Effective Access Time

Page Fault Rate $0 \le p \le 1$

if $p = 0$ no page faults
if $p = 1$, every reference is a fault

Effective Access Time (EAT)

EAT = $(1 - p)$ x memory access
    + p (page fault overhead
    + [swap page out]
    + swap page in
    + restart overhead)

Lets say memory access (including TLB and extra page table accesses) = 20 nsecs
overhead at both ends say 10000 instructions $\approx$ 10000nsecs
swap page in = 4 msecs = 4 000 000 nsecs
50% of time have to swap page out = 2 000 000 nsecs

So EAT = $(1 - p)$ x 20 + p(10000 + 6 000 000)
    $\approx$ 20 + 6 000 000p nsecs

How often do we want page faults?

With

EAT $\approx$ 20 + 6 000 000p nsecs

If we want the EAT to be only half as slow again as real memory we need:

p = 10 / 6 000 000 $\approx$ 0.0000016

So one page fault every 600,000 memory accesses.

Even though the estimates were very approximate we see that we don't want page faults to happen very often.

With page sizes of 4K – 8Kbytes we need lots of frames or lots of repeated access to make the speed acceptable.

Reducing page faults

Different processes have different memory access patterns and therefore different numbers of pages they need to have in memory at one time.

There is a minimum number we must have e.g. with the `add @x, @y` instruction we saw earlier we need at least 10 frames for each process (otherwise this instruction may never complete).

We can allocate frames equally or proportionally (depending on size or priority).

We can set minimum and maximum numbers per process.

We really need the currently required pages in real memory.

Working sets

We talked of the notion of *locality of reference*.

The working set of a process is the collection of pages needed in real memory in order to keep the process running.

If we observe a process running over a short period of time (a window) we can record the page accesses the process makes. This is a picture of the page's working set.

The trick is getting the window the right size:
if it is too small not enough pages are included in the working set
if it is too big too many pages are included

Approximate with interval timer + a reference bit

A reference bit is available in the page table entry in some architectures to indicate the page has been accessed (read or written) since it was cleared.

Example: window = 1,000msec
Timer interrupts after every 500msec.
Keep in memory 2 bits for each page.
Whenever a timer interrupts read and then set the values of all reference bits to 0.
If one of the bits in memory = 1 $\Rightarrow$ page in working set.

PFF

We can also use the Page Fault Frequency to control the number of frames allocated to a process.

As the number of frames increases the number of page faults drops rapidly at first, then there reaches a point where adding more frames hardly alters the rate at which paging occurs. We set upper and lower bounds and add or remove frames to stay within them.

Also see the diagram on pg 431.
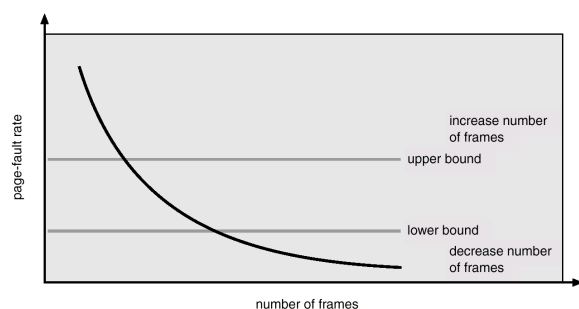
Choosing pages for replacement

When there are no free frames to bring in a page the system has to pick one to replace.

There are two main ways of selecting frames for replacement.

- Global – any frame allocated to any process can be chosen

- Local – chosen frames must come from the processes own allocated frames

There are different consequences for these:

With the global method the number of frames for a process varies depending on its behaviour and the behaviour of the other processes. (The same process can run with widely varying speed due to other processes taking some of its frames.)

With the local scheme there are less frames to choose from.

Normally global replacement is chosen.

We still have to pick

So of all currently occupied frames which one is chosen.

We have some preferences:

pages that are read-only or haven't been modified don't have to be written back to disk (this saves on swapping time)

page table entries commonly have a dirty-bit to indicate the frame has been changed since the page was loaded

pages that aren't going to be accessed again in the near future (so we don't end up with another page fault on the page we just moved out) – unfortunately we can't see into the future so we rely on recent behaviour

if we have a referenced bit we might use this to get an approximation

Selection algorithms

Want lowest page-fault rate.

Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

In all our examples, the reference string is

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

and we have 3 frames.

Random
- it treats every process fairly
- easy to implement
- with enough pages the method won't replace pages just about to be used too frequently

page request	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	=	=	=	=	2	5	=	2	=	=	5
frame 1		2	=	4	=	=	=	=	=	3	=	=
frame 2			3	=	=	=	=	1	=	=	4	=

FIFO

Keep a list of pages in a queue. Remove the one at the head put new ones at the tail.

- simple
- very important pages (such as part of the operating system) which are referenced frequently will be paged out just as frequently as pages which are hardly ever referred to
- Belady's anomaly – increasing the number of frames occasionally increases the number of page faults

page request	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	=	=	4	=	=	5	=	=	=	=	=
frame 1		2	=	=	1	=	=	=	=	3	=	=
frame 2			3	=	=	2	=	=	=	=	4	=

page request	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	=	=	=	=	=	5	=	=	=	4	=
frame 1		2	=	=	=	=	=	1	=	=	=	5
frame 2			3	=	=	=	=	=	2	=	=	=
frame 3				4	=	=	=	=	=	3	=	=

Least Recently Used – LRU

Based on the assumption that a page not used recently will not be used in the near future.

In this example not as good as FIFO – generally better.

Why can't LRU suffer from Belady's anomaly?

page request	1	2	3	4	1	2	5	1	2	3	4	5
frame 0	1	=	=	4	=	=	5	=	=	3	=	=
frame 1		2	=	=	1	=	=	=	=	=	4	=
frame 2			3	=	=	2	=	=	=	=	=	5

Very expensive – need to have hardware that keeps track of last access time for each page.

Or maintain a list of pages and move a page to the top of the list when accessed. Lots of moves.

Use the referenced bit – originally clear, set when the page is used.

Keep regular track (additional reference bits)

Every 100 msecs (say) move the referenced bit into the high bit of a value (say 8 bits), shifting all bits to the right and clear the referenced bit for every page.

e.g.

R: 1   referenced byte: 0 0 0 1 1 1 1 1

becomes

R: 0   referenced byte: 1 0 0 0 1 1 1 1

The pages with the lowest numbers have either been used the longest time ago (or not used as regularly).

Can select randomly from lowest valued or use a FIFO strategy to choose.

Second chance (clock algorithm)

FIFO – but if a page has a 1 in its referenced bit when it is chosen we don't replace it, but clear its referenced bit instead (and change its arrival time to be now).

Commonly implemented as a circular queue – see Figure 9.17.

Least Frequently Used – LFU

- maintain a count of memory accesses for each page

- keep heavily used pages

- Pages can stay around after they are needed – can decrease the count over time.
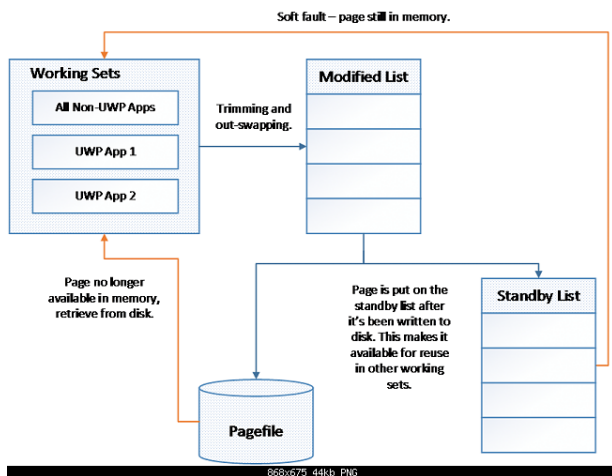
Most Frequently Used – MFU

- Pages with very few accesses may have just been brought in to memory.

Neither is commonly used.

Death Row

- Put frames into a replacement pool according to FIFO selection.

- Keep track of which page is in each frame.

- If a page is accessed while its frame is in the replacement pool then retrieve it.

There is no penalty for paging from disk in this situation.

Windows VMM

Soft fault – page still in memory.

Working Sets
- All Non-UWP Apps
- UWP App 1
- UWP App 2

Trimming and out-swapping.

Modified List

Page no longer available in memory, retrieve from disk.

Page is put on the standby list after it's been written to disk. This makes it available for reuse in other working sets.

Standby List

Pagefile

868x675 44kb PNG

http://www.tenforums.com/windows-10-news/17993-windows-10-memory-compression.html

Windows VMM

The VMM – virtual memory manager runs in the background maintaining memory policies.

It keeps track of the free list of frames and the zeroed list.

Processes have working-set maximums and minimums.

The process is guaranteed its working-set minimum.

If the number of frames allocated is below the maximum the system will allocate it more frames (if it can).

If there aren't enough free frames then working-sets are trimmed to their minimum value.

Default working-set size is 30 – VMM occasionally steals pages to see if the page is in the working-set.

Privileged processes can lock pages in real memory

useful for real-time processes and device drivers

Clustering – when a page is brought in the pages around it are also brought in.

Windows prefetching

When an application is started Windows observes the pages and files referenced in the first 10 seconds.

It keeps track of these and will load all such pages the next time the application is started.

It also defrags these files every few days.

Thrashing

If the sum of the number of pages of the working sets of all processes in the system exceeds the number of frames we are in deep trouble.

This causes thrashing.

Every page fault causes a page from the working set of a process to be removed.

By definition the removed page is going to be accessed soon causing another page fault.

And so on.

It severely affects the amount of work that can be done.

Any process that falls below the number of pages in its working set should be suspended and swapped out (it is not going to get any work done anyway).
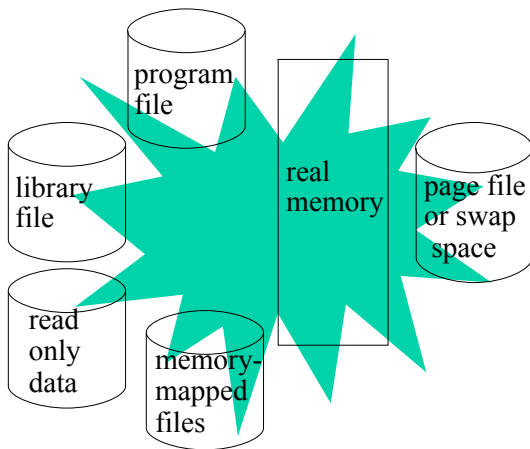
Batch system thrashing

If a batch system is set up to increase the number of programs running in the system at a time if the CPU utilisation gets too small we can get thrashing very easily.

thrashing

CPU utilization

degree of multiprogramming
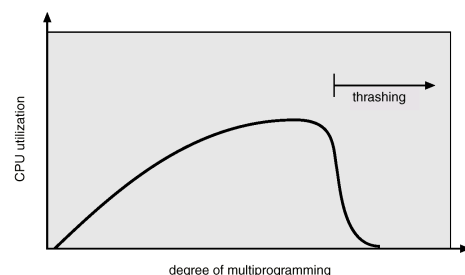
The location of a process' memory

Addressable memory in a UNIX or Windows type process is scattered in different places.

The file pages haven't been loaded yet or have been "paged out".

The diagram gets even more complicated with distributed shared memory and NUMA (non-uniform memory).

Before next time

Read from the textbook
14.1, 14.2 – Goals and Principles of Protection
14.3 – Domain of Protection
14.4 – Access Matrix