Principles of Operating Systems

Virtual Memory – Paging to Disk (+ Additional details)

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2024

References

- These slides are adapted from the slides of Renaud Lachaize
- Chapters of *Operating Systems: Three Easy Pieces*
  - Chapter 21: Swapping: Mechanisms
  - Chapter 22: Swapping: Policies

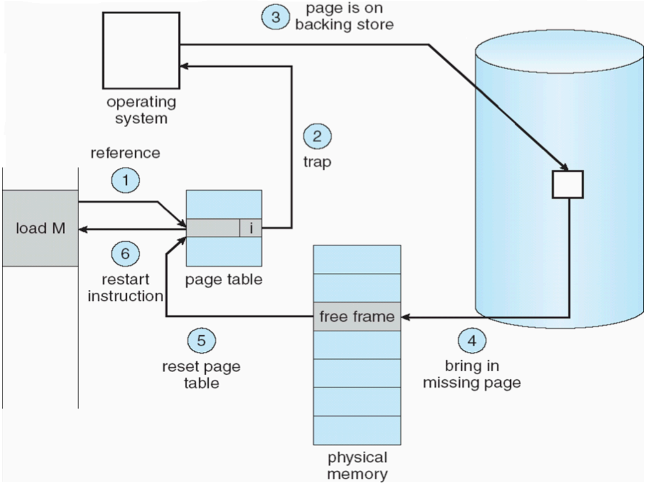
Agenda

- Paging to disk principles
- Choosing what to fetch
- Choosing what to eject
- Further problems
- Memory-mapped files

Paging to disk

- **Motivation**: **use secondary storage (disk) to provide a virtual memory with a larger capacity than the physical memory**
- The RAM acts like a cache for the disk

Paging to disk (continued)



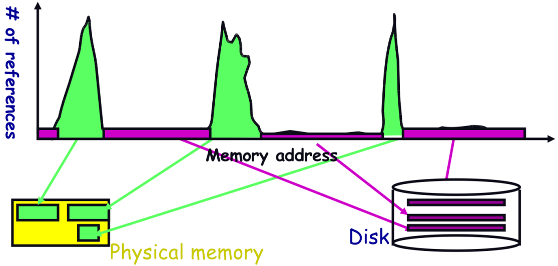
About the eviction of a page

- If the physical memory is full, bringing a new a page to memory requires removing another page
- Two cases to consider:

About the eviction of a page

- If the physical memory is full, bringing a new a page to memory requires removing another page
- Two cases to consider:
  - The page is clean (most recent modifications are already stored on disk)
    - Replace the content with the new page
  - The page is dirty (the only valid copy is in memory)
    - The content needs to be written to disk before replacement (takes more time)
    - It is faster to evict a clean page
- About the evicted page:
  - Mark page as not present
  - Store enough information in the PTE to find the page on Disk
  - Will need to be loaded again next time it is accessed (Page fault)

Working set model



- The disk is much, much slower than memory
  - Goal: run at memory speed, not disk speed
- 90/10 (or 80/20) rule: 10% of memory gets 90% of memory references
  - So, keep that 10% in real memory, the other 90% on disk
  - How to pick which 10%?

Some challenges

What to fetch?

- Just needed page or more?

What to evict?

What to fetch?

- Bring in page that caused page fault

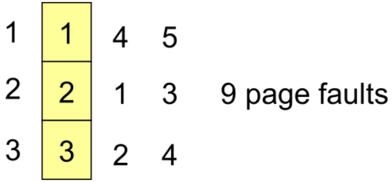
- Pre-fetch surrounding pages?
  - In many cases, reading two disk blocks is approximately as fast as reading one
  - If application exhibits spatial locality, then big win to store and read multiple contiguous pages

- Also, keep a pool of zero-filled pages
  - Frequently required for new pages in process stacks, heaps, and anonymously mmapped memory
  - Zeroing them only on-demand is slower
  - So many OSes zero the free pages while CPU is idle

What to evict? Selecting pages
Straw man: FIFO eviction

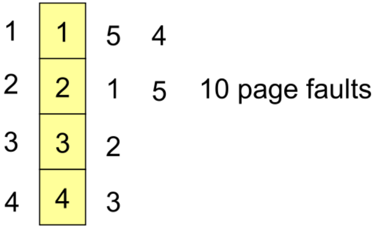
- **Evict oldest page fetched in system**

- Example - consider the following reference string:
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With a capacity of 3 physical pages: 9 page faults

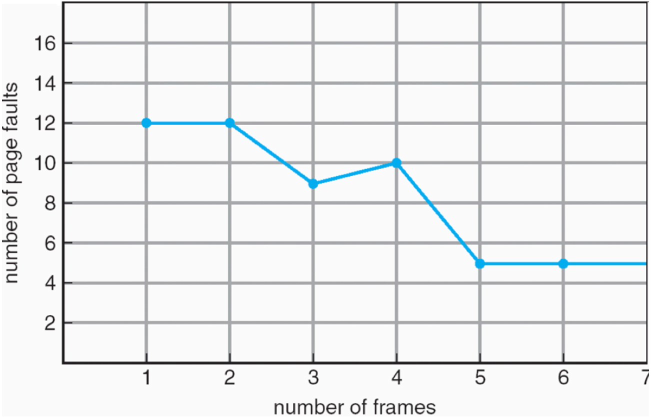


What to evict? Selecting pages
Straw man: FIFO eviction

- Evict oldest page fetched in system
- Example - consider the following reference string:
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With a capacity of 3 physical pages: 9 page faults
- **With a capacity of 4 physical pages: 10 page faults**



Selecting physical pages
Belady’s anomaly

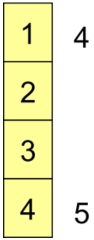


More physical memory does not always mean fewer faults!

Optimal page replacement

- **What is optimal (if you knew the future)?**
  - **Replace page that will not be used for the longest period of time**

- Example – with reference string
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages

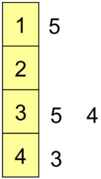


6 page faults

LRU page replacement

- **Approximate optimal with least recently used**
  - **Because past often predicts the future**

- Example – with reference string
  - 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages: 8 page faults



- Problem 1: can be pathologic– example?
  - Looping over memory (then want MRU eviction)

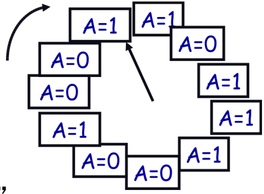
- Problem 2: How to implement?

Straw man LRU implementations

- Idea 1: Stamp PTEs with timer value
  - E.g., using the CPU cycle counter
  - Automatically write value to PTE on each page access
  - (When page selection is needed) Scan page table to find oldest counter value = LRU page
  - Problem: would dramatically increase the memory traffic
- Idea 2: Keep doubly-linked list of pages
  - On access, remove page, place at tail of list
  - Problem: again, very expensive
- **What to do?**
  - **Just approximate LRU, don't try to do it exactly**

Clock algorithm

- **Use "accessed" bit supported by most hardware**
  - E.g., Intel x86 processors will write 1 to "A" bit in PTE on first access
  - Software managed TLBs like MIPS can do the same
- Do FIFO but skip accessed pages

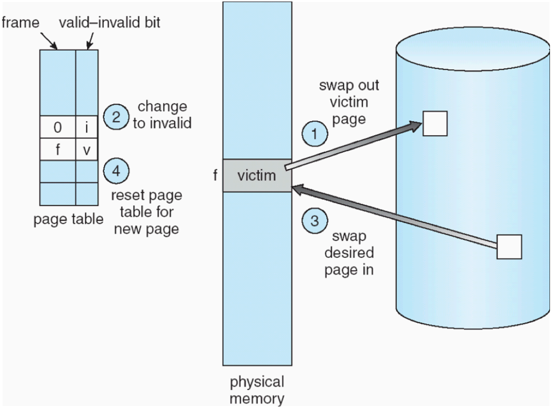


- Keep pages in circular FIFO list
- Scan:
  - If page's "A" bit == 1, set to 0 and skip
  - Else, if "A" == 0, evict
- A.k.a. "second-chance replacement"

Other replacement algorithms

- **Random eviction**
  - Very simple to implement
  - Not overly horrible results (avoids Belady and pathological cases)
- **LFU (least frequently used) eviction**
  - Instead of just "A" bit, count the number of times each page is accessed
  - Least frequently accessed page must not be very useful (or maybe was just brought in and is about to be used)
  - Decay usage counts over time (for pages that fall out of usage)
- **MFU (most frequently used) algorithm**
  - Idea: page with the smallest count was probably just brought in and has yet to be used (so it should not be evicted)
- Neither LFU nor MFU used very commonly

Naïve paging



- Naïve page replacement: 2 disk I/Os per page fault

Page buffering

- Idea: reduce number of I/Os on the critical path

- Keep pool of free page frames
  - On fault, still select victim page to evict
  - But read fetched page into already free page
  - Can resume execution while writing out victim page
  - Then add victim page to free pool

- Can also yank pages back from free pool
  - Contains only clean pages, but may still have data
  - If page fault on page still in free pool, recycle

Thrashing

- Thrashing: processes on system require more memory than it has
  - Each time one page is brought in, another page, whose contents will be soon referenced, is thrown out
  - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - I/O devices at 100% utilization but system not getting much useful work done

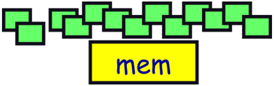
- What we wanted: virtual memory as large as the disk with access time as low as the one of the physical memory
- What we have: memory with access time of the disk ☹

Reasons for thrashing

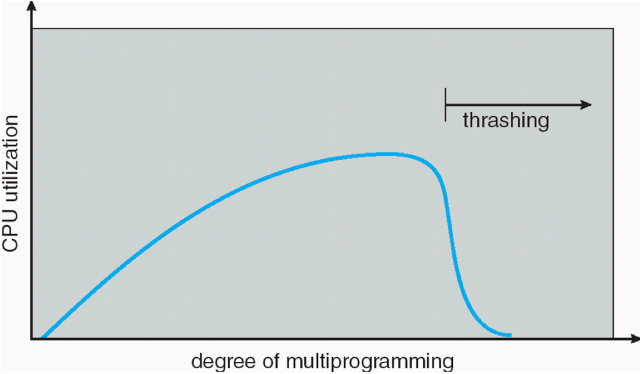
- Process does not reuse memory, so caching does not work (past != future)

- Process does reuse memory, but it does not "fit"

- Individually, all processes fit and reuse memory, but too many for system
  - At least, this case is possible to address (see next slides)



Multiprogramming and thrashing



- Need to shed load when thrashing

Dealing with thrashing

- **Approach 1: working set**
  - Thrashing viewed from a caching perspective: given locality of reference, how big a cache does the process need?
  - Or: how much memory does process need in order to make reasonable progress (its working set size)?
  - Only run processes whose memory requirements can be satisfied

- **Approach 2: page fault frequency (PFF)**
  - Thrashing viewed as poor ratio of "page fetch" to "useful work"
  - PFF = page faults / instructions executed
  - If PFF rises above threshold, process needs more memory. If not enough memory on the system, swap out.
  - If PFF sinks below threshold, memory can be taken away

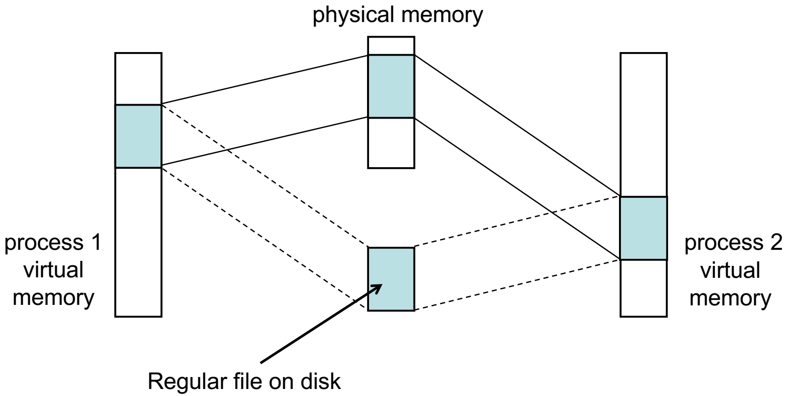
Memory-mapped files

- **Key idea**: associate an address range within an address space (a.k.a. *"memory area"/"region"/"zone"*, and sometimes *"segment"*) with the contents of a "backing" file (or a portion of a backing file)
- **Useful**
  - For the OS, when building the contents of an address space
  - For the application programmers (makes code simpler and/or more efficient)
  - See details in the next slides
- **Two different kinds of backing files**
  - **Regular (persistent) files**:
    - Initial page bytes come from this file
    - Updated bytes may (or may not, depending on settings) be propagated to the backing file (and become persistent)
  - **Fake file full of zeros, called "demand-zero" or "anonymous"**
    - Does not need to be read from disk
    - Once the page is modified (dirtied), treated like any other page
    - Updates are not persistent

Memory-mapped files (continued)

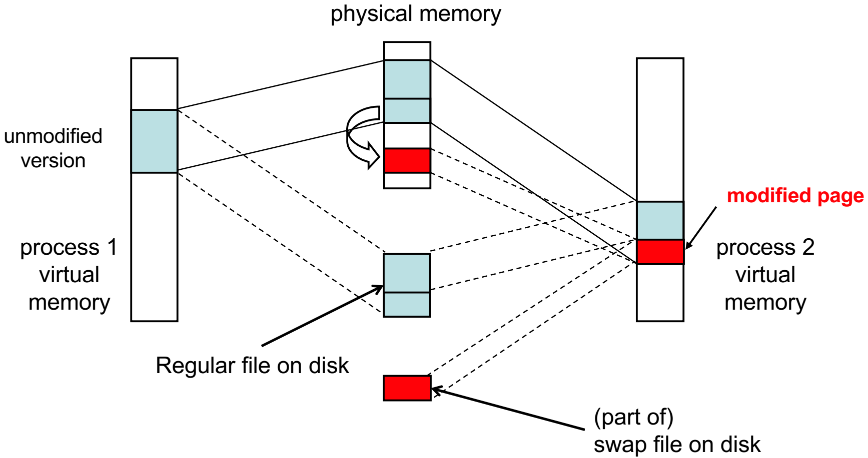
- Different levels of sharing/visibility
  - **Shared mapping**
    - Single copy in physical memory
    - Several processes can share it
    - Updates from a given process are visible by the other processes with the shared mapping
    - Updates are propagated to the backing regular file
  - **Private mapping**
    - Initially, only a single copy in memory
    - When a page is modified, a new page is allocated to store the new version
    - Updates from a given process are not visible by the other processes (with a shared or a private mapping)
    - Updates are <u>not</u> propagated to the backing regular file

Memory-mapped file
Shared mapping



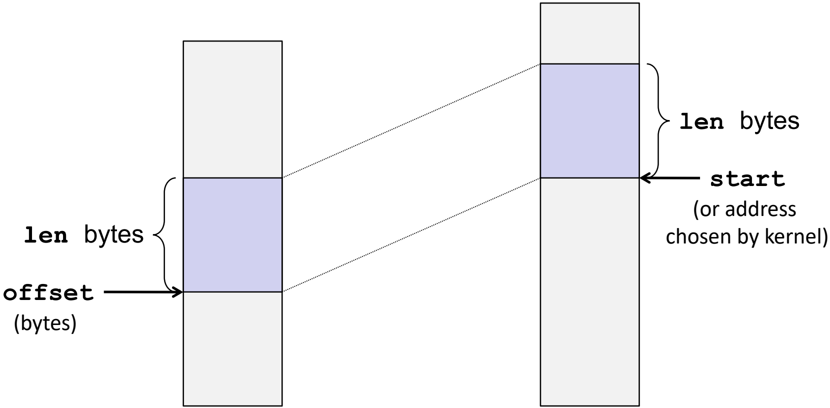
- Notice that different processes can map the file at different addresses

Memory-mapped file
Private mapping



The mmap system call

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Disk file specified by file descriptor **fd**

Process virtual memory

The mmap system call (continued)

```
void *mmap(void *start, int len, int prot, int flags,
           int fd, int offset)
```

- return value: starting address of mapping
  - or **`MAP_FAILED`** if error
- **`fd`**: open file descriptor corresponding to the file to be mapped
- **`start`**: hint for the starting address of the mapping
  - The kernel may choose a different address
  - Typically set to **`NULL`**, to let kernel choose address
- **`len`**: size of the mapping (in bytes)
- **`offset`**: offset relative to the start of the file (in bytes)
- **`prot`**: protection rights (for whole mapped region):
  - **`PROT_READ`**, **`PROT_WRITE`**, **`PROT_EXEC`**, **`PROT_NONE`**
  - Can combine several rights using bitwise OR (e.g., **`PROT_READ | PROT_WRITE`**)
- **`flags`**:
  - **`MAP_PRIVATE`**: private mapping
  - **`MAP_SHARED`**: shared mapping
  - **`MAP_ANONYMOUS`**: anonymous memory (**`fd`** should be -1), i.e. “demand-zero” mapping
    - Option that can be combined (bitwise OR) with either **`MAP_PRIVATE`** or **`MAP_SHARED`**

The mmap system call
Purposes of the various types of memory mappings

Visibility of modifications	*Mapping type*	
	File	**Anonymous**
Private	Initializing memory from contents of file	Memory allocation
Shared	Sharing data between processes or Memory-mapped file I/O (accessing a file without explicit **`read`**/**`write`** calls)	Sharing memory between processes (of the same family)

The mmap system call
Purposes of the various types of memory mappings (cont.)

- **Private-file**: initializing memory from contents of file
  - Example: program/library data (global static variables)
    - Modifications must not be visible from other processes (each process has its own copy)
- **Private-anonymous**
  - Used to allocate new, zero-filled memory region, with private modifications (e.g., memory heap)
- **Shared-file**
  - Memory mapped I/O: e.g., reading and (persistently) modifying a file without having to explicitly use **`read`**/**`write`**/**`fread`**/**`fwrite`** ...
  - (Persistent) shared buffer for data exchange between (arbitrary) processes
- **Shared-anonymous**
  - (Non persistent) shared buffer for data exchange between related processes (e.g., parent-child) – such a mapping can only be transmitted via “family inheritance” (through **`fork`**)

The mmap system call
Details on swapping

- **What happens when a dirty page within a memory mapped region must be swapped out (to disk)?**

- **The location on disk depends on the type of mapping**
  - **File-shared**: update the corresponding (regular) file
  - **File-private**: store the modified page in the swap file
  - **Anonymous-shared**: store the modified page in the swap file
  - **Anonymous-private**: store the modified page in the swap file

- Note:
  - The size of the swap file (on disk) + the total size of the physical memory provide an upper bound on the maximum (global) amount of virtual memory that can be allocated by the OS
  - The swap file is stored on disk (and is thus persistent) but its contents are discarded upon each reboot