

Lecture 6 Virtual Memory

Modern OS does not simply use the paging mechanism to allow programs to load in different places of the main memory.

Instead, a number of interesting ideas are used to boost the performance of the system. We learn these mechanisms in this lecture.

References:

- [OSC] 10.1–10.6 (Virtual Memory).
- [ULK] pp. 216–228 (Page Fault Exception Handler), pp. 456–459 (What is swapping, ..., When to Perform Page Swap-Out).

POS(0230A)

POS(0230A)-6.1

What to do at page fault?

Hardware **exceptions** are handled like **interrupts**:

- The **ISR table is searched** to find the right ISR to call.
- The **memory address accessed** is stored in a register of the CPU.
- The **address of the faulting instruction is pushed** onto the kernel mode stack so that we can return to the faulting code if desired.
- The machine is switched to **kernel mode** and start **running the ISR**.

The question is **what to do in the ISR**. The obvious one:

Terminate the program, since it has accessed invalid memory.

In reality, most OS use the ISR to change the page table, to make the access valid! Why?

POS(0230A)-6.2

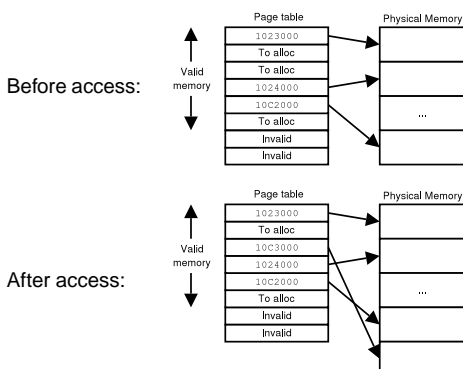
POS(0230A)-6.3

Page fault 1: Delayed allocation

- Many programs allocate a big array, but eventually only access a few entries that it really needs, or access most entries only after a long time.
- If the kernel immediately gives that space to the program, the unused memory are wasted until the program exits.
- One strategy: **don't allocate frame when the page is allocated**. Instead, delay frame allocation until the program **accesses** the page.
- Thus some of the **pages have no corresponding frames**.
- When the program accesses them, a **page fault** results. The operating system allocates the needed frame and restarts the instruction.

One special case of this technique is used for **stack** allocation. We don't know how much stack we need, so we allocate a small one, and enlarge it when a page fault occurs.

Schematics



Actually, the distinction between To alloc and Invalid is not made in the page table (it just store "not present"). Instead, it is made in the memory region descriptor.

POS(0230A)-6.4

POS(0230A)-6.5

Page fault 2: Copy-on-write

We extend the idea of delayed operations to copying: copy-on-write.

- In order to copy a page to another, the **page table entry** is copied, so that the page frame is shared between two or more pages.
- Of course, the user will not expect that writing to one page affect the other. The straight-forward implementation thus won't work.
- Instead, the shared page is configured to be **read-only**.
- When one of the pages is **written**, a page fault occur. The OS allocates a new frame, perform the copy, and restart the faulting instruction.

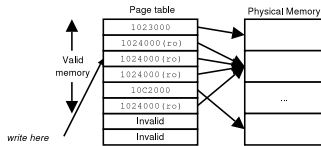
Thus the real copy of frame is delayed until write. This technique is commonly used for duplication the address space on a `fork` system call.

Question: can we use copy-on-write to do delayed allocation?

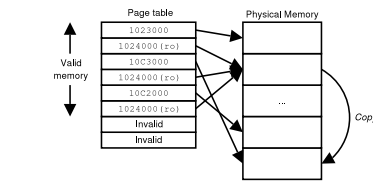
In general, this is not done. Why?

Schematics

Before writing the shared page #2:



After writing the shared page #2:



The OS must store the number of page table entries pointing to each frame, usually in a "frame descriptor". (Why needed?)

POS(0230A)-6.6

Page fault 3: Demanded loading

In the second lecture, we mentioned a system call `mmap()`, which allows **files to be accessed like memory**, i.e., without further system calls.

How this behaviour is achieved? Will we need to load the whole mmapped region to memory? No, instead we play tricks with the page table.

- When `mmap()` is called, the file is not loaded. Instead, the memory descriptor of the process is modified to associate some pages to the file.
- When those memory is **accessed**, **page fault** occurs, and the corresponding blocks in the file is loaded into **frames**.
The file is loaded "on-demand", so the name of the technique.
- When `munmap()` is called (or when process is terminated), the modified frames are optionally **written back** to the file.

Program (and shared library) loading is also done using memory maps.

POS(0230A)-6.7

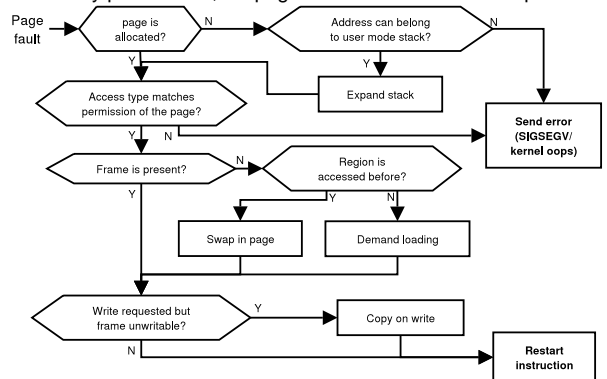
Page fault 4: Swapping

- If we can move files to frames on page fault, why not do the reverse?
- When the OS needs a frame and found that all frames are in use (i.e., there is a **shortage of frames**), it frees a frame by putting its content (the *victim*) to disk.
- If the corresponding page is later **accessed**, a page fault is generated, and another frame is allocated again to hold the page.
- The OS must make three types of decisions: **when** to swap something out, **how** to **select a victim**, and **where** to place the victim.
- This is very performance sensitive: wrong decision can **slow down** the system a real lot.
In case of demand loading, each page is loaded at most once. For swapping, a page can be swapped in and out many times.

POS(0230A)-6.8

Summing all: Linux page fault handler

With so many possibilities, the page fault handler is a bit complicated.



POS(0230A)-6.9

Swapping: a quantitative analysis

Typical disk access time is 10ms, while typical memory access time is 60ns if uncached and 5ns if cached.

Thus we are talking about difference of **6 orders** of magnitude. Compare the expected access time for fault rate of 0%, 0.001% and 1%. Let's assume uncached memory (cached memory makes the difference larger).

- 0%: of course, access cost is 60ns.
- 0.001%: $E(t_a) = 0.00001 \times 10ms + 0.99999 \times 60ns = 160ns$.
- 1%: $E(t_a) = 0.01 \times 10ms + 0.99 \times 60ns = 100\mu s$.

Thus the performance is **nearly proportional to the page fault rate**. If servicing page faults take most CPU time, **thrashing** occurred.

It is important to keep page fault rate extremely low.

POS(0230A)-6.10

Why swapping is important?

If swapping is so costly, why we want it anyway?

- Swapping allows programs **larger than the main memory size** to be executed.
- Swapping allows **more programs** to be concurrently running.
- (Most important) Swapping frees some memory of the computer so that **disk cache can be larger**, increasing the overall performance.

But can the performance gain cover the performance loss to page faults?

- Most programs exhibit strong "**locality of reference**". I.e., during a short period of time, the number of different pages accessed is small.
- Thus it is really possible to keep fault rate as low as 10^{-6} to make the cost low enough.

POS(0230A)-6.11

Who should suffer—Optimal Replacement strategy

How the OS makes use of locality of reference? The key to answer: choose the **right victim** to swap out.

I.e., the second question we mentioned earlier.

- This is called **page replacement**: one new page is swapped in, one old page is swapped out.
- Wrong page replacement strategy is costly: if we replace a page that is about to be accessed, we need to swap it back very soon.
- This idea leads us to the **optimal replacement strategy**: to replace the page that **will not be accessed for the longest period of time**.
- The unlucky fact is that **optimal replacement strategy is unimplementable**, since it requires knowledge about the future.
Note the future tense of the criteria.

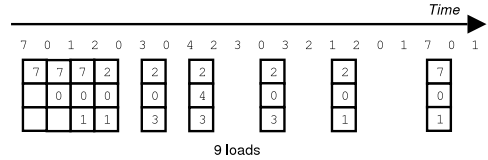
POS(0230A)-6.12

Optimal Replacement: example

We present examples for page-replacement strategy by using a **reference string**, which lists out the pages accessed by the CPU during some time. E.g.,

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

It means page 7 is accessed first, then page 0, then page 1, 2, 0, and so on. Using optimal replacement with 3 frames:



E.g., request 4 (page 2), frames contain page 7, 0 and 1; page 7 will not be accessed for the longest (until request 18), so it is replaced.

POS(0230A)-6.13

FIFO

If we can't do optimal replacement, what we can do?

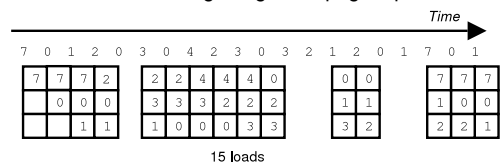
- The OS does know **the time when the frame is allocated**.
Because it is the OS who allocates the frame.
- The **FIFO** (First In First Out) algorithm uses the frame allocation time for choosing the victim.
- The algorithm **always choose the oldest page** to swap out. So the OS keep a **time-ordered queue** of frames of all processes.
- **FIFO algorithm**: The frame at the head of the queue is used to load the new page, and the page originally stored there is evicted.
Rationale: this gives the most time for any loaded page to be accessed. But the problem is that not all page needs the same amount of time in memory.

Advantage: very **simple algorithm**, and requires **no hardware support** other than paging.

POS(0230A)-6.14

FIFO example

Using the same reference string, we get this page replacement events:



This is not as good as the optimal algorithm, requiring 6 more loads.

Note the first 3 loads are always needed, so the numbers of "non-trivial" loads are 6 and 12 respectively. Thus FIFO doubles the miss rate here.

But this is very sensitive to the number of frames available. Try 4 frames.

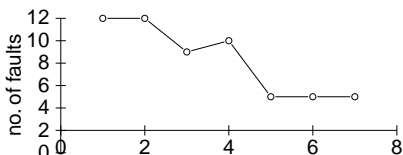
POS(0230A)-6.15

Belady's anomaly

There is a strange behaviour of FIFO: **giving more frames does not necessarily reduce number of faults**—it can **increase** it!

This is called **Belady anomaly**.

Suppose a program needs 5 pages to run, with reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. We plot the number of faults against the number of frames available.



Belady's anomaly is not evil, but it is a good indication that **better replacement algorithm** exists.

POS(0230A)-6.16

LRU

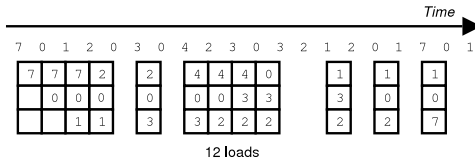
If one is allowed to use the **last access time** of each frame, one can approximate the optimal strategy better by taking locality into account.

- If the program exhibit locality, **the page that will not be used for the longest tends not to be used for the longest in the past** as well.
We use the history to predict the future.
- Algorithm: replace the page that is not used for the longest (i.e., **least recently used**). We call the algorithm **LRU**.
- Note that **this is only a heuristic**: there is no reason why this is the "best" page replacement algorithm.
Try make out a reference string that FIFO performs better!
- In practice **LRU** behaves much better than **FIFO** in most situations.

POS(0230A)-6.17

LRU example

The same reference string for LRU:



This is somewhat in-between the optimal replacement (an unachievable goal) and FIFO (the one needing less info).

POS(0230A)-6.18

Difficulty of LRU

Unluckily, even LRU is **not implementable** in most computers.

- Merely **using** a frame won't invoke the OS: there is no page fault.
- Most computers can't tell you when a page is used. Most do have a **reference bit** in the page table to **mark that the page as used**, though. Still remember the bits in the page table entries that are always 0?
- Kernel mode code is capable to **clear that reference bit** and to **check whether that bit is set** by some process accessing that frame.
- One way to approximate LRU is to **periodically examine all reference bits**, and update the last access time of used frames. The OS would then keep the last access time of each frame.
- But there is an efficiency problem: in each period, the reference bits of **all frames has to be checked** (e.g., 1G system has $1G/4k=256k$ frames).

POS(0230A)-6.19

Second chance and related algorithm

To deal with the efficiency problem, we try to **improve FIFO** instead.

- The **second chance algorithm** organize all pages as a list like FIFO.
- Everytime a swap-out is required, **the head of the list is removed**.
- If its reference bit is set, the OS clear it, and **put the page back into the tail of the list**, and the steps are repeated to find a better victim. Rationale: the page is used recently enough.
- Otherwise, the frame corresponding to the page is chosen as victim.

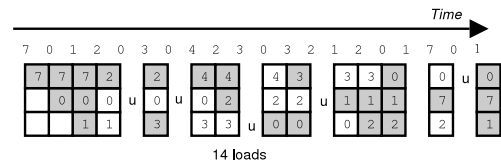
Many computers also have a **modified (dirty) bit** in the page table, set when there is a write to the page.

A clean page do not need to be written back to cache, unless its cache entry is already overwritten. Thus many operating systems modify the second chance algorithm to **favour evicting clean pages**.

POS(0230A)-6.20

Second chance algorithm example

The same reference string for 2nd chance algorithm (shaded = marked, 'u' = marking only):



E.g., request 4 (page 2), next frame to check is 1 (page 7), unmarked, then 2 (page 0), unmarked, then 3 (page 1), unmarked, then 1 again (page 7), evicted.

Request 6: next frame to check is 2 (page 0), it is unmarked, then 3 (page 1), evicted.

POS(0230A)-6.21

When to swap out

- Local strategy: allocate a **fixed number of frames** for each process. Swap out occurs if those frames available to a process are all used. The OS decides how many frames to allocate by the access statistics of each process.

Better for **batched** systems: performance of a process depend only on its own characteristics, not on that of others.

But Windows 2000 use it anyway.

- Global strategy: **consider all pages together**. When page fault occurs requiring a new frame but none is free, the system chooses a used frame from either some process (swap out) or some I/O buffer using second-chance algorithm.

Better for **interactive** systems: frames of inactive processes is given to active processes for better performances.

POS(0230A)-6.22