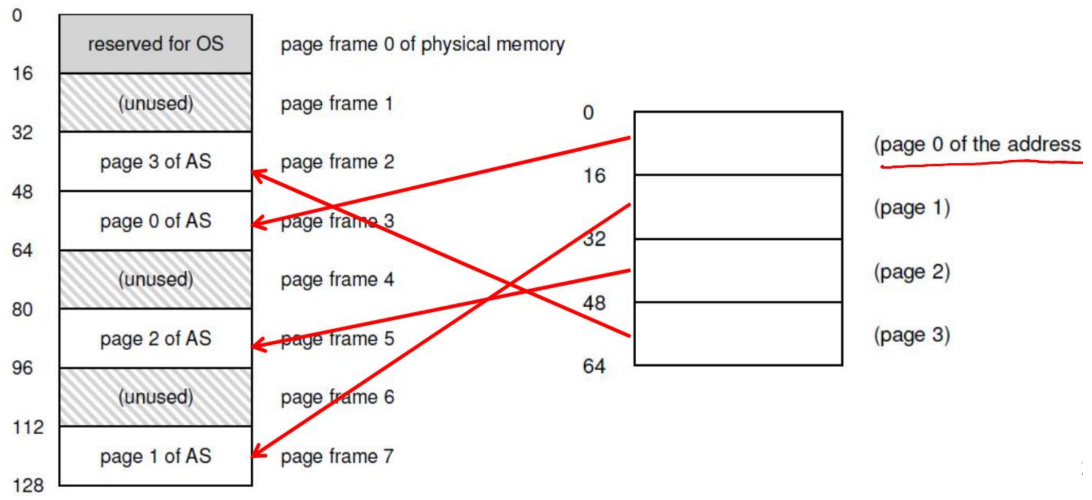
LECTURE 23
FEB 28, 2024

Paging

- Allocate memory in fixed size chunks ("pages")
- Avoids external fragmentation (no small "holes")
- Has internal fragmentation (partially filled pages)

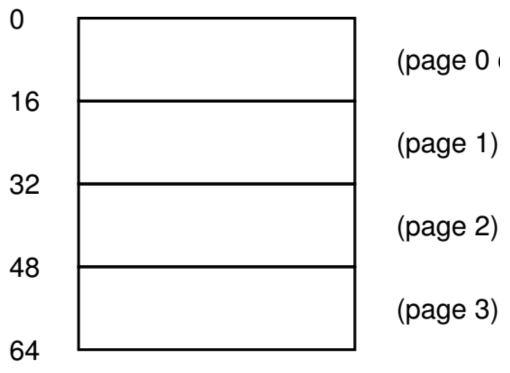


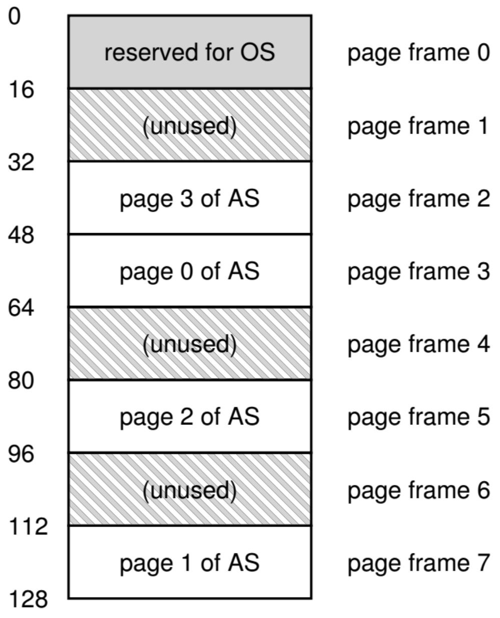
Paging Data structures

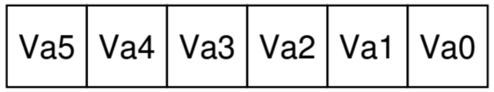
- **Pages are fixed size, e.g., 4K**
  - Virtual address has two parts: virtual page number and offset
  - Least significant 12 (log2 4K) bits of address are page offset
  - Most significant bits are *page number*
- **Page tables**
  - Map virtual page number (VPN) to physical page number (PPN)
    - VPN is the index into the table that determines PPN
    - PPN also called page frame number
  - Also includes bits for protection, validity, etc.
  - One page table entry (PTE) per page in virtual address space

Example

```
movl <virtual address>, %eax
```



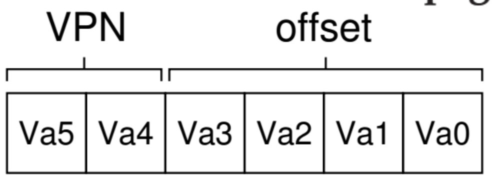




Virtual address space of the process is 64 bytes, we need **6 bits** total for our virtual address (2^6 = 64).

The page size is 16 bytes in a 64-byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that.

virtual page number (VPN)

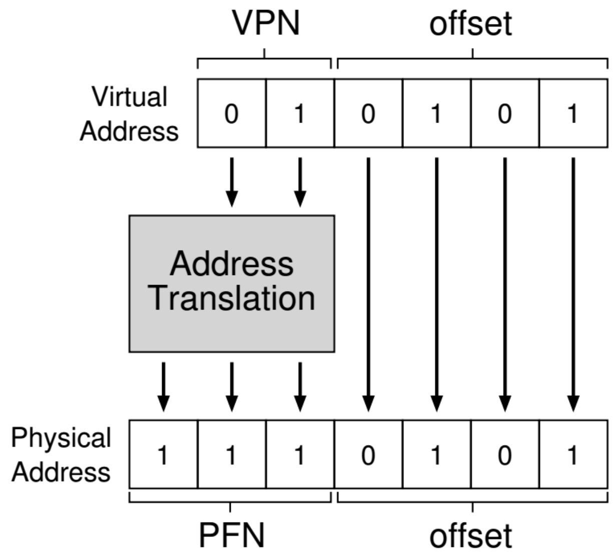


offset just tells us which byte *within* the page we want

Address Translation in Hardware

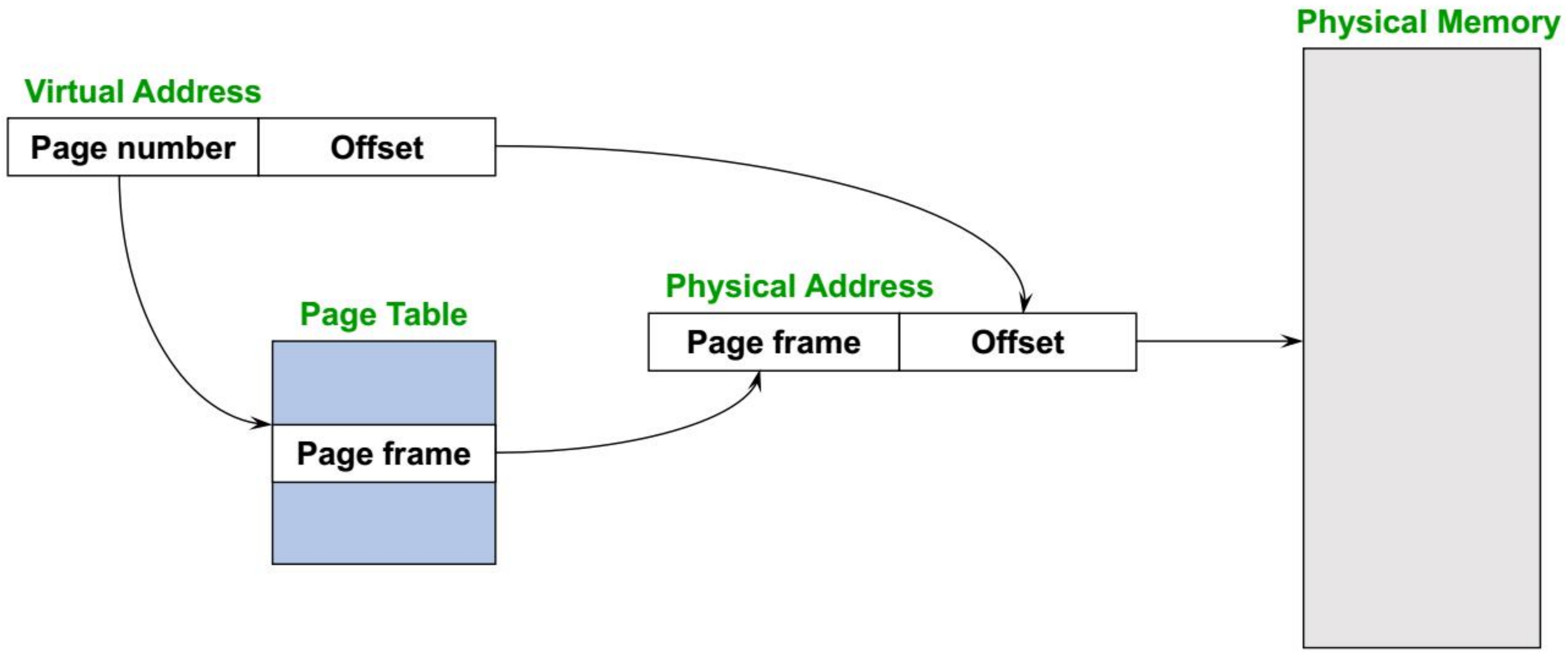
- Most significant bits of VA give the VPN
- Page table maps VPN to PFN
- PA is obtained from PFN and offset within a page
- MMU stores (physical)address of start of page table, not all entries.
- “Walks” the page table to get relevant PTE

```
movl 21, %eax
```



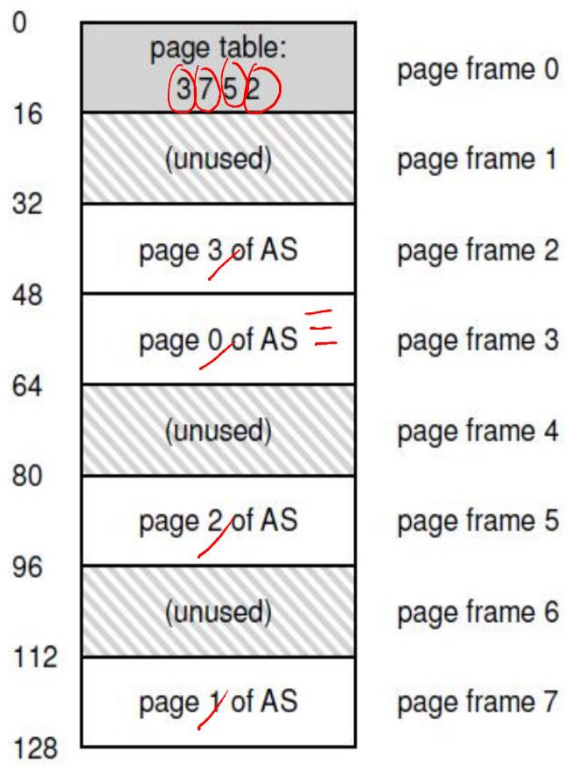
1110101 (117 in decimal)

Page Lookups



Page Table

- Per process data structure to help VA-PA translation
- Array stores mappings from virtual page number (VPN) to physical frame number (PFN) – E.g.,VP0 ➡PF 3, VP 1➡PF 7
- Part of OS memory (in PCB)
- MMU has access to page table and uses it for address Translation
- OS updates page table upon context switch
- Per process data structure to help VA-PA translation



Page Table Entries

Simplest page table: linear page table

- Page table is an array of page table entries, one per virtual page
- VPN (virtual page no.) is index into this array
- Each PTE contains PFN (physical frame number) and few other bits

– Valid bit: is this page used by process?

– Protection bits: read/write permissions

– Present bit: is this page in memory? (more later)

– Dirty bit: has this page been modified?

– Accessed bit: has this page been recently accessed?

What happens on a memory access ?

```
movl 21, %eax
```

First translate the virtual address (21) into the correct physical address (117).

Before fetching the data from address 117, the system must first fetch the proper page table entry from the process's page table, perform the translation, and then load the data from physical memory. [Hardware must know where the page table is for the currently-running process]

Single **page-table base register** contains the **physical address** of the starting location of the page table.

PTEAddr = Page Table Base Register + (VPN*sizeof(PTE))

Virtual Page Number- used as an index into the array of PTEs pointed to by the page table base register

Now, physical address is known, the hardware can fetch the PTE from memory, extract the PFN, and concatenate it with the offset from the virtual address to form the desired physical address.

```
offset   = VirtualAddress & OFFSET_MASK
PhysAddr = (PFN << SHIFT) | offset
```

The hardware can fetch the desired data from memory and put it into register **`eax`**.

The program has now succeeded at loading a value from memory!

What happens on a memory access ?

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

What happens on a memory access ?

- CPU requests code or data at a virtual address
- MMU must translate VA to PA
  – First, access memory to read page table entry
  – Translate VA to PA
  – Then, access memory to fetch code/data
- Paging adds overhead to memory access
- Solution? A cache for VA-PA mappings

Translation Lookaside Buffer (TLB)

A cache of recent VA-PA mappings

•To translate VA to PA, MMU first looks up TLB

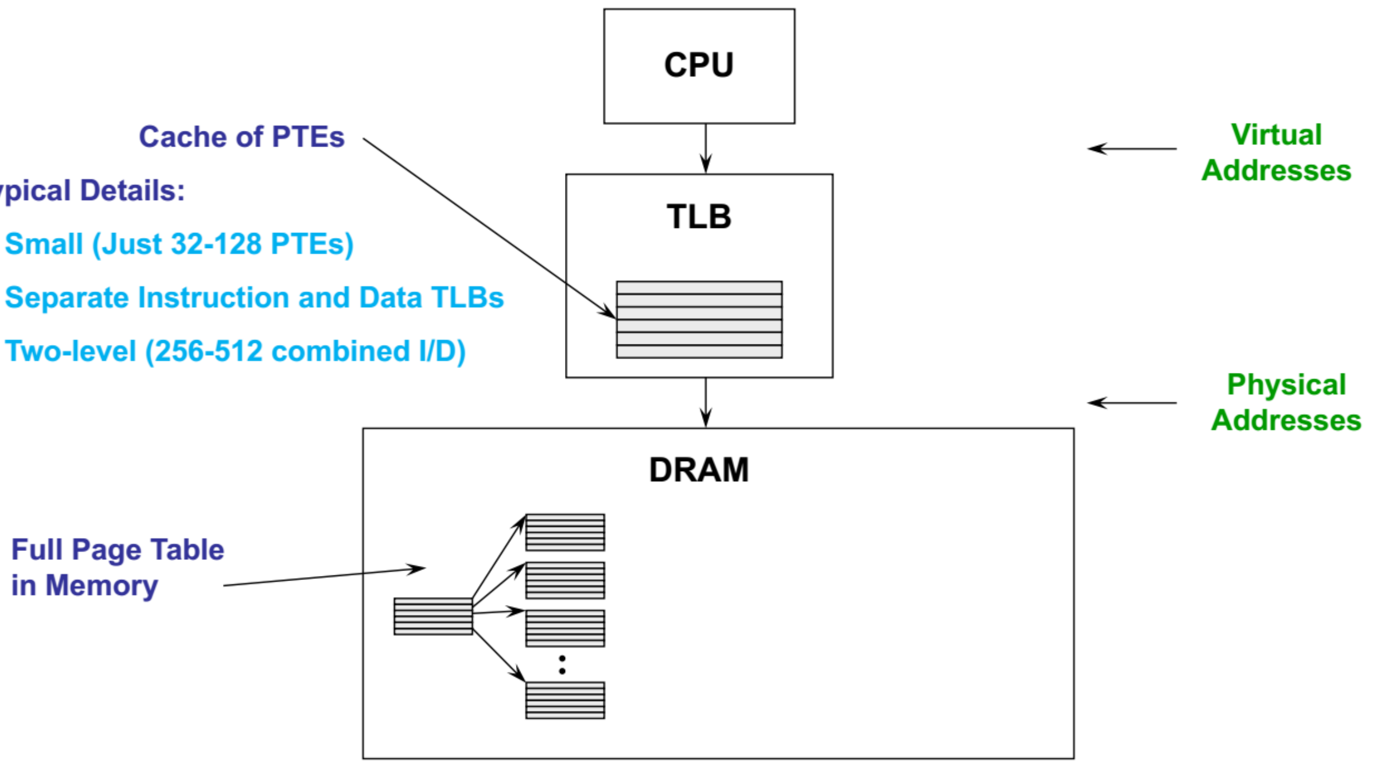
•If TLB hit, PA can be directly used

•If TLB miss, then MMU performs additional memory accesses to “walk” page table

•TLB misses are expensive (multiple memory accesses)

–Locality of reference helps to have high hit rate

•TLB entries may become invalid on context switch and change of page tables



Typical Details:

- Small (Just 32-128 PTEs)
- Separate Instruction and Data TLBs
- Two-level (256-512 combined I/D)

Paging Advantages

- **Easy to allocate memory**
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem
- **Easy to swap out chunks of a program**
  - All chunks are the same size
  - Use valid bit to detect references to swapped pages
  - Pages are a convenient multiple of the disk block size

Limitations

- **Can still have internal fragmentation**
  - Process may not use memory in multiples of a page
- **Memory reference overhead**
  - 2 or more references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (more later)
- **Memory required to hold page table can be significant**
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - Solution – page the page tables (more later)

How are page tables stored in memory

- What is typical size of page table?

–32 bit VA, 4 KB pages, so 2^32 / 2^12 = 2^20 entries

–If each PTE is 4 bytes, then page table is 4MB

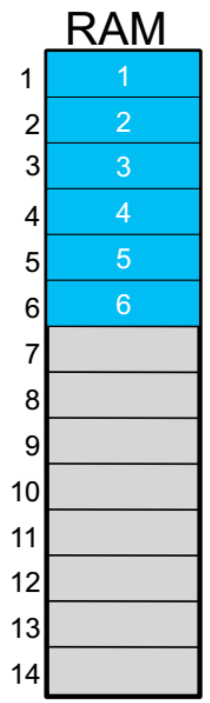
–One such page table per process!

- How to reduce the size of page tables?

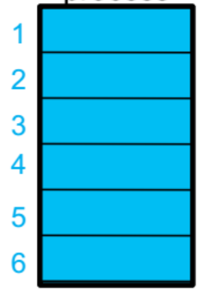
–Larger pages, so fewer entries

- How does OS allocate memory for such large tables?

–Page table is itself split into smaller chunks!



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.

Managing Page tables

- **Size of the page table for a 32-bit address space w/ 4K pages is 4MB**
  - This is far far too much overhead for each process
- **How can we reduce this overhead?**
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
- **How do we only map what is being used?**
  - Can dynamically extend page table…
  - Does not work if addr space is sparse (internal fragmentation)
- **Use another level of indirection: two-level page tables**

Bigger Pages ?

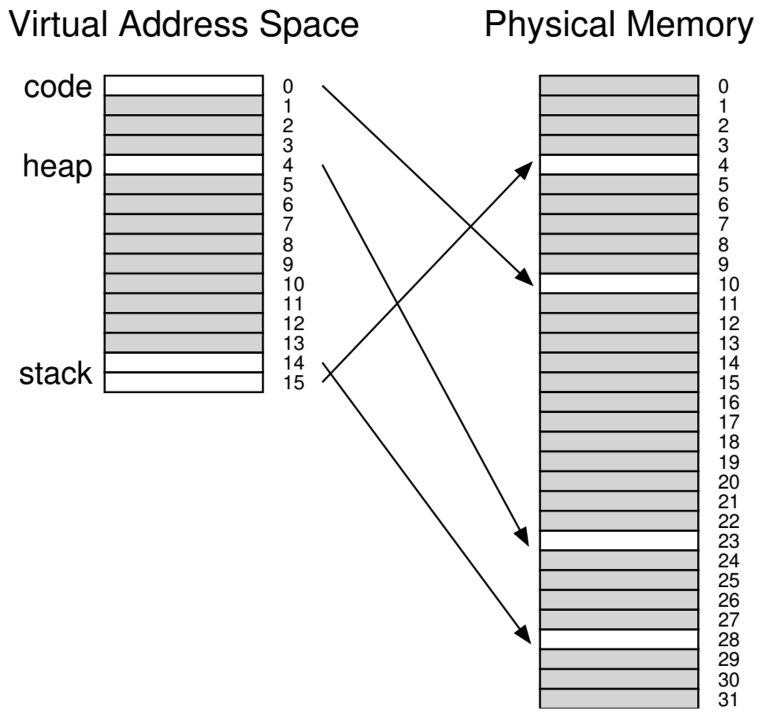
32-bit address space again, but this time assume 16KB pages instead of 4KB

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)

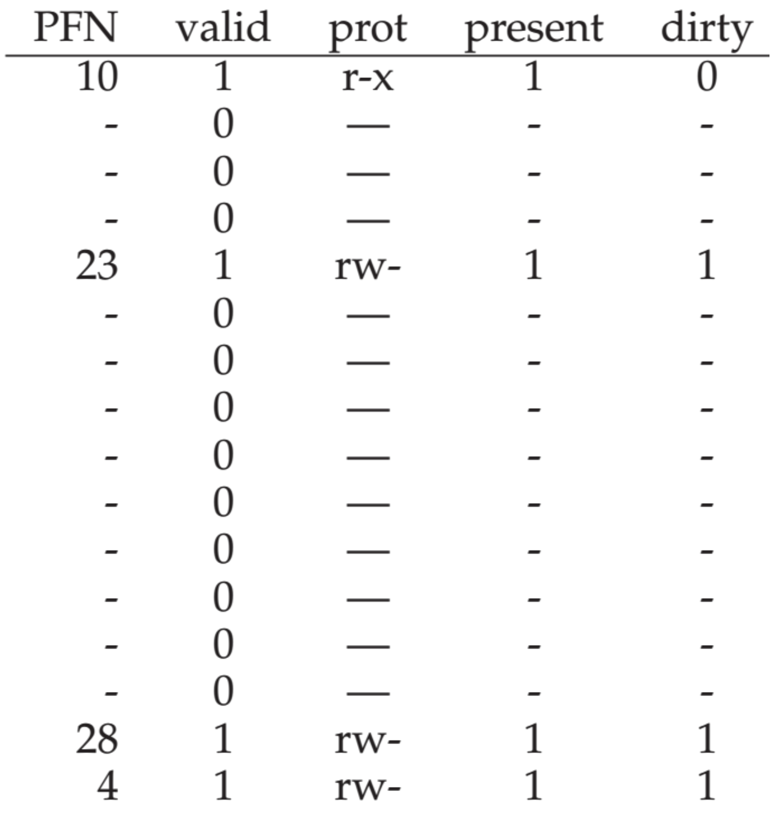
Applications endup allocating pages but only using little bits and pieces of each, and memory quickly fills up with these overly-large pages.

Invalid regions ?



Assume we have an address space in which the used portions of the heap and stack are small.
For the example, we use a tiny 16KB address space with 1KB 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	—	-	-
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!]

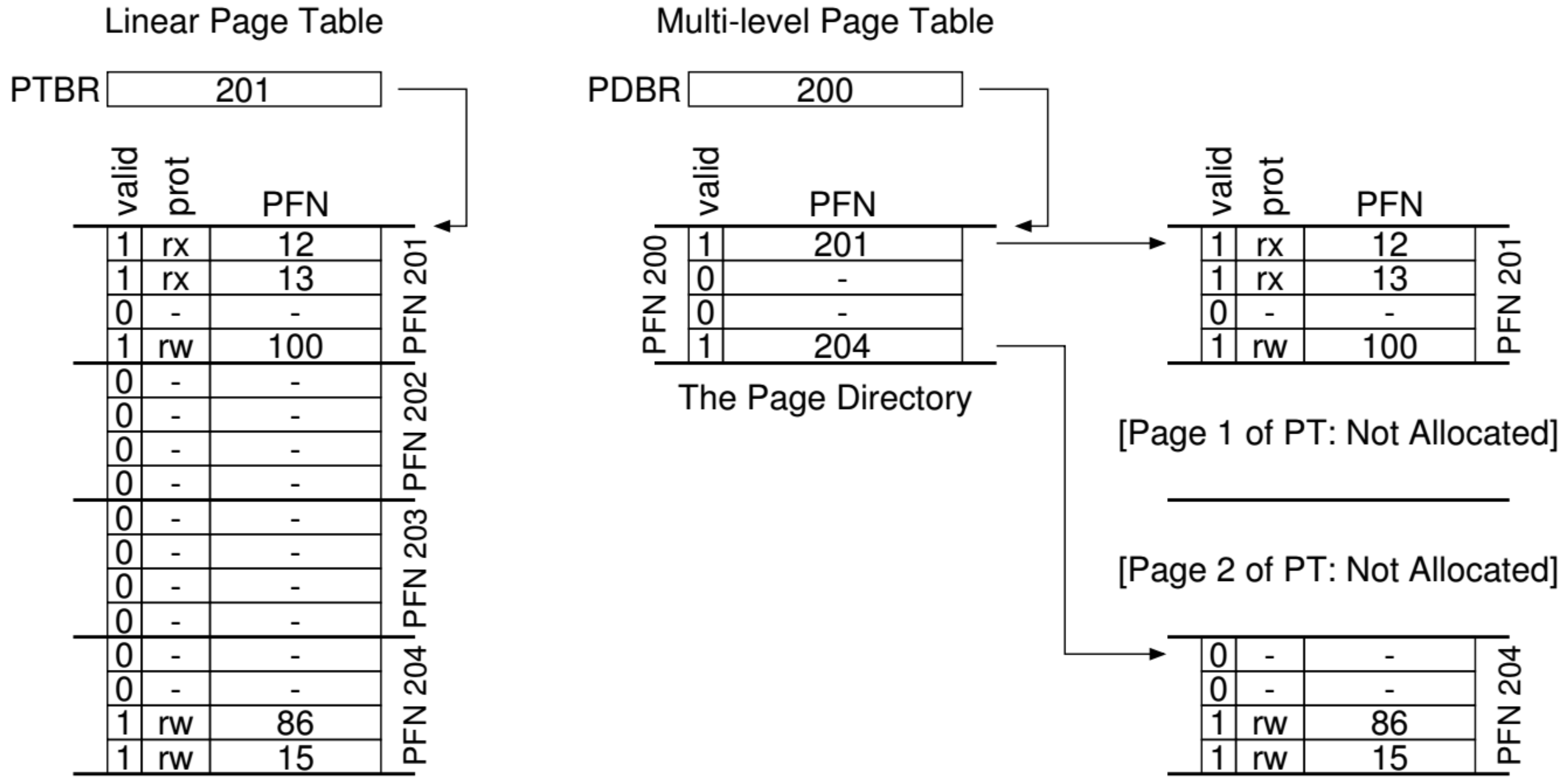
To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the **page directory**.

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



- 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 p**roportion 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

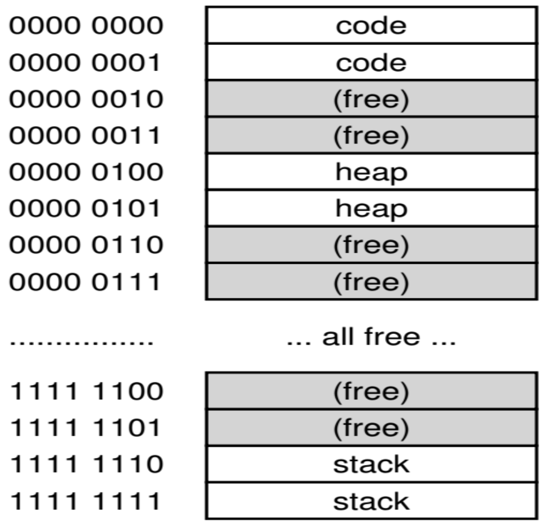
Example

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.



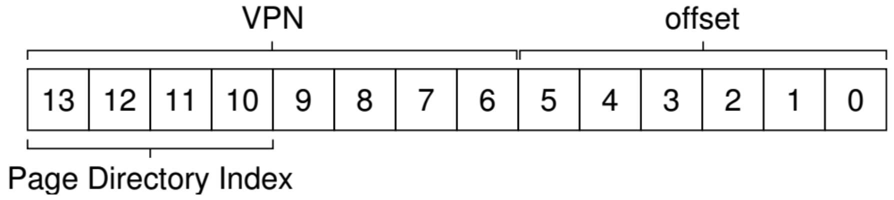
Given that we have **64**-byte pages, the 1KB page table can be divided into **16** 64-byte pages; **each page can hold 16 PTEs.**

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.

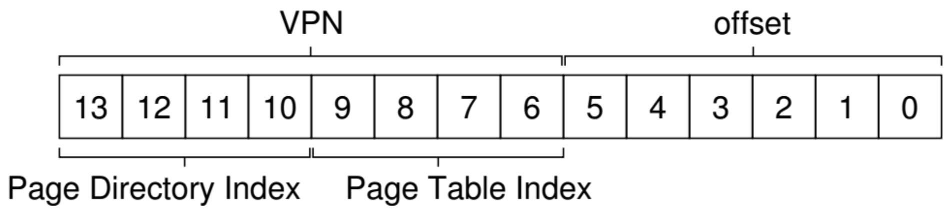
Four bits of the VPN to index into the directory; we use the top four bits of the VPN



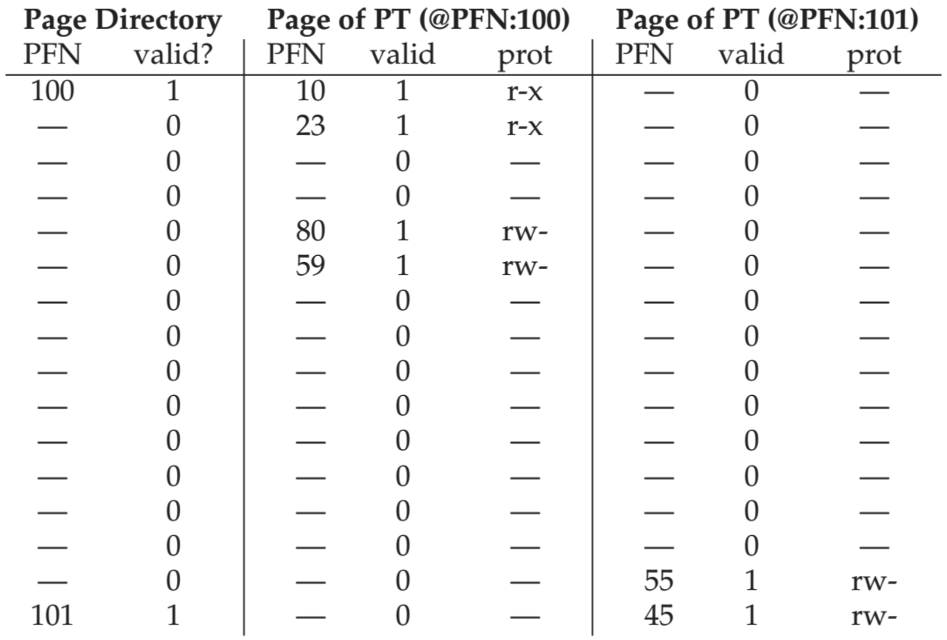
Page director index from VPN -- use it to find the address of the page-directory entry (PDE) : **`PDEAddr = PageDirBase + (PDIndex * sizeof(PDE))`**

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.



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 the other flag bits. 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

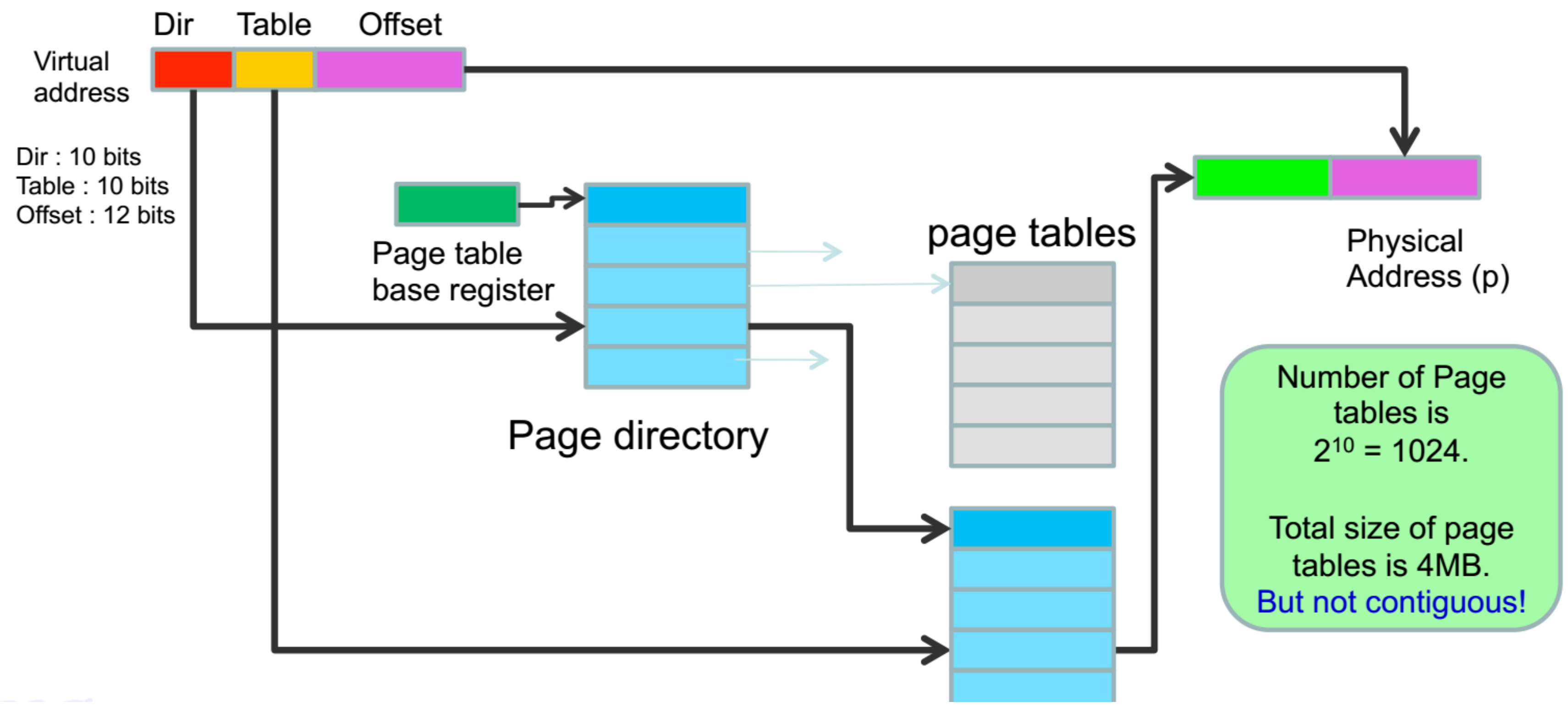
Each PTE has frame number (21 bits) and flags(10 bits)≈4 bytes.

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


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!

Inverted Page Tables

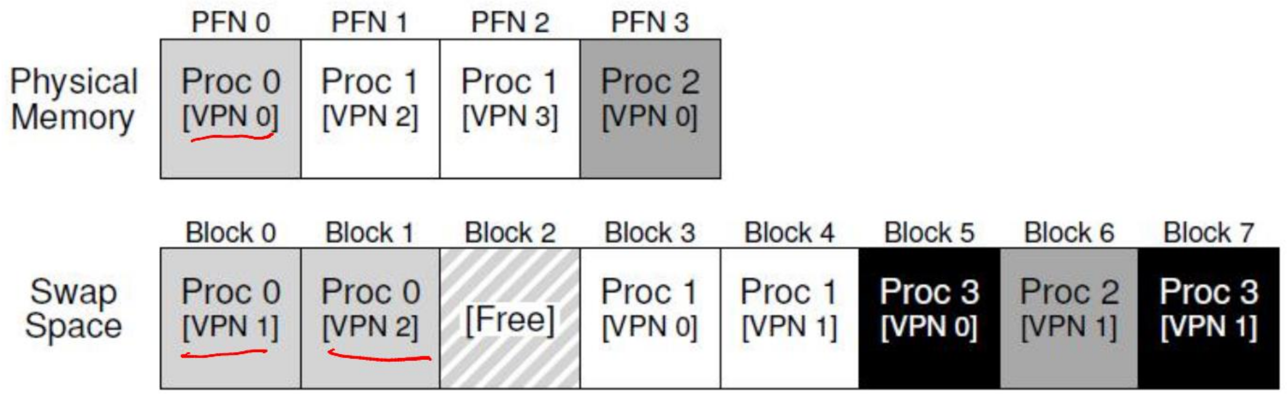
- Instead of having many page tables (one per process of the system), we keep a single page table that has an entry for each physical page of the system.
- The entry tells us which process is using this page, and which virtual page of that process maps to this physical page.
- Finding the correct entry 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

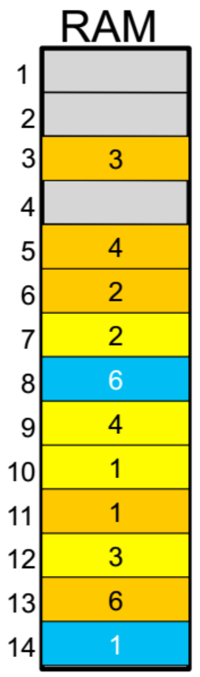


•Reserve some space on the disk for moving pages back and forth. Referred as swap space, because we swap pages out of memory to it and swap pages into memory from it.

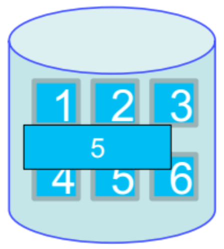
•OS can read from and write to the swap space, in page-sized units [OS will need to remember the disk address of a given page]

• 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



Swap space
(on disk)



process page table in RAM

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

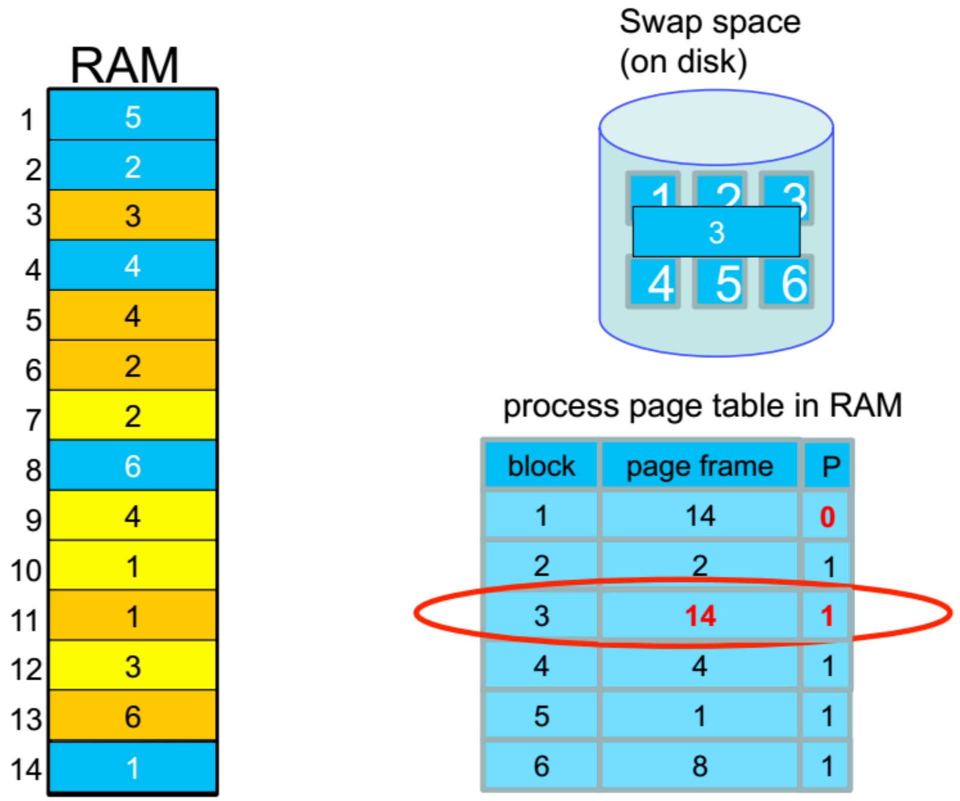
present bit

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



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

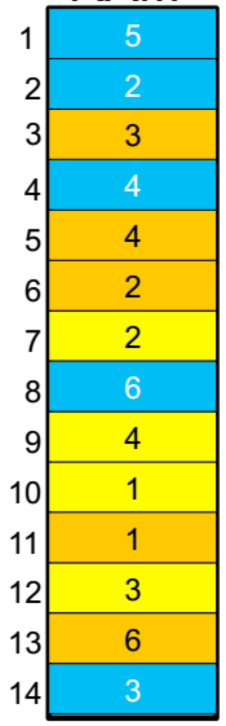
If there are no pages free
for a new block to be loaded,
the OS makes a decision to
remove another block from RAM.

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

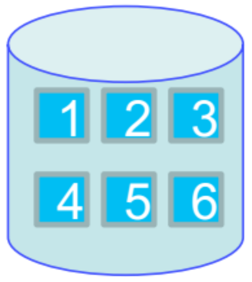
Some replacement policies are
* First in first out
* Least recently used
* Least frequently used

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

RAM



Swap space
(on disk)



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.