EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

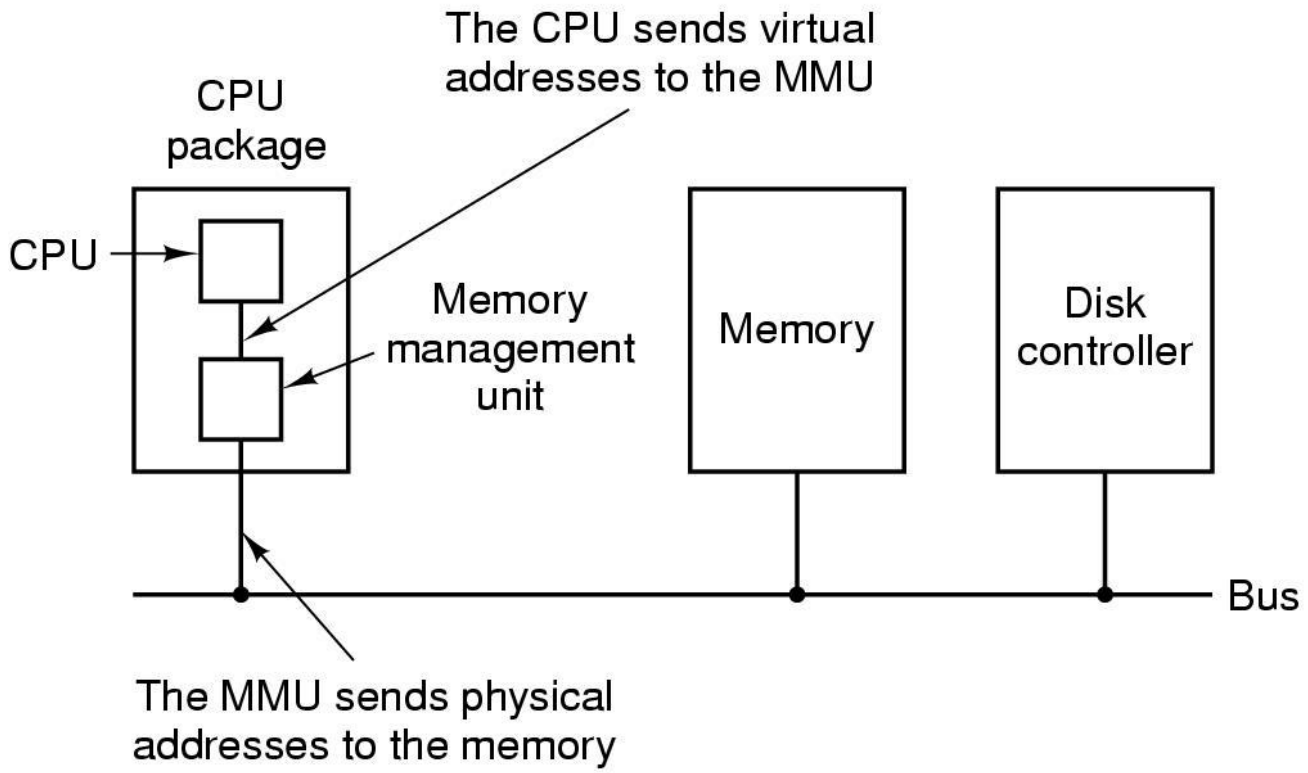
Review of paging

Paging

- In most situations, *paged virtual addressing* technique is used rather than just segmentation
- Maps contiguous virtual addresses to a discontiguous set of physical pages
- Virtual and physical memory is broken up into fixed-sized units (commonly, $0x1000_{16}$ == $4{,}096_{10}$ bytes)
  - Virtual units called *pages*
  - Physical units called *page frames*

Memory Management Unit (MMU)

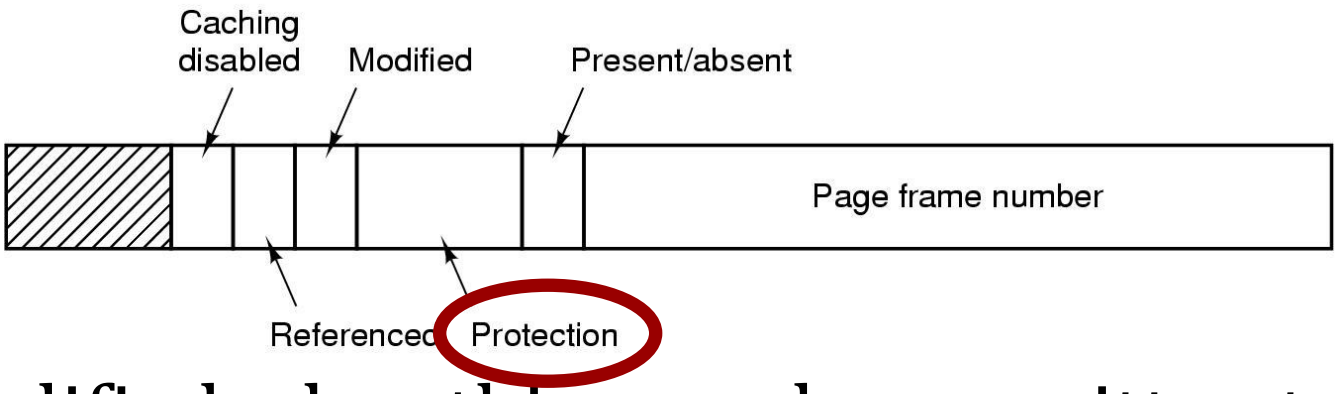
Automatically performs the mapping from virtual addresses to physical addresses



Memory Management Unit

- CPU hardware that performs the virtual to physical translations.
- It has to maintain lots Virtual to Physical translation entries, one for each virtual page.
- Its specialized hardware that performs extremely fast translations

Information per translation entry



- Modified – has this page been written to?
  - If so, we will need to write to disk before evicting
- Referenced – has anyone used this page?
- Caching disabled – used if physical page is used for device I/O
- Page Frame Number - physical address
- Missing from this picture is the Virtual page number, its implicit in the location

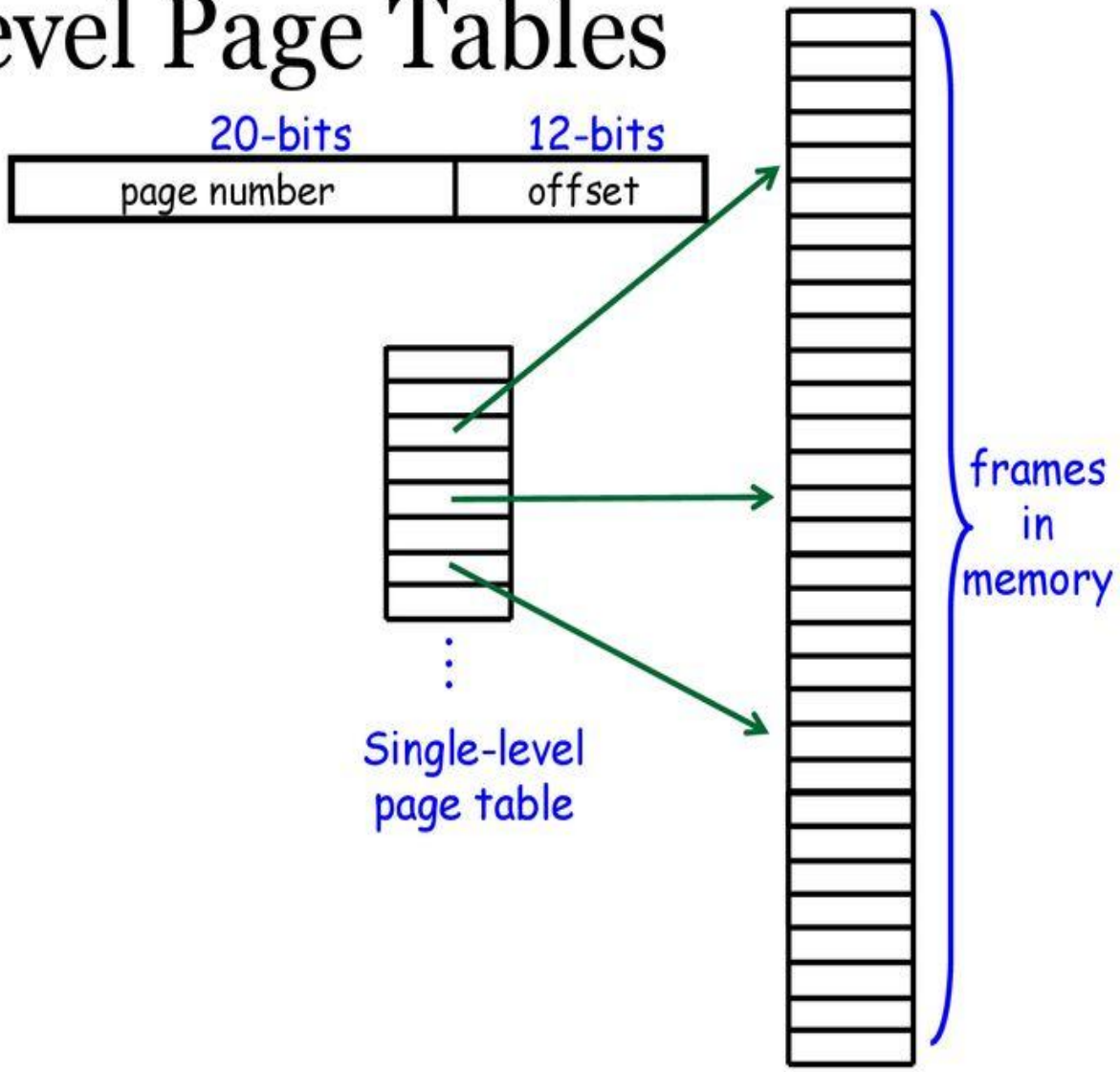
Unfortunately, everything doesn't fit in the MMU

- The MMU needs an entry for every valid virtual page.
- Way too big for the MMU hardware.
- So, we store everything in memory in a table known as the "page table".
- Whenever a virtual to physical translation isn't in the MMU cache, it reads translation into the MMU from memory
- The MMU is a small high-speed cache of translations

Single Level Page Tables

- Contains one entry for each virtual page.
- The upper portion of the virtual address(page number) is the index into the page table
- Virtual address space is designed to be far larger than ever required (32 bits ;-))
- HW requires the huge page table to be in physically contiguous memory(not feasible)
  - a 32-bit/4GB virtual address space requires 1million 4-byte entries!
  - a 4TB virtual address space requires 1 billion entries!! Most systems support much larger than this
  - Reliably providing large regions of physically contiguous memory is difficult for the kernel!

Single-Level Page Tables

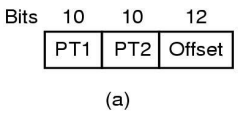


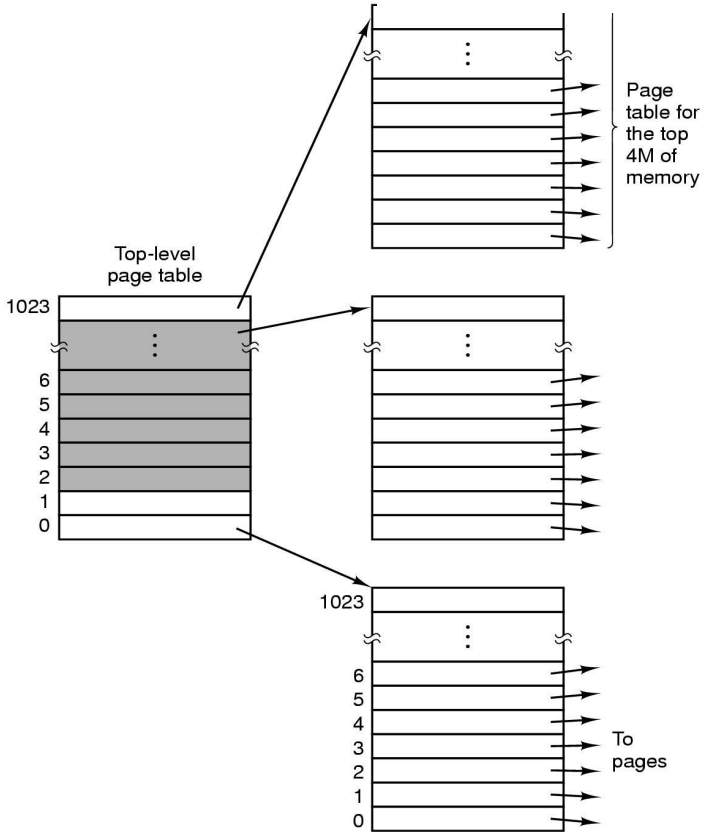
Multilevel Page Tables

- Single level is very inefficient if the virtual address space is expected to be sparse (i.e., not many mapped pages)
- Instead, multi-level page tables are used
  - The virtual address now has multiple indices into the multiple levels of the page table
  - This allows us to only allocate tables for portions of the space that are used
  - Eliminates the need to store page tables in physically contiguous memory.
- Page tables themselves are stored in pages of memory called page table pages
- Page table pages are not pre-allocated, they are allocated on demand the first time needed.

Multilevel Page Tables

- 32 bit virtual address
- PT1: Top-level index, 10 bits
- PT2: Second-level index, 10 bits
- Offset: 12 bits
- Page size: 4KB
- Second-level maps 4MB (1024 entries of 4KB)
- Top-level maps 4GB (1024 entries of 4MB)



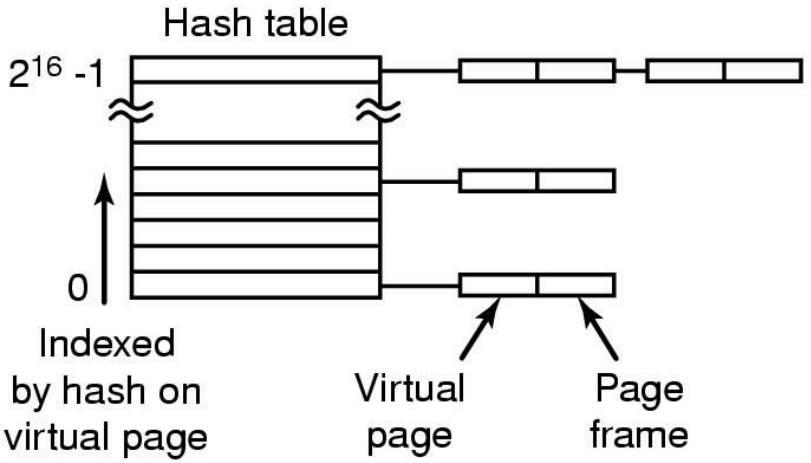


Inverted Page Table

- When virtual pages are too many, maintaining a page table is not feasible
  - In a worst case scenario, with a very sparse virtual address space, the number of page table pages can equal or even exceed the number of physical pages mapped in the address space.
  - This is a huge memory consumption problem if many processes are sparse.
- Solution: Inverted Page Table
  - One entry per physical page frame
  - Each entry contains a pair <process, virtual page>
- Address cannot be resolved simply by looking for an index in a table
- When process n accesses page p, the table must be scanned for an entry <n, p>
- Solution
  - TLB should catch most of the accesses
  - Table hashed on virtual address to resolve the mapping

Inverted Page Table

- Introduced by PowerPC, SPARK, Itanium
- Advantages:
  a. less overhead sparse; only one field
  b. can be way more efficient (book wrong)
- On Power, actually table no links…; HW can search in parallel
- Challenge: supporting large pages



What happens if translation not in table?

Page Faults

- What happens if we try to access a page that is not mapped?
- The MMU notices, and we raise a CPU exception called a *page fault*
- Control is passed to the OS (via interrupt) to decide what to do
  - Kill the process if the virtual address is not allowed(segmentation fault)
  - Find some physical page to map to it

Programs Bigger than Memory

- Note that this gives us a way to have programs that don't all fit into memory at once
- We can just map in the parts of the program we're using right now
- If we reference a page that isn't mapped we incur a page fault which locates the appropriate page and maps it.
- If all physical memory is in use we can free another page by ``*paging*'' it to disk.
  - way cheaper than swapping whole process

Types of Page Faults

- **Minor page fault** – can be serviced by just creating the right mapping
- **Major page fault** – must load in a page from disk to service
- **Segmentation fault** – invalid address accessed; can't service so we usually just kill the program (dynamic stack expansion uses this but just extend the stack section, we don't kill the process)

Translation Look-aside Buffer(TLB): Today's MMU

Translation Look-aside Buffers

- Translation Look-aside Buffer (TLB)
  - hardware device that allows fast access without using the page table
  - very fast cache that holds a subset of the processes page table entries.
- Small number of entries (e.g., 64-512) accessible as an associative memory(CPU hardware searches all TLB entries in parallel).
- Checked by hardware before doing a page table walk
- If lookup succeeds (hit), the page is accessed directly
- If TLB lookup fails (miss), the page table is used and the corresponding entry in TLB is added
- When an entry is taken out of the TLB, the modified bit is updated in the corresponding entry of the page table
- TLB management can be done both in hardware (MMU) or in software (by the OS)

Translation Look-aside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB on x86

- Can look this up on Linux with the `x86info` command:

```
x86info –c
```

TLB on x86

- Can look this up on Linux with the `x86info` command:

```
Cache info
TLB info
 Instruction TLB: 4K pages, 4-way associative, 64 entries.
 Data TLB: 4KB or 4MB pages, fully associative, 32 entries.
 Data TLB: 4KB pages, 4-way associative, 64 entries
 Data TLB: 4K pages, 4-way associative, 512 entries.
```

- Note that the x86 keeps multiple separate TLBs for code vs. data as well as different page sizes

Software TLB Management

- In some architectures (e.g., SPARC, MIPS), the TLB is managed by software
- TLB entries are explicitly loaded by the OS
- If we look up an address and it's not in the TLB (a *TLB miss*) we raise a TLB fault
- The OS then has to fill in the missing TLB entry

Software TLB Management

- Why manage the TLB explicitly instead of letting hardware do it?
  - The MMU can be much simpler, which saves space on the CPU that can be used for other things
  - Flexibility – the OS can choose its own algorithms for which TLB entries to evict and add
- But: this is generally slower than hardware-managed TLBs

It's a tradeoff!

TLBs and Context Switching

- TLBs map a virtual page to a physical page frame
- But, once we change to a new process, these mappings are no longer valid, and a *TLB flush* occurs
- This makes context switching more expensive – the first few memory accesses a process makes will have to be serviced by walking the page tables (i.e., 3x slower memory)

Global TLBs

- Similar to the Global Segment some TLB entries are marked as “Global” so they don’t get flushed during context switches
- Global TLB entries are used to map the kernel’s virtual address space.
- We never want to flush the kernel’s TLB entries during a context switch, every process needs the kernel.

Tagged TLBs

- On most architectures today, each TLB entry can be associated with a *tag* that says what address space it belongs to.
- Think of this *tag* as being the process PID
- Now we don't have to flush the TLB when switching address spaces
- This can help make context switching faster – some TLB entries might still be valid when we switch back to a process

Physical Memory Consumption and Reclaiming

- The system exhausts all available RAM very quickly.
  - Typical Linux system is constantly under memory pressure and reclaiming pages of RAM.
- All memory is freed at boot-time.
- Kernel memory consumption:
  - Kernel data structures
  - Kernel stacks for each thread
  - Page tables
  - File system metadata(data structures describing files)
  - Module loading
- User memory consumption :
  - Filesystem data (i.e., file system cache/pagecache)
  - Anonymous and mapped file page faults
- When all physical memory is exhausted
  - User memory is reclaimed
  - Few kernel data structures can be reclaimed
- The constant/ongoing memory allocation and reclaiming activity is referred to as “Paging Dynamics”

Page Replacement Algorithms

***Page fault (or fs read/write)* forces choice**

- When free memory is exhausted which page should be removed to make room for an incoming page?

Criteria:

- Modified page must first be saved
- Unmodified just overwritten
- Better not to choose an often used page that will probably need to be brought back in soon

Thrashing:

- Constantly swapping in and out memory pages
- Large negative impact on system performance
- i.e., more time is spent on moving data from memory to disk (and back) than making progress in the program

Optimal Page Replacement Algorithms

- Determine how far in the execution of the program a page will be needed
- Replace page needed at the farthest point in the future
  - Optimal but unrealizable
- Useful to compare with other algorithms
  - Log page use on first execution of the program
  - Develop a scheduling for following executions (with same inputs)

"*Prediction is very difficult, especially about the future*"
Niels Bohr

R and M Bits

- Each page has a Reference (R) bit and a Modified (M) bit

- R and M bits can be provided by the hardware or can be managed in software
  - Initial process with no pages in memory
  - When page is loaded
    - R bit is set
    - READONLY mode is set
  - When modification is attempted a fault is generated
    - M bit is set
    - READ/WRITE mode is set

Not Recently Used (NRU) Page Replacement Algorithm

- When process starts, R and M bit are set to 0
- Periodically R bit is cleared to take into account for pages that have not been referenced recently
- Pages are classified according to R and M
  - Class 0: not referenced, not modified
  - Class 1: not referenced, modified (R bit has been cleared!)
  - Class 2: referenced, not modified
  - Class 3: referenced, modified
- NRU removes page at random from lowest numbered non empty class
- Easy to implement but not terribly effective

First-in First-out (FIFO) Page Replacement Algorithm

- Maintain a linked list of all pages
- List is ordered by loading time
- The page at the beginning is replaced
  - Oldest one

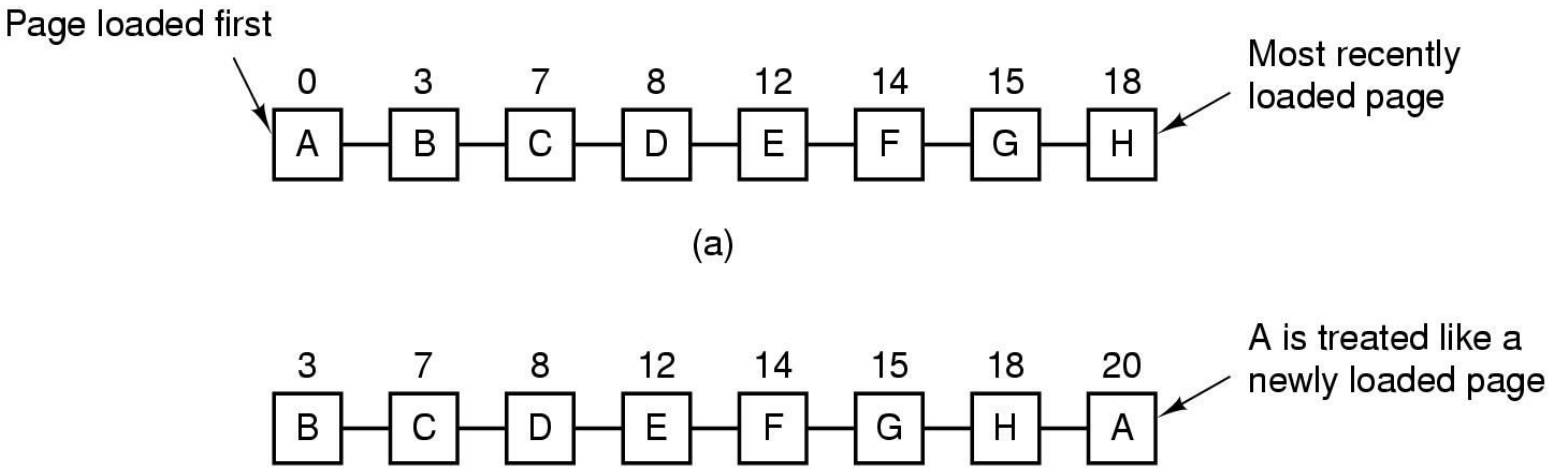
- Advantage
  - Easy to manage
- Disadvantage
  - Page in memory the longest may be often used

Second Chance Page Replacement Algorithm

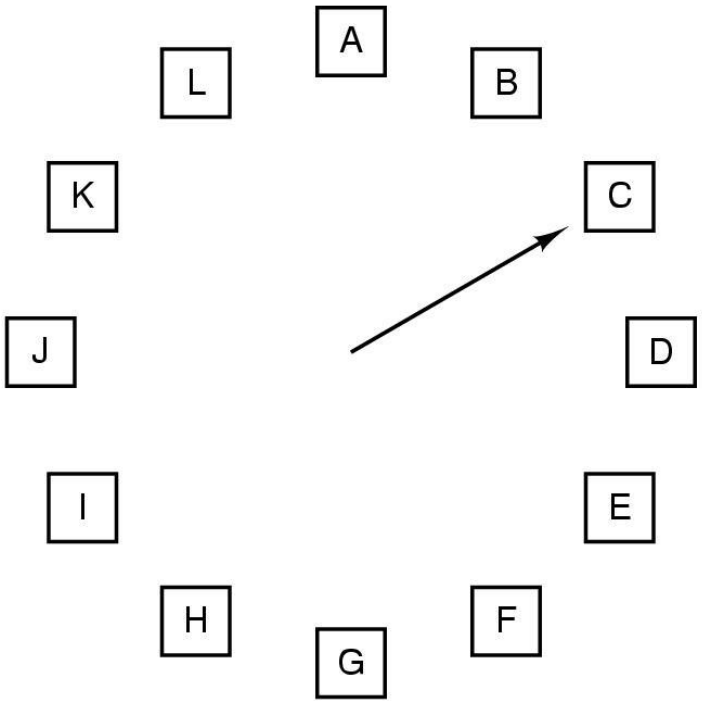
- Pages in list are sorted in FIFO order
- R bits are cleared regularly
- If the R bit of the oldest page is set it is put at the end of the list
- If all the pages in the list have been referenced the page that was “recycled” will reappear with the R bit cleared and will be thrown away

Second Chance Page Replacement Algorithm

Page list if fault occurs at time 20, A has R bit set



Clock
Page Replacement Algorithm



- Same concept as Second Chance, different implementation
- Hand points to oldest page
- When a page fault occurs
  - If R=0: evict the page
  - If R=1: clear R and advance hand until page with R=0 is encountered

Least Recently Used (LRU) Page Replacement Algorithm

- Assumption: pages used recently will be used again soon
  - Throw out page that has been unused for longest time

Problem?

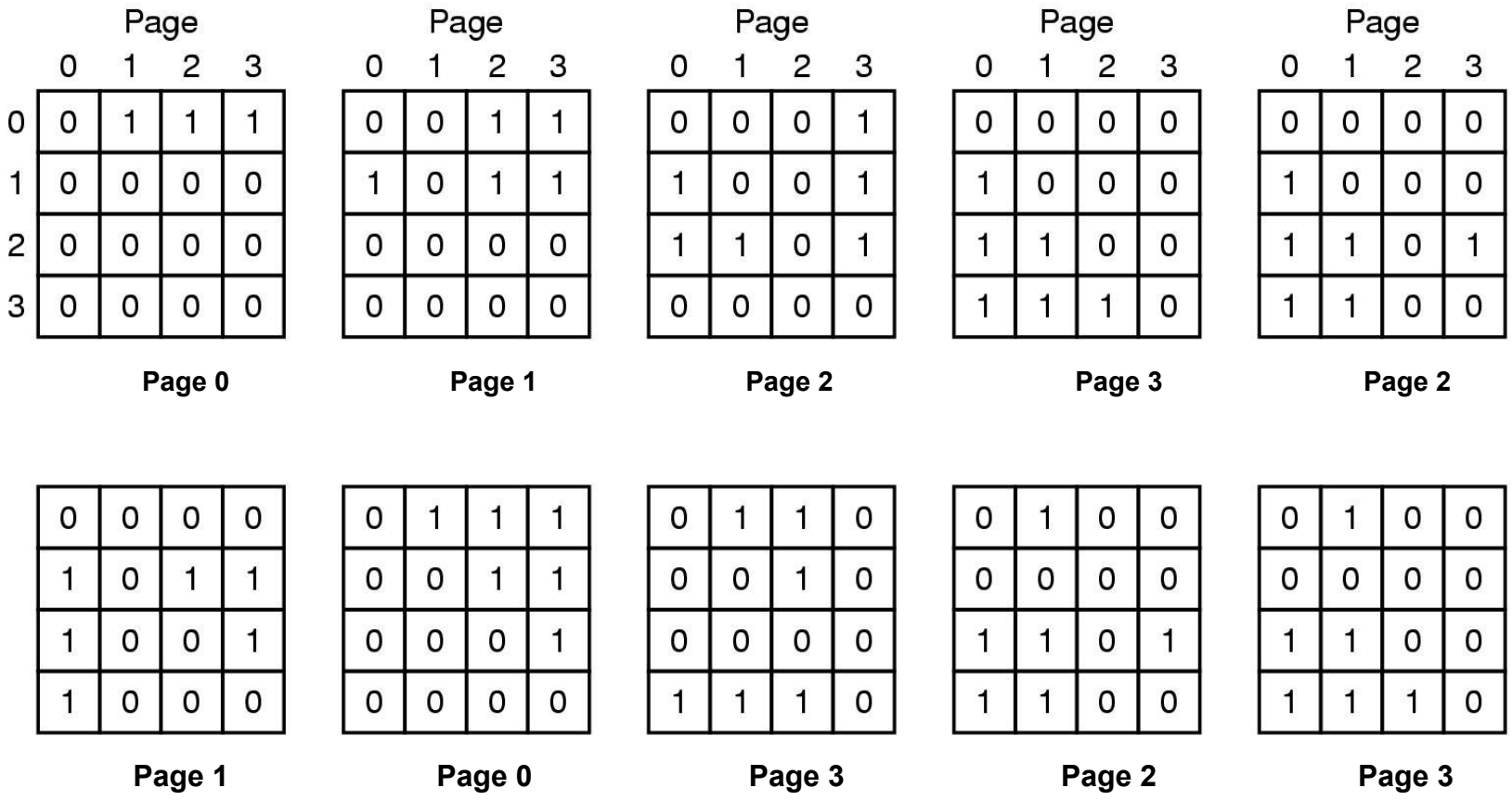
- Very expensive: Must keep a linked list of pages
  - Most recently used at front, least at rear
  - Update this list every memory reference !
- Alternative
  - Maintain global counter that is incremented at each instruction execution
  - Maintain similar counter in each page table entry
  - If page is referenced copy global counter in page counter
  - Choose page with lowest value counter

Least Recently Used (LRU) Page Replacement Algorithm

Another alternative:

- Maintain matrix *M* with *n* x *n* bits, where *n* is the number of page frames
- If page *j* is accessed
  - set to 1 all the bits in the corresponding row ($M_{j,i} = 1, i = 1 \ldots n$)
  - set to 0 all the bits in the corresponding column($M_{i,j} = 0, i = 1 \ldots n$)
- At any moment the page with the row containing the lowest value is the least recently used

Least Recently Used (LRU) Page Replacement Algorithm

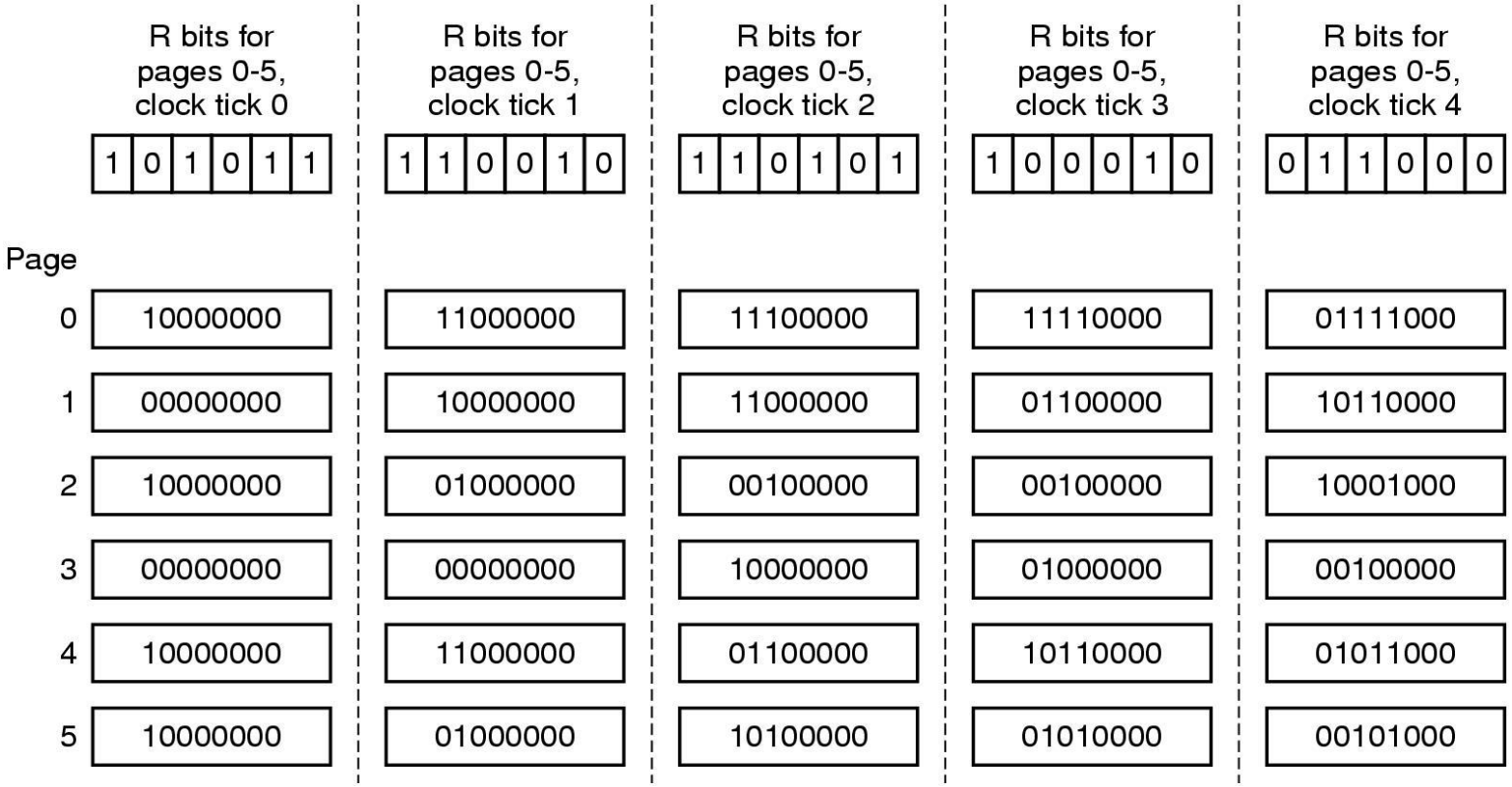


Not Frequently Used (NFU) Page Replacement Algorithm

- A counter is associated with each page
- At each clock interval, the counter is incremented if the page has been referenced (R=1)
- The page with the lowest counter is removed
- Problem:

  pages that have been heavily used in the past will always maintain high counter values
- Need for an aging mechanism
  - First shift the counter
  - Then set bit in most significant position if referenced

Not Frequently Used (NFU) Page Replacement Algorithm



Linux memory management

Linux Virtual Memory Management

- **x86_64 is really a 48-bit virtual/46-bit physical address system**
- **Space for future expansion up to 64-bits of both virtual and physical**

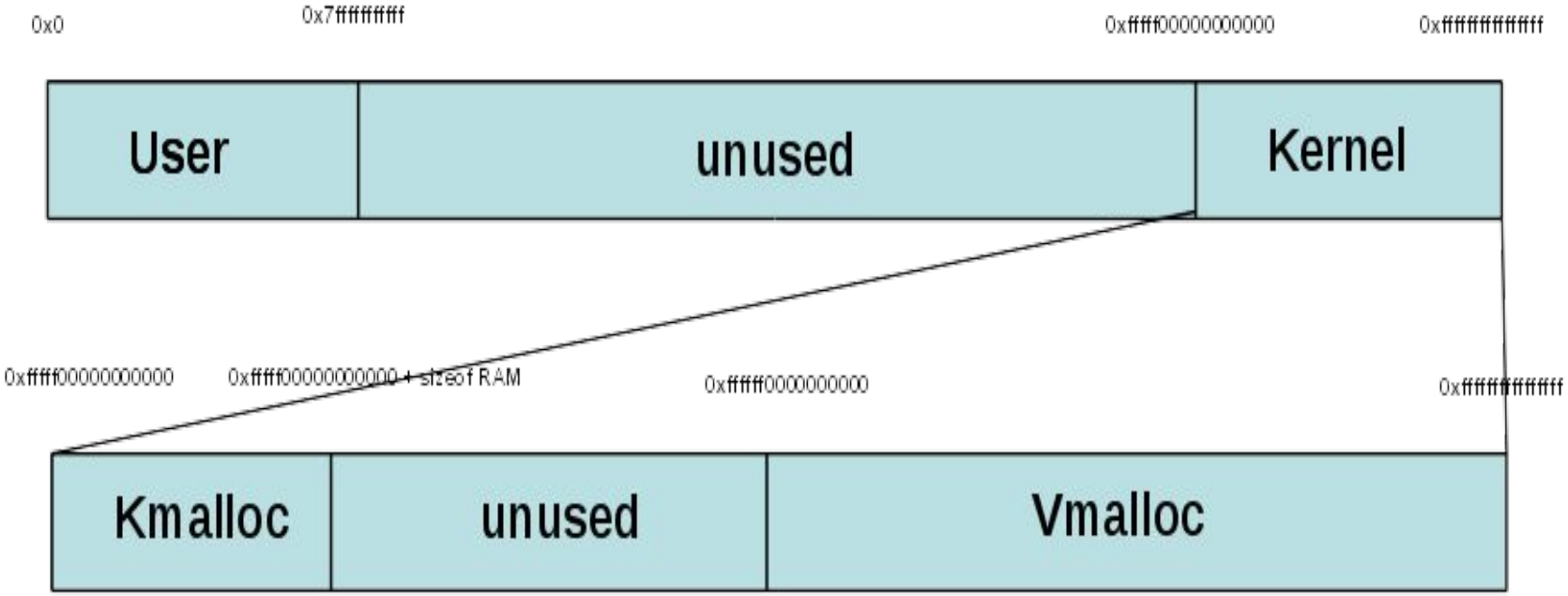
User: Virtual size = 2^47/128TB
0x0 - 0x7fffffffffff(2^47)

Kernel: Virtual size = 2^47/128TB
0xffff800000000000 - 0xffffffffffffffff
Kmalloc: lower ½ of kernel(2^46 or 64TB)
physical memory directly mapped here.
Used for kernel allocations: 0 - 4MB

Vmalloc: upper ½ of kernel(2^46 or 64TB)
Modules are loaded here.
Used for kernel allocations: 4MB - 32TB

Unused: Giant gap between User-End & Kernel-Start

Linux Virtual memory layout

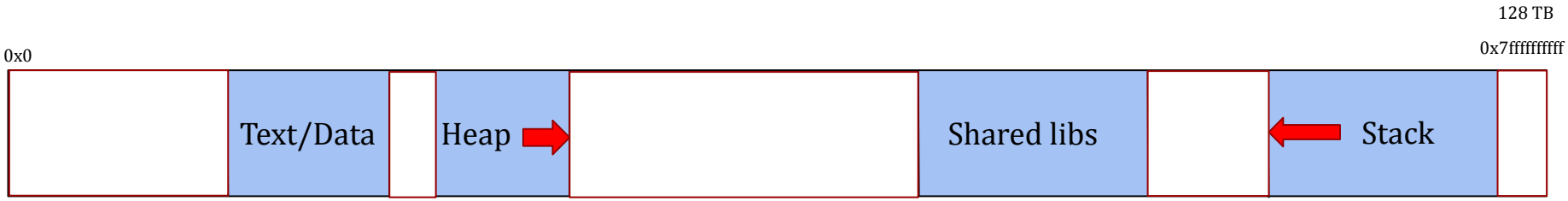


2 Types of User Virtual Memory

- Mapped File Virtual Memory
  - A file that is mapped into the virtual address space
  - Created via mmap(fd=FD) system call.
  - Destroyed via munmap() system call

- Anonymous Virtual Memory
  - Any virtual memory not backed by a file(stack, heap, uninitialized data, etc.)
  - Created via mmap(fd=NULL), sbrk(), brk(), malloc()
  - Destroyed, via munmap(), sbrk(), brk(), free(), exit()

Linux user virtual address space




```
[lwoodman@lwoodman linux]$ cat /proc/self/maps
55a31c484000-55a31c486000 r--p 00000000 fd:00 3801429                    /usr/bin/cat
55a31c486000-55a31c48a000 r-xp 00002000 fd:00 3801429                    /usr/bin/cat
55a31c48a000-55a31c48c000 r--p 00006000 fd:00 3801429                    /usr/bin/cat
55a31c48c000-55a31c48d000 r--p 00007000 fd:00 3801429                    /usr/bin/cat
55a31c48d000-55a31c48e000 rw-p 00008000 fd:00 3801429                    /usr/bin/cat
55a31cdb2000-55a31cdd3000 rw-p 00000000 00:00 0                          [heap]
7f2b31d8c000-7f2b31dae000 rw-p 00000000 00:00 0
7f2b31dae000-7f2b3f2de000 r--p 00000000 fd:00 3806989                    /usr/lib/locale/locale-archive
7f2b3f2de000-7f2b3f2e0000 rw-p 00000000 00:00 0
7f2b3f2e0000-7f2b3f306000 r--p 00000000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f306000-7f2b3f455000 r-xp 00026000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f455000-7f2b3f4a0000 r--p 00175000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a0000-7f2b3f4a1000 ---p 001c0000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a1000-7f2b3f4a4000 r--p 001c0000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a4000-7f2b3f4a7000 rw-p 001c3000 fd:00 3813237                    /usr/lib64/libc-2.32.so
7f2b3f4a7000-7f2b3f4ad000 rw-p 00000000 00:00 0
7f2b3f4c4000-7f2b3f4c5000 r--p 00000000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4c5000-7f2b3f4e6000 r-xp 00001000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4e6000-7f2b3f4ef000 r--p 00022000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4ef000-7f2b3f4f0000 r--p 0002a000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7f2b3f4f0000-7f2b3f4f2000 rw-p 0002b000 fd:00 3815120                    /usr/lib64/ld-2.32.so
7ffcb575a000-7ffcb577b000 rw-p 00000000 00:00 0                          [stack]
7ffcb577d000-7ffcb5781000 r--p 00000000 00:00 0                          [vvar]
7ffcb5781000-7ffcb5783000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
[lwoodman@lwoodman linux]$
```

Linux Physical Memory Management

struct page: one for each page of RAM
mem_map: array of struct pages
buddy page allocator: pages coalesced from 4KB to 4MB

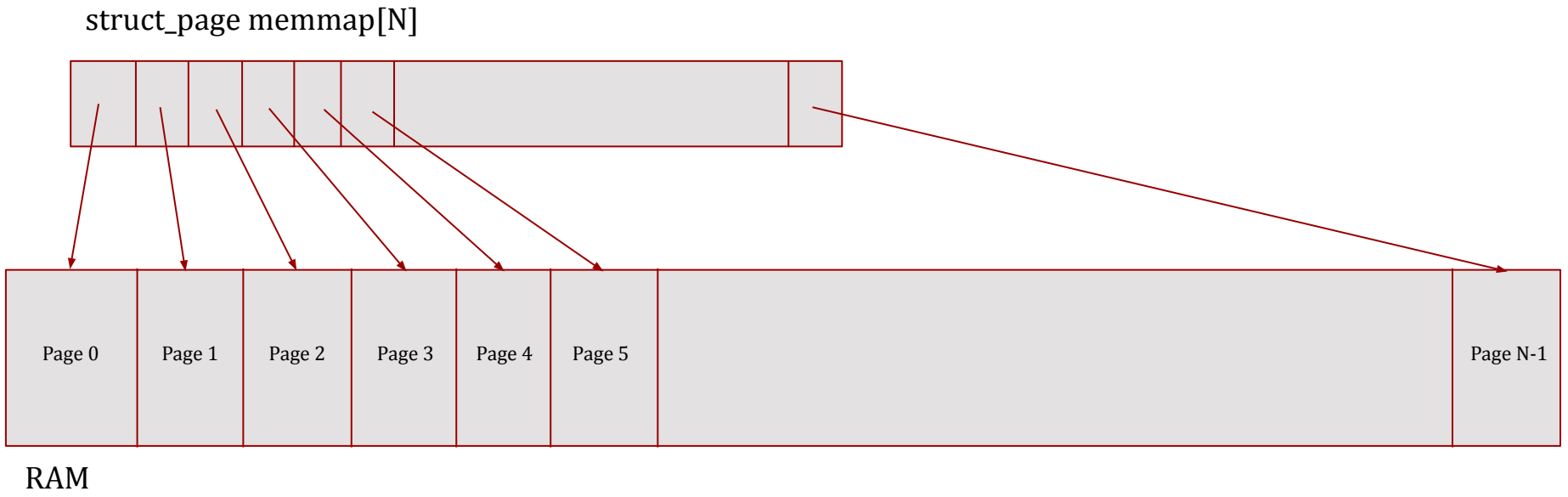
- 11 buckets of physically contiguous RAM
- every allocation can split up a large bucket/page
- every free tries to coalesce into largest bucket

DMA: 0*4kB 2*8kB 1*16kB 2*32kB 2*64kB 0*128kB 2*256kB 1*512kB 1*1024kB 1*2048kB 1*4096kB = 8416kB
DMA32: 3997*4kB 3042*8kB 2544*16kB 1995*32kB 1033*64kB 232*128kB 137*256kB 102*512kB 68*1024kB 1*2048kB 0*4096kB = 399652kB
Normal: 3997*4kB 3042*8kB 2544*16kB 1995*32kB 1033*64kB 232*128kB 137*256kB 102*512kB 68*1024kB 1*2048kB 0*4096kB = 399652kB

- All RAM is freed/put free list at boot-time
- RAM is continuously allocated/removed from free list for kernel and user processes
- RAM is eventually exhausted and therefore reclaimed from users and few select kernel locations and freed.

Linux memmap

- Linux maintains a struct_page to track each page of RAM
- The mem_map is an array of struct_pages
- The Nth page of RAM is tracked by memmap[N]
- Each struct_page is 64 bytes and describes one 4096 byte page of RAM
  - The mem_map consumes 1/64 of all the RAM(64/4096)



The ‘struct page’ fields

struct page (8 longwords)

flags	**_count**	**_mapcount**	**private**
mapping	**index**	**lru.next**	**lru.prev**

The ‘struct page’ definition is in the <linux/mm.h> header

along with the declaration for the ‘mem_map[ ]’ array