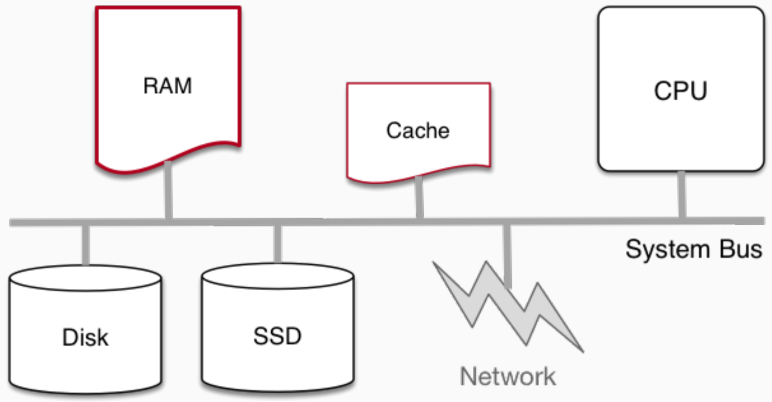
Memory

Systems typically contain 4-16GB of volatile RAM



Plus a hierarchy of smaller cache memory -on or off the CPU chip.

Running Programs without an Operating System

- Many small embedded system run without operating system.
- Single program running, probably written in C.
- Devices (sensors, switches, . . .) often wired at particular address.
- E.g can set motor speed by storing byte at 0x100400.
- Program accesses (any) RAM directly.
- Development and debugging tricky.
- Widely used for simple micro-controllers.

Running Processes with only User/Kernel Space

- Operating system need only simple hardware support.
- Program accesses RAM directly but a part of address space (kernel space) accessible only in a privileged mode.
- System call enables privileged mode and passes execution to operating system code in kernel space.
- Privileged mode disabled when system call returns.
- Privileged mode support - could be a bit in a special register
- Only one process resident in RAM at any time - switching between processes slow .
- Operating system must write out all memory of old process to disk and read all memory of new process from disk.
- OK for some uses, but inefficient in general.
- Little used in modern computing.

Running Processes with Protected Memory Segments

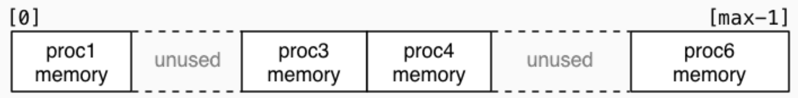
- Hardware support could limit process accesses to particular region (segment) of RAM.
- Programs would access RAM directly, but RAM belonging to other processes & kernel protected
- Allows multiple processes to be resident in RAM
- O/S can swap execution between them quickly.
- BUT - process doesn't know where in RAM it will be..
- Programs can't use absolute memory address (relocatable code) or addresses have to be modified before they are run.
- Major limitation - much more workable if processes can all have same view of memory.
- Little used in modern computing.

Memory management

- Big idea - disconnect address processes use from actual RAM address.
- Operating system translates every address a process uses to an actual RAM address.
- Convenient - each processes has same virtual view of RAM.
- But can have multiple processes in RAM simultaneously.
- Can load part of processes into RAM on demand.

Memory management with Memory Segments

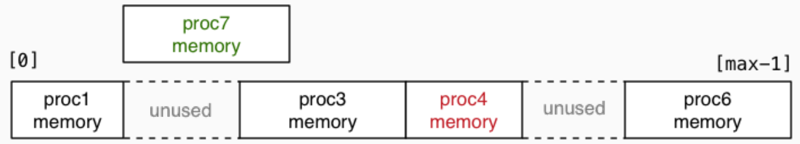
Consider a scenario with multiple processes loaded in memory:



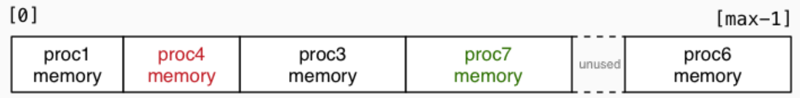
- Every process is in a contiguous section of RAM, starting at address *base* finishing at address *limit*.
- Each process sees its own address space as [0 .. psize-1]
- process can be loaded anywhere in memory without change
- When it accesses memory we add *base* to the address and check that is $<$ *limit*
- Easy to add hardware support.

Memory Management

Consider the same scenario, but now we want to add a new process

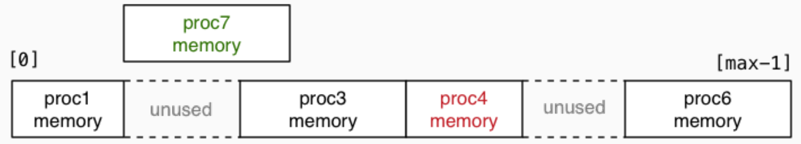


- The new process doesn't fit in any of the unused slots
- Could move some process to make a single large slot
- Can not make efficient use of RAM (fragmentation).
- Little used in modern computing.

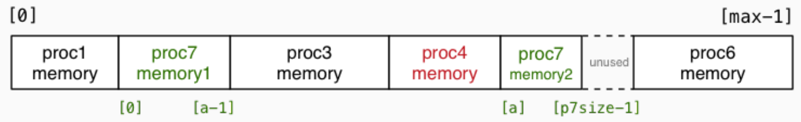


Split Process Memory over Multiple Regions

Idea: split process memory over multiple parts of physical memory.



becomes

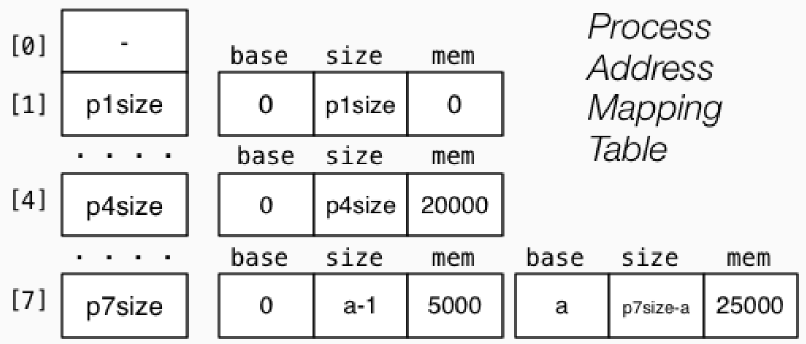


Split Process Memory over Multiple Regions

Implications for splitting process memory across physical memory

- each chunk of process address space has its own *base*
- each chunk of process address space has its own *size*
- each chunk of process address space has its own *memory* location

Need a table of process/address information to manage this, e.g.



Split Process Memory over Multiple Regions

Under this scheme, address mapping calculation is complicated make hardware support is difficult:

```
Address processToPhysical(pid, addr) {
  Chunk chunks[] = getChunkInfo(pid);
   for (int i = 0; i < nChunks(pid); i++) {
     Chunk *c = &chunks[i];
     if (addr >= c->base && addr < c->base+c->size)
       break;
   }
   uint offset = addr - c->base;
   return c->mem + offset;
}
```

Memory Management

Address mapping would be simpler if all chunks were same size

- call each chunk of address space a *page*
- all pages are the same size P (*PageSize*)
- page i has addresses A in range $i * P \leq A < (i+1) * P$

Also leads to a simpler address mapping table:

- each process has an array of page entries
- each page entry contains start address of one page
- can compute index of relevant page entry by (A/P)
- can compute offset within page by $(A\%P)$

Memory Management Review

Reminder: process addresses $\leftrightarrow$ physical addresses

- process has (virtual) address space 0..N-1 bytes
- memory has (physical) address space 0..M-1 bytes
- both address spaces partitioned in P byte pages
- process address space contains $K = \lceil N/P \rceil$ *pages*
- memory address space has $L = \lceil M/P \rceil$ *frames*

Mapping:

- takes address (Vaddr) in process address space
- returns address (Paddr) in memory address space

Address Mapping

Mapping from process address to physical address:

```
Address processToPhysical(pid, Vaddr) {
   PageInfo pages[] = getPageInfo(pid);
   uint pageno = Vaddr / PageSize; // int div
   uint offset = Vaddr % PageSize;
   return pages[pageno].mem + offset;
}
```

Computation of **pageno**,**offset** can be done much faster if *Pagesize* $== 2^n$

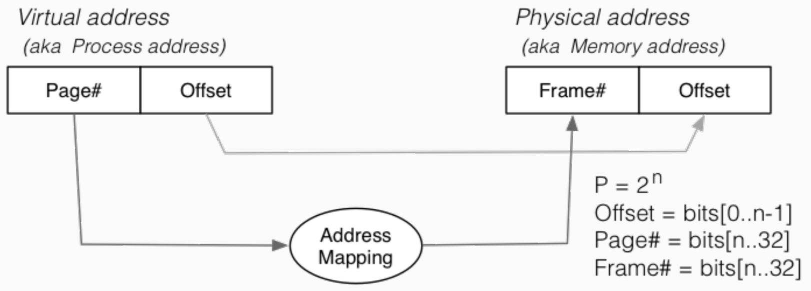
Note that we assume **PageInfo** entries with more information ....

Address Mapping

Page table entries typically do not store physical address

- to save space, just store frame number F
- compute physical address via (P * F + Offset)

If $P == 2^n$, then address mapping becomes



Virtual Memory

A side-effect of this type of virtual → physical address mapping

- don't need to load all of process's pages up-front
- start with a small memory "footprint" (e.g. `main` + stack top)
- load new process address pages into memory *as needed*
- grow up to the size of the (available) physical memory

The strategy of ...

- dividing process memory space into fixed-size pages
- on-demand loading of process pages into physical memory

is called *virtual memory*

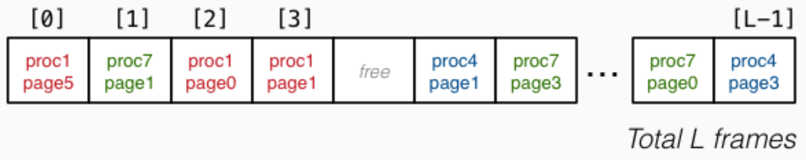
Virtual Memory

Pages/frames are typically 512B .. 8KB in size

In a 4GB memory, would have ≈ 1 million × 4KB frames

Each frame can hold one page of process address space

Leads to a memory layout like this (with *L* total pages of physical memory):



When a process completes, all of its frames are released for re-use

Virtual Memory

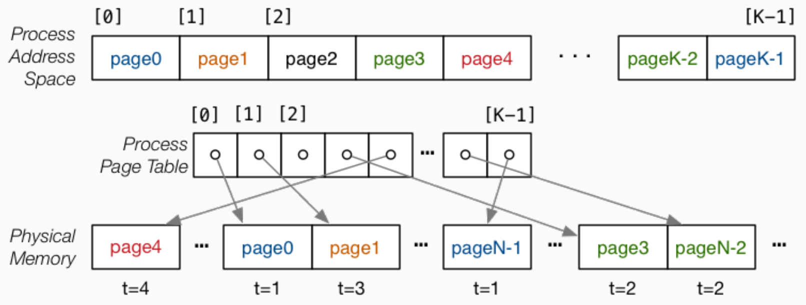
How to arrange mapping process address → physical address?

Consider a per-process page table, e.g.

- each page table entry (PTE) contains
  - page status . . . Loaded, IsModified, NotLoaded
  - frame number of page (if Loaded)
  - . . . maybe others . . . (e.g. last accessed time)
- we need $\lceil ProcSize/PageSize \rceil$ entries in this table

Virtual Memory

Example of page table for one process:



Timestamps show when page was loaded.

Virtual Memory

Virtual address to physical address mapping {(more detail)}:

```
typedef struct {int status, int frameNo, ...} PageData;

PageData *AllPageTables[maxProc];
// one entry for each process

Address processToPhysical(pid, Vaddr) {
   PageData *PageTable = AllPageTables[pid];
   int pageno = PageNumberFrom(Vaddr);
   int offset = OffsetFrom(Vaddr);
   if (PageTable[pageno].status != Loaded) {
     // load page into free frame
   // set PageTable[pageno]
   }
   int frame = PageTable[pageno].frameNo;
   return frame * P + offset;
}
```

An Aside: Working Sets

Consider a new process commencing execution . . .

- initially has zero pages loaded
- load page containing code for `main()`
- load page for `main()`'s stack frame
- load other pages when process references address within page

Do we ever need to load all process pages at once?

An Aside: Working Sets

From observations of running programs . . .

- in any given window of time, process typically access only a small subset of their pages
- often called *locality of reference*
- subset of pages called the *working set*

Implications:

- if each process has a relatively small working set, can hold pages for many active processes in memory at same time
- if only need to hold some of process's pages in memory, process address space can be larger than physical memory

Virtual Memory

We say that we "load" pages into physical memory

But where are they loaded from?

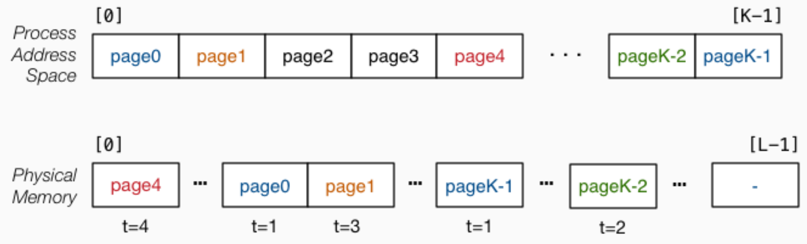
- code is loaded from the executable file stored on disk into read-only pages
- some data (e.g. C strings) also loaded into read-only pages
- initialised data (C global/static variables) also loaded from executable file
- pages for uninitialised data (heap, stack) are zero-ed
  - prevents information leaking from other processes
  - results in uninitialised local (stack) variables often containing 0

Consider a process whose address space exceeds physical memory

Virtual Memory

We can imagine that a process's address space . . .

- exists on disk for the duration of the process's execution
- and only some parts of it are in memory at any given time

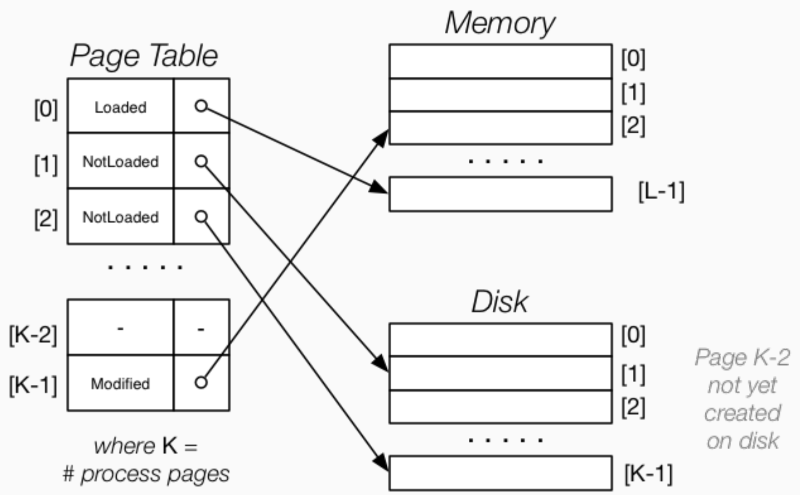


Transferring pages between disk↔memory is **very** expensive

- need to ensure minimal reading from / writing to disk

Virtual Memory

Per-process page table, allowing for some pages to be not loaded



Virtual Memory

Recall the address mapping process with per-process page tables

```
Address processToPhysical(pid, Vaddr) {
   PageData *PageTable = AllPageTables[pid];
   uint pageno = PageNumberFrom(Vaddr);
   uint offset = OffsetFrom(Vaddr);
   if (PageTable[pageno].status != Loaded) {
      // load page into free frame
      // set PageTable[pageno]
   }
   uint frame = PageTable[pageno].frameNo;
   return frame * P + offset;
}
```

What to do if the page is not loaded?

Page Faults

Requesting a non-loaded page generates a {*page fault*}.

One approach to handling a page fault . . .

- *find a free (unused) page frame in memory and use that*

```
// load page into a free frame ...
else {
  frameno = getFreeFrame();
  p->frameNo = frameno;
  p->status  = Loaded;
}
```

- *Assumes that we have a way of quickly identifying free frames*
- *Commonly handled via a* {free list}
- Reminder: frames allocated to a process become {*free*} *when process exits*

No Free Page

What happens if there are currently no free page frames

What does `getFreeFrame()` do?

Possibilities:

- *suspend* the requesting process until a page is freed
- *replace* one of the currently loaded/used pages

Suspending requires the operating system to

- mark the process as unable to run until page available
- switch to running another process
- mark the process as able to run when page available

Page Replacement

What happens when a page is replaced?

- if it's been modified since loading, save to disk ** \ {(in the disk-based virtual memory space of the running process)}
- grab its frame number and give it to the requestor

How to decide which frame should be replaced?

- define a "usefulness" measure for each frame
- grab the frame with lowest usefulness

** we need a flag to indicate whether a page is modified

```
#define NotLoaded           0x00000000
#define Loaded              0x00000001
#define IsModified          0x00000002
```

Page Replacement

Factors to consider in deciding which page to replace

- best page is one that won't be used again by its process
- prefer pages that are read-only (no need to write to disk)
- prefer pages that are unmodified (no need to write to disk)
- prefer pages that are used by only one process (see later)

OS can't predict whether a page will be required again by its process

But we do know whether it has been used recently (if we record this)

Useful heuristic: *LRU replacement*

- a page not used recently may not be needed again soon

Page Replacement

Factors for choosing best page to replace are *heuristic*

What happens if ...

- we replace a page which is soon used again
- this causes us to replace another page
- and the second page is soon used again ...?

Thrashing = constantly swapping pages in and out of memory

The working set model plus LRU helps avoid *thrashing*

- recently used page is likely to be used again soon
- not recently used page is unlikely to be used again soon

Page Replacement

LRU is one replacement strategy. Others include:

First-in-first-out (FIFO)

- page frames are entered into a queue when loaded
- page replacement uses the frame from the front of the queue

Clock sweep

- uses a reference bit for each frame, updated when page is used
- maintains a circular list of allocated frames
- uses a "clock hand" which iterates over page frame list
  - skipping and resetting reference bit in all referenced pages
- page replacement uses first-found unreferenced frame

{**Exercise: Page Replacement**}
Show how the page frames and page tables change when

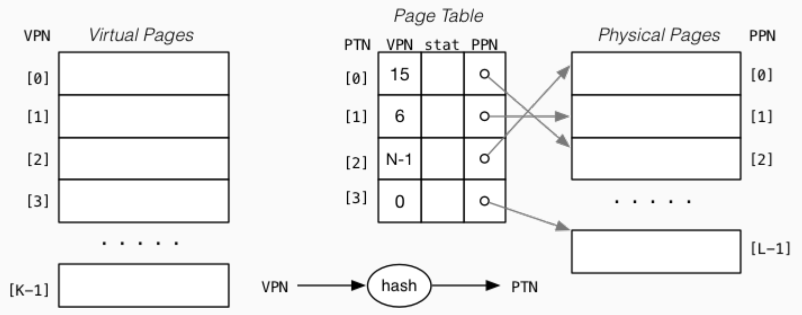
- **there are 4 page frames in memory**

Virtual Memory

Page tables (PTs) revisited ...

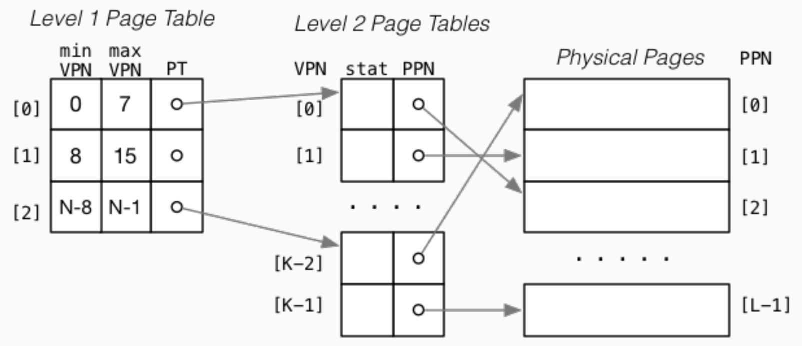
- a virtual address space with K pages needs K PT entries
- since K may be large, do not want to store whole PT
- especially since working set tells us $n \ll K$ needed at once

One possibility: PT with $n < K$ entries and hashing



Virtual Memory

Alternative strategy: multi-level page tables

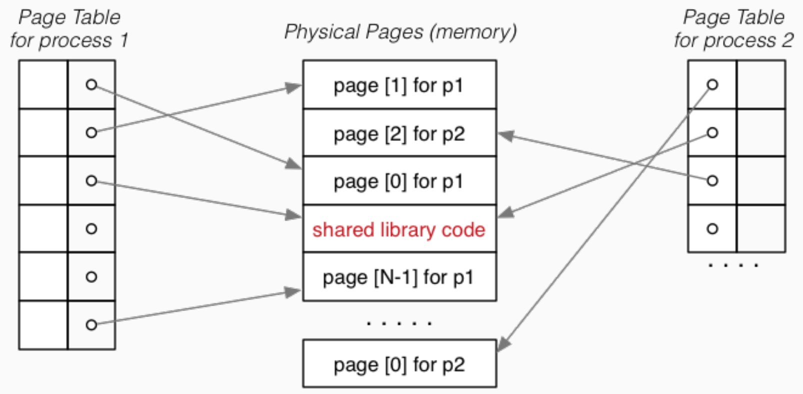


Effective because not all pages in virtual address space are required
(e.g. the pages between the top of the heap and the bottom of the stack)

Virtual Memory

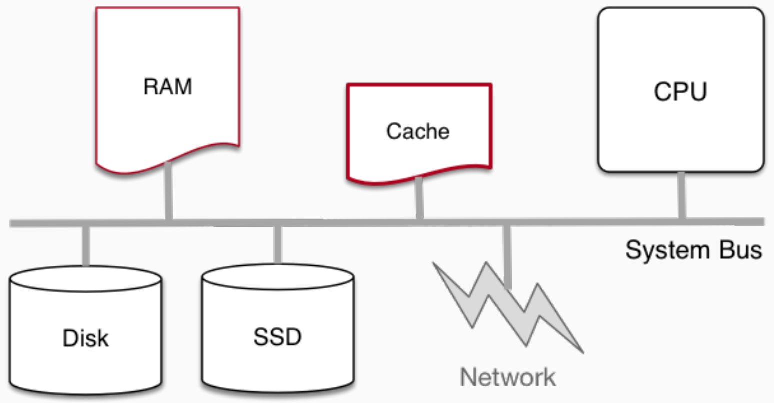
Virtual memory allows sharing of read-only pages (e.g. library code)

- several processes include same frame in virtual address space



Cache Memory

Cache memory = small*, fast memory* close to CPU



Small = MB, Fast = 5 × *RAM*

Cache Memory

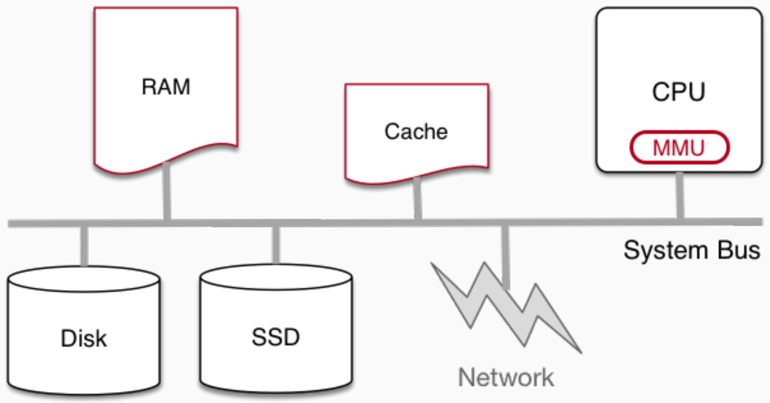
Cache memory

- holds parts of RAM that are (hopefully) heavily used
- transfers data to/from RAM in blocks (*cache blocks*)
- memory reference hardware first looks in cache
  - if required address is there, use its contents
  - if not, get it from RAM and put in cache
  - possibly replacing an existing cache block
- replacement strategies have similar issues to virtual memory

Memory Management Hardware

Address translation is very important/frequent

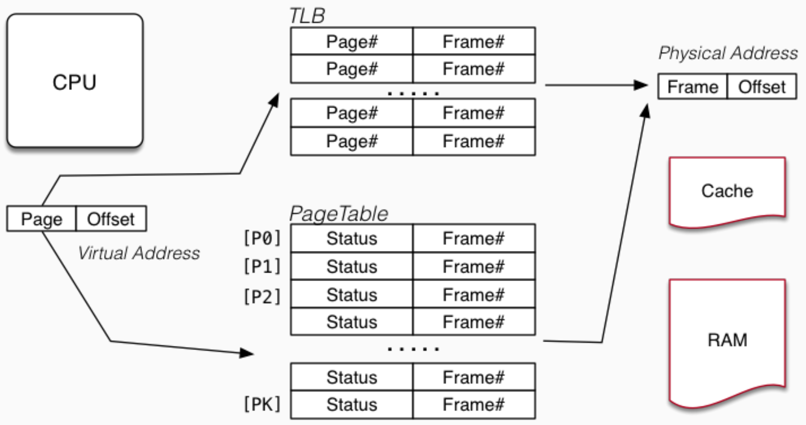
- provide specialised hardware (MMU) to do it efficiently
- sometimes located on CPU chip, sometimes separate



Memory Management Hardware

TLB = translation lookaside buffer

- lookup table containing (virtual,physical) address pairs



CPU
TLB
Page# Frame#
Physical Address
Frame Offset
Cache
Page Offset
Virtual Address
PageTable
[P0] Status Frame#
[P1] Status Frame#
[P2] Status Frame#
[PK] Status Frame#
RAM