Virtual Memory

①

Previous: Required entire process to be in memory

not always needed: locality of reference

Loc of Ref:

Programs spend most time in small piece of code

Knuth: 90% of time in 10% of code

Don't need entire AS in mem @ once

Idea: keep unused pages on ~~slow~~ disk!

Process can run even if all pages not loaded!

OS: must manage which pages in mem, disk

carefully:

⇒ memory hierarchy

size	cost		
32 (in the room)	$$$	registers ↓	1 cycle (a few ns)
a few MB (next door)	$$	cache ↓	a few cycles (10's of ns)
a few GB (down the hall (long hall))	$	main memory ↓	100s of ns
many GB (1000 miles)	¢	disk	a few ms

[PAGE CACHE]

Implication: Page Access

needed: H/W + S/W

page table: present bit

[VA | off]

↓

check TLB $\xrightarrow{\text{match!}}$ [PA | off]

↓ TLB fault

check page table $\xrightarrow{\text{match}}$ [PA | off] update TLB

↓ page fault (trap to OS)

(page not in memory)

fetch page from disk

update page table

update TLB

} problem: continue (restart instruction)

problem: must kick out old page

which one?

Continuing a Process : Mechanisms

Can be difficult

Depends on ISA

simple case:

load <addr>, R_2 → page fault on addr

⋮

just restart : ok

dest / to, srce / from

copy <$addr_1$>, <$addr_2$>, R_1 ← size

(R_dst), (R_src), R_size

only ok if copy is not destructive

[move +(SP), R_2] auto increment

H/W support helps

OS Decisions: Policies

Page Selection: when to bring page from disk to memory

Replacement: which page(s) to move to disk?

Selection:

Demand Paging: load page only on fault, wait until referenced

start up w/ none in memory

[No OS choice] Request paging: user-specified / control overlays

really hard.

Prepaging: Load page before referenced

hard to know when this is a good idea

Clustering: when bringing in 1 page, bring in many (e.g. code pages)

Page Replacement

Optimal: Throw out page that won't be accessed for longest time in future
→ best if access pattern is known
(not practical)
ideal

simple

Random: Pick any @ random
easy to implement, may work OK

FIFO: simple : throw out oldest page
Fair per page

recency based

LRU: Throw out page that hasn't been used in longest time
(past predicts future ?)
w/ locality, LRU approximates OPT

frequency based

LFU: least frequently used
when bad?

MFU: used little, must be needed soon!

E.g./ Ref string A B C A B D A D B C B

FIFO

A B C ~~A~~ B C

A
B
D replace! D B C
A replace! D A C
D
B replace! D A B
C C A B
B fault —

OPT

A B C

F A B D

A B C or C B ~~B~~
fault

LRU

A B C

F A B D
LRU

C B D
fault

String

1 2 3 4 1 2 5 1 2 3 4 5

A B C D A B E A B C D E

FIFO : 3

1 2 3

~~1~~ ~~2~~ ~~3~~

~~4~~ ~~1~~ ~~2~~ 9

5 3 4

FIFO : 4

4 + 4 + 2 = 10

~~1~~ ~~2~~ ~~3~~ ~~4~~

~~5~~ ~~1~~ 2 3

4 5

>> Belady's Anomaly <<

Implementing LRU

TLB miss , page fault
TLB miss , page hit
TLB hit , page ~~hit~~ ~~fault~~ hit
TLB ~~miss~~ hit , page ~~[illegible]~~ ~~fault~~ fault

Software Perfect LRU:

Keep ordered list of pages
on memory reference, move page to front of list
Replacement: remove page from back of list

H/W Perfect LRU:

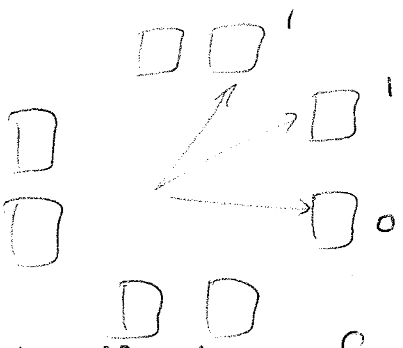
Register / page
Memory ref: put system clock in register   clock overflow
Replacement: scan, find oldest

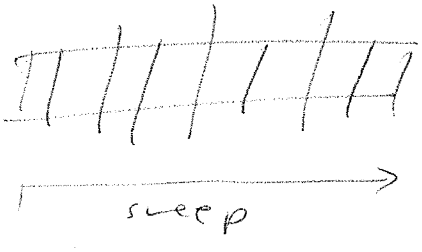
=> Impractical to support efficiently
=> approximate: old page, not oldest

Clock Algorithm "second chance"

H/W
Keep use / reference bit per page
On memory reference, set bit

S/W
Replacement: Look for page w/ use bit cleared
Scan: traverse all pages, clearing bits over time





> what if too fast? too slow?

Extension : Dirty Bit

Add another bit: dirty

page is dirty if it's been modified

when replaced, must be written to disk

⇒ Prefer kicking out clean : why?

Problems can occur when:

clean pages ref'd frequently (code)

⇒ might get thrown out instead of lesser used dirty page

Extension : Dirty Bit

Add another bit: dirty

page is dirty if it's been modified

when replaced, must be written to disk

⇒ Prefer kicking out clean : why?

Problems can occur when:

clean pages ref'd frequently (code)

→ might get thrown out instead of lesser used dirty page

How to Allocate Across Processes?

2 ways
Global Replacement
Per-process replacement

Global

All pages lumped in single pool
(all compete for page frames) ⇒ Fault: replace *any* page
\+ flexible, no limits
\- one process can dominate

Per-Process (Local)

Each process has separate pool
fixed # of pages, frac of phys mem
Page fault: replace your page
\+ isolation
\- inefficient

⇒ also, could do per user

Hybrid (VAX/VMS)

Per-process limits
when removed from per-process list
⇒ Global list
only when removed from here ⇒ disk
miss in per-process list checks global list

Over commiting Memory

Processes: actively accessing N+1 pages
only N page frames

Cycle / ~~of~~

(Ref page not in memory
replace page w/ new one)

=> **Thrashing**

System just moving pages back and forth
(many) each { load instr / instruction fetches } => disk!

Illusion of virtual memory breaks
(no longer an "infinite" amt of fast mem)

=> **E.g.**

H : % of hits
C_{mem} : in-mem page access (10-~100ns)
C_{fault} : page not in mem ~25 ms (3-18 ms)

Overall cost =

$$H \cdot C_{mem} + (1-H)\,C_{fault}$$

if ~~just~~ $H = 97\%$

=> OCost = 750 µs (7500 X in-mem)

=> even small miss rate is a problem

System : does not know it has
too much work

Page replacement:
view is too narrow
(only which page to replace)

Note say:
Student analogy: too many courses
drop one!
(or focus on one/few @ a time,
ignore others
⇒ allows progress)

OS approach: "Admission Control"
don't let everyone run
only run those who fit in memory
rest are swapped to disk

how to detect?
certain amt of paging going on?

what are potential problems?
starvation: large memory jobs get stuck waiting
single mem hog: never runs?

Different Approach : working sets

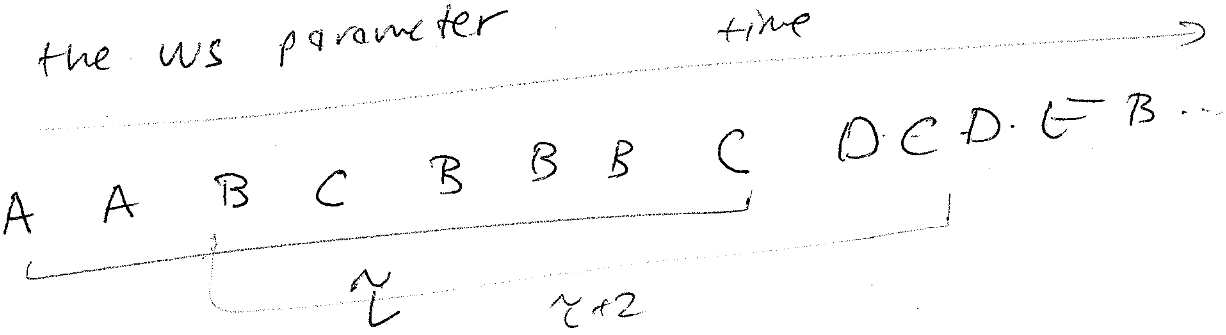
Informally: "working set" is...

- > collection of pages ref'd frequently at a given time by process
- > must be resident to avoid thrashing

Locality:
recent use ⇒ future?

More formally:
(unique) pages ref'd by process w/ in last τ seconds

τ : the ws parameter



ws = ABC

ws = BCD

Goals:

Ensure WS / process remains in memory

Pages not in WS : Discard whenever

Long-term:
Process not executed unless
WS resident in main memory

Balance set

Keep __all__ processes in mind

Two groups:

__Active__ : WS is loaded

__Inactive__ : WS on disk

"Balance set":

sum of WS's across __active__ processes

__Long-term__:

__Policy #2:__

if Bset > mem,

move some active → inactive

__Policy #2:__

when to move processes

back from inactive ⇒ active?

Implementing WS

Hard to know real WS
- what pages have been accessed
  - w/in last τ seconds?
- analogy to capacitor
  - charge on mem ref,
  - discharge if not ref'd
  - if charge is "low", not in WS

Really, leverage use bits
- + idle time per page
- idle time: amt of CPU received by
  - process since last access
- Periodically scan resident pages
  - if use bit on, clear idle time
  - if off, add CPU (since last scan)
    - to idle time
  - clear use bit (set~~E~~ to off)
- How often to scan?
  - too fast?
  - too slow?

Unresolved

what should τ be?

too large?

small?

How to pick active processes?

How to compute is pages are shared?

How much memory is needed?

Trends

Memory is cheap

- ⇒ Not under contention as in early machines
- ⇒ If system pages a lot, something is wrong

Large page sizes

⇒ UltraSPARC 8K, 1MB

why? ~~⇒~~ fewer pages to manage

⇒ more internal frag

Larger Virtual AS

64-bit VA

32-bit limits some apps (a few GB)

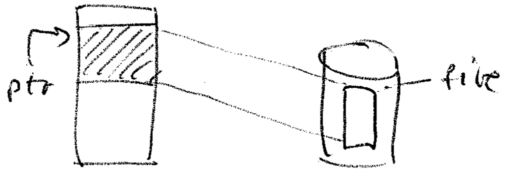
Page tables: must be sparse, or inverted

Integrated VM / File I/O

⇒ madvise()

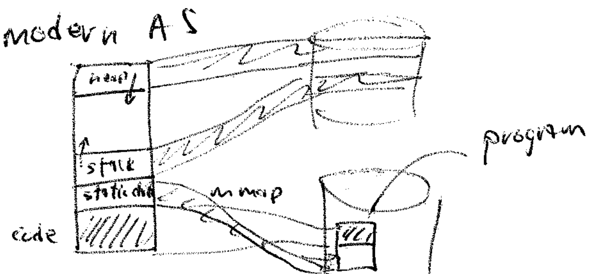
LRU: worst case

mmap()



access: fault, data read from file

update: dirty, event. flush to disk

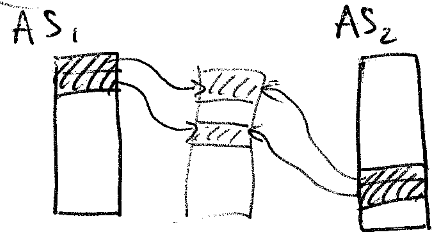


Advanced Topics

Sharing Memory

shmctl() shmget()
shmat()

AS₁ AS₂



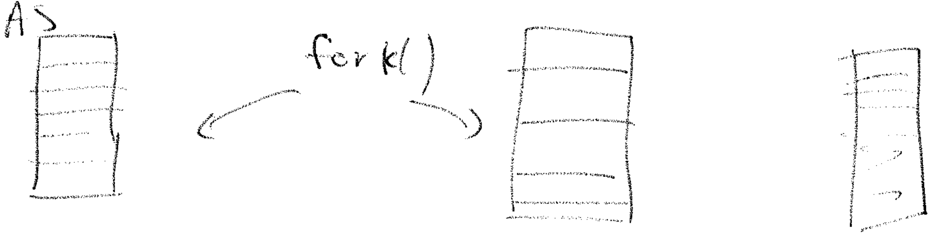
~~Multiprocessors~~

~~> Some memory faster to access than others~~
~~=> page migration~~

Copy-on-write

exact copy

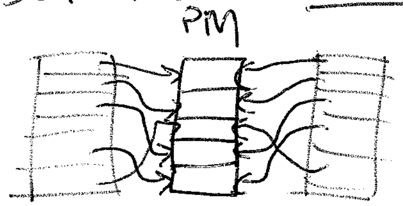
"ls"
execv()



(overlay w/ new image)

=> wasteful : why go to all the work of that copy

=> solution: lazy copy



> mark page as read only (protection)
> when written to, catch fault,
=> ~~write~~ make copy

Advanced Topics

> Software emulated "use" bit

Q) how to tell if page has been used by process recently?

H/W: use bit

S/W: use protection bits!

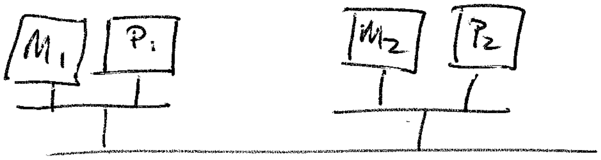
mark all as unreadable

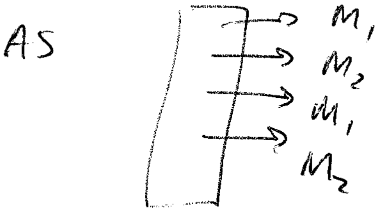
when accessed, fault, OS notes that page has been read_written (dirty)

pages that have not been unmarked can be thrown out!

> Multiprocessors

Some memory is cheaper to access than others





⇒ how to fix?

> page migration (data → process)

> process migration (process → data)