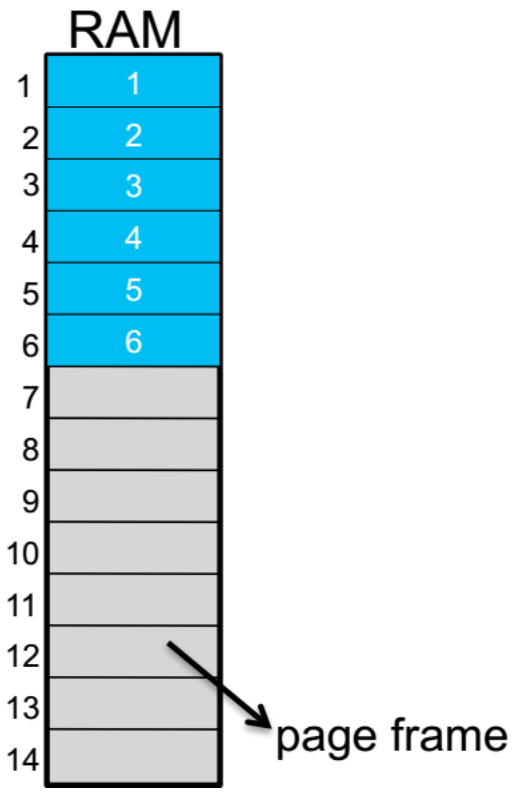
CS304 Operating Systems

DR GAYATHRI ANANTHANARAYANAN

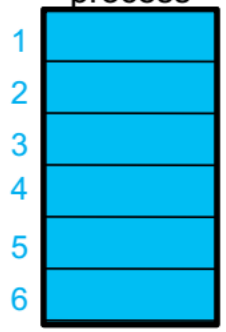
gayathri@iitdh.ac.in

Materials in these slides have been borrowed from textbooks and existing operating systems courses

17th March, Paging (Contd..)



Virtual address space of process



process page table

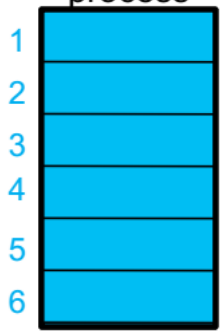
block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

If size of virtual address space is 4 GB (2^32).

If size of each page frame is 4 KB (2^12)

What is the size of the page table?

Virtual address space of process



process page table

block	page frame
1	1
2	2
3	3
4	4
5	5
6	6

If size of virtual address space is 4 GB (2^32).

If size of each page frame is 4 KB (2^12)

Number of entries in page table? ***2^20***

If size of each page entry is 4 bytes, 4MB of contiguous memory is required.

Bigger Pages ?

32-bit address space again, but this time assume 16KB pages instead of 4KB

18-bit VPN + 14-bit offset -- each PTE (4 bytes) -- 2^ 18 entries in our linear page table -- total size of **1MB** per page table [Reduction of a factor of 4 !]

Big pages lead to waste within each page, a problem known as **internal fragmentation**(as the waste is internal to the unit of allocation)

Applications endup allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages.

Problem of invalid regions

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Code page (VPN 0) is mapped to physical page 10, the heap page (VPN 4) to physical page 23, & two stack pages at the other end of the address space (VPNs 14 and 15) are mapped to physical pages 28 and 4, respectively.

Most of the page table is unused, full of i**nvalid entries**. What a waste! And this is for a tiny 16KB address space.

32-bit address space and all the potential wasted space --??

Multi-level Page tables

Chop up the page table into **page-sized units**; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. [Page table is itself split into smaller chunks!]

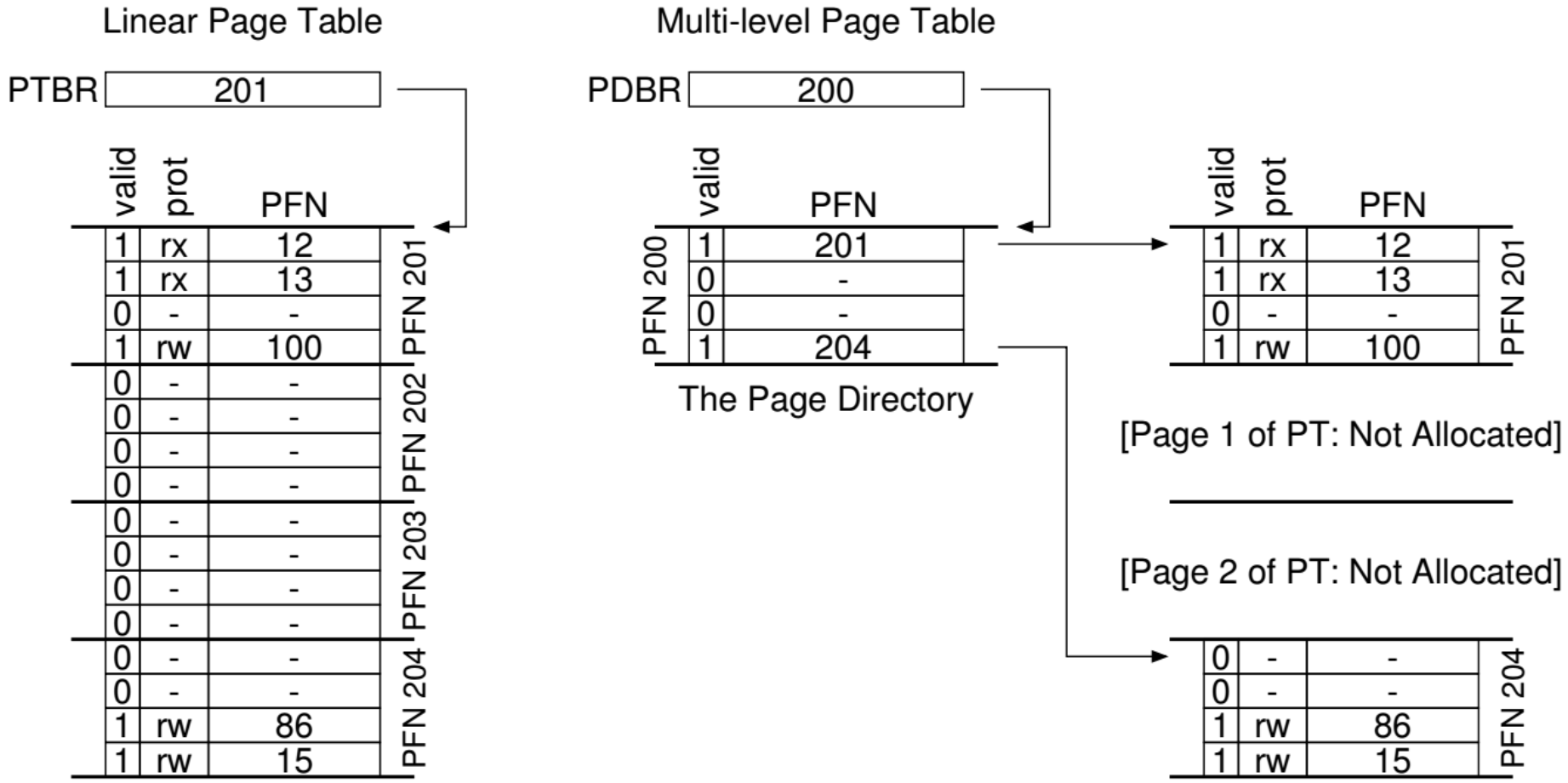
To track whether a page of the page table is valid (and if valid,where it is in memory), use a new structure, called the **page directory**.

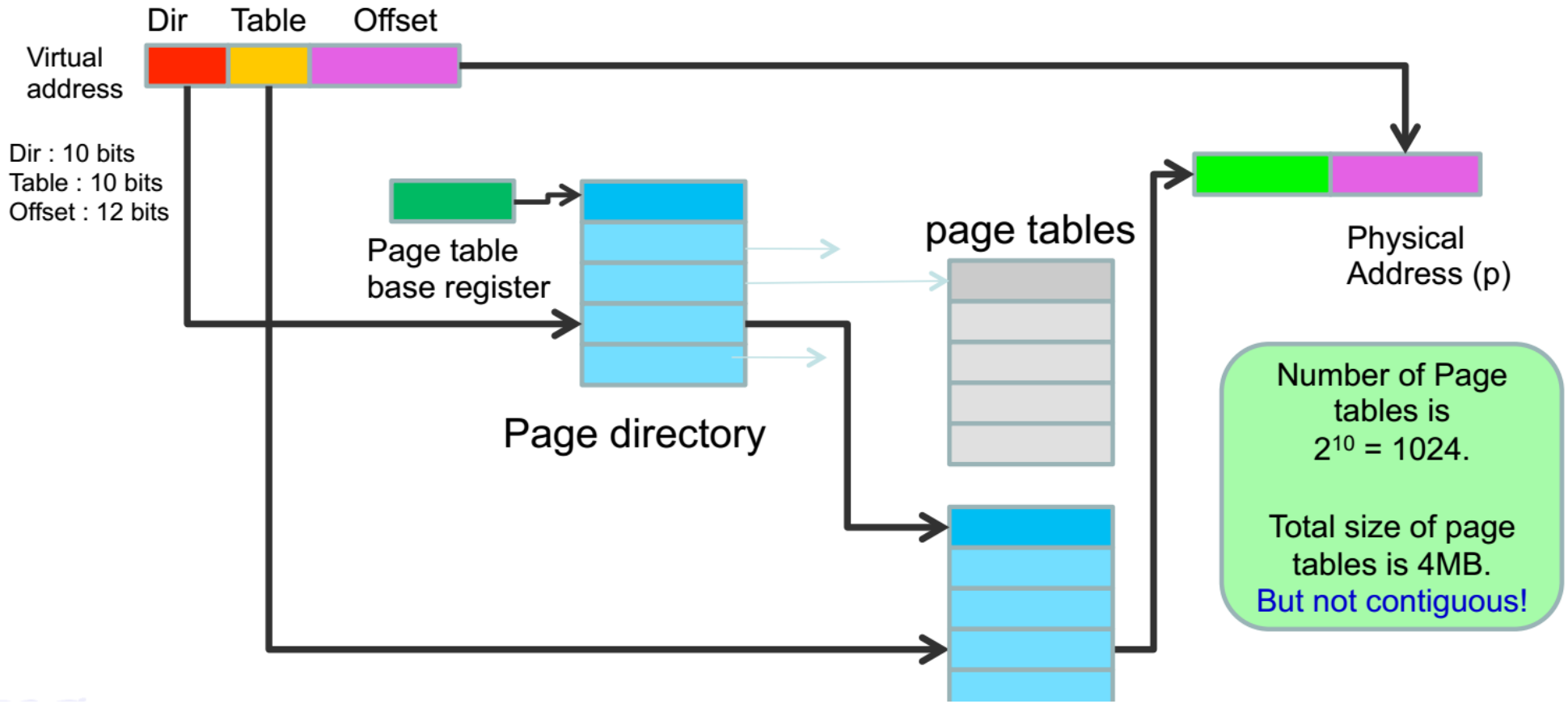
The page directory either can be used to tell you where a page of the page table is, or that the entire page of the page table contains no valid pages.

Crux:

- A page table is spread over many pages
- An "outer" page table or page directory tracks the PFNs of the page table pages

Multilevel page tables




Dir
Table
Offset
Virtual address
Dir : 10 bits
Table : 10 bits
Offset : 12 bits
Page table base register
Page directory
page tables
Physical Address (p)
Number of Page tables is 2^{10} = 1024.
Total size of page tables is 4MB.
But not contiguous!

• The page directory, in a simple two-level table, contains one entry per page of the page table. It consists of a number of page directory entries (PDE).

•A PDE has a valid bit and a page frame number (PFN), similar to a PTE.

•Meaning of this valid bit is slightly different: if the PDE is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN)is valid

• In at least one PTE on that page pointed to by this PDE, the valid bit in that PTE is set to one. If the PDE is not valid (i.e., equal to zero), the rest of the PDE is not defined.

Advantages

Multi-level table only allocates page-table space in proportion to the amount of address space you are using -- generally compact and supports sparse address spaces.

If carefully constructed, each portion of the page table fits neatly within a page, making it easier to manage memory -- the OS can simply grab the next free page when it needs to allocate or grow a page table.

With a multi-level structure, we add a level of indirection through use of the page directory, which points to pieces of the page table; that indirection allows to place the page-table pages wherever we would like in physical memory [No contiguous free memory is required]

Disadvantages

On a TLB miss, two loads from memory will be required to get the right translation information from the page table (one for the page directory, and one for the PTE itself).

Time-space trade-off -- We wanted smaller tables [got them but not for free]; although in the common case (TLB hit), performance is obviously identical, a TLB miss suffers from a higher cost with a smaller table.

Complexity -- Page-table lookups are more complicated in order to save valuable memory

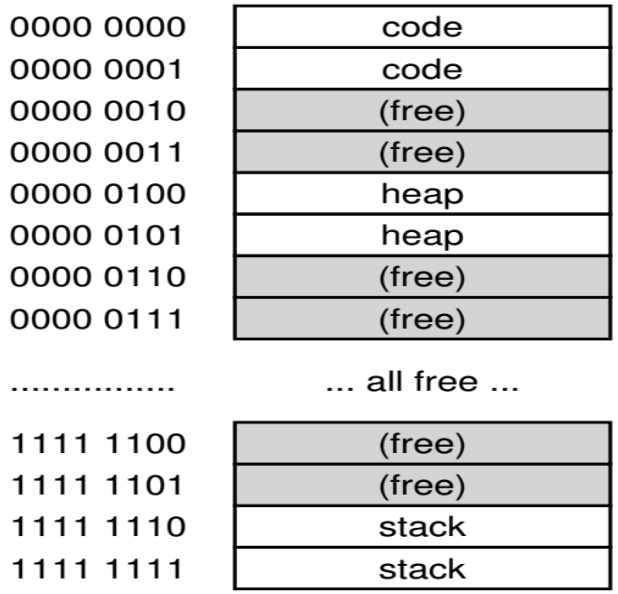
Example

Imagine a small address space of size **16KB**, with **64-byte** pages.

14-bit virtual address space, with 8 bits for the VPN and 6 bits for the offset.

A linear page table would have 2^8 (256) entries, even if only a small portion of the address space is in use.

Page table is **1KB** (256 × 4 bytes) in size.



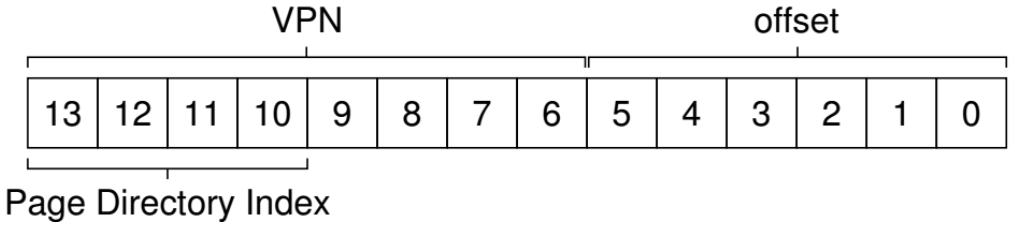
Given that we have **64**-byte pages, the 1KB page table can be divided into **16** 64-byte pages; **each page can hold 16 PTEs.**

How to take a VPN and use it to index first into the page directory and then into the page of the page table ?

Our page table in this example is small: 256 entries, spread across 16 pages.

The page directory needs one entry per page of the page table -- 16 entries.

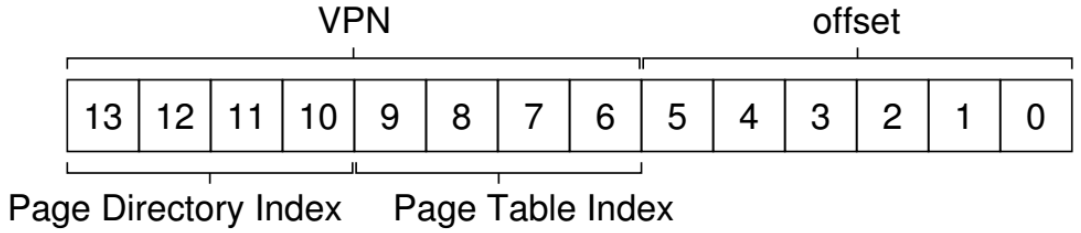
Four bits of the VPN to index into the directory; we use the top four bits of the VPN



Page director index from VPN -- use it to find the address of the page-directory entry (PDE) : **`PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))`**

Now, we have to fetch the pagetable entry (PTE) from the page of the page table pointed to by this pagedirectory entry.

To find this PTE, we have to index into the portion of the page table using the remaining bits of the VPN



Page-table index (PTIndex for short) can then be used to index into the page table itself, giving us the address of our PTE: PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Trivia

Consider a system with paging-based memory management, whose architecture allows for a **4GB virtual address space for processes.** The size of logical pages and physical frames is **4KB**. The system has **8GB** of physical RAM. The system allows a maximum of **1K (=1024)** processes to run concurrently. Assuming the OS uses hierarchical paging, calculate the maximum memory space required to store the page tables of all processes in the system. Assume that each pagetable entry requires an **additional 10 bits (beyond the frame number)** to store various flags. Assume page table entries are rounded up to the nearest byte. Consider the memory required for both outer and inner page tables in your calculations

Number of physical frames =2^33/2^12=2^21

Each PTE has frame number (21 bits) and flags(10 bits)≈4 bytes.

The total number of pages per process is2^32/2^12=2^20

so total size of innerpage table pages is2^20×4= 4MB.

Each page can hold2^12/4=2^ 10PTEs, so we need 2^20/2^10 PTEs to point to inner page tables, which will fit in a single outer page table.

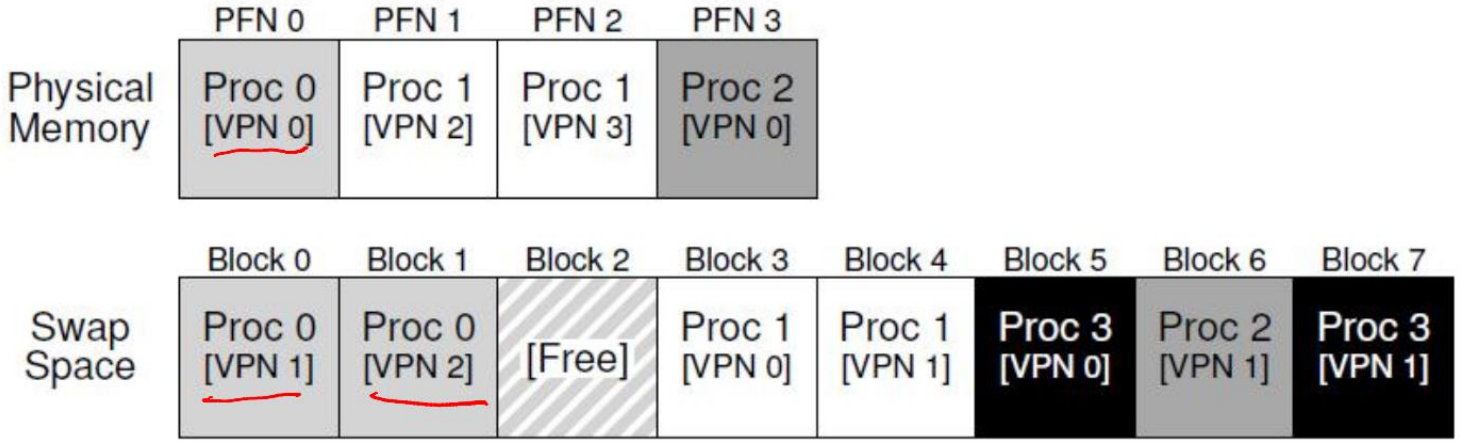
So the total size of page tables of one process is 4MB+ 4KB. For 1K process, the total memory consumed by page tables is 4GB + 4MB

Inverted Page Tables

- Instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system.
- The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.
- Finding the correct entry is now a matter of searching through this data structure.
- A linear scan would be expensive, and thus a hash table is often built over the base structure to speed up lookups. **[PowerPC is one example of such an architecture]**

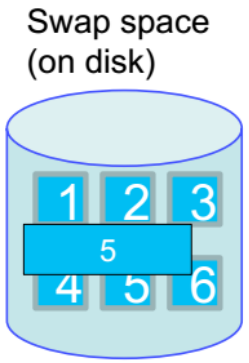
Is main memory always enough ?

- Are all pages of all active processes always in main memory?
  - Not necessary, with large address spaces
- OS uses a part of disk (swap space) to store pages that are not in active use



- Reserve some space on the disk for moving pages back and forth. Referred as swap space, because we swap pages out of memory to it and swap pages into memory from it.
- OS can read from and write to the swap space, in page-sized units [OS will need to remember the disk address of a given page]
- The size of the swap space is important – determines the maximum number of memory pages that can be in use by a system at a given time.

Demand paging



process page table in RAM

block	page frame	P
1	14	1
2		0
3		0
4		0
5	1	0
6	8	1

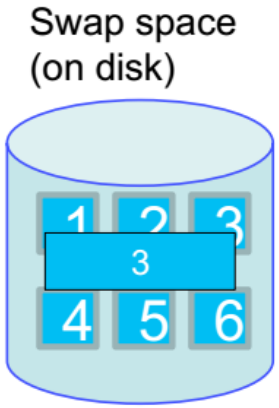
present bit

Pages are loaded from disk to RAM, only when needed.

A 'present bit' in the page table indicates if the block is in RAM or not.

If (present bit = 1){ block in RAM}
else {block not in RAM}

If a page is accessed that is not present in RAM, the processor issues a page fault interrupt, triggering the OS to load the page into RAM and mark the present bit to 1



process page table in RAM

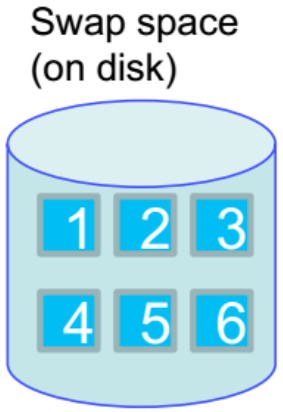
block	page frame	P
1	14	0
2	2	1
3	14	1
4	4	1
5	1	1
6	8	1

If there are no pages free for a new block to be loaded, the OS makes a decision to remove another block from RAM.

This is based on a replacement policy, implemented in the OS.

Some replacement policies are
* First in first out
* Least recently used
* Least frequently used

The replaced block **may** need to be written back to the swap (swap out)



process page table in RAM

block	page frame	P	D
1	14	0	1
2	2	1	1
3	14	1	0
4	4	1	1
5	1	1	0
6	8	1	1

The **dirty bit**, in the page table indicates if a page needs to be written back to disk

If the dirty bit is 1, indicates the page needs to be written back to disk.

Page Fault

The act of accessing a page that is not in physical memory is referred to as a page fault

- Present bit in page table entry: indicates if a page of a process resides in memory or not
- When translating VA to PA, MMU reads present bit
- If page present in memory, directly accessed
- If page not in memory, MMU raises a trap to the OS –page fault

Page Fault Handling

- Page fault traps OS and moves CPU to kernel mode
- OS fetches disk address of page and issues read to disk
  - OS keeps track of disk address (say, in page table)
- OS context switches to another process
  - Current process is blocked and cannot run
- When disk read completes, OS updates page table of process, and marks it as ready
- When process scheduled again, OS restarts the instruction that caused page fault

Summary: what happens on memory access

- CPU issues load to a VA for code or data
  - –Checks CPU cache first
  - –Goes to main memory in case of cache miss
- MMU looks up TLB for VA
  - –If TLB hit, obtains PA, fetches memory location and returns to CPU (via CPU caches)
  - –If TLB miss, MMU accesses memory, walks page table, and obtains page table entry
    - If present bit set in PTE, accesses memory
    - If not present but valid, raises page fault. OS handles page fault and restarts the CPU load instruction
    - If invalid page access, trap to OS for illegal access

Subtle point

While the I/O is in flight, the process will be in the blocked state.

Thus, the OS will be free to run other ready processes while the page fault is being serviced. Because I/O is expensive, this overlap of the I/O (page fault) of one process and the execution of another process

Another way a multiprogrammed system can make the most effective use of its hardware

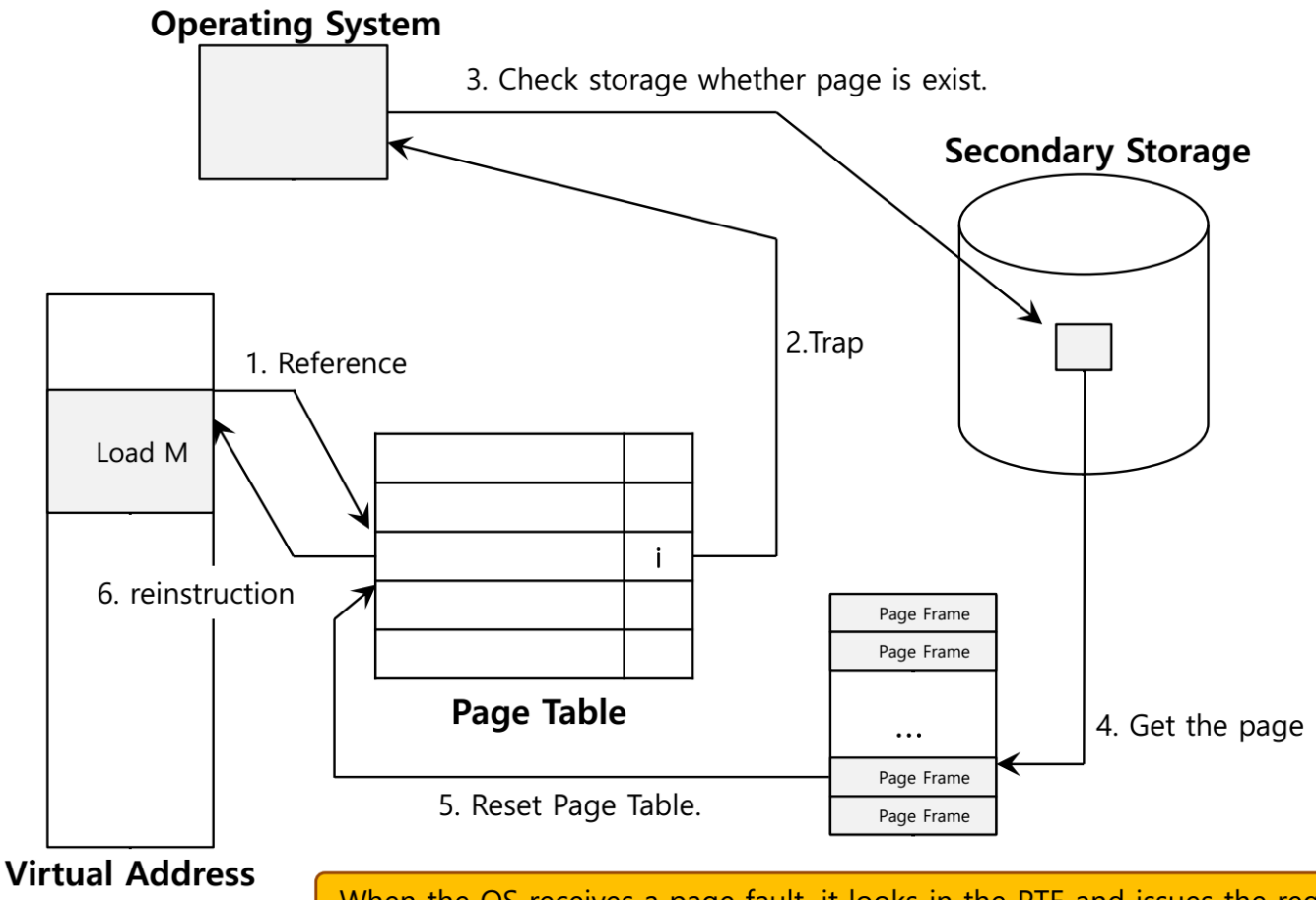
More Complications in a Page fault

- When servicing page fault, what if OS finds that there is no free page to swap in the faulting page?
- OS must swap out an existing page (if it has been modified, i.e., dirty) and then swap in the faulting page–too much work!
- OS may proactively swap out pages to keep list of free pages handy
- Which pages to swap out? Decided by page replacement policy.

March 22nd

- How to run process when not enough physical memory?
- When should a page be moved from disk to memory?
- What page in memory should be replaced?
- How can the LRU page be approximated efficiently?

Page Fault Control Flow



When the OS receives a page fault, it looks in the PTE and issues the request to disk.

- The OS must find a physical frame for the soon-be-faulted-in page to reside within.
- If there is no such page, waiting for the replacement algorithm to run and kick some pages out of memory.

- OS waits until memory is entirely full, and only then replaces a page to make room for some other page
  - This is a little bit unrealistic, and there are many reason for the OS to keep a small portion of memory free more proactively.

Swap Daemon, Page Daemon

To keep a small amount of memory free, most operating systems have some kind of high watermark(HW) and low watermark(LW) to help decide when to start evicting pages from memory.

How this works :

When the OS notices that there are fewer than LW pages available, a background thread that is responsible for freeing memory runs.

The thread evicts pages until there are HW pages available.

The background thread, sometimes called the swap daemon or page daemon, then goes to sleep, after it has freed some memory for running processes and the OS to use.

- Memory pressure forces the OS to start paging out pages to make room for actively-used pages.
- Deciding which page to evict is encapsulated within the replacement policy of the OS.

Cache Management

- Goal in picking a replacement policy for this cache is to minimize the number of cache misses.
- The number of cache hits and misses let us calculate the *average memory access time(AMAT)*.

$$AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D)$$

Arguement	Meaning
T_M	The cost of accessing memory
T_D	The cost of accessing disk
P_{Hit}	The probability of finding the data item in the cache(a hit)
P_{Miss}	The probability of not finding the data in the cache(a miss)

Given that main memory holds some subset of all the pages in the system, it can be viewed as a cache for virtual memory pages in the system.

Minimize the number of times that we have to fetch a page from disk

or

Maximize the number of times a page that is accessed is found in memory

Virtual Memory Policies

Goal: Minimize number of page faults

Page faults require milliseconds to handle (reading from disk)

Implication: Plenty of time for OS to make good decision

OS has two decisions

Page selection

When should a page (or pages) on disk be brought into memory?

Page replacement

Which resident page (or pages) in memory should be thrown out to disk?

Page replacement policies

- **Optimal:** replace page not needed for longest time in future (not practical!)
- **FIFO:** replace page that was brought into memory earliest (may be a popular page!)
- **LRU/LFU:** replace the page that was least recently (or frequently) used in the past

Page Selection

Demand paging: Load page only when page fault occurs

- Intuition: Wait until page must absolutely be in memory
- When process starts: No pages are loaded in memory
- Problems: Pay cost of page fault for every newly accessed page

Prepaging (anticipatory, prefetching): Load page before referenced

- OS predicts future accesses (oracle) and brings pages into memory early
- Works well for some access patterns (e.g., sequential)
- Problems?

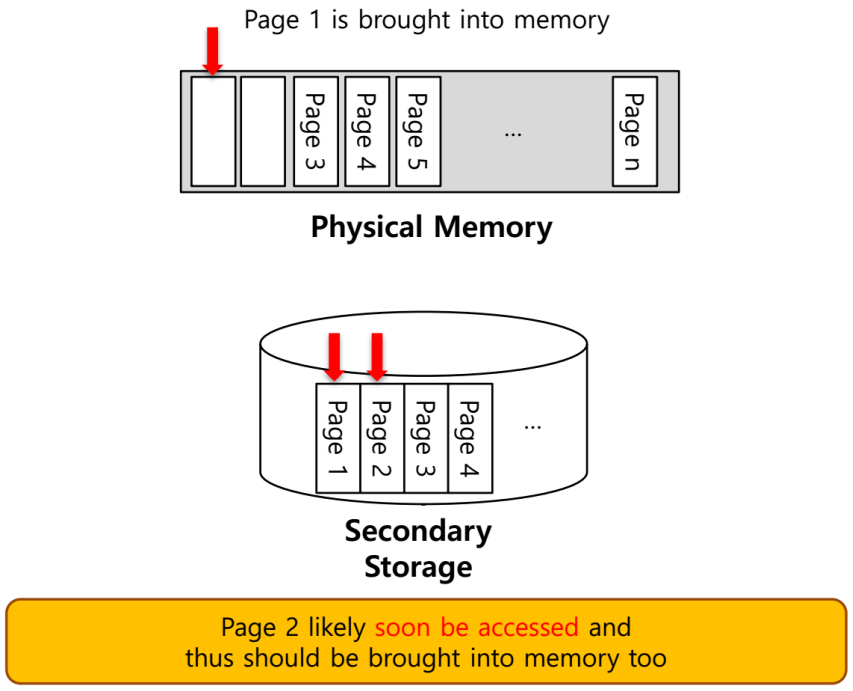
Hints: Combine above with user-supplied hints about page references

- User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
- Example: madvise() in Unix

When should a page be brought from disk into memory?

Prefecthing/Prepaging

- The OS guess that a page is about to be used, and thus bring it in ahead of time.



Page Replacement

Which page in main memory should selected as victim?

- Write out victim page to disk if modified (dirty bit set)
- If victim page is not modified (clean), just discard

OPT: Replace page not used for longest time in future

- Advantages: Guaranteed to minimize number of page faults
- Disadvantages: Requires that OS predict the future; Not practical, but good for comparison

FIFO: Replace page that has been in memory the longest

- Intuition: First referenced long time ago, done with it now
- Advantages: Fair: All pages receive equal residency; Easy to implement (circular buffer)
- Disadvantage: Some pages may always be needed

LRU: Least-recently-used: Replace page not used for longest time in past

- Intuition: Use past to predict the future
- Advantages: With locality, LRU approximates OPT
- Disadvantages:
  - Harder to implement, must track which pages have been accessed
  - Does not handle all workloads well

Tracing the Optimal Policy

Assume a program accesses the following stream of virtual pages:

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Assuming a cache that fits three pages

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{54.6\%}$

Future is not known.

Tracing the FIFO Policy

Assume a program accesses the following stream of virtual pages:

Reference Row

0 1 2 0 1 3 0 3 1 2 1

Assuming a cache that fits three pages

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss		3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate is $\frac{Hits}{Hits+Misses} = \mathbf{36.4\%}$

Even though page 0 had been accessed a number of times, FIFO still kicks it out.

Belady's Anomaly

Interesting reference stream that behaved a little unexpectedly

The memory-reference stream: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**.

The replacement policy they were studying was FIFO.

The interesting part: how the cache hit rate changed when moving from a cache size of 3 to 4 pages.

Belady's Anomaly

- We would expect the cache hit rate to increase when the cache gets larger. But in this case, with FIFO, it gets worse.

Reference Row

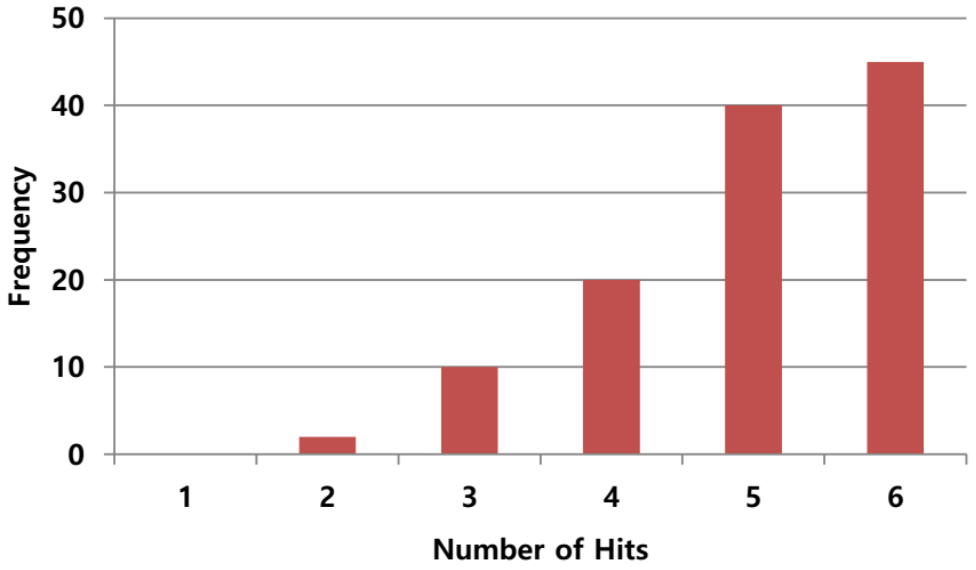
1 2 3 4 1 2 5 1 2 3 4 5

Random Policy

- Picks a random page to replace under memory pressure.
  - It doesn't really try to be too intelligent in picking which blocks to evict.
  - Random does depends entirely upon how lucky Random gets in its choice.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	3	2,0,1
2	Hit		2,0,1
1	Hit		2,0,1

- Sometimes, Random is as good as optimal, achieving 6 hits on the example trace.



Random Performance over 10,000 Trials

Need for better ?

FIFO or Random is likely to have a common problem: it might kick out an important page, one that is about to be referenced again.

FIFO kicks out the page that was first brought in; if this happens to be a page with important code or data structures upon it, it gets thrown out anyhow, even though it will soon bepaged back in.

Thus, FIFO, Random, and similar policies are not likely to approach optimal;

Something smarter is needed

Using History

- Lean on the past and use **history**.
  - Two type of historical information.

Historical Information	Meaning	Algorithms
recency	The more recently a page has been accessed, the more likely it will be accessed again	LRU
frequency	If a page has been accessed many times, It should not be replcaed as it clearly has some value	LFU

Principle of locality

An observation about programs and their behavior.

What this principle says --

Programs tend to access certain code sequences (e.g., in a loop) and datastructures (e.g., an array accessed by the loop) quite frequently;

Spatial locality -- states that if a page P is accessed,it is likely the pages around it (say P−1 or P+ 1) will also likely be accessed.

Temporal locality -- states that pages that have been accessed in the near past are likely to be accessed again in the near future.

Use history to figure out which pages are important, and keep those pages in memory when it comes to eviction time.

Using History: LRU

- Replaces the least-recently-used page.

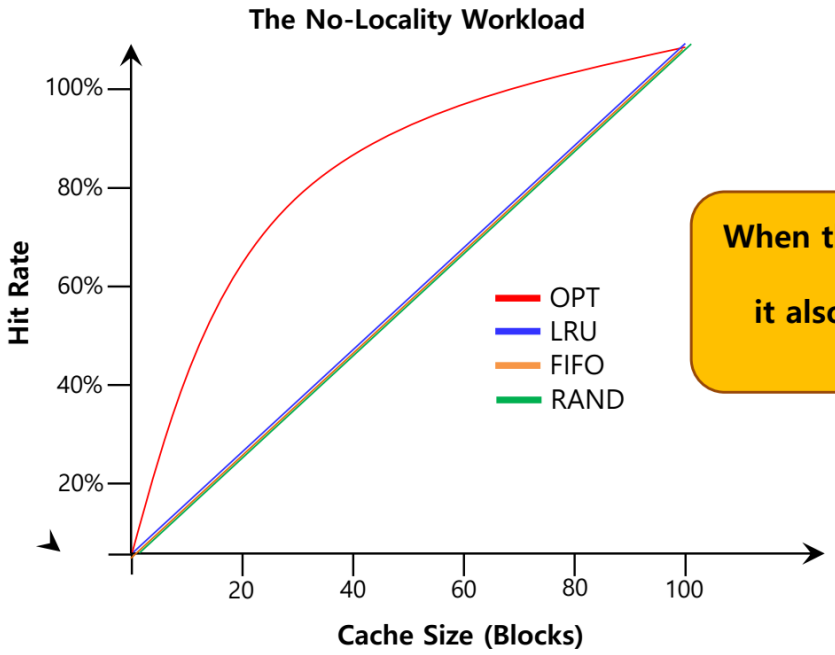
Reference Row

0 1 2 0 1 3 0 3 1 2 1

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

The No-locality workload

- Each reference is to a random page within the set of accessed pages.
  - Workload accesses 100 unique pages over time.
  - Choosing the next page to refer to at random



When the cache is large enough to fit the entire workload, it also doesn't matter which policy you use.

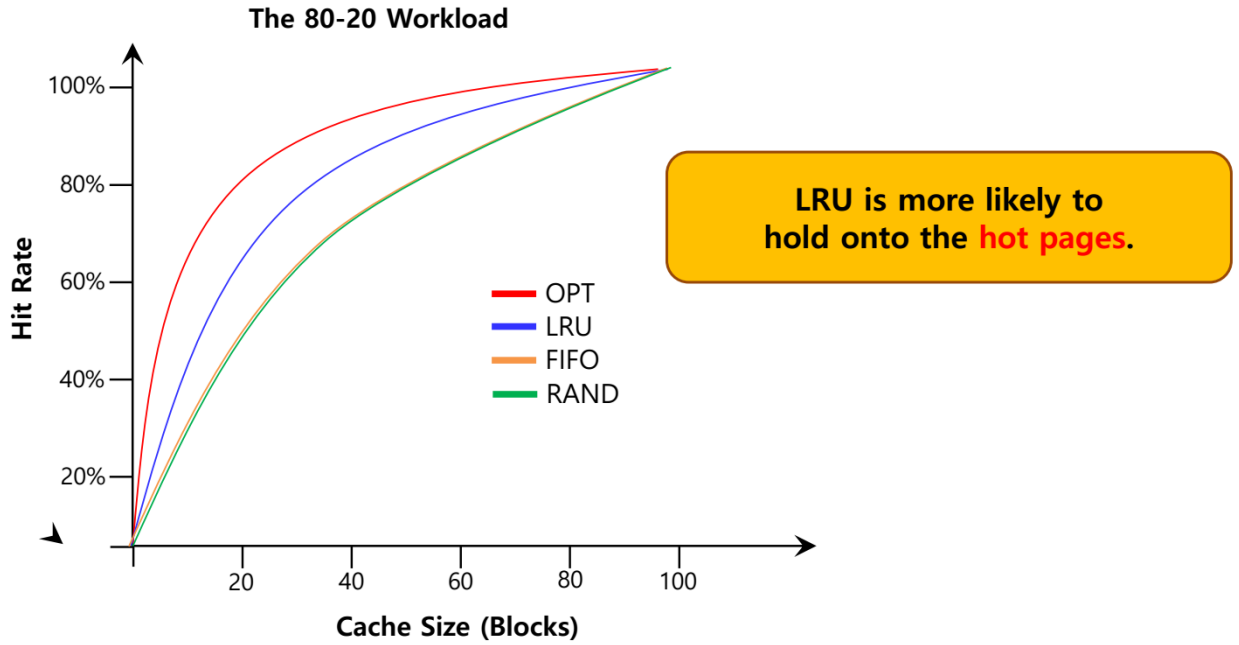
Overall, 10,000 pages are accessed.

Cache size varied from very small (1 page) to enough to hold all the unique pages (100 page)

It doesn't matter much which realistic policy you are using; LRU, FIFO, and Random all perform the same, with the hit rate exactly determined by the size of the cache.

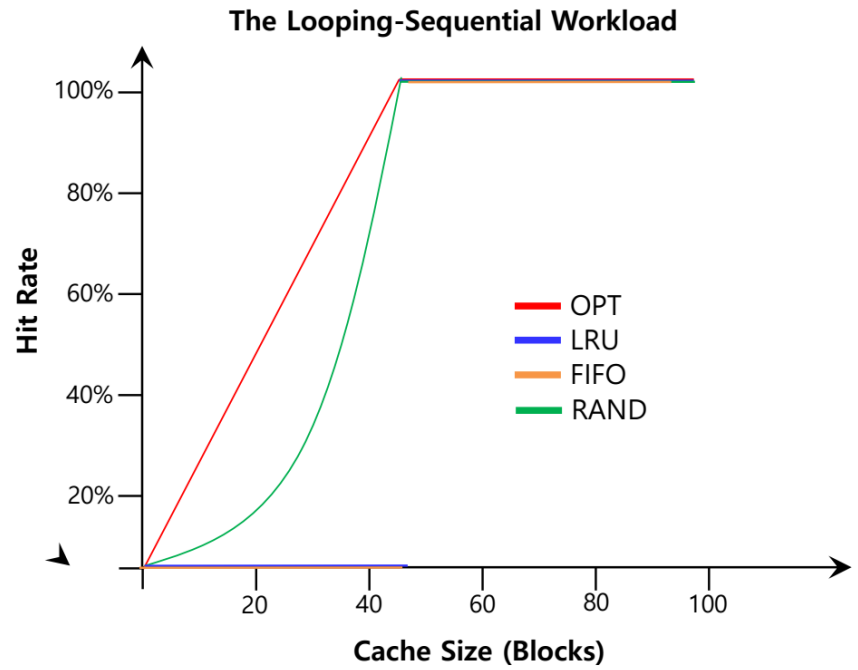
80-20 Workload

- Exhibits locality: 80% of the reference are made to 20% of the page
- The remaining 20% of the reference are made to the remaining 80% of the pages.



Looping Sequential

- Refer to 50 pages in sequence.
  - Starting at 0, then 1, … up to page 49, and then we Loop, repeating those accesses, for total of 10,000 accesses to 50 unique pages.



FIFO & LRU under a looping-sequential workload, kick out older pages; unfortunately, due to the looping nature of the workload, these older pages are going to be accessed sooner than the pages that the policies prefer to keep in cache

How is LRU implemented ?

- OS is not involved in every memory access
  - –how does it know which page is LRU? [To keep track of which pages have been least- and most-recently used, the system has to do some accounting work on every memory reference]
- Hardware help and some approximations
- MMU sets a bit in PTE (“accessed” bit) when a page is accessed
- OS periodically looks at this bit to estimate pages that are active and inactive
- To replace, OS tries to find a page that does not have access bit set –May also look for page with dirty bit not set (to avoid swapping out to disk)

Software Perfect LRU

- OS maintains ordered list of physical pages by reference time
- When page is referenced: Move page to front of list
- When need victim: Pick page at back of list
- Trade-off: Slow on memory reference, fast on replacement

Hardware Perfect LRU

- Associate timestamp register with each page
- When page is referenced: Store system clock in register
- When need victim: Scan through registers to find oldest clock
- Trade-off: Fast on memory reference, slow on replacement (especially as size of memory grows)

In practice, do not implement Perfect LRU -- LRU is an approximation anyway, so approximate more Goal: Find an old page, but not necessarily the very oldest

Historical Algorithms

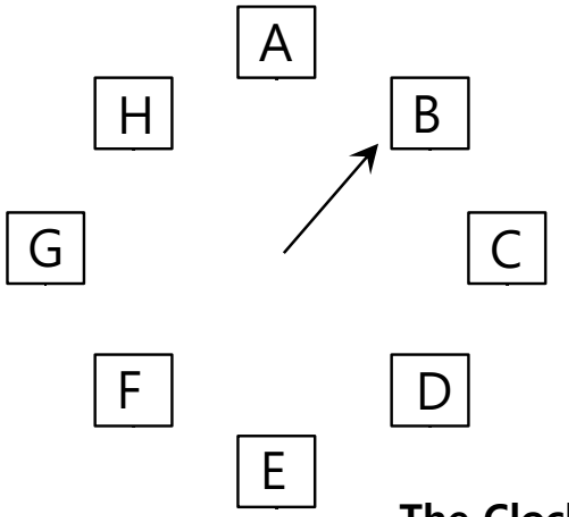
- To keep track of which pages have been least-and-recently used, the system has to do some accounting work on **every memory reference.**
  - Add a little bit of hardware support.

Approximating LRU

- Require some hardware support, in the form of a **use bit**
  - Whenever a page is referenced, the use bit is set by hardware to 1.
  - Hardware never clears the bit, though; that is the responsibility of the OS
- Clock Algorithm
  - All pages of the system arranges in a circular list.
  - A clock hand points to some particular page to begin with.

Clock Algorithm

- The algorithm continues until it finds a use bit that is set to 0.



Use bit	Meaning
0	Evict the page
1	Clear **Use bit** and advance hand

The Clock page replacement algorithm

When a page fault occurs, the page the hand is pointing to is inspected.
The action taken depends on the Use bit

Clock algorithm

Hardware

Keep use (or reference) bit for each page frame

When page is referenced: set use bit

Operating System

Page replacement: Look for page with use bit cleared (has not been referenced for awhile)

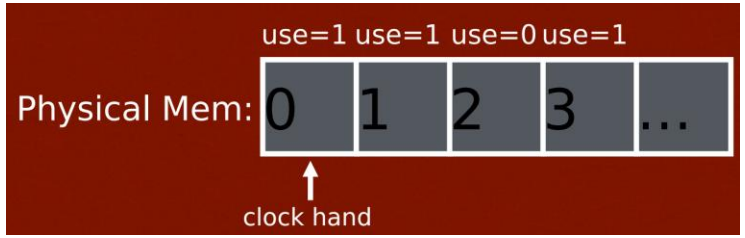
Implementation:

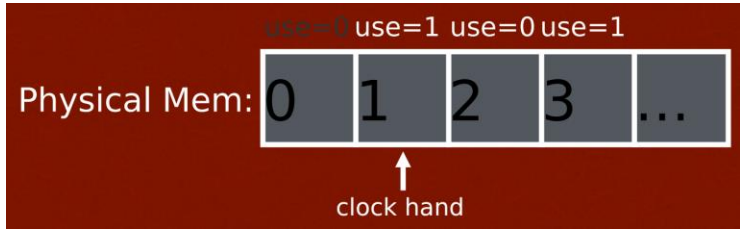
Keep pointer to last examined page frame

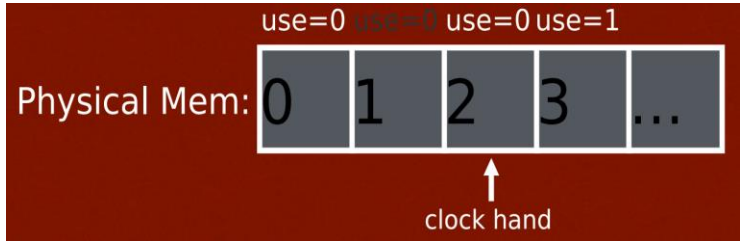
Traverse pages in circular buffer

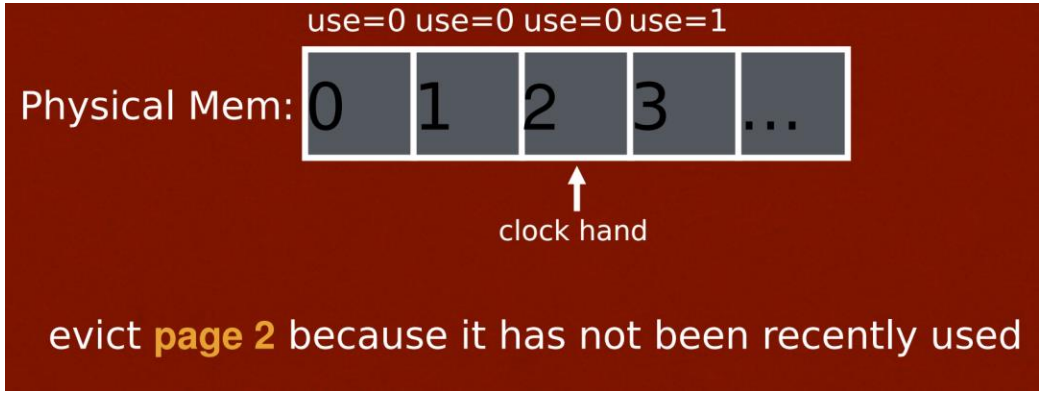
Clear use bits as search

Stop when find page with already cleared use bit, replace this page


use=1 use=1 use=0 use=1
Physical Mem:
0
1
2
3
...
clock hand

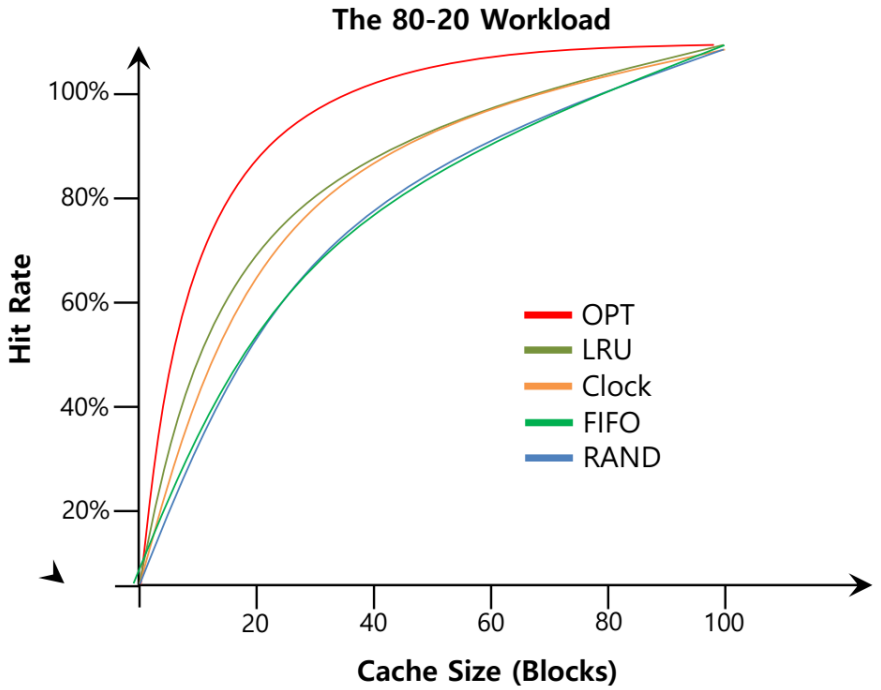

use=0 use=1 use=0 use=1
Physical Mem:
0
1
2
3
...
clock hand


use=0 use=0 use=0 use=1
Physical Mem:
0
1
2
3
...
clock hand


use=0 use=0 use=0 use=1
Physical Mem:
0
1
2
3
...
clock hand
evict page 2 because it has not been recently used

Workload – Clock Algorithm

- Clock algorithm doesn't do as well as perfect LRU, it does better then approach that don't consider history at all.



Clock extensions

Replace multiple pages at once

Intuition: Expensive to run replacement algorithm and to write single block to disk

Find multiple victims each time and track free list

Add software counter (“chance”)

Intuition: Better ability to differentiate across pages (how much they are being accessed)

Increment software counter if use bit is 0

Replace when chance exceeds some specified limit

Use dirty bit to give preference to dirty pages

Intuition: More expensive to replace dirty pages

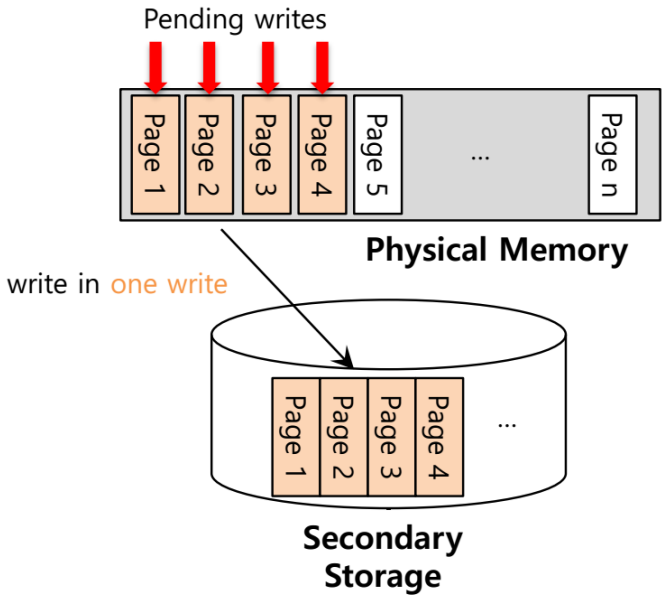
Dirty pages must be written to disk, clean pages do not

Replace pages that have use bit and dirty bit cleared

Algo could be modified to scan for pages that are both unused and clean to evict first; failing to find those, then for unused pages that are dirty, and so forth.

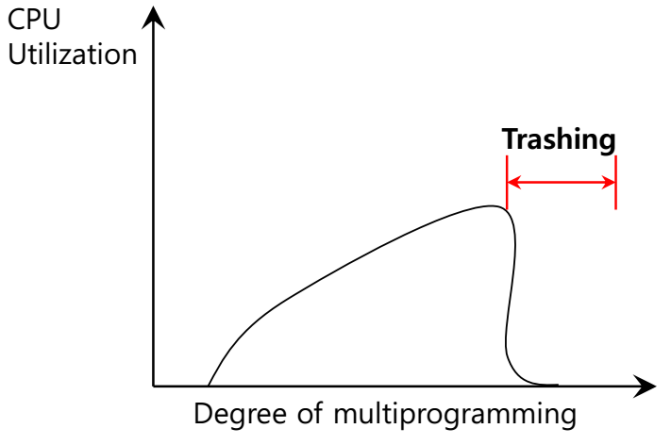
Clustering/Grouping

- Collect a number of pending writes together in memory and write them to disk in one write.
  - Perform a **single large write** more efficiently than **many small ones**.



Thrashing

- Memory is oversubscribed and the memory demands of the set of running processes exceeds the available physical memory.
  - Decide not to run a subset of processes.
  - Reduced set of processes working sets fit in memory.



Current systems take more aggressive approach to memory overload.

For example, some versions of Linux run an **out-of-memory killer** when memory is oversubscribed;

this daemon chooses a memory-intensive process and kills it, thus reducing memory.

While successful at reducing memory pressure, this approach can have problems,

if, for example, it kills the X server and thus renders any applications requiring the display unusable.

Conclusion

Illusion of virtual memory:

Processes can run when sum of virtual address spaces > amount of physical memory

Mechanism:

Extend page table entry with “present” bit

OS handles page faults (or page misses) by reading in desired page from disk

Policy:

Page selection – demand paging, prefetching, hints

Page replacement – OPT, FIFO, LRU, others

Implementations (clock) perform approximation of LRU

March 24, Free Space Management

malloc library (managing pages of a process's heap) or

the OS itself (managing portions of the address space of a process)

How should free space be managed, when satisfying variable-sized requests?

What strategies can be used to minimize fragmentation?

What are the time and space overheads of alternate approaches?

Assumptions

`void*malloc(size_t size)` takes a single parameter, **size**, which is the number of bytes requested by the application; it hands back a pointer (of no particular type, or a void pointer) to a region of that size (or greater)

`void free(void*ptr)` takes a pointer and frees the corresponding chunk.

Note the implication of the interface:

the user, when freeing the space, does not inform the library of its size;

the library must be able to figure out how big a chunk of memory is when handed just a pointer to it.

The space that this library manages is known historically as the **heap**,

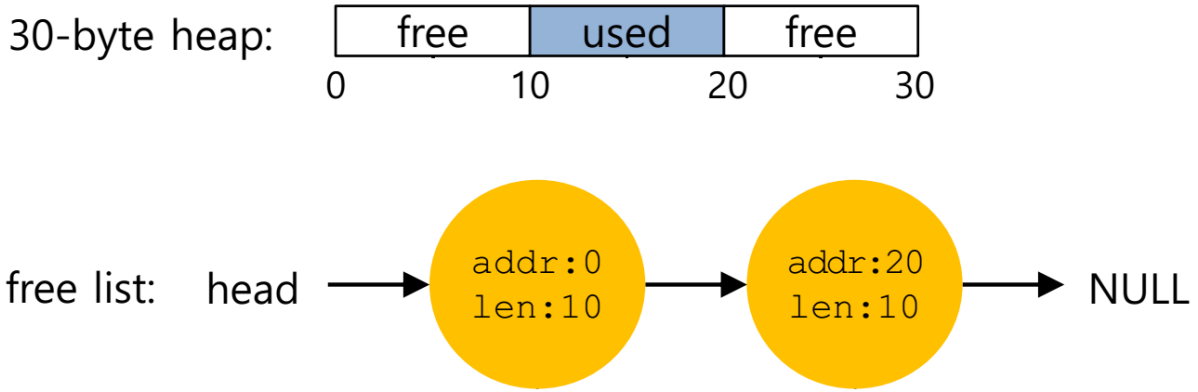
the generic data structure used to manage free space in the heap is some kind of free list.

once memory is handed out to a client, it cannot be relocated to another location in memory.

For example, if a program calls malloc()and is given a pointer to some space within the heap, that memory region is essentially “owned” by the program (and cannot be moved by the library) until the program returns it via a corresponding call to free().

Variable Sized allocation

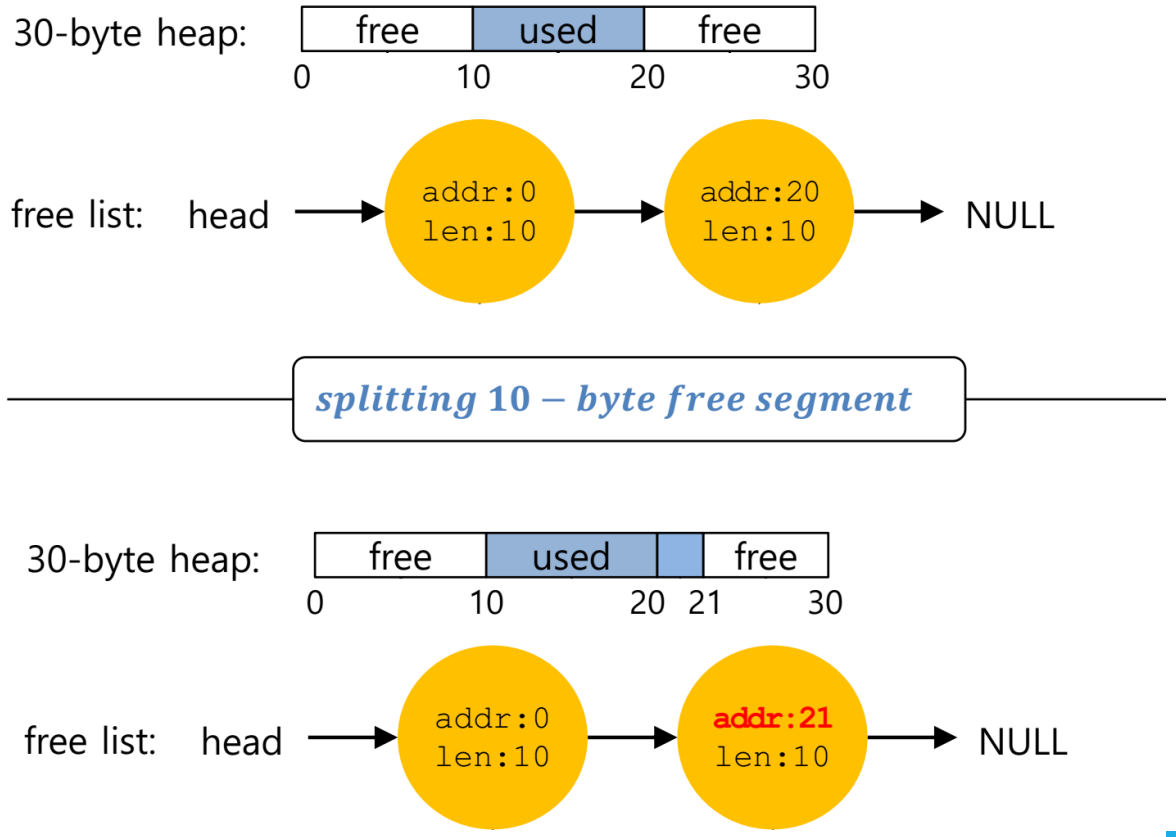
- Given a block of memory, how do we allocate it to satisfy various memory allocation requests?
- This problem must be solved in the C library
  - Allocates one or more pages from kernel via brk/sbrk or mmap system calls
  - Gives out smaller chunks to user programs via malloc
- This problem also occurs in the kernel
  - Kernel must allocate memory for its internal data structures



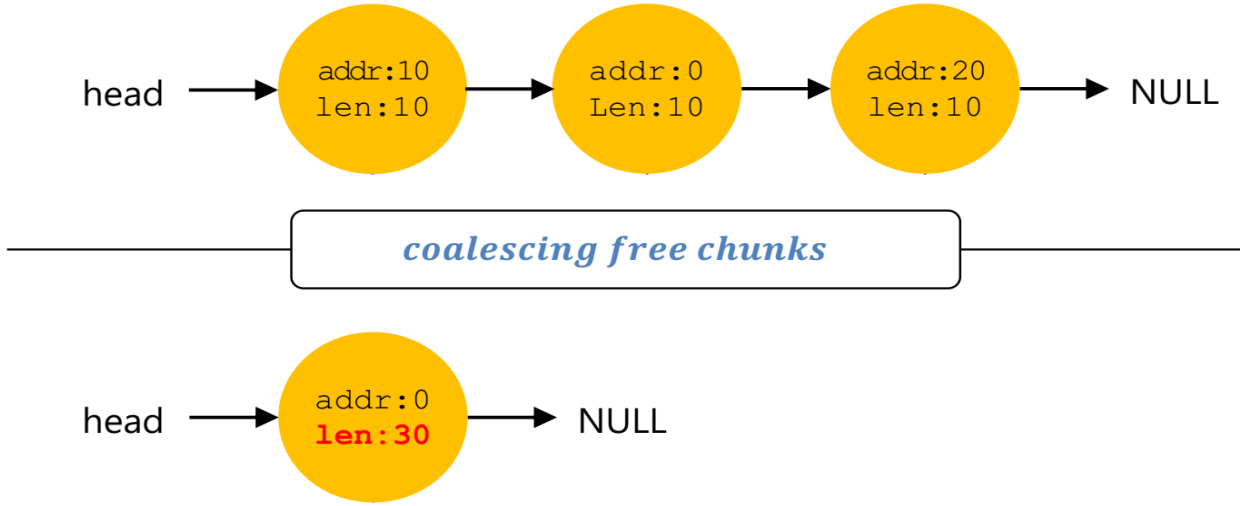
In this case, the total free space available is 20 bytes; unfortunately, it is fragmented into two chunks of size 10 each. As a result, a request for 15 bytes will fail even though there are 20 bytes free

Splitting and Coalescing

- Two 10-bytes free segment with **1-byte request**



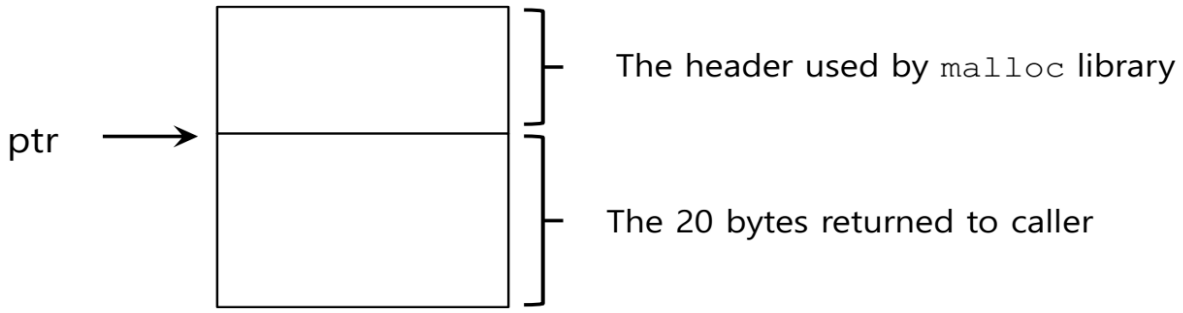
- If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



Tracking the size of allocated regions

- The interface to `free(void *ptr)` does **not take** a **size parameter**.
  - How does the library **know the size** of memory region that will be back **into free list**?
- Most allocators store **extra information** in a **header** block.

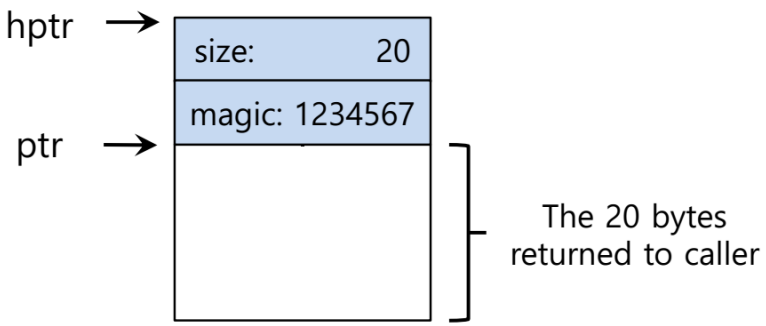
```
ptr = malloc(20);
```



An Allocated Region Plus Header

Header details

- The header minimally **contains the size** of the allocated memory region.
- The header may also contain
  - Additional pointers to speed up deallocation
  - A magic number for integrity checking



Specific Contents Of The Header

```
typedef struct __header_t {
        int size;
        int magic;
} header_t;
```

A Simple Header

- The **size** for free region is the **size of the header plus the size of the space** allocated to the user.
  - If a user **request `N` bytes**, the library searches for a free chunk of **size `N` plus the size of the header**

- Simple pointer arithmetic to find the header pointer.

```
void free(void *ptr) {
        header_t *hptr = (void *)ptr - sizeof(header_t);
}
```

Free list

Free space is managed as a list

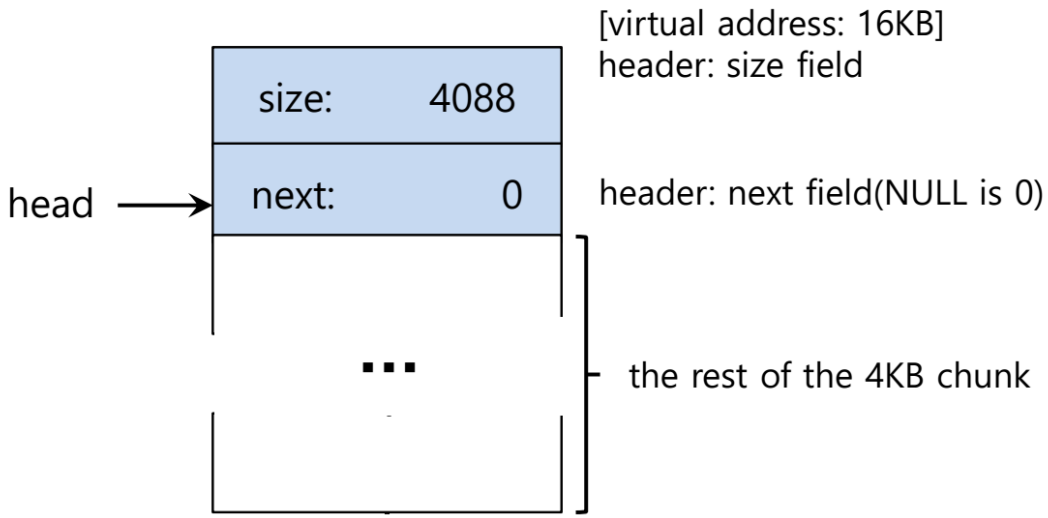
–Pointer to the next free chunk is

embedded within the free chunk

- The library tracks the head of the list

–Allocations happen from the head

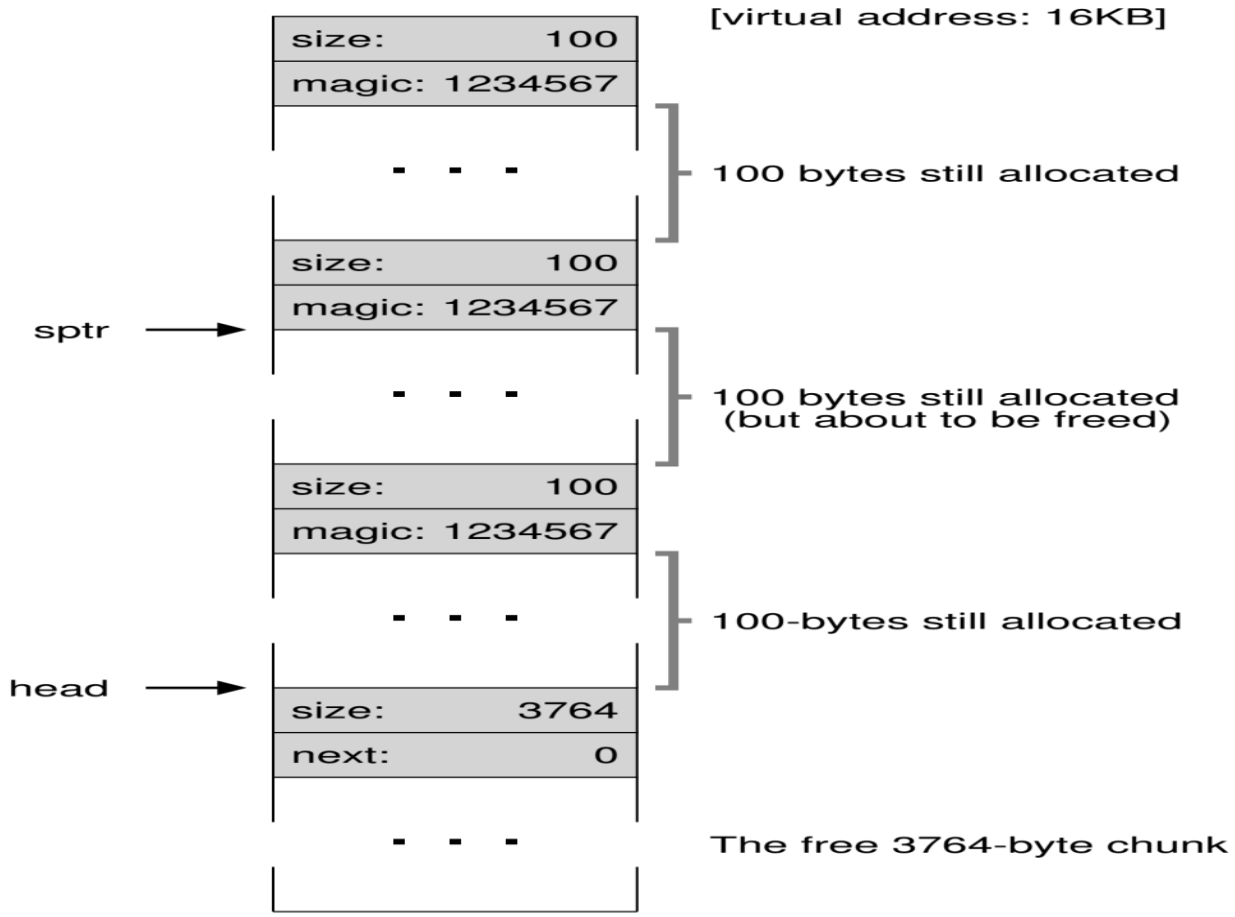
```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                                MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```



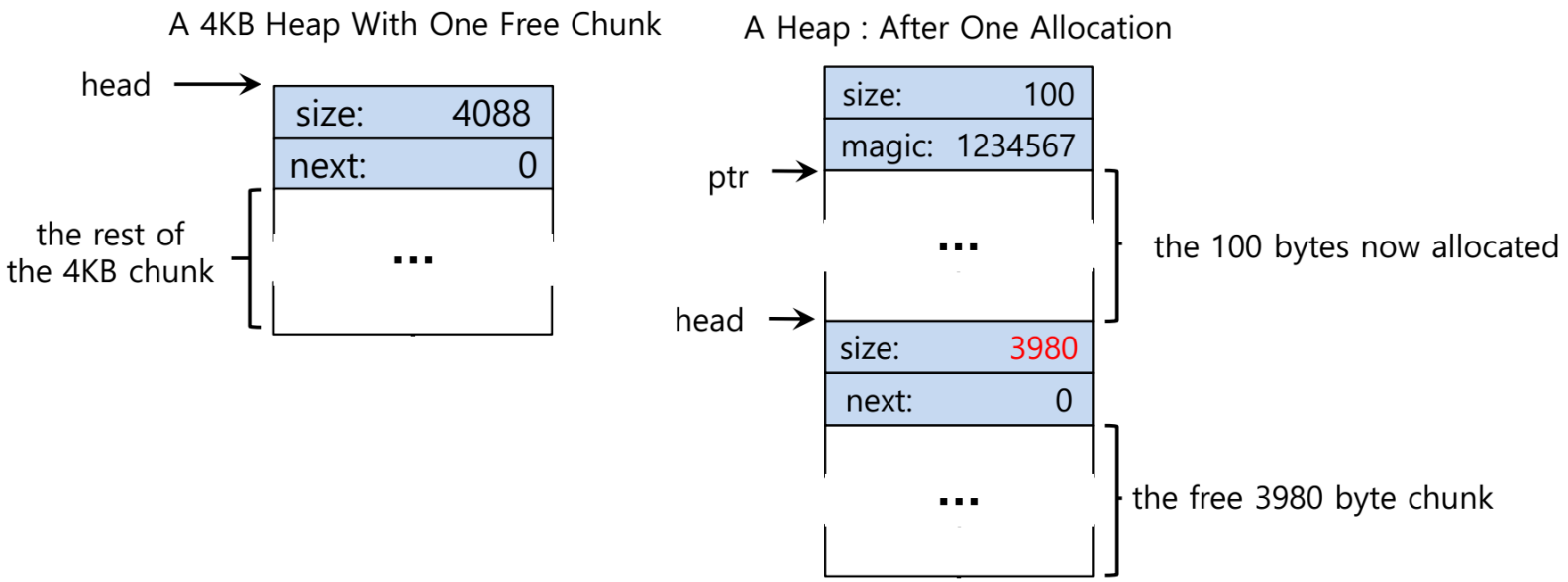
- If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request.

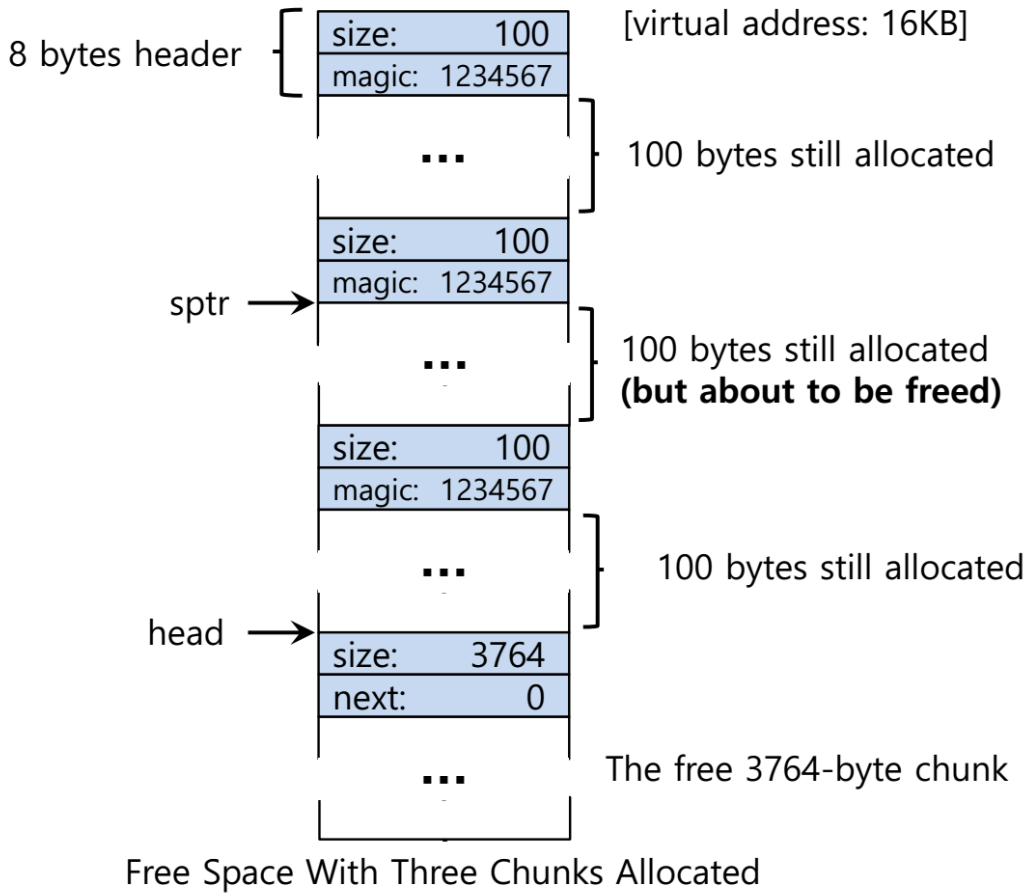
- The library will
  - **Split** the large free chunk into two.
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of free chunk in the list.

Free space - Three chunks allocated



- Example: a request for 100 bytes by `ptr = malloc(100)`
  - Allocating 108 bytes out of the existing one free chunk.
  - shrinking the one free chunk to 3980(4088 minus 108).





Free Space With Three Chunks Allocated

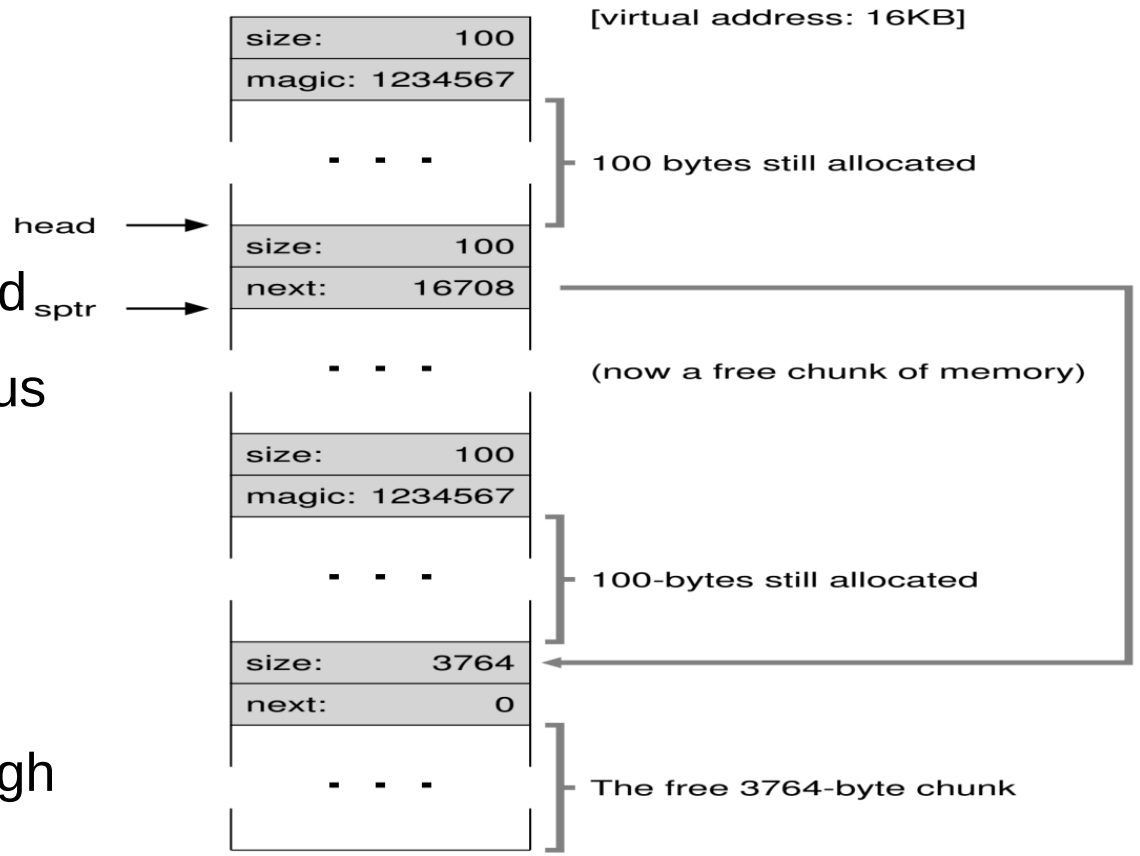
•Suppose 3 allocations of size 100 bytes each happen.

Then, the middle chunk pointed to by `sptr` is freed

•What is the free list?–It now has two non-contiguous elements

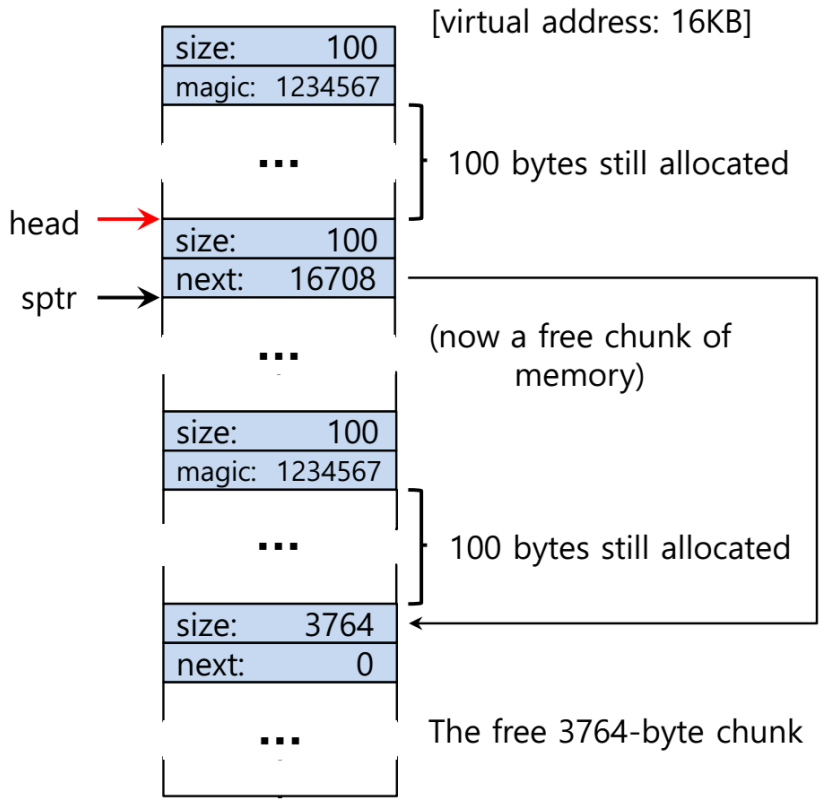
•Free space may be scattered around due to fragmentation

–Cannot satisfy a request for 3800 bytes even though we have the freespace



- Example: `free(sptr)`

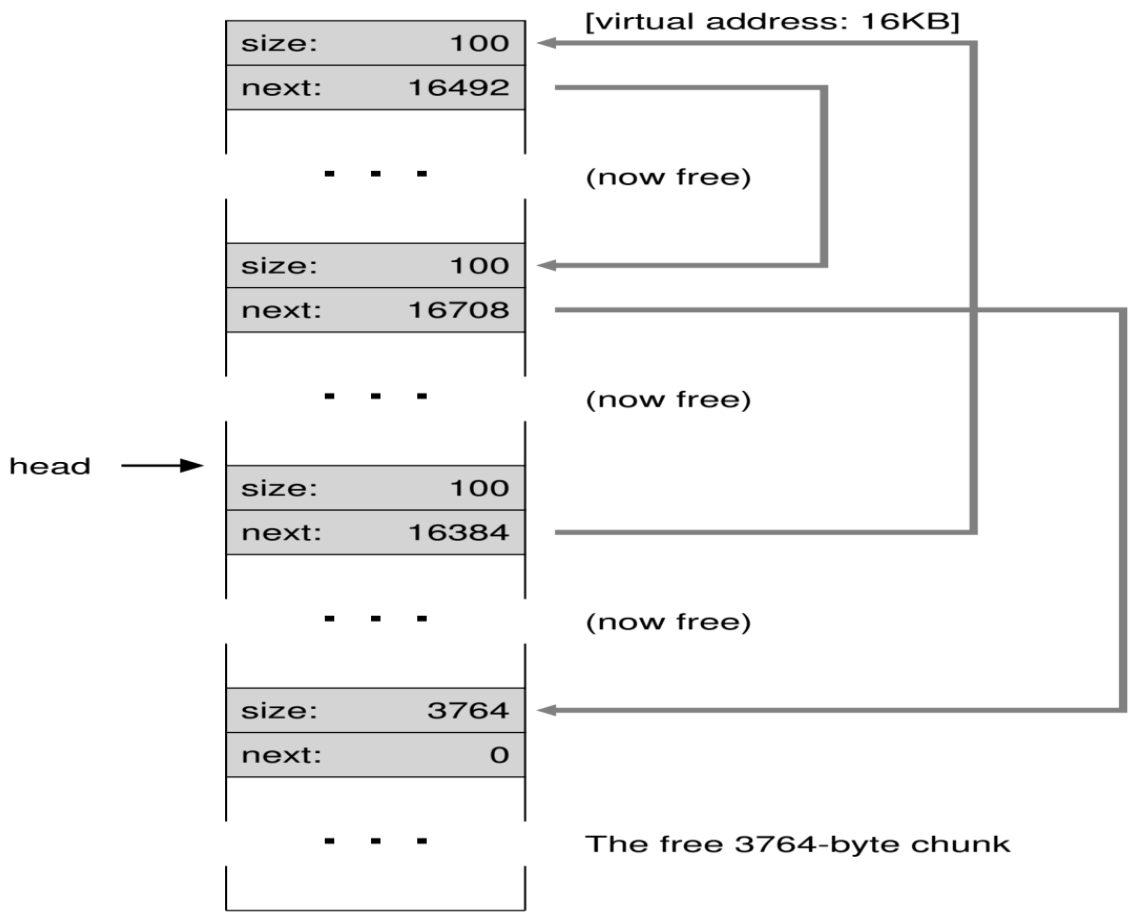
  - The 100 bytes chunks is **back into** the free list.
  - The free list will **start** with **a small chunk**.
    - The list header will point the small chunk



Coalescing

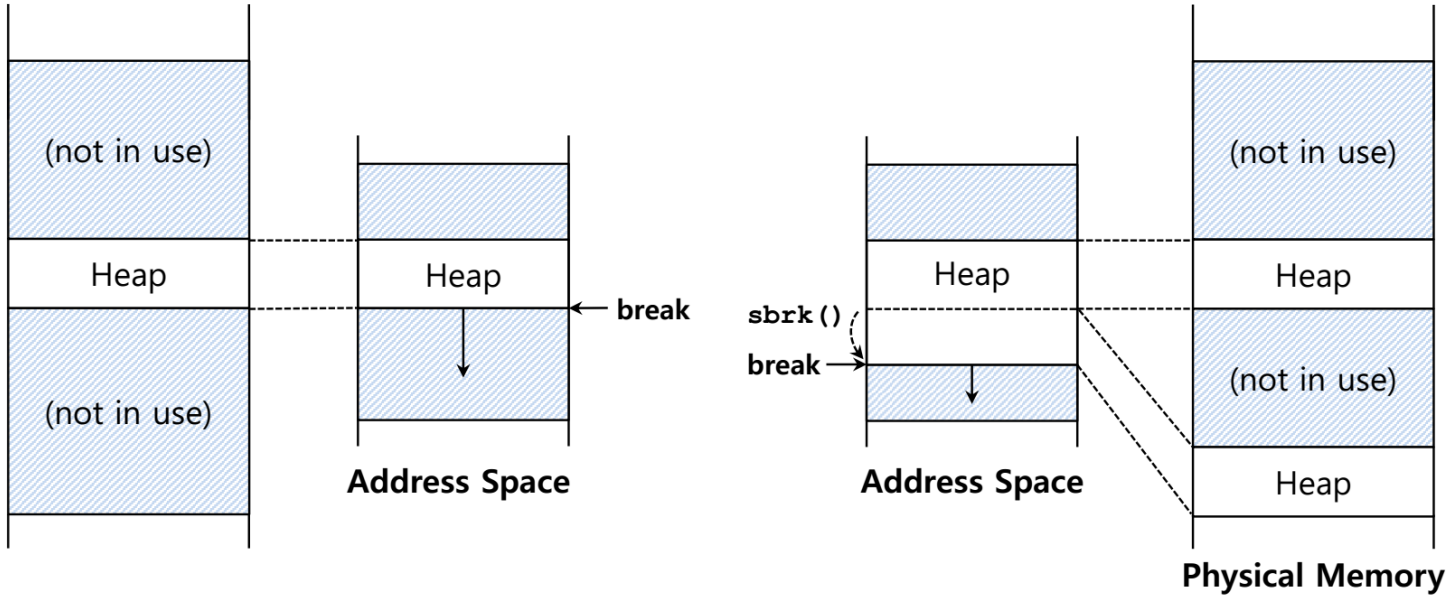
Suppose all the three chunks are freed

- The list now has a bunch of free chunks that are adjacent
- A smart algorithm would merge them all into a bigger free chunk
- Must split and coalesce free chunks to satisfy variable sized requests



Growing the heap

- Most allocators **start** with **a small-sized heap** and then **request more** memory from the OS when they run out.
  - e.g., `sbrk()`, `brk()` in most UNIX systems.



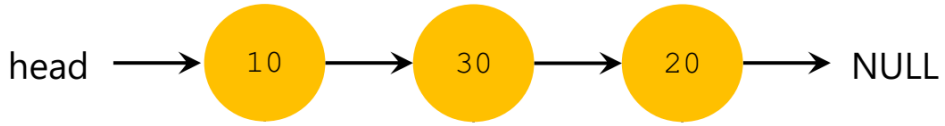
Managing Free space – Basic strategies

- Best Fit:
  - Finding free chunks that are **big or bigger than the request**
  - Returning the **one of smallest** in the chunks **in the group** of candidates
- Worst Fit:
  - Finding the **largest free chunks** and allocation the amount of the request
  - **Keeping the remaining chunk** on the free list.

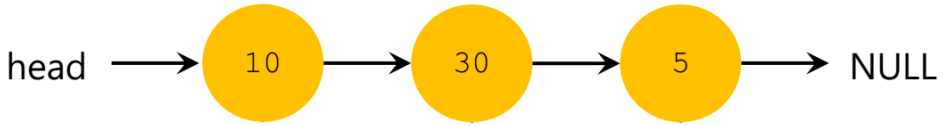
First Fit : Simply finds the first block that is big enough and returns the requested amount to the user

Next Fit : Instead of always beginning the first-fit search at the beginning of the list, the next fit algorithm keeps an extra pointer to the location within the list where one was looking last.

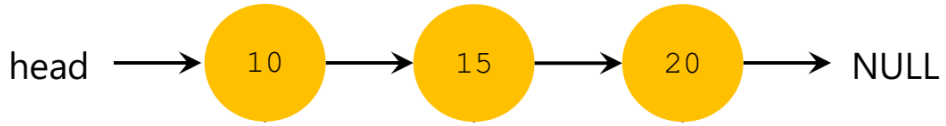
- Allocation Request Size 15



- Result of Best-fit

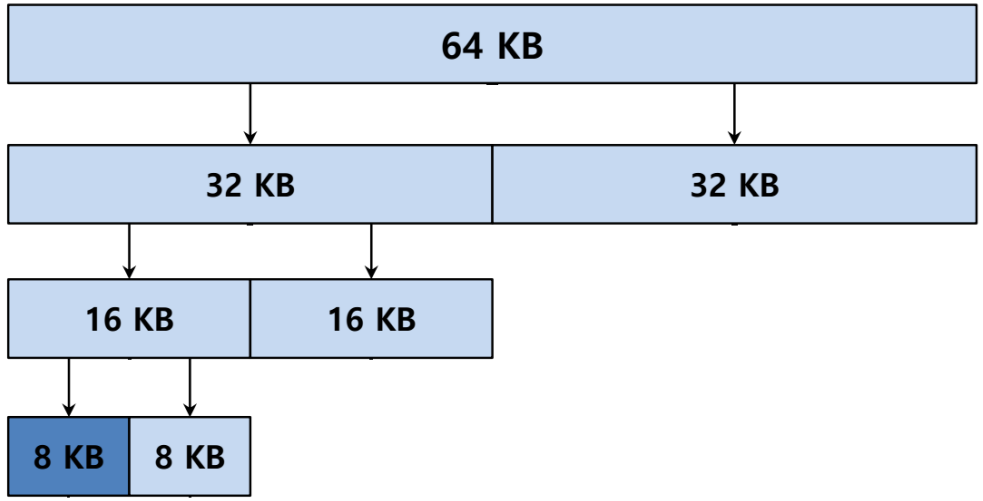


- Result of Worst-fit



Binary Buddy allocator

- Binary Buddy Allocation
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**.



64KB free space for 7KB request

Allocate memory in size of power of 2 –E.g., for a request of 7000 bytes, allocate 8KB chunk

•Why? 2 adjacent power-of-2 chunks can be merged to form a bigger power-of-2 chunk –E.g., if 8KB block and its "buddy" are free, they can form a 16KB chunk

Fixed size allocations

Memory allocation algorithms are much simpler with fixed size allocations

•Page-sized fixed allocations in kernel:–Has free list of pages–Pointer to next page stored in the free page itself

•For some smaller allocations (e.g., PCB), kernel uses a slab allocator

–Object caches for each type (size) of objects

–Within each cache, only fixed size allocation

–Each cache is made up of one or more “slabs”

•Fixed size memory allocators can be used in user programs also (instead of generic malloc)