Virtual Memory

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 at once

Idea: keep unused pages on disk!
Process can run even if all pages not loaded:
OS: must manage which pages in mem, disk
carefully:

⇒ memory hierarchy

size cost $/$ registers ↓ cycle (a few ns)
32 $/32a
a few MB (at random)
↓ cache (10% of ns)
next door

↓ main memory

GB down the hall (600 ns)

many GB (1000 miles)
disk

100s of ms

PAGE CACHE

Implication: Page Access

VA off

check TLB match!

TLB fault

check page table match

↓ page fault (trap to OS)
(page not in memory)

fetch page from disk update page table
update TLB

needed: H/w + S/w

page table: present bit

problem: must pick out old page which one?
Continuing a Process: Mechanisms

Can be difficult
Depends on ISA

Simple case:

\[
\text{load} (\text{addr} \rightarrow R2) \rightarrow \text{page fault on addr}
\]

\[\text{just restart: OK}\]

\text{copy} (\text{addr}, \text{Raddr}, \text{Rsize})

\[\text{dest} \text{ size from src}
\]

\[\text{only ok if copy is not destructive}\]

\[
\text{move} (+\text{(SP)}, R2)
\]

\[\text{auto increment}\]

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

start up w/ none in memory, wait until referenced

\[\text{No OS}\]

Request paging: user-specified/control

really hard, overlays

Pre-paging: 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 assessed for longest time in future → best if access pattern is known (not practical)

LFU: Least Frequently Used

LRU: Throw out page that hasn't been used in longest time (past predicts future?)

MFU: Used little must be needed soon!

E.g. Ref String: A B C A B D A D B C B

FIFO

OPT

LRU

A B C

A B D

A B D

A B C

A B D

A B D
>> Belady's Anomaly <<
Implementing LRU

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 per page
- Memory ref: put clock in register
- Replacement: scan and 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?
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
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 referenced frequently (code)
- Might get thrown out instead of less 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) \( \Rightarrow \) 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

\( \Rightarrow \) also, could do per user

Hybrid (VAX/VMS)

Per-process limits when removed from per-process list
\( \Rightarrow \) Global list
only when removed from here \( \Rightarrow \) disk
miss on per-process list checks global list
Overcommitting Memory

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

Cycle

- Ref page not in memory
- Replace page with new one

⇒ Thrashing

- System just moving pages back and forth
- (many)
- Each load instruction = disk
- Instruction fetches

Illusion of virtual memory breaks
- (no longer an "infinite" amount of first men)

⇒ E.g.,

- \( H \): % of hits
- \( C_{\text{mem}} \): in-mem page access \( \sim 100 \, \text{ns} \)
- \( C_{\text{fault}} \): page not in mem \( \sim 25 \, \text{ms} \)
- \( (3-18 \, \text{ms}) \)

Overall cost =

\[ 
H \cdot C_{\text{mem}} + (1-H) \cdot C_{\text{fault}} 
\]

if \( H = 97\% \)

⇒ \( \Omega \text{Cost} = 750 \, \text{ms} \)

(\( 7500 \times \) 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 pages going on?

what are potential problems?
starvation: large memory jobs get stuck waiting
single men hog: never runs
Different Approach: working sets

Informally: "working set" is a collection of pages refilled frequently or resident to avoid thrashing.

More formally: by process / in list / is

Locality: recent use ⇒ refill

Process parameter

Refill margins to avoid

Page sets

Us = ABC

Us = BCB

Us = ABC

Us = BCB

Us = ABC

Us = BCB

Us = ABC

Us = BCB
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 #1:

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 2 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 to off)

How often to scan?
too fast?
too slow?
Unresolved

what should \( n \) 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 OK / MB

Why? fewer pages to manage
⇒ More internal frag

Larger Virtual AS

64-bit VA

Page tables:
must be sparse, or inverted

Integrated VM / File I/O

mmap()

Access: fault
data read from file
Update: dirty, event
Flush to disk

madvise()

cdev: worst case
Advanced Topics

Sharing Memory

Multiprocessors

Copy-on-write

⇒ wasteful: why go to all the work of that copy

⇒ solution: lazy copy

Advanced Topics

> Software emulated "use" bit

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

H/W: use bit

S/W: use protection bits!
mark all as unreadable
even accessed, fault, or
notes that page has been read, written
pages that have not been
unmarked can be thrown out!

> Multiprocessors makes sense to access

Some memory is cheaper than others

\[
\begin{array}{c|c|c}
M_1 & P_1 & M_2 \\
\hline
M_1' & | & M_2' \\
\end{array}
\]

=> how to fix?

> page migration (data \rightarrow process)

> process migration (process \rightarrow data)