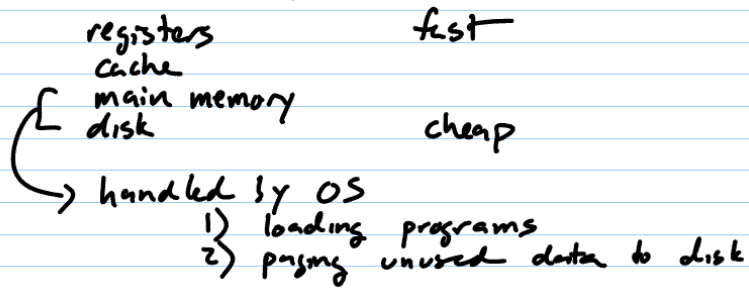


Memory hierarchy

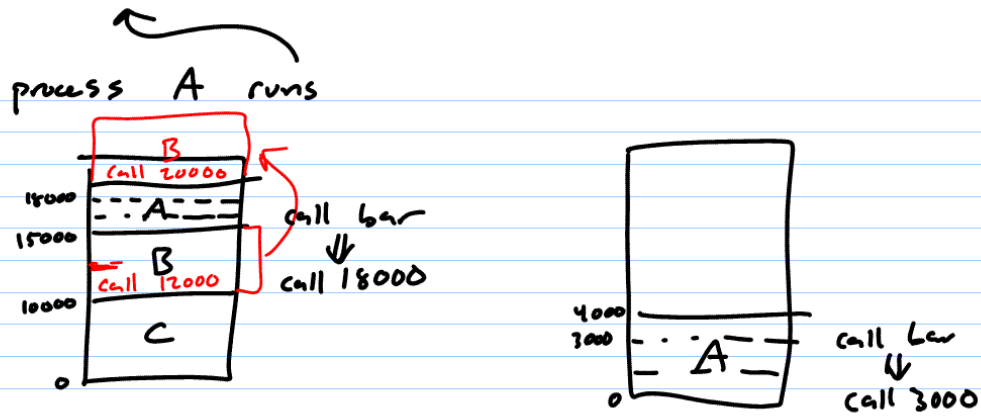


other issues for memory manages

- 1) protection — make sure process A can't trash B's data
- 2) relocation of loaded programs

↪ int foo()
 {
 bar(); → call addr of bar
 }
 call 3050





monoprogramming: protect OS from running process

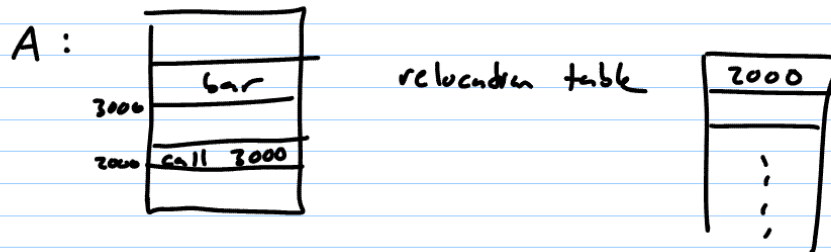
simple schemes for protection: base + limit registers
 only changed in kernel mode
 on access to memory loc. x ,
 check $base \leq x < base + limit$

works for multiprogramming protection (mono: base always 0
 multi: base start addr of process)

| OS PCBs : | A | B | C |
|-----------|-------|-------|-------|
| base | 15000 | 10000 | 0 |
| limit | 4000 | 5000 | 10000 |

on context switch (transfer control to next process)
OS loads values from PCB into HW base/limit regs

relocation: add to each relocatable a relocation table giving offsets to address that need to be changed as programs are moved around



helpful hardware feature: relocatable code

↳ address given as offsets instead of absolute

call 3000 @ addr 2000 becomes call +1000

Virtual Memory

Each process has own private virtual address space divided into pages
 A even if loaded at physical addr 15000
 call bar is left as call 3000
 MMU translates virtual addr 3000 to physical addr 15000

paging allows processes to grow w/o moving around
 and allows parts of process to be **on disk / parts in mem**

Page Table: tells MMU how to translate virtual \rightarrow physical address

virtual addr space divided into pages
 page table map pages to frames (blocks of physical memory)

translation: 1) figure out what page a virtual addr is in
 2) look up in page table the corresponding frame
 3) glue together to get physical addr

page table: present/absent bit
 protection bits (read/write/executable)
 referenced/modified bit (optional)

page table is unique to each process

on context switch, OS must tell MMU about new page table

MMU can have own memory to hold page table
OS copies its page table to MMU hardware
or have a single reg that points to page table
in main memory

Ex: System w/ 64 kB physical memory
 Virtual address space 64 kB per process
 page size = frame size = 8 kB (virtual addr space divided into 8 pages of 8 kB each; physical mem is 8 frames, 8 kB each)



$$8192 = 2^{13}$$

call 8192 =

| | |
|-----|--------------|
| 001 | 000000000000 |
|-----|--------------|

 ↓ mmu
 call 24576

↑
 index into page table
 physical addr

| | |
|-----|--------------|
| 011 | 000000000000 |
|-----|--------------|

A's virtual addr 16400
 ||

| | |
|-----|--------------|
| 010 | 000000010000 |
|-----|--------------|

offset 16 in page 2

| | |
|-----|--------------|
| 101 | 000000010000 |
|-----|--------------|

↓
 physical addr 40976

Ex: 32-bit virtual addr space (4GB)
 32-bit physical addr (4GB)

page size 16kB

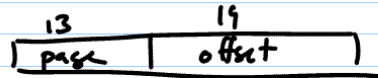
| | |
|------|--------|
| 18 | 14 |
| page | offset |

 virtual addr

each process has $2^{18} = 256K$ pages
 page table then has 256K entries

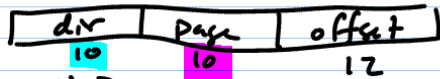
(@ 32 bits per entry 1 MB per process
4 bytes for page table)

page size 512 kB
= 2^{19} bytes



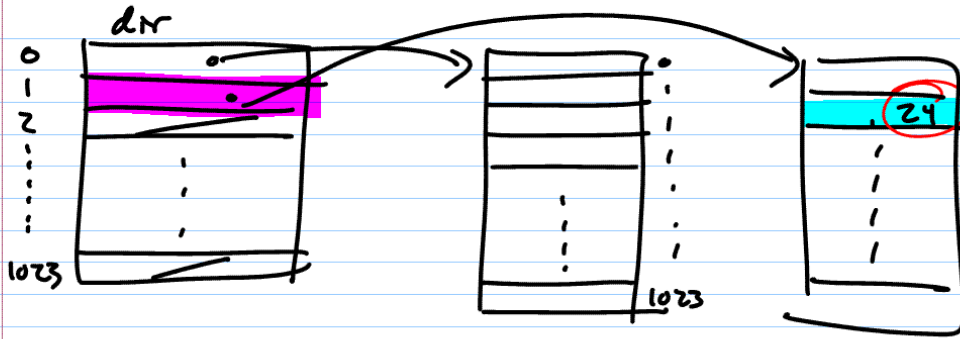
each process has 2^{13} pages
= 8192
(@ 4 bytes/entry,
32 kB per page table)

to deal with large page tables, have multi-level page table



page size 4 kB
↳ 2^{12} bytes

dir used as index into top-level page dir
page dir entries contain pointers to smaller page tables
↳ 2^{10} dir entries; each points to 2^{10} page table entries
1024



process A uses 2000 pages - only uses 2 dir entries

virtual addr

00401080

3072 entries total
(would have been 2^{20}
in single-level
system)

dir entry : 000000001
page : 000000001

00018080 physical addr

virtual addr

08000000
000010000

=> page fault

single-level is faster b/c fewer table lookups needed

solution: Translation Lookaside Buffer (TLB)
 - cache for page table entries

add (0040, 0018) to TLB

Suppose process A, B share a lot of code (iostream, string, cmath)

