Lecture 9:

Memory Management II:
Paging and Segmentation

`www.cl.cam.ac.uk/Teaching/2001/OSFounds/`

Lecture 9: Wednesday 24th October 2001

Today's Lecture

In the last lecture, we considered the question:

> How do we manage memory when sharing the CPU between many processes?

But we saw the problem of fragmentation, and a possible solution: compaction.

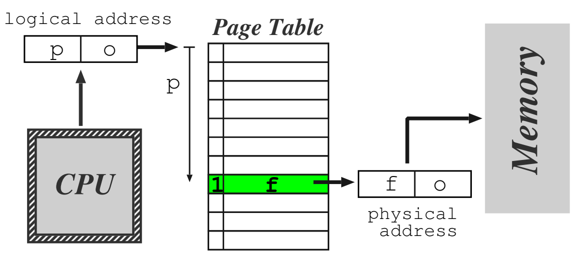
Today we'll look at another idea:

> What if we allow a process to reside in **non-contiguous** memory?

We'll consider two possible methods:

1. **Paging**, and
2. **Segmentation**.

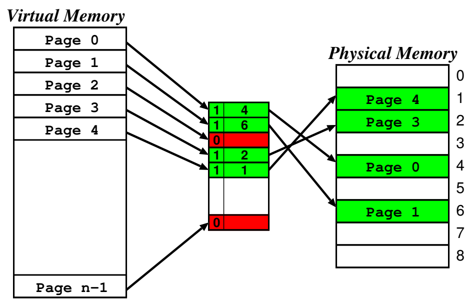
Paged Virtual Memory



Another solution is to allow a process to exist in **non-contiguous memory**, i.e.

- divide physical memory into relatively small blocks of fixed size, called **frames**
- divide logical memory into blocks of the same size called **pages** (typical value is 4K)
- each address generated by CPU is composed of a page number p and page offset o.
- MMU uses p as an index into a **page table**.
- page table contains associated frame number f
- usually have $|\mathtt{p}| \gg |\mathtt{f}| \Rightarrow$ need valid bit.

Paging Pros and Cons



- ✔ memory allocation easier
- ✘ OS must keep page table per process
- ✔ no external fragmentation (in physical memory at least)
- ✘ but get **internal fragmentation**
- ✔ clear separation between user and system view of memory usage.
- ✘ additional overhead on context switching

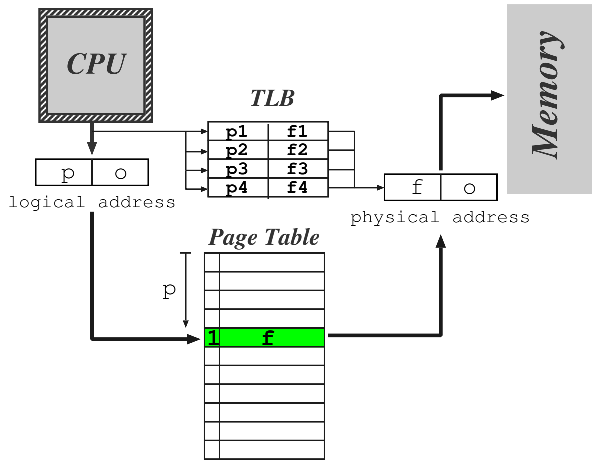
Structure of the Page Table

Different kinds of hardware support can be provided:

- Simplest case: set of dedicated relocation registers
  - one register per page
  - OS loads the registers on context switch
  - fine if the page table is small. . . but what if have large number of pages ?
- Alternatively keep page table in memory
  - only one register needed in MMU (page table base register (PTBR))
  - OS switches this when switching process
- Problem: page tables might still be very big.
  - can keep a page table length register (PTLR) to indicate size of page table.
  - or can use more complex structure (see later)
- Problem: need to refer to memory **twice** for every 'actual' memory reference. . .

⇒ use a **translation lookaside buffer (TLB)**

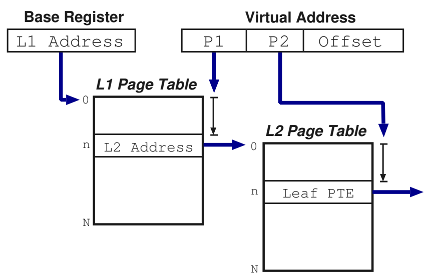
TLB Operation



- On memory reference present TLB with logical memory address
- If page table entry for the page is present then get an immediate result
- If not then make memory reference to page tables, and update the TLB

Multilevel Page Tables

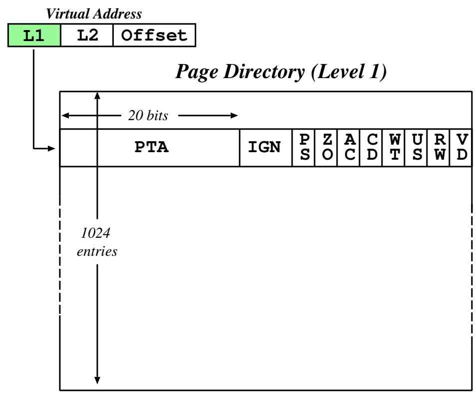
- Most modern systems can support **very large** $(2^{32}, 2^{64})$ address spaces.
- Solution: split page table into several sub-parts
- Two level paging—page the page table



- For 64 bit architectures a two-level paging scheme is not sufficient: need further levels.

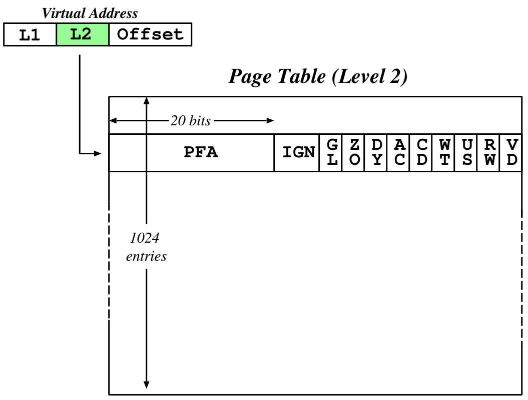
  (even some 32 bit machines have > 2 levels).

Example: x86



- Page size 4K (or 4Mb).
- First lookup is in the **page directory**: index using 10 most significant bits.
- Address of page directory stored in internal processor register (cr3).
- Results (normally) in the address of a **page table**.

Example: x86 cont.



- Use next 10 bits to index into page table.
- Once retrieve page frame address, add in the offset (i.e. the low 12 bits).
- Notice page directory and page tables are **exactly one page each themselves**.

Protection Issues

- Associate protection bits with each page—kept in page tables (and TLB).

  e.g. one bit for read, one for write, one for execute.
- May also distinguish whether may only be accessed when executing in *kernel mode*, e.g.

Frame Number	K	R	W	X	V

- At the same time as address is going through page hardware, can check protection bits.
- Attempt to violate protection causes h/w trap to operating system code
- As before, have **valid/invalid** bit determining if the page is mapped into the process address space:
  - if invalid ⇒ trap to OS handler
  - can do lots of interesting things here, particularly with regard to sharing. . .

Shared Pages

Another advantage of paged memory is code/data sharing, for example:

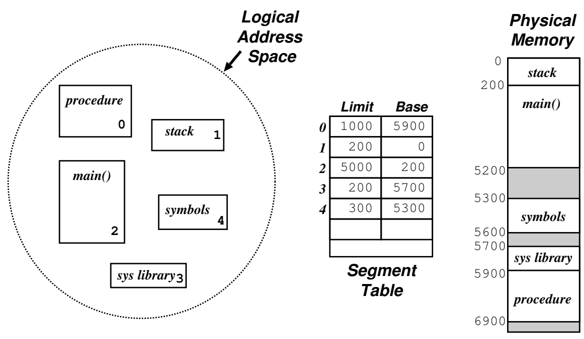
- binaries: editor, compiler etc.
- libraries: shared objects, dlls.

So how does this work?

- Implemented as two logical addresses which map to one physical address.
- If code is **re-entrant** (i.e. stateless, non-self modifying) it can be easily shared between users.
- Otherwise can use **copy-on-write** technique:
  - mark page as read-only in all processes.
  - if a process tries to write to page, will trap to OS fault handler.
  - can then allocate new frame, copy data, and create new page table mapping.
- (may use this for lazy data sharing too).

Requires additional book-keeping in OS, but worth it.

Segmentation



- User prefers to view memory as a set of segments of no particular size, with no particular ordering.
- **Segmentation** supports this user-view of memory — logical address space is a collection of (typically disjoint) segments.
- Segments have a name (or a number) and a length—addresses specify segment and offset.
- Contrast with paging where user is unaware of memory structure (all managed invisibly).

Implementing Segments

- Maintain a segment table for each process:

Segment	Access	Base	Size	Others!

- If program has a very large number of segments then the table is kept in memory, pointed to by **ST base register STBR**.
- Also need a **ST length register STLR** since no. of segs used by different programs will differ widely
- The table is part of the process context and hence is changed on each process switch.

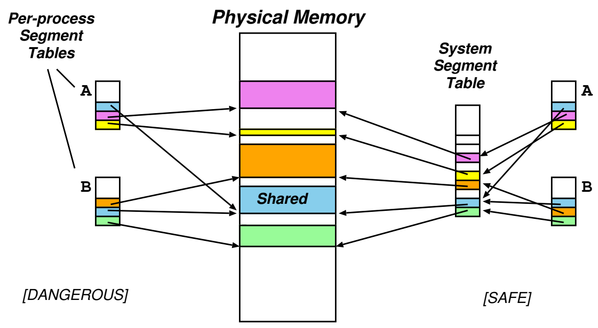
Algorithm:

1. Program presents address (s, d). Check that $s <$ STLR. If not, fault
2. Obtain table entry at reference $s+$ STBR, a tuple of form (b_s, l_s)
3. If $0 \leq d < l_s$ then this is a valid address at location (b_s, d), else fault

Sharing and Protection

- Big advantage of segmentation is that protection is **per segment**; i.e. corresponds to logical view.
- Protection bits associated with each ST entry checked in usual way, e.g.
  - instruction segments (should be non-self modifying!) thus protected against writes etc.
  - place each array in own seg ⇒ array limits checked by hardware
- Segmentation also facilitates sharing of code/data:
  - each process has its own STBR/STLR
  - sharing is enabled when two processes have entries for the same physical locations.
  - for data segments can use copy-on-write as per paged case.
- Several subtle caveats exist with segmentation — e.g. jumps within shared code.

Sharing Segments



Sharing segments:

- wasteful (and dangerous) to store common information on shared segment in each process segment table
- assign each segment a unique **System Segment Number (SSN)**
- process segment table simply maps from a Process Segment Number (PSN) to SSN

External Fragmentation Returns. . .

- Long term scheduler must find spots in memory for all segments of a program.
- Problem now is that segs are of variable size ⇒ leads to **fragmentation**.
- Tradeoff between compaction/delay depends on average segment size
- Extremes:
  - each process 1 seg — reduces to variable sized partitions, or
  - each byte 1 seg separately relocated — quadruples memory use!
- Fixed size small segments ≡ paging!
- In general with small average segment sizes, external fragmentation is small.

Summary

You should now understand:

- Paging, and
- Segmentation.

Next lecture: I/O

Background Reading:

- Silberschatz et al.: – Sections 9.4& 9.5.