Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Operating Systems

Part 1: Virtualization – 5) Memory 3

Revision: `master@d08e360 (20220915-171922)`

BTI1341 / Fall 2022-23

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Outline

Introduction

Remember the unrealistic assumptions from *03-memory-1:*

1. A user address space must be placed *contiguously* in physical memory
2. Its size *must not be too big*, i.e. it is smaller than physical memory
3. Every address space is exactly *the same size*

Address spaces, *address translation*, *dynamic relocation*, and finally *paging*, freed us from assumptions 1 and 3.

We now introduce the mechanism of **swap space**, which allows an address space to be larger than physical memory. We also introduce the **page replacement policies** required for this.

Finally, we close memory virtualization (and Part 1 of the course) with a short look at the Linux replacement policy.

Swap Space

Introducing Swapping

The key points underlying the mechanism of swap space and swapping are:

- Extend the physical memory hierarchy with an additional level of memory space.
- Use that space to temporarily store portions of the address space *currently not in use*.

This additional space should be larger than main memory. For this reason, in general the *hard disk drive* will be used. Clearly, this will be *slower* than normal memory (RAM).

Swap Space

> **Definition**
> The swap space is reserved *disk space* used by the OS to *swap out* pages from memory to it or to *swap in* pages from it to memory. It will be assumed that the OS can read and write *page-sized units* to/from it. This requires the OS to know the disk address of a given page.

Swap space is typically implemented as a dedicated *swap partition* or as a *swap file*. It is assumed, that there is enough disk space for as many pages as required.

☞ *Note that e.g. code sections loaded from an executable file are not required to be swapped out: they can be loaded again from the executable file itself when required.*

Swap Space Example

The figure below shows an example of physical memory (4 pages), with 3 active processes (PIDs 0 to 2), and swap space (8 pages), with some swapped-out pages of the same processes.

Additionally, the swap space contains an empty block and two pages of process 3, which is *completely swapped out* (inactive).

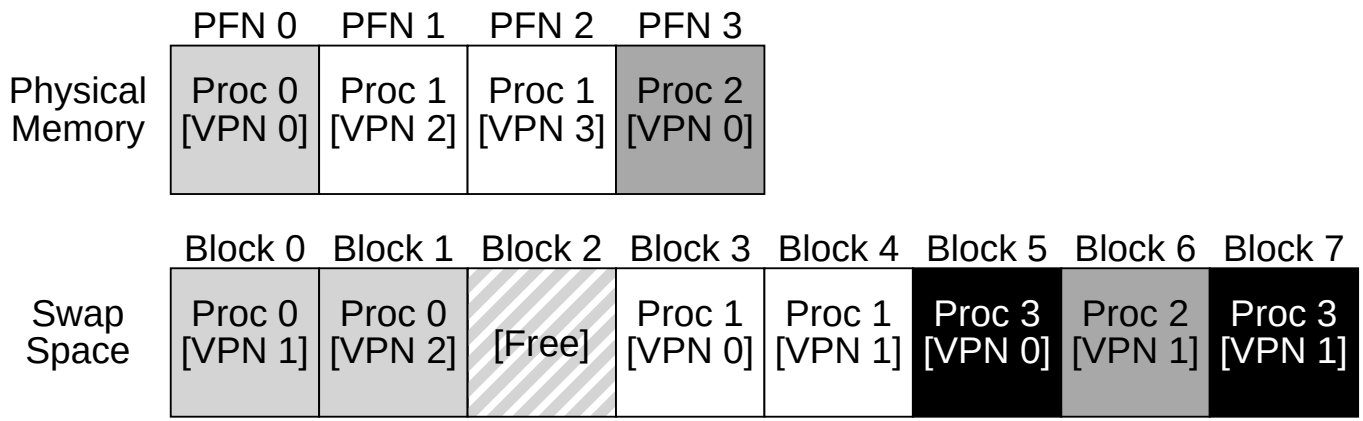

Figure: Physical Memory and Swap Space

Courtesy of [ADAD18]

The Present Bit

Remember from *04-memory-2* how addresses are translated:

1. By using the VPN from the virtual address, the TLB is consulted and hopefully returns a match (TLB hit).
2. If successful, access memory directly.
3. If not, use PTBR to find page table and retrieve the PTE corresponding to the VPN.
   - Then, using PFN from PTE, access memory.

With swap space, it is possible that a given *page is not in memory*, resulting in additional steps to be carried out.

Specifically, the present bit in the *PTE* is used to indicate if a page is in memory or has been swapped out. If it is set, address translation occurs as given above. *If it is not set*, a page fault occurs!

The Page Fault

A page fault occurs when a requested page is not available in memory. The HW (MMU) then delegates the problem to the OS, for different reasons:

- Accessing a disk is complicated
  - Requires driver logic for the disk
  - Potentially for different HW types (e.g. SATA, SCSI)
- Organization of swap space would have to be defined by the HW otherwise.
- Accessing a disk is slow anyway, there is no real benefit of doing it in the MMU.

The mechanism used for the delegation is similar to what we have seen so far: The HW *traps* into the OS, where the page-fault handler takes care of the situation.

Handling the Page Fault

After a page fault has occurred, the page-fault handler proceeds to load the missing page into memory:

1. Using the *PTE*, the *disk address* of the missing page can be determined. As the page is not in memory, the *PFN field* can be used for storing it.
2. The page-fault handler performs the necessary *disk I/O* to load the page from the swap space.
3. It then *updates the PTE's PFN*, this time using the memory address.

While handling the page fault, the process is in *blocked* state, the OS is thus free to schedule other processes in the mean time.

If there is *no free space available* in memory, some other page must first be evicted (i.e. swapped out). Details will be given in Section "Replacement Policies".

Page Fault Algorithm (Hardware)

The following pseudocode shows the MMU+TLB addressing algorithm with paging:

```
VPN = (VirtualAddress & VPN_MASK) >> OFFSET_SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits) == True)
    Offset = VirtualAddress & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << OFFSET_SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)

else    // TLB Miss
  PTEAddr = PTBR + (VPN * sizeof(PTE))
  PTE = AccessMemory(PTEAddr)

  if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
  else
    if (CanAccess(PTE.ProtectBits) == False)
      RaiseException(PROTECTION_FAULT)
    else if (PTE.Present == True)
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
    else if (PTE.Present == False)
      RaiseException(PAGE_FAULT)
```

Page Fault Algorithm (Software)

The following pseudocode shows the algorithm for the page-fault handler:

```
PFN = FindFreePhysicalPage()

if (PFN == -1)          // no free page found
  PFN = EvictPage()     // run replacement algorithm

// update PTE
DiskRead(PTE.DiskAddr, PFN)
PTE.Present = True
PTE.PFN = PFN

RetryInstruction()
```

Optimizations

It is not efficient to perform swapping on a page-per-page basis. Instead, often there is some OS thread, which:

1. Runs from time to time and checks the amount of free pages available.
2. If it is less than a specific minimum, it swaps out pages until reaching the minimum amount again.

This enables *clustering of page swaps*: more disk reads/writes can be executed together, which in turn helps to obtain better overall swapping performance.

Replacement Policies

Introduction

If there would always be a free page available, swapping would be trivial. However, due to memory pressure in a busy OS, decisions about *which page to evict* have to be taken (c.f. `EvictPage()` from Slide 12).

As we have seen before with strategies for free-space management (e.g. first fit, best fit, ...), various replacement policies guiding those decisions are conceivable. We will now have a look at:

- The optimal policy
- FIFO
- Random
- Least Recently/Frequently Used (LRU/LFU)
- Clock

Cache Management

Main memory can be thought of as a *cache for pages* (holding a subset of all pages). A good replacement policy thus tries to maximize cache hits (or minimize cache misses).

To see, why this is important, consider the average memory access time :[1] $\text{AMAT} = T_{\text{M}} + (P_{\text{Miss}} \cdot T_{\text{D}})$

Assuming 100ns for T_{M} and 10ms for T_{D}:

- A hit rate of 90% ($P_{\text{Miss}} = 0.1$) leads to 1.0001ms AMAT
- A hit rate of 99.9% ($P_{\text{Miss}} = 0.001$) leads to 10.1μs AMAT

Clearly, already a tiny miss rate quickly becomes the dominant factor for AMAT and must be avoided.

[1] T_{M} and T_{D} are the times required for memory and disk access, P_{Miss} the probability of a cache miss.

Optimal Policy

The *optimal* replacement policy is straightforward:

Replace the page that will be accessed *furthest in the future*.
This works, as all other pages will be accessed earlier and can thus be considered more important.

As we have seen for scheduling policies, the future is not known. In practice it *cannot be determined* in advance, which page will be the one to be accessed furthest in the future. Thus, we use this policy only for comparison.

Optimal Policy Example

Assume a cache of size 3 and the following sequence of page accesses: 0,1,2,0,1,3,0,3,1,2,1

Access	Hit/Miss	Evict	Cache
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	2	0,1,3
0	Hit		0,1,3
3	Hit		0,1,3
1	Hit		0,1,3
2	Miss	3	0,1,2
1	Hit		0,1,2

Optimal Policy Example (cont.)

Hit rate: $\frac{6}{11} = 54.5\%$

Looking at the example trace, we see that the first three misses cannot be avoided (the cache needs to be filled initially). These are called cold-start misses or compulsory misses .

Independent of the cache state, the same also applies whenever a page is accessed for the *first time*. In the example, this is the case for the first access to page 3.

Hit rate ignoring compulsory misses: $\frac{6}{7} = 85.7\%$

FIFO Policy

A simple policy is the FIFO policy : Pages are evicted in the order they where loaded; the first page loaded is the first to be evicted:

Access	Hit/Miss	Evict	Cache
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		0,1,2
1	Hit		0,1,2
3	Miss	0	1,2,3
0	Miss	1	2,3,0
3	Hit		2,3,0
1	Miss	2	3,0,1
2	Miss	3	0,1,2
1	Hit		0,1,2

Hit rate: $\frac{4}{11} = 36.4\%$, or $\frac{4}{7} = 57.1\%$ without compulsory misses.

Random Policy

The random policy simply choses the page to evict randomly. Its performance entirely depends on the random numbers drawn and the sequence of pages accessed. The following figure shows 10'000 trials for the example sequence used so far:

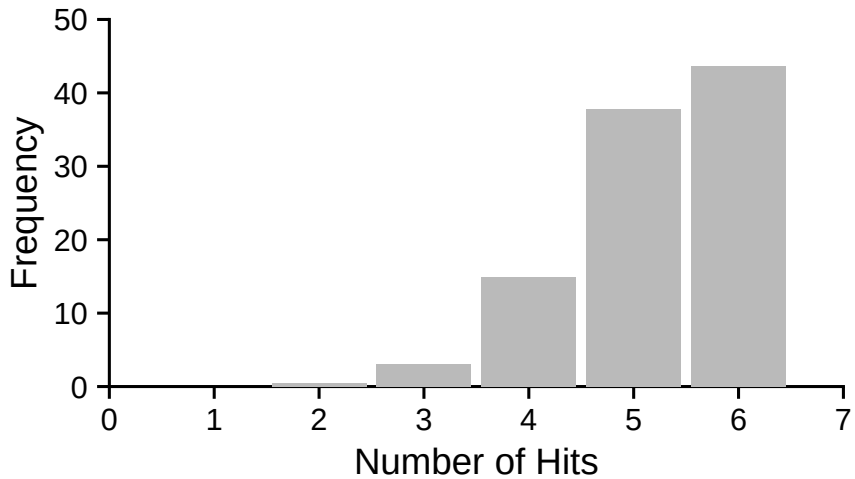

Figure: Histogram for 10'000 random trials

Courtesy of [ADAD18]

Random performs quite well: Over 40% of the time, it is as good as optimal (6 hits).

Using History to Decide

There is a problem with simple policies like FIFO or random: An *important, i.e. frequently used page* might get evicted (e.g. a page containing code).

The solution here is, as for scheduling policies, to base the decisions on the *past behavior* observed. Possible options:

Frequency: How many times has a page been accessed?

Recentness: When was the last time a page has been accessed?

Again, due to *locality of reference*, it is likely that pages which have been accessed recently or often, will be accessed again in near future.

Least-Recently Used (LRU) Policy

The two related policies Least-Frequently-Used (LFU) and Least-Recently-Used (LRU) exploit locality of reference. The following is the example for LRU:

Access	Hit/Miss	Evict	Cache
0	Miss		0
1	Miss		0,1
2	Miss		0,1,2
0	Hit		1,2,0
1	Hit		2,0,1
3	Miss	2	0,1,3
0	Hit		1,3,0
3	Hit		1,0,3
1	Hit		0,3,1
2	Miss	0	3,1,2
1	Hit		3,2,1

Hit rate: $\frac{6}{11} = 54.5\%$, or $\frac{6}{7} = 85.7\%$ without compulsory misses.

Clock Policy

In practice, implementing LRU/LFU is *impossible*, as it requires tracking the least frequently accessed page on *every* page access.

The clock policy is an approximation for LRU, requiring some *HW support*:

- There is a use bit or reference bit *per page*, e.g. in the PTE.
- When a page is accessed, the *HW sets it to 1*.
- To evict a page, the OS searches all bits clockwise:
  - If a *bit is 1*, the page was recently used. The *OS clears the bit* and continues searching.
  - It a *bit is 0*, the page was not recently used and *can be evicted* (or all pages where scanned and cleared).

In addition, often the dirty bit is used to avoid evicting pages which need to be written back to disk first (performance).

More Policies and Thrashing

Virtual memory subsystems not only use policies for page replacement, but also to decide when to bring in pages to memory or when to write them out to disk.

Demand paging simply loads a page when it is accessed, while prefetching tries to load pages in advance, which *may soon be accessed*. Similar decisions can be made for writing pages to disk, c.f. Slide 13.

Finally, the situation can occur that memory is oversubscribed as the running processes simply require *more memory than available*. This is referred to as thrashing , which is difficult to handle. Either, some processes are simply killed, or some form of admission control takes place, preventing certain processes to be run for the moment.

Workload Examples

Without Locality

This workload accesses 100 unique pages 10'000 times, choosing the next page at random. The cache size is varied from 1 to 100. It can be seen, that without locality, all (except optimal) perform the same. Clearly also, when the cache is large enough, all achieve 100% hit rate.

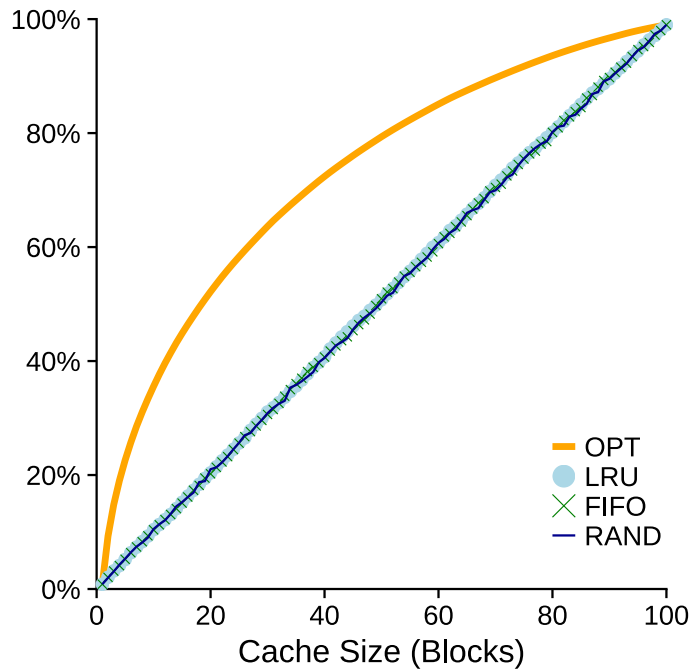

Figure: Workload Without Locality

Courtesy of [ADAD18]

80-20

In this workload, again 100 unique pages are accessed: 80% of the accesses are for 20% of the pages, while 20% of the accesses for the remaining 80%. The history information LRU uses improves its performance compared to random and FIFO.

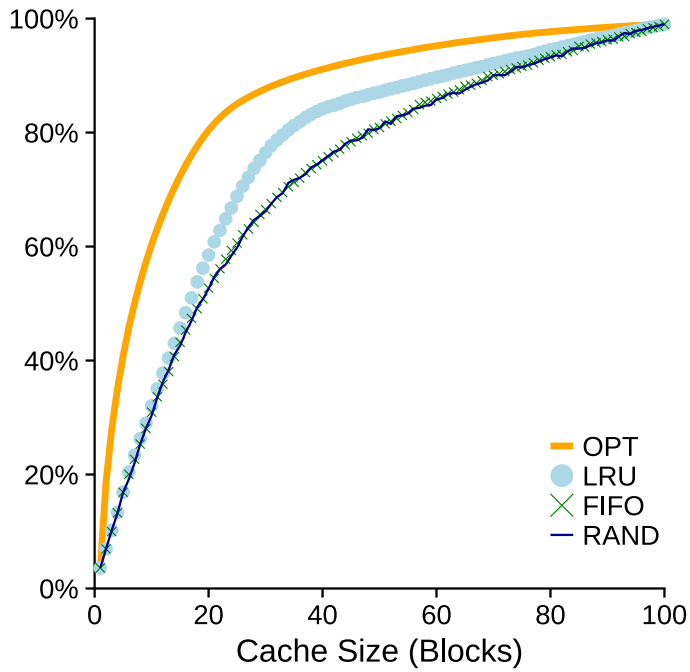

Figure: 80-20 Workload

Courtesy of [ADAD18]

80-20 with Clock

This is again the 80-20 workload, but with clock policy for comparison. The implementation used for the clock algorithm randomly scans pages when searching one with a use bit set to 0.

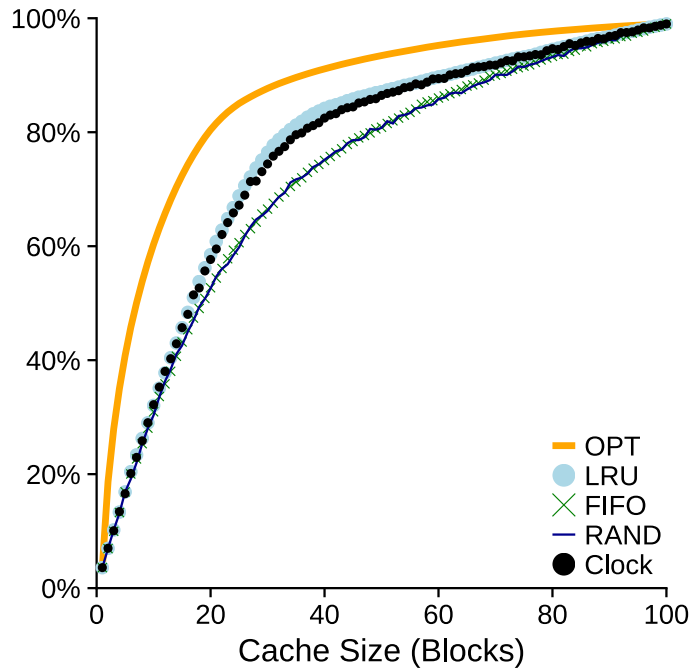

Figure: 80-20 Workload with Clock

Courtesy of [ADAD18]

Looping Sequential

This workload repeatedly loops over 50 pages in sequence (0,1,...,49), also accessing a total of 10'000 pages. It is a worst-case scenario for LRU and FIFO; random performs a bit better as it has no inherent structure.

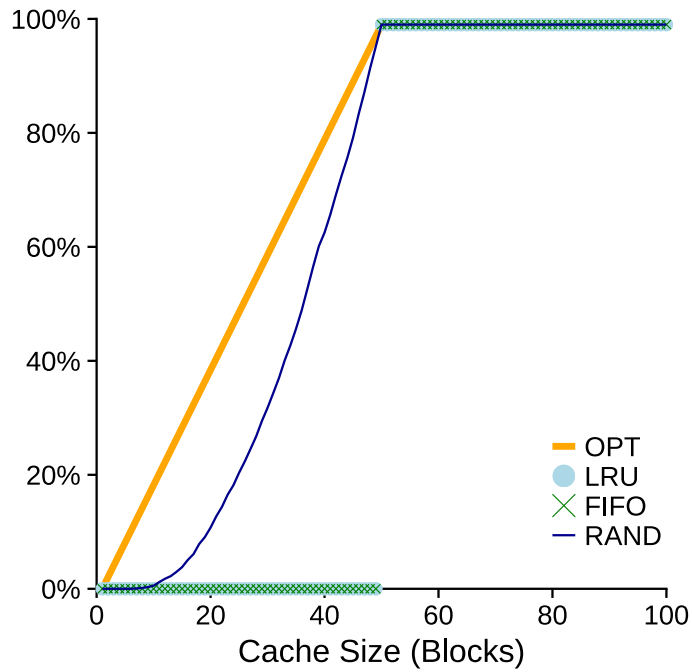

Figure: Looping Sequential Workload

Courtesy of [ADAD18]

Linux Replacement Policy

Linux Replacement Policy

Linux uses a modified version of *2Q Replacement* ([JS94]):

- On first access, a page is placed in the *inactive list*.
- If it is accessed again, it is placed in the *active list*.
- When replacement of pages is required, pages from the inactive list are selected.
- Periodically, pages are moved from the active to the inactive list, keeping the active list to $\sim \frac{2}{3}$ of the page cache.
- An LRU approximation similar to the clock algorithm is used.

2Q specifically fixes an issue when sequentially accessing large files (requiring lots of pages, all of which are then accessed only once).

Appendix

Bibliography

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

[JS94] Theodore Johnson and Dennis Shasha, *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm*, Proceedings of the 20th International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '94, Morgan Kaufmann Publishers Inc., 1994, p. 439–450.