50004 - Operating Systems - Lecture 11

Oliver Killane

23/11/21

Lecture Recording

Lecture recording is available here

Page Replacement Algorithms Continued...

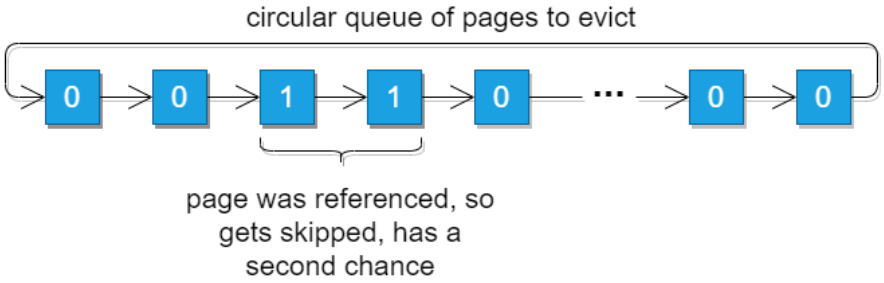
Least Recently Reused Page

Each page has a counter, when referenced, copy clock to counter. When replacing, choose page with lowest counter.
3 frames, 9 page faults

Access:	1	2	3	4	1	2	5	1	2	3	4	5
Page Fault:	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N
Frame 0:	1	1	1	4	4	4	5	→	→	5	5	→
Frame 1:	N	2	2	2	1	1	1	→	→	3	3	→
Frame 2:	N	N	3	3	3	2	2	→	→	2	4	→

As proper **LRU** is expensive (requires a search for lowest counter & to store counters), so we use approximations.

- **Reference Bit**
  - Each page has reference bit r, inbitially $r = 0$.
  - When a page is referenced $r = 1$ (done by **MMU**).
  - Periodically reset bits.
  - When evicting, choose a page with $r = 0$.
- **Second Chance** Evict pages in order, but if $r = 1$ give a second chance to stay in memory.



  - Uses a reference bit r and usesd clock replacement
  - If page to be replaced (in clock order) has $r = 1$, set $r = 0$, leave in memory, search for another page.

Counting algorithms

Work by keeping a counter of the number of references.

- **LFU - Least Frequently Used**
  - Replace page with the smallest count.

  - May replace very new page just brought into memory.
  - Never forgets heavy page usage (reset counters or use ageing)

- **MFU - Most Frequently Used**
  - Replace page with largest count.
  - Newly accessed pages have low count, are prioritised.
  - Pages barely used hog memory & heavily used pages evicted quickly.

Locality of Reference

Programs tend to request the same pages across space & time. We need to design for this to ensure good performance.

If we do optimise for this, it could result in **thrashing**

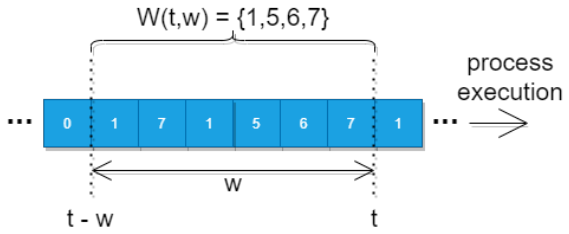
Thrashing

where memory virtual memory is so overused, that there are constant page faults, and paging routines (OS replacing). This destroys performnance.

- Excessive page faults & page replacement causing low CPU utilisation, low performance.
- Program gets slowed by constant access to slow secondary storage.

Working Set Model

A **working set** of pages $W(t, w)$ is a set of pages referenced by a process while running at time interval $t - w$ to t.



We can then use this in our clock replacement algorithm, keeping track of pages in a working set.

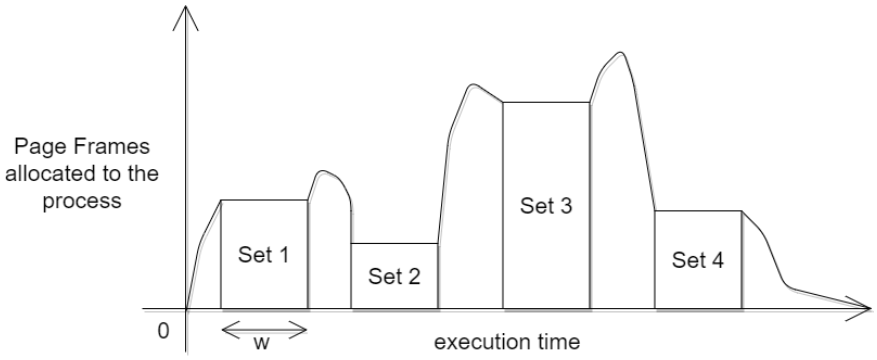
At each page fault:

1. $r = 1$ Set $r = 0$ and move to next page.
2. $r = 0$
   (a) Calculate age.
   (b) If $age < w$ (working set age) continue (page is in working set).
   (c) If $age > w$, if page clean, replace, else write back and replace.

Effectively we only replace pages if they haven't be referenced in w time, so that we can take advantage of temporal locality.

Working Set Size (w)

Processes transition between working sets (different parts of the program use different sets of pages). Hence allocations tend to look like:



Here we do not deallocate pages referenced within w time, at different points in the program the number of allocated page frames differ.

In transitions the number of allocated pages tend to be higher as pages from two sets are referenced within w time.

- Working set too large - too many allocations, process uses too much memory.
- Working set too small - tooo few allocations, lots of page faults, slow.
- Programs transition between working sets (e.g program moves to a new phase, uses different structures).
- Dont want to misallocate - OS allocates for pages the program dosen't actually want anymore.

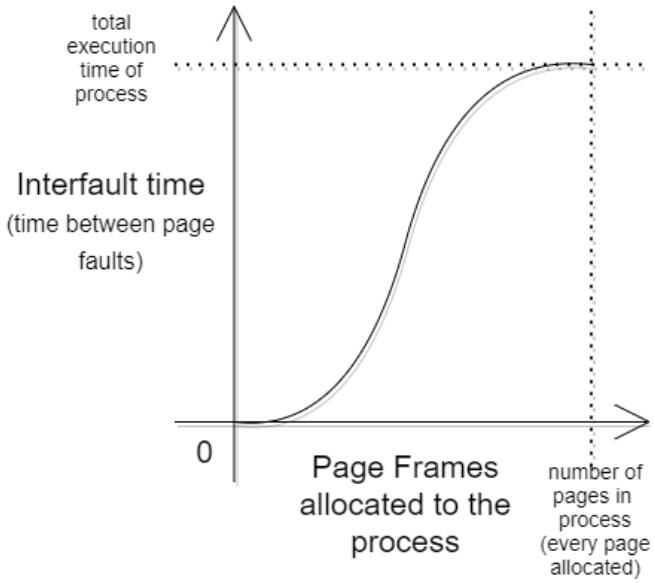
A program can transition between working sets, e.g:

```
1  int main(int argc, char **argv)
2  {
3      /* assume struct big uses memory over several pages. */
4      struct big big1;
5      struct big big2;
6
7      init_big_structure(&big1);
8
9      /* first working set. Using pages associated with big1. */
10     while(some_condition) {
11         /* code using the big1. */
12     }
13
14     /* transition, e.g copying from big1 to big 2, using pages from both. */
15
16     /* second working set. Using pages associated with big2.*/
17     while(some_condition) {
18         /* code using the big2. */
19     }
20 }
```

As we allocate more pages, the number of page faults will decrease (so interfault time will increase). When we allocate all pages in the process there will be no page faults, so interfault time is execution

time.

This generally follows:



Local and Global Page Replacement

- **Local Strategy**
  When replacing a page, a process can choose some page that belongs to the same process.

  Requires some partitioning to determine how many pages each process owns/can be allocated. The most popular techniques are:

  – fixed partitioning - each process gets a fixed size
  – balanced set algorithms

  As processes manage page faults independently, it is more scalable than global (at most considering one process' pages when replacing).

- **Global Strategy**
  Can choose any page from any process.

  – memory is dynamically shared between processes.
  – Initially allocate memory proportional to process size.
  – Use page fault frequency to tune allocation (more page faults → more allocation).

  Uses all pages to determine replacement (slower).

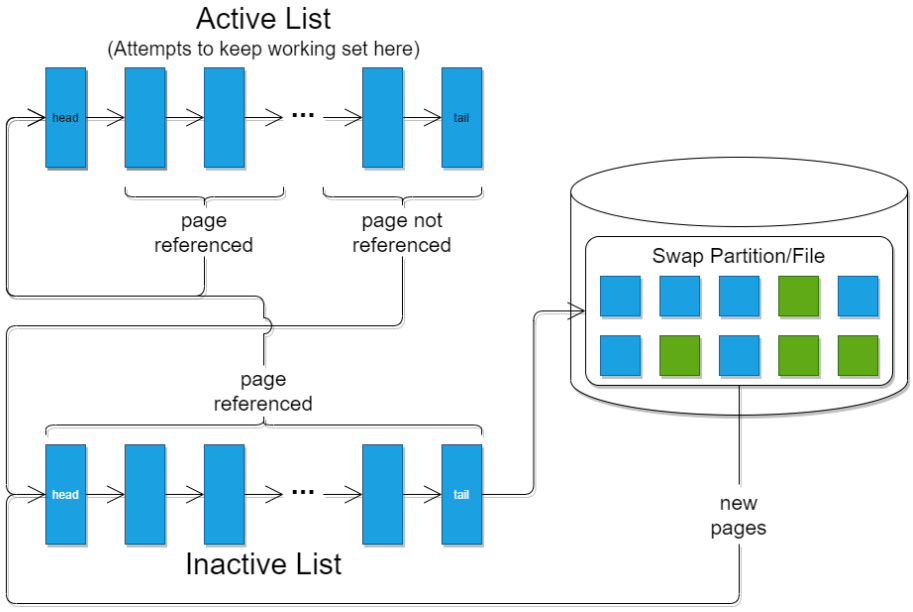
  More efficient overall than local. e.g demanding process's page replaces page of process idling/not using the page.

There is no agreed best solution & it is partially dependent on the scheduling strategy.

- Windows is local
- Linux is global

Linux Page Replacement

Uses a variation of the **clock algorithm** to approximate **LRU** page replacement.



- **kswapd** Swap Daemon
  - Pages in inactive list reclaimed when memory low.
  - Uses dedicated swap partition or file
  - Handles locked & shared pages.
- **pdflush** Kernel Thread
  - Periodically flushes dirty pages to disk from the inactive list.
  - Clean pages can just be freed, no need to write back.
- **Reference bit** Used to determine when pages are moved between active & inactive lists.

Example:

Given 3 empty frames and a reference string of virtual memory accesses:

1 2 1 3 2 1 4 3 1 1 2 4 1 5 6 2 1

Assume demand paging. **LRU replacement policy - 11 page faults**

Access:	1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
Page Fault:	Y	Y	N	Y	N	N	Y	Y	N	N	Y	Y	N	Y	Y	Y	Y
Frame 0:	1	1	→	1	→	→	1	1	→	→	1	1	→	1	1	2	2
Frame 1:	N	2	→	2	→	→	2	3	→	→	3	4	→	4	6	6	6
Frame 2:	N	N	→	3	→	→	4	4	→	→	2	2	→	5	5	5	1

Second chance with clock algorithm - 9 page faults

Access:		1	2	1	3	2	1	4	3	1	1	2	4	1	5	6	2	1
Page Fault:		Y	Y	N	Y	N	N	Y	N	Y	N	Y	N	N	Y	Y	N	Y
Frame 0:	N	$1_{r=1}$	$1_{r=1}$	→	$1_{r=1}$	→	→	$4_{r=1}$	→	$4_{r=1}$	→	$4_{r=0}$	$4_{r=1}$	→	$5_{r=1}$	$5_{r=1}$	$5_{r=1}$	$5_{r=0}$
Frame 1:	N	N	$2_{r=1}$	→	$2_{r=1}$	→	→	$2_{r=0}$	→	$1_{r=1}$	→	$1_{r=0}$	→	$1_{r=1}$	$1_{r=0}$	$6_{r=1}$	$6_{r=1}$	$6_{r=0}$
Frame 2:	N	N	N	→	$3_{r=1}$	→	→	$3_{r=0}$	$3_{r=1}$	$3_{r=1}$	→	$2_{r=1}$	→	→	$2_{r=0}$	$2_{r=0}$	$2_{r=1}$	$1_{r=1}$