

Page Replacement

Module 12

Allocation & Replacement

We have one last pending issue:



- For each process, only a few pages are going to be in physical memory
 - Which ones?
 - How many?
 - How do we decide?

Allocation - in brief

- We will need to allocate X frames to each process (we'll determine X later)
- We only bring pages into memory when references - “on demand” paging
- We keep track of pages that have been modified, and only store them back to disk if they are “dirty”

Page Fault: Revisited

- When a page is not in memory, the OS must find the page on the backing store
- Once found, it is loaded into a frame.
 - What if there are no available frames?
 - We must select a page to be moved back to the backing store... a “victim”



Which page should be removed?

- Could be selected at random
- But... can we reason a little about what would be a good choice?
- How about the page that won't be used in the future?

Reference Strings and Simulation

- Before we evaluate any replacement strategies, we should agree on a “counting” method:
 - Memory references come into the MMU as a stream of addresses
 - From our perspective (now), it's not the address that is important... it's the page!

Reference Strings

0100, 0432,
0101, 0612, 0102, 0103, 0104,
0101, 0611, 0102, 0103, 0104,
0101, 0610, 0102, 0103, 0104,
0101, 0609, 0102, 0105

What might this code be doing?

How many page faults if we only have 1 frame available?

If Page Size == 100 Bytes, then we are looking at references to page # 1, 4, 1, 6, 1, 6, 1, 6, 1

We don't concern ourselves with consecutive requests to the same page

Simulating the optimal outcome:

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Frame Allocation: 3 Frames

Replacement Strategy: Replace frame that won't be used for the longest time.

That “optimal” word again...



- Everything is easy when you can predict the future!
- The OS doesn't know which addresses will be accessed in the future!
- We must approximate

History repeats itself

- The MMU has a R (referenced) and M (modified) bit for each page (in page table)
- These bits are set to 1 when the page is read/written to.
- They stay at 1 until OS clears them.

NRU Algorithm

Class 0: Not referenced, Not Modified

Class 1: Not references, Modified

Class 2: Referenced, Not Modified

Class 3: Referenced, Modified

- OS periodically (quantum) clears all R bits
 - Don't clear M bits - they are needed for swapping
- On replacement, **randomly** select a page with lowest class #

FIFO Page Replacement

- Simply put, hold page #'s in a queue
- When you need to select one to evict, select the front of the queue.
- When a page is loaded, place it at the end of the queue.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Using 3 frames...

The problem with FIFO...

- The problem with FIFO is that it can replace a **heavily** used page.
- **Second - Chance FIFO:**
 - Using the R bit, if a page has $R = 1$, don't replace it and put it at back of queue... (and clear bit)
 - Periodically (quantum) clear all bits

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Using 3 frames...

NRU/Second-Chance

- NRU requires a search of “class #”
 - NRU then “punts” on the question of which one if multiple in same class...
-
- Second-Chance is more deterministic, and also more efficient.

Least-Recently Used

Second-Chance and NRU approximate the idea that if you've use a page, you'll use it again.

However - they do not really distinguish between heavily used, and just pages that got used once...

Least Recently Used

LRU selects the page that hasn't been used for the longest time... but it's expensive:

- 1) In hardware, each page table gets a timestamp (integer).
- 2) On **each reference**, set the page table's timestamp
- 3) On page fault, search for the smallest number

Between Second-Chance and full LRU

We can get a good compromise by allocating a small set of bits (as opposed to 64!) for each page table index

1. On each memory reference, set left-most bit to 1
2. On each quantum, shift all bits right - replacing left-most bit with 0
3. On replacement, choose largest value

Back to Allocation

OK... so how many frames does a process get?

- Not all processes are created equal - some are typically using more pages than others.
- Page faults destroy overall performance, so every attempt must be made to ensure all pages that are being used, are **memory resident**

Working Sets

- While not guaranteed, it is well understood that programs usually reference only a small fraction of their pages during a given “phase”.
 - This idea is called “locality”
 - If 99% of memory accesses over a period of time are in 4 pages - then the process should have 4 frames - no more, no less.

No more, no less

- If we allocate too much - then other programs are deprived frames
- If we allocate too few, we'll "thrash"
 - Thrashing: Frequent Page Faults

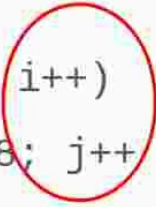
Determining Working Set

Normally it's simply a matter of counting number of pages referenced over time **K**

- **Key Point:** If we cannot satisfy the current working set, we **suspend the process.**
 - Eliminates Thrashing
 - Optimized overall performance

What programmers can do...

```
int data[128][128];  
  
for ( i = 0; i < 128; i++)  
    for ( j = 0; j < 128; j++ )  
        data[i][j] = 0;
```



Page size = 128 integers (1024 bytes)

A simple switch of i
and j can have
damaging effects on
your runtime!