up or right, meaning we are stuck (deadlock), a is already locked in thread 2 and now thread 1 is trying to claim it aswell, how sad...

In this example we can always move up or right.



It's important to note that moving along is not forbidden, think of it this way: it's first moving past p(c) that a thread actually locks c so standing at (p(c),p(c)) is not a deadlock since it first (p(c),p(b)) where c is actually locked and we would not be allowed to move right, however we could just move up instead.

This could not cause a deadlock, since we can move along the lines (green lines indicate these paths)



Context switching

Only one process gets to run at a time, but we regularly switch between available processes. Doing this often and rapidly creates the illusion of simultaneous execution.

Pausing a process, saving its entire state, then resuming some other process based on its saved state.

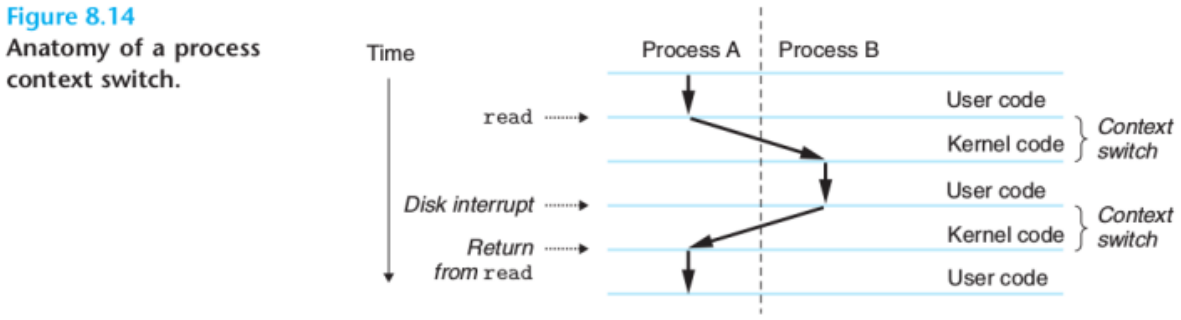
So what do we need to save?

1. All registers, including control registers.
2. Contents of memory.

So when do we do this?

- Regular timer interrupts transfer control to the kernel, whose scheduler decides the next process to run.

- Scheduling is a big and interesting topic that we don't have time to go into.



Fork

See man fork() for more info.

- Each process in Unix has a process ID (PID).
- Each process has a parent.
- ...except the initial process (init) with PID 1.
- A process may have multiple children.
- Implies processes are organised as a tree (pstree command shows it).
- Creating processes: fork().
- Terminating current process: exit().
- Loading program code from disk into current process: exec().
- Waiting for a specific child to die: waitpid().
- Getting PID of running process: getpid().

The child process and the parent process run in separate memory spaces. At the time of fork() both memory spaces have the same content. fork() is called once but returns twice.

Fork returns -1 on failure, 0 if child, 0> for parent (the parent gets the ID of the child). if (fork() == 0) - It's the child that goes into this if statement.

Mutexs are just variables so they are a part of this virtual address space.

File descriptors are also inherited, including open file status flags, current file offset, and signal-driven I/O attributes

Threads are not copied into the new process

OBS: the states of mutexes, condition variables, and other pthreads objects are part of the parent's virtual address space, and so they are inherited to the child! This may cause deadlocks.

- print uses a buffer (a variable) before actually printing, so this buffer may be inherited and the child will print too.
- if a mutex is locked and not unlocked before child is created, the mutex may never be unlocked.

  - **Idea**
    - When process terminates, it still consumes system resources
      - Examples: Exit status, various OS tables
    - Called a "zombie"
      - Living corpse, half alive and half dead
  - **Reaping**
    - Performed by parent on terminated child (using `wait` or `waitpid`)
    - Parent is given exit status information
    - Kernel then deletes zombie child process
  - **What if parent doesn't reap?**
    - If any parent terminates without reaping a child, then the orphaned child will be reaped by **`init`** process (pid == 1)
    - So, only need explicit reaping in long-running processes
      - e.g., shells and servers

  `If (waitpid(pid, NULL, 0)) > 0)` will be true when the child of pid has executed, in terms of the code example:

In the parent process, `waitpid` is used to wait for the child process to finish. Once the child process completes, it checks again and creates

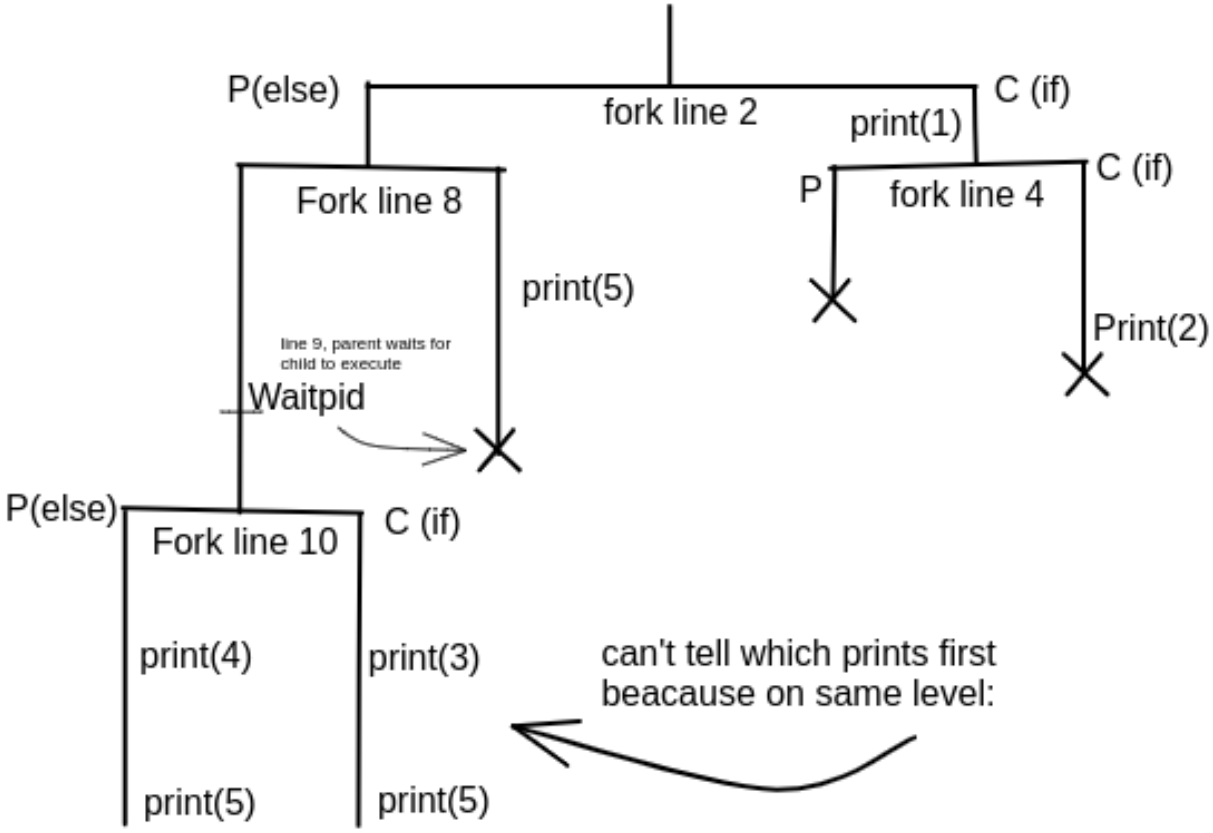
a new child process. The second child process prints "3", and the parent process prints "4".

the parent can print 4 and 5
children can print 1, 2 and 3

```c
int main () {
  if (fork() == 0) {
    printf("1");
    if (fork() == 0) {
      printf("2");
    }
  } else {
    pid_t pid = fork();
    if (waitpid(pid, NULL, 0) > 0) {
      if (fork() == 0) {
        printf("3");
      } else {
        printf("4");
      }
    }
    printf("5");
  }
}
```

Make a graph: Make a fork in the graph each time fork() is called. Beware of the level of each thing when making the graph, we see that print(4), print(3) are on the same level, therefore we can't be certain

that one prints before the other, the same goes for the 2 print(5).



Threads

Race condition: code is dependent on the sequence or timing of other uncontrollable events such as multiple threads accessing the same variable, leading to unexpected results.

context switched, just like processes

thread context: Thread ID, stack, stack pointer, PC, condition codes, and GP registers
the remaining process context is shared between the thread: Code, data, heap, and shared library segments of the process virtual address space. Open files and installed handlers.

Atomicity

Atomicity is a guarantee of isolation from concurrent processes. Additionally, atomic operations commonly have a succeed-or-fail definition — they either successfully change the state of the system, or have no apparent effect. That is atomic functions doesn't need

mutexes while non-atomic functions need mutexes
if multiple thread

remember that threads die with their process, this is not the same for forks.

```c
#include <unistd.h>

#include <stdio.h>
int x = 0;
void* worker(void* p) {
        for (int i = 0; i < 1000; i++)
                x++;
        return NULL;
}
int main() {
        pthread_t t;
        pthread_create(&t, NULL, worker, NULL)
        pthread_join(t);
        pthread_create(&t, NULL, worker, NULL);
        pthread_join(t);
        printf("%d\n", x);
}
```

`pthread_create(&t, NULL, worker, NULL);`: Creates a new thread (`t`) that will execute the `worker` function. The `NULL` arguments mean default thread attributes and no argument for the worker function. We now have the main thread an worker thread

`pthread_join(t);`: Waits for the thread represented by `t` to finish before proceeding. wait for thread termination.

In this example the a thread is created then terminated when x has been incremented a 1000 times. `pthread_join` ensures that no other code is executed before the thread is executed. After the thread is done a new thread is created.

Beacause of `pthread_join(t);` there can be no race condition

```
int x = 1;
void* thread(void* arg) {
        x = 2;
        printf("%d\n", x);
        return NULL;
}
int main() {
        printf("%d\n", x);
        pthread_t tid;
        pthread_create(&tid, NULL, thread, NULL);
        fork();
        printf("%d\n", x);
        exit(0);
}
```

In this example the worker thread would start working, but the thread in main would keep progressing aswell which would means that it could die before the worker thread does anything. Therefore the output is non deterministic (race condition).

"1" will be printed at least one time

Fork will have the child print one time, the parent process will also print one time after the fork.

Depending on whether the worker thread gets to run or not another print is made

We will therefore either print 3 or 4 times

if the worker thread newer runs any code, 3 prints are made and none of them are "2"

if the worker thread runs before the parent process (main thread) and the child: 3 of the 4 prints are "2"

its possible for the "2" to be printed 0-3 times.

Best way to go about solving this sort of question:

there aren't really any ONE way to figure this out, but a great way is just to look at the code an decipher the way the code could execute.

- Look at each function related to processes (what does it do)
- Look at each function relating to threads (what does it do and how does it impact the output)
- Use the man page to look up functions (open WSL terminal, bash terminal, mac terminal or bash terminal in VScode if non other are available)

Memory

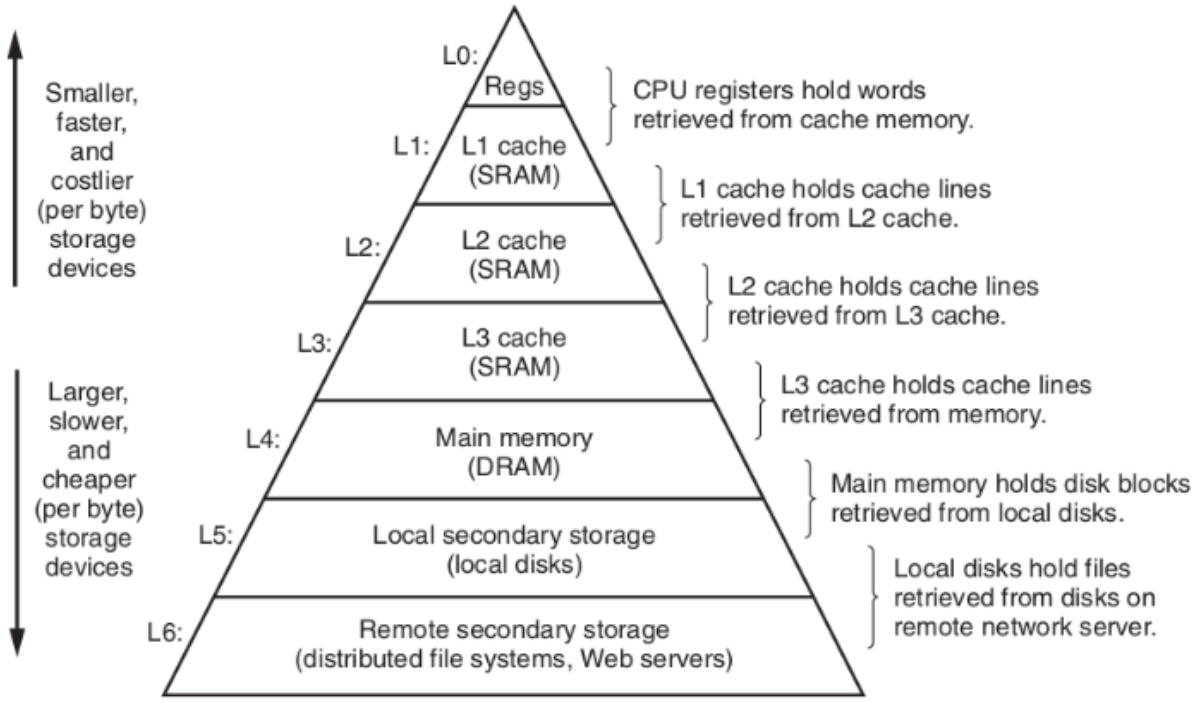


Figure 1.9 **An example of a memory hierarchy.**

SRAM and DRAM: Key differences

SRAM: Static random access memory

DRAM: Dynamic random access memory
RAM is volatile memory - the content is lost when the power is gone.

1. SRAM is an on-chip memory whose access time is small while DRAM is an off-chip memory which has a large access time. Therefore SRAM is faster than DRAM.
2. DRAM is available in larger storage capacity while SRAM is of smaller size. SRAM is expensive whereas DRAM is cheap.
3. The cache memory is an application of SRAM. In contrast, DRAM is used in main memory.
4. DRAM is highly dense. As against, SRAM is rarer.
5. The construction of SRAM is complex due to the usage of a large number of transistors. On the contrary, DRAM is simple to design and implement.
6. In SRAM a single block of memory requires six transistors whereas DRAM needs just one transistor for a single block of memory.
7. DRAM is named as dynamic, because it uses capacitor which produces leakage current due to the dielectric used inside the capacitor to separate the conductive plates is not a perfect insulator hence require power refresh circuitry. On the other hand, there is no issue of charge leakage in the SRAM.
8. Power consumption is higher in DRAM than SRAM. SRAM operates on the principle of changing the direction of current through switches whereas DRAM works on holding the charges.

Disk Storage

Rotational latency:
Time it takes for the disk to rotate so that the head can
read the first bit of the target sector.
Maximum Rotational latency:
Time it takes for the disk to do an entire rotation. Average is half of that

Cache info

Always has the following order
Tag | Index | Offset

Cache organisation

- N-way set associative: N: how many blocks there are in each set.
- Direct mapped/One-way set associative: one block for each set, so each memory location is mapped to one location in the cache.
- Fully associative: cache is one set, a memory location can be mapped to anywhere in the cache.

$\text{offset} = \log_2(\text{blocksize in bytes}) = \log_2(32) = 5$
$\text{cache size in bytes} = \text{KiB} * 1024$
$n = \text{associativity}$
$\text{set count} = \frac{\text{cachesize in bytes}}{n \cdot \text{blocksize in bytes}}$
$index = \log_2(\text{set count})$
$tag = \text{bits in address} - (offset + index)$

Use script to calculate size of each.

Number of lines pr. set = associativity

Virtual memory

Use DRAM as a cache for parts of a virtual address space
Uses memory efficiently by caching virtual memory pages

- Efficient only because of locality
- Simplifies memory management and programming
- Simplifies protection by providing a convenient interpositioning point to check permissions

Each process gets the same uniform linear address space that cannot be corrupted by other processes

Isolates address spaces

- One process can’t interfere with another’s memory
- User program cannot access privileged kernel information and code
  Simplifying memory allocation
- Each virtual page can be mapped to any physical page
- A virtual page can be stored in different physical pages at different times

Virtual addresses

Page faults are handled by software (kernel code), meaning we have flexibility.

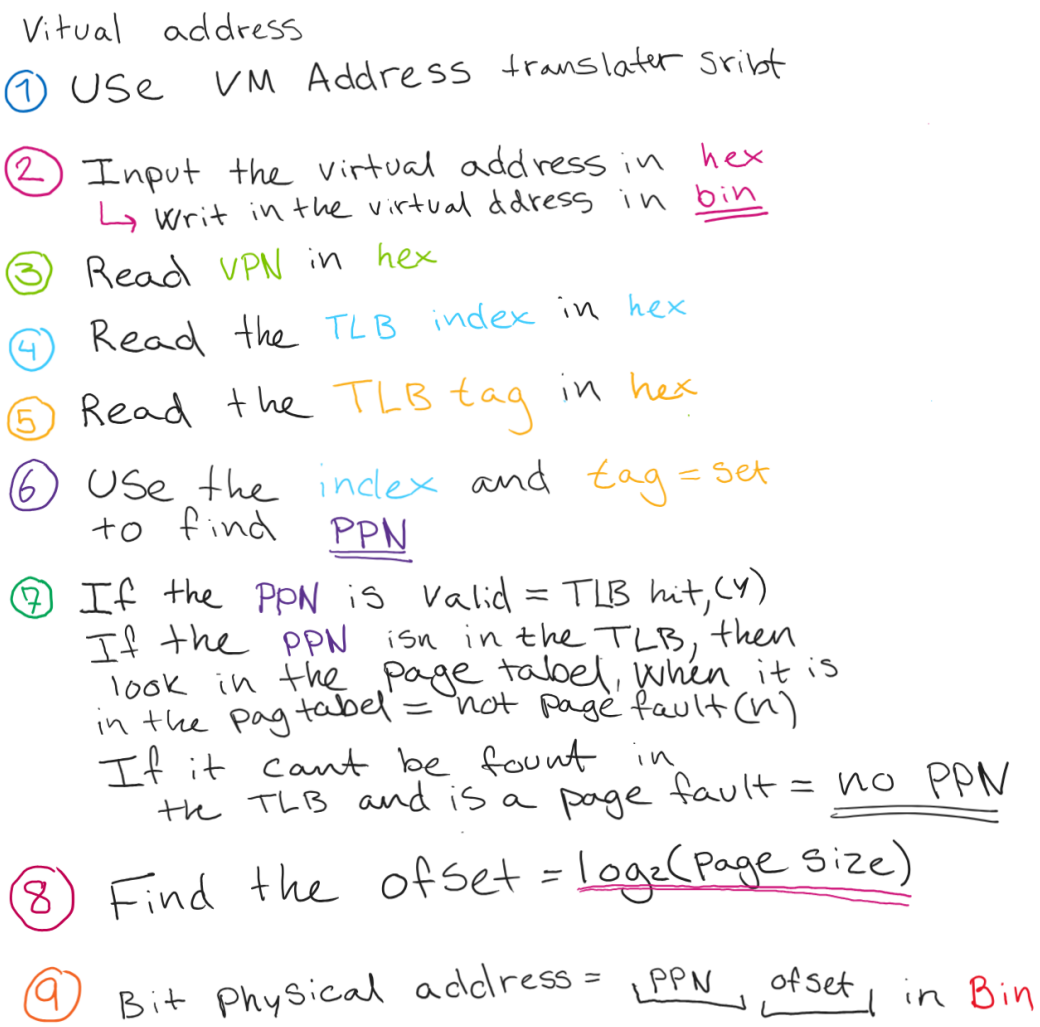
- Page fault handler can update the page table based on kernel data and policy.

mmap() is powerful but inflexible:

- Smallest granularity of allocation is a page.
- Is a system call, so fairly slow.
  Instead programmers use dynamic memory allocators (e.g. malloc()) to aquire memory at runtime.
- Run entirely in user space-not part of the kernel.
- Acquires memory via mmap() and sbrk ().
  Region of virtual memory managed by such an allocator is known as the heap.
- No relation to the datastructure known as a heap.
- May have multiple heaps; heap might not be contiguous.

- When we say the heap, we mean whatever malloc() manages by default.
  CompComponents of the virtual address (VA)
- TLBI: TLB index
- TLBT: TLB tag
- VPO: Virtual page offset
- VPN: Virtual page number
  Components of the physical address (PA)
- PPO: Physical page offset (same as VPO)
- PPN: Physical page numberonents of the virtual address (VA)
- TLBI: TLB index
- TLBT: TLB tag
- VPO: Virtual page offset
- VPN: Virtual page number
  Components of the physical address (PA)
- PPO: Physical page offset (same as VPO)

- PPN: Physical page number



TLB hit:

- check TLB index som set.
- Er der et TLB tag som matcher index, og er den valid?
- hvis ja - skriv Y, ellers N
- aflæs PPN og skriv ind
- Hvis ikke i TLB table aflæs andet skema
  Page fault?
- Aflæs andet skema.
- Hvis skemaet har din VPN - aflæs tilsvarende PPN
- Hvis valid -> Page fault = N
- Hvis invalid -> Page fault = Y
- Hvis VPN ikke kan findes, page fault = Y

mmap () syscall allows processes to map virtual memory.

- Can map files to memory, or make anonymous mapping.
- Can share memory between processes.
  malloc () is a userspace memory manager.
- Not a system call itself.
- Requests memory from kernel with mmap () and sbrk () and then parcels it out.
- Internal fragmentation is when allocated blocks have wasted space.
- External fragmentation is when free space is split into many small blocks.

Cache table

Cache organisation

- N-way set associative: N: how many blocks there are in each set.
- Direct mapped/One-way set associative: one block for each set, so each memory location is mapped to one location in the cache.
- Fully associative: cache is one set, a memory location can be mapped to anywhere in the cache.

Each address has tag, index and offset.

For this type of question use the script for calculating cache addresses:

Number of lines pr. set = associativity

usally the cache size is given in Kibibytes or other unit, convert this to byte.

1 KB = 1 Kibibyte (KiB) = 1024 bytes
1 MB = 1024KB = 1,048,576 bytes
1 GB = 1 000 000 000 bytes
1 Kbit = 1 kilobit = 125 bytes

Managing state of tags

If the index and tag is the same it's a hit. in a 4 way associative we keep 4 tags in each index, getting a hit is a good thing since it means we don't have to go into memory which is slow.

Replacement strategies

- First in first out (FIFO): First block in is deleted first.
- Last in first out (LIFO): Last block in is deleted last.
- Least recently used (LRU): The last used block is deleted first. In the state of tags we order the tags from most recently used to least recently used.
- Most recently used (MRU): The latest used block is deleted first.
- Random replacement (RR): A random block is deleted.

0 : {0x10, 0x0, 0x1, 0x11}
Means that in index we have recently encounted 0x10 and not encountered 0x11 for some time.

If a new address comes in such as 0x12, 0x11 would be removed since we only have space for 4 in the cache:

0 : {0x12, 0x10, 0x0, 0x1}

if we now encounter 0x0 agian it gets moved to the front since thats the last we saw, this would be a hit:

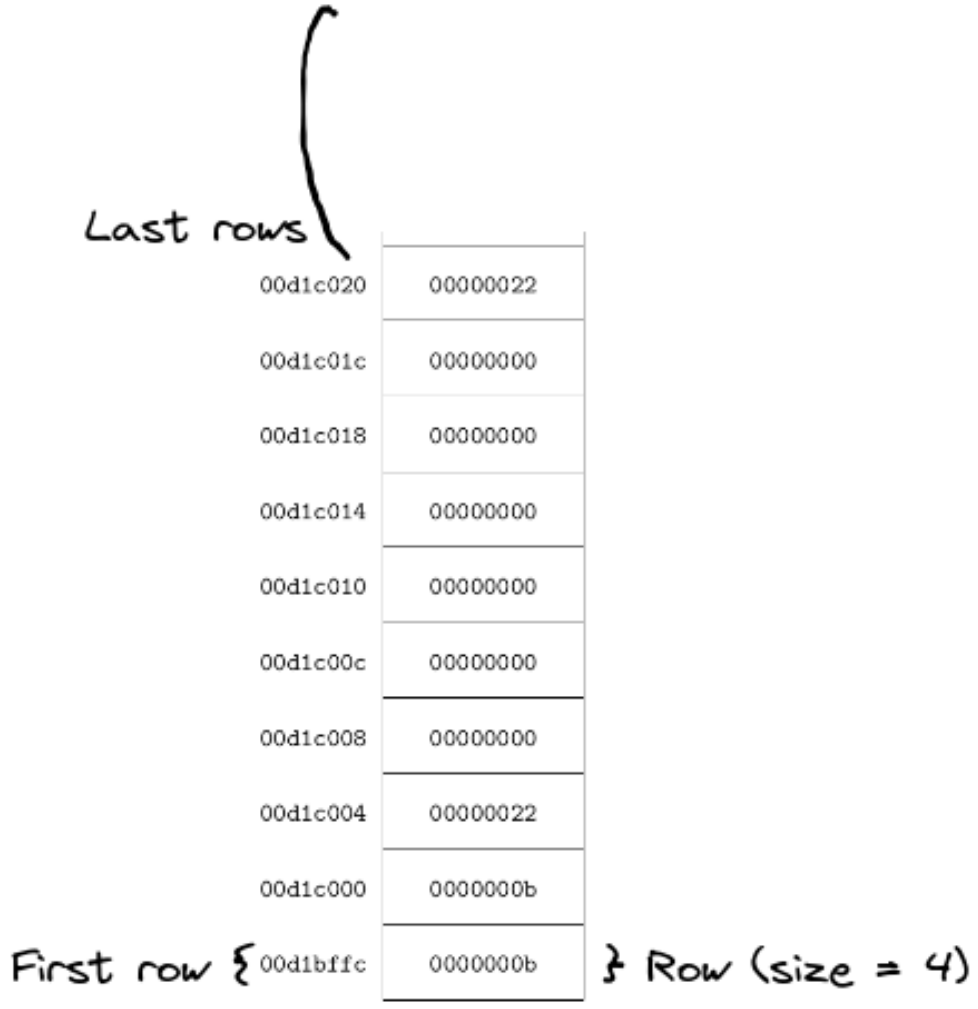
0 : {0x0, 0x12, 0x10, 0x1}

Heap

1. Understanding the Heap Structure:

- Rows in the heap can be headers, footers, or payload, with each row having a size of 4, each row has an address.

- Blocks are collections of rows with a header, footer, and optional payload (sometimes blocks must have payload meaning they must be bigger than 8, which is the size of a block with only a header and footer).
- The heap grows from bottom up.
- Block sizes must be multiples of 8.
- When one block ends the next block the next starts. By finding header of one block you also know the next row will be Header of next block.



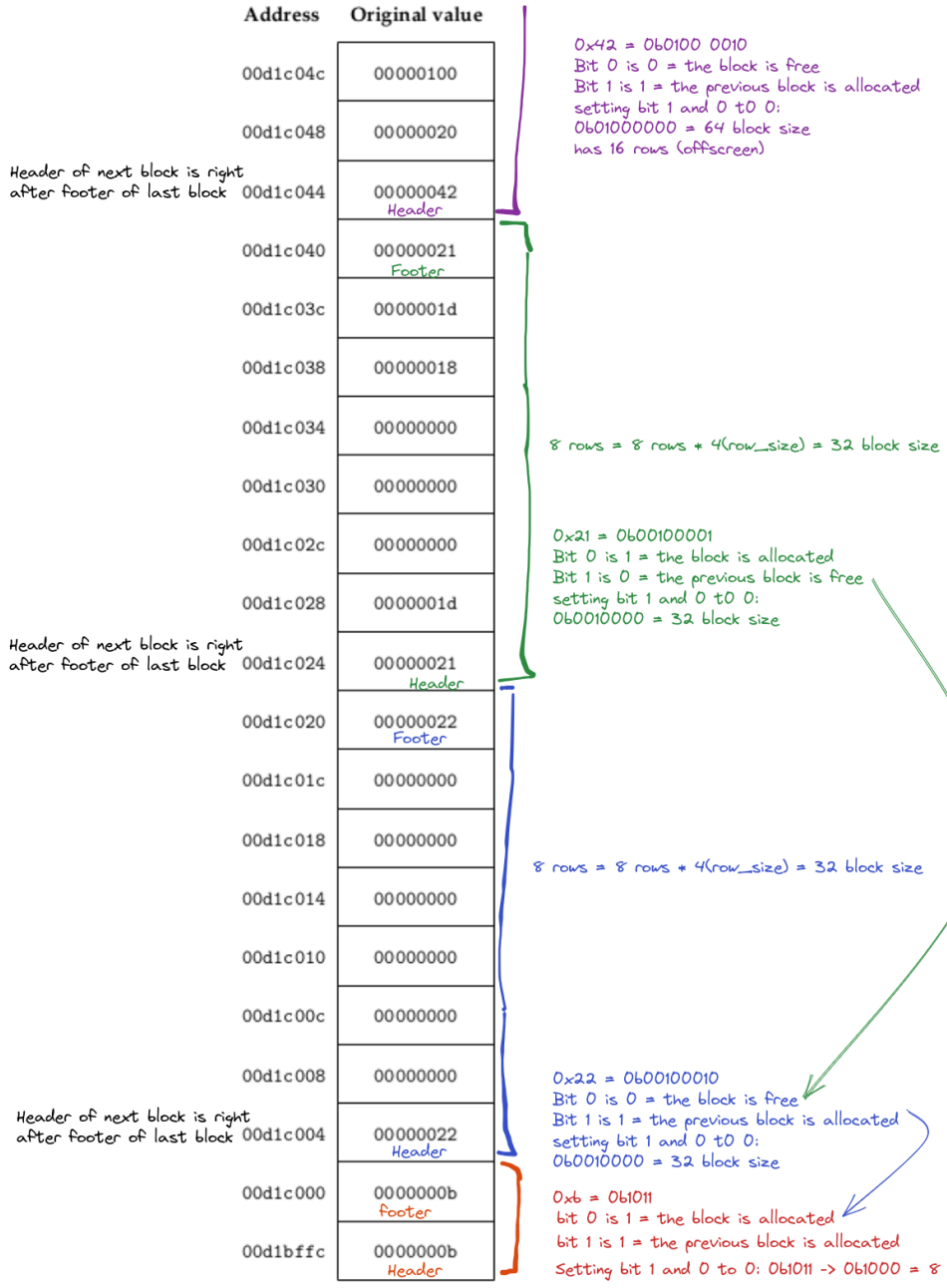
2. Analyzing Header and Footer Bits:

- Bit 0 in the header/footer indicates block usage (1 for allocated, 0 for free).
- Bit 1 indicates the usage of the previous block.
- Bit 2 is unused and always set to 0.

- The block size is determined when the first 3 bits are set to 0. (this could change so be sure to read the text carefully)

  example: 0x23 = 0b0010 0011
  bit 0 is 1: the block is allocated
  bit 1 is 1: the previous block is allocated
  setting bit 0 and 1 to 0 we get the size: 0b00100000 = 32

  32 is a multiple of 8 the block would therefore have 8 rows since each row has a size of 4, therefore 8 * 4 = 32.
  of those 8 rows 2 rows are for the header and footer

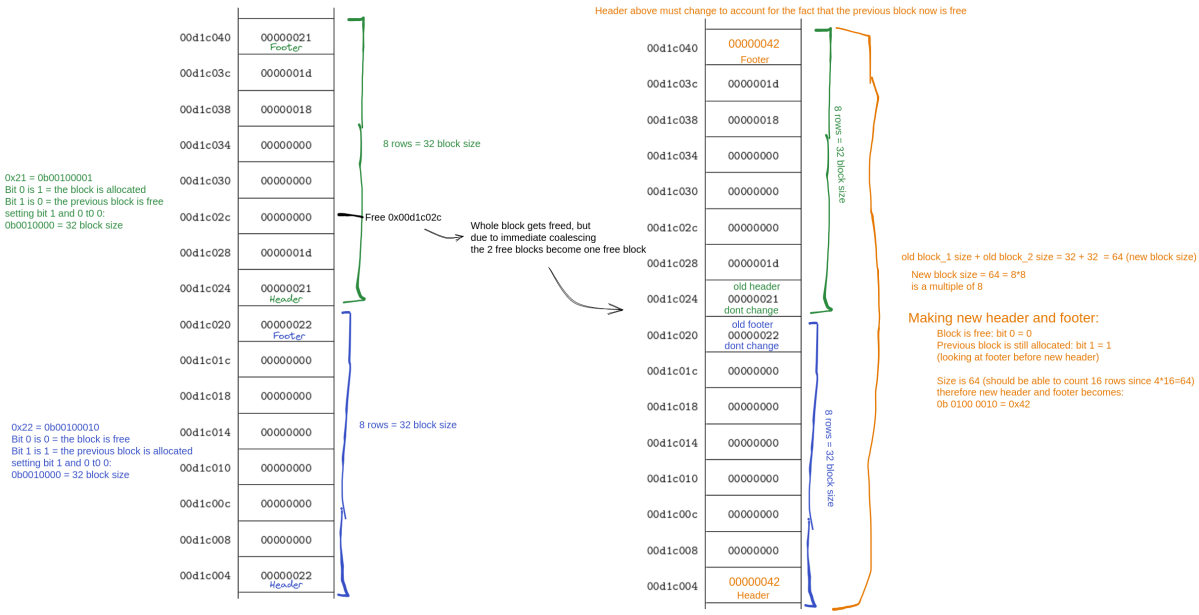


3. Steps to Solve the Assignment:

- Start by finding the first block (header or footer) from the bottom.
- Check if it's a valid block: Is the previous block allocated? Is the size correct (multiple of 8)?
- Find other blocks by moving from one block's footer to the next block's header, considering that blocks are right next to each other.

4. Operations:

- **immediate coalescing
- When freeing a block, check for adjacent free blocks immediately.
- If neighboring blocks are free, merge them into a single larger block.
- Ensure that the new block's header and footer are adjusted for the combined size, if 2 blocks have size 16, new size will be 32. alternatively count the amount of rows in each block (each row is 4 in size), *this not always possible(some rows of a block may be offscreen) so its better to look at header and footer of each block, determine their sizes, then add them up. See example below*
- Retain the headers and footers of the merged blocks in the payload area. That means the headers and footers in the middle of the block stay unchanged
- Update the adjacent block's (the block above) header and footer bits to reflect the merged block's status.
- Verify that the merged block adheres to any specified constraints, such as not creating blocks with zero payload.
- Adjust the size and allocation status(bit 1 and 0) of the resulting block to accurately reflect the merged space.



- **Free Operation:**
  1. Find header and footer of the block to free (the whole block which contains the address to free has to be freed, however

only the header and footer of the block will change).
2. Implement immediate coalescing if adjacent blocks are free.
3. Change header and footer bits to indicate the block is free.
4. Update adjacent block header and footer bits.
5. Recalculate the size and update the header/footer accordingly.

- **Malloc Operation:**
  1. Find the block where the address is located, this block will be allocated or at least part of the block.
  2. Round up the requested size to the nearest multiple of 8.
  3. Determine the rows needed for the new block (considering header, footer, and payload). malloc(n) bytes meaning reserving n bytes of payload and 8 for the header and footer, therefore we malloc n+8.
  4. if the free block is 32 for example, and you malloc(8), you get 2 new blocks one that that is 16 bytes and alllocated and another thats 16 bytes and free still, they previously free 32 byte block has been split.
  5. if malloc ends up creating a empty free block (only header and footer) you have to take additional rows to avoid having empty blocks (*only if the text specifies that there can be no empty blocks*). See an example below.
  6. Change header and footer bits to indicate the block is allocated.
  7. Update the adjacent block's header and footer bits.
- **Realloc Operation:**
  1. Perform a combination of free and malloc.
  2. Free the existing block.
  3. Malloc a new block with the required size.
  4. Implement immediate coalescing if necessary.
  5. Adjust header and footer bits, considering the new block's size and allocation status.

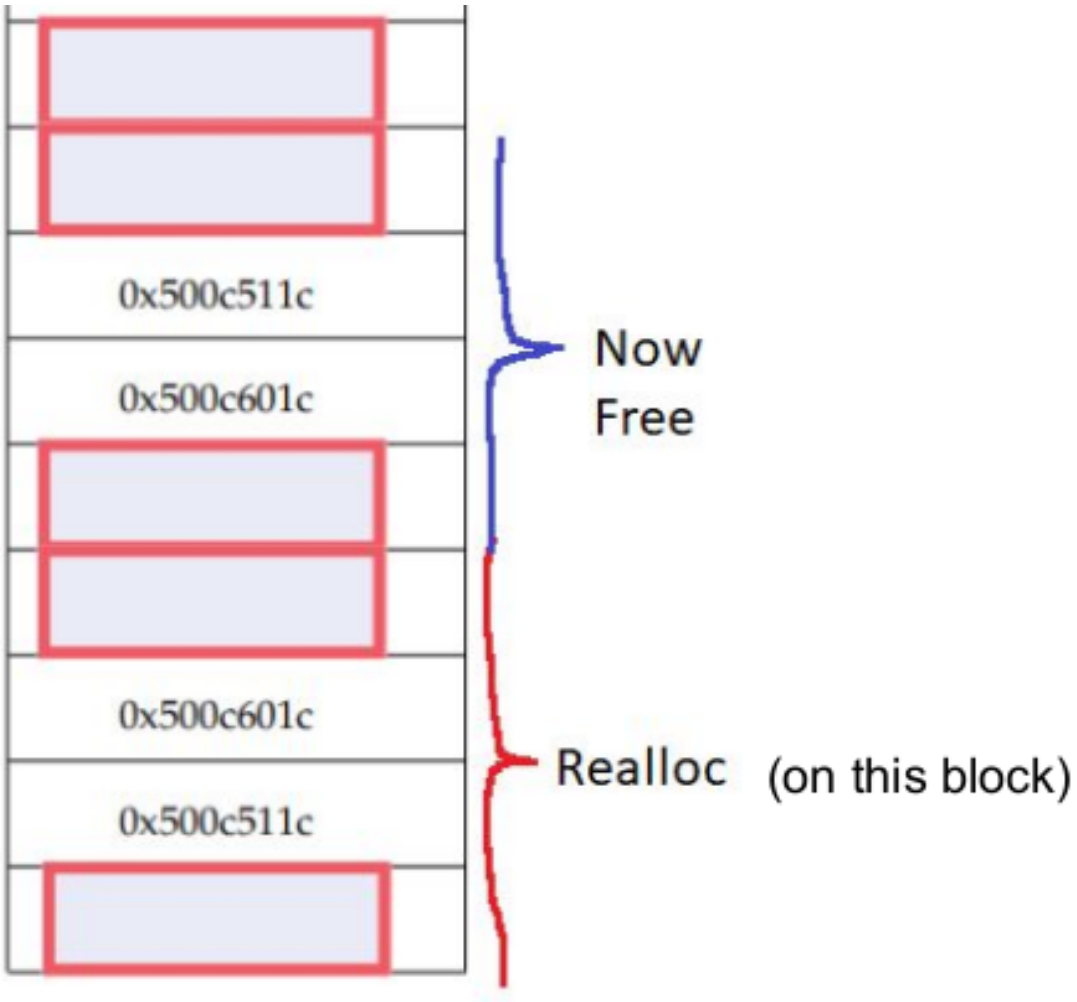
5. Internal Fragmentation:

- Understand that rounding up to the nearest multiple of 8 can cause internal fragmentation.
- Example: If reallocating 12 bytes, you may end up with a block of 16 bytes, causing 4 bytes of internal fragmentation.

6. Example:

Exam 20/21

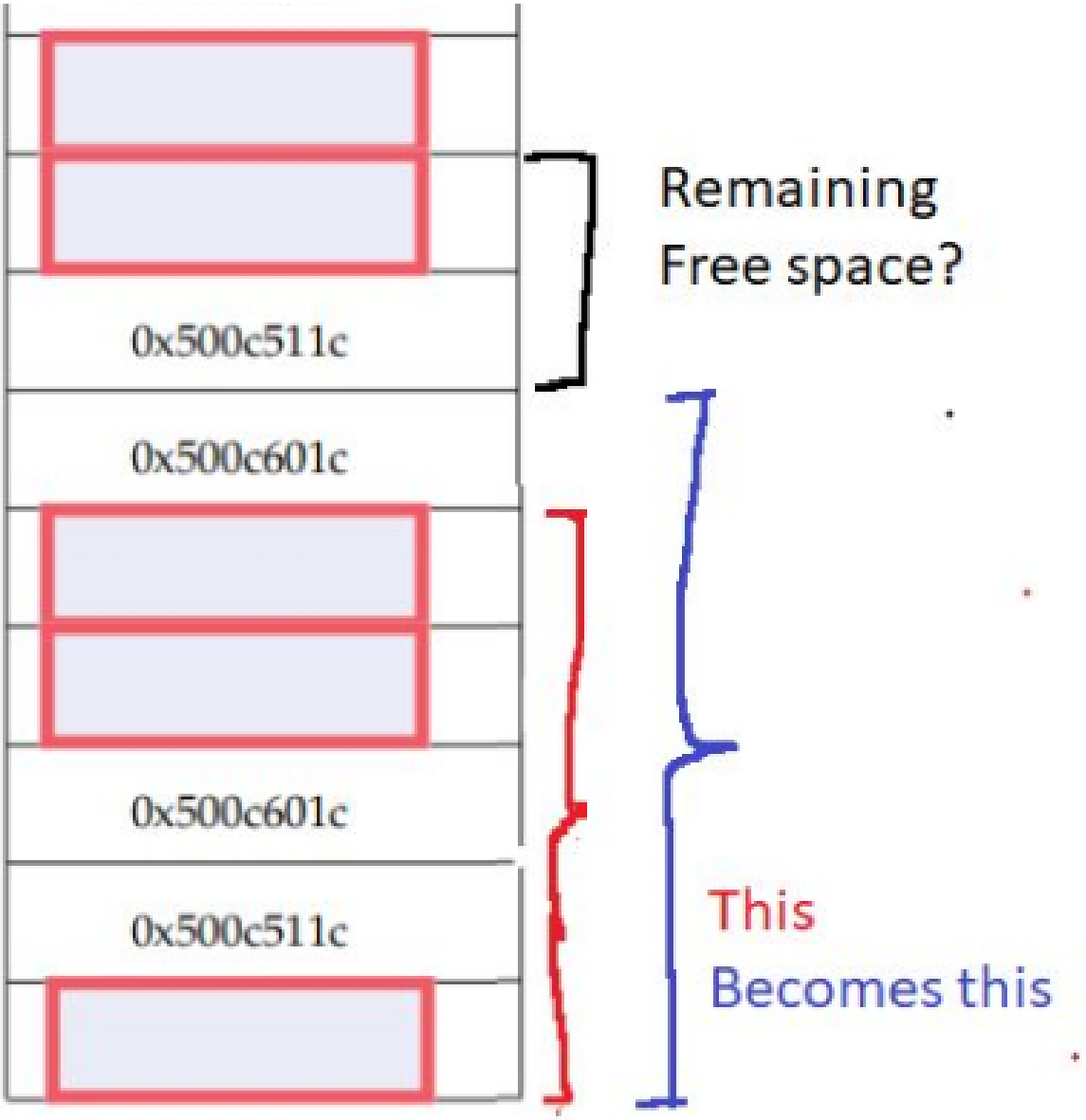
- Follow the given examples to understand how to apply the mentioned steps and concepts:
- we want have freed some memory (blue) and want to realloc 12 bytes (red)



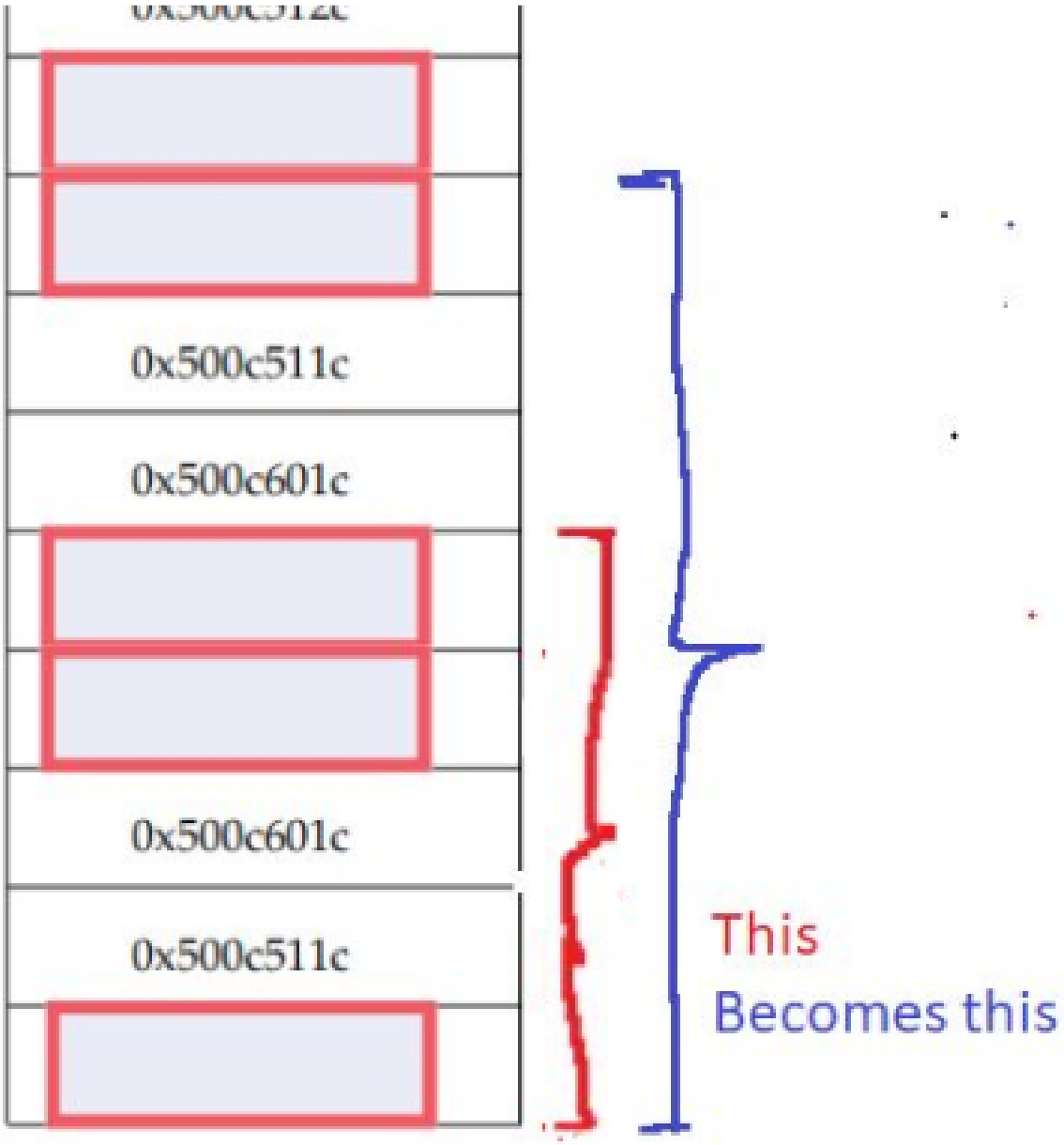
We have to take 16(realloc) + 4(header) + 4(footer) = 24 bytes, which is 6 rows (each row has size 4).

the black is the remaining free space, but since there is a block above the free space, it would mean we would get a block with size 2 * 4 = 8.

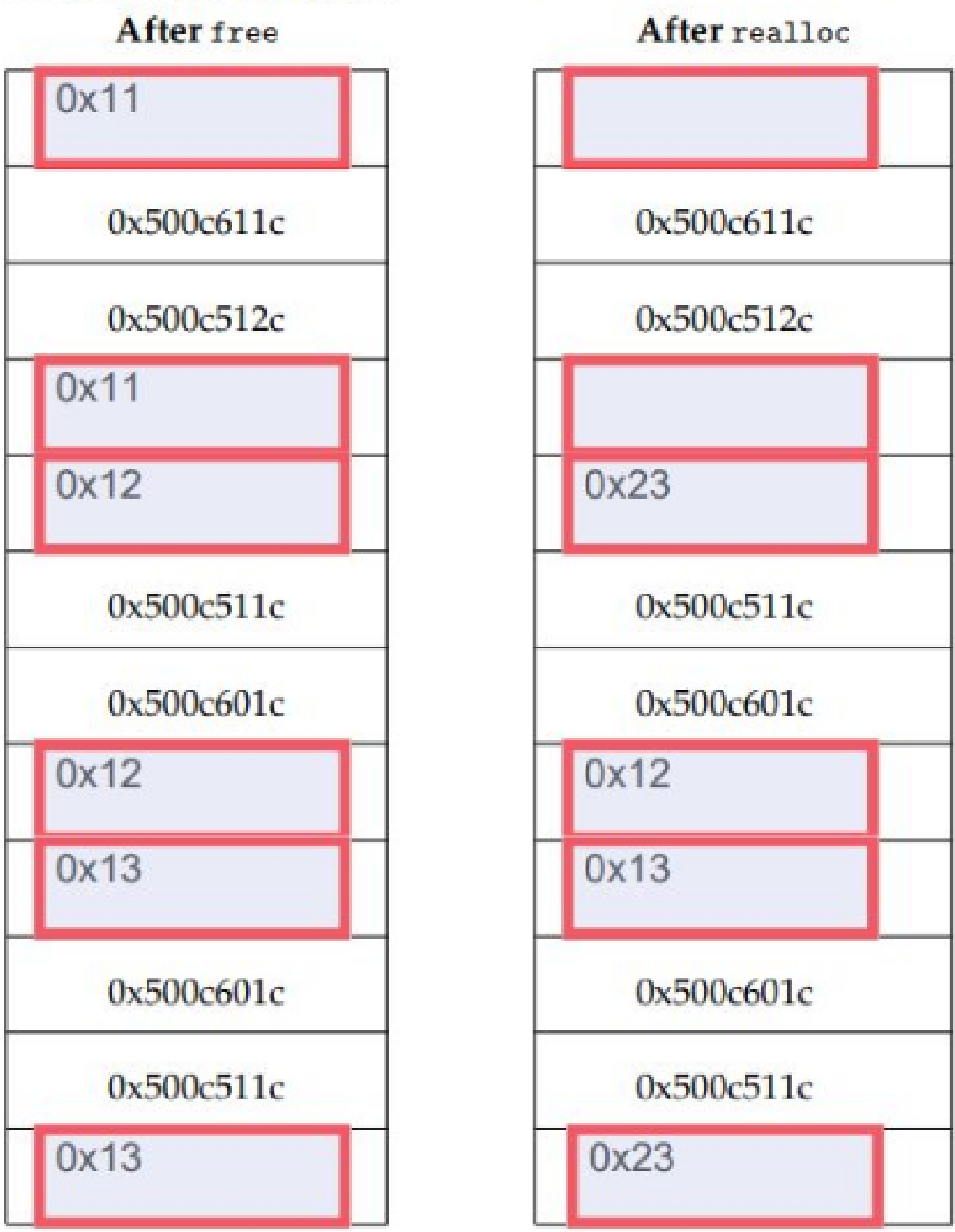
Exam text specifies that "we must never create blocks with 0 payload", this means we must not have a block that's only a header and footer, therefore we also need to take the remaining free space when re-allocating.



Therefore:



Therefore we get a block that is 16 + 16 in size, we remember that the old footer (0x13) and header (0x12) don't change since the 2 blocks merge into 1 due to immediate coalescing



7. Additional Tips:

- Pay attention to the specific requirements of the assignment, such as restrictions on block sizes or the presence of payload in certain cases.

- Be meticulous in tracking block sizes, allocation status, and immediate coalescing.