Computer Systems - Notes Week 5

Ruben Schenk, ruben.schenk@inf.ethz.ch

January 4, 2022

Chapter 9: Demand Paging

9.1 Basic Mechanism

Demand paging uses page faults to exchange virtual pages on demand between physical pages in main memory, and locations on a larger *page file* held on cheaper persistent storage. This is the traditional use of virtual memory: making a small main memory plus a large disk look like a large main memory. Demand paging is *lazy:* it only loads a page into memory when a virtual address in the page has been touched by the processor – a demand pager was sometimes called a lazy swapper for this reason.

The process works roughly as follows:

```
# Algorithm 9.2: Demand paging: page fault handling
    # On a page fault with faulting VPN v_fault:
1:  if there are free physical pages then:
2:      p <- get_new_pfn()
3:  else:
4:      p <- get_victim_pfn()
5:      v_old <- VPN mapped to p
6:      invalidate all TLB entries and page table mappings to p
7:      if p is dirty (modified) then:
8:          write contents of p into v_old's area in storage
9:      end if
10: end if
11: read page v_fault in from disk into physical page p
12: install mapping from v_fault to p
13: return
```

9.2 Paging Performance

The performance of a demand paging systems is critically dependent on how many page faults are generated for a workload. The goal is to minimize these page faults. The critical part of Algorithm 9.2 for performance is what happens in `get_victim_pfn()`, in other word what the *page replacement algorithm* is.

The **page replacement policy** of a demand paging systems is the algorithm which determines which physical page (the *victim page*) will be used when paging a virtual page in from storage.

The average time taken to access memory, over all memory references in a program, is the **Effective Access Time.** Consider a page fault rate p, $0 \leq p \leq 1.0$. If $p = 0$, we have no page faults. Similarly, if $p = 1$, then every memory reference causes a page fault. Then, the Effective Access Time is given by:

$$EAT = ((1 - p) \times m) + (p \times (o + m)),$$

where m is the memory access latency, and o is the paging overhead.

Example: Suppose $m = 50ns$, and on average o is $4ms$. Then the EAT in nanoseconds is $(((1p) \times 50) + (p \times 4'000'050))$. If only one access in 1'000 causes a page fault, i.e. $p = 0.01$, then $EAT = 4\mu s$, and we have a slowdown over main memory of a factor of 80.

A **reference string** is a trace of page-level memory accesses. A given page replacement algorithm can be evaluated relative to a given reference string.

9.3 Page Replacement Policies

```
# Algorithm 97: Optimal page replacement
    # When a victim page is required
1: return the virtual page that will not be referenced again for the longest period of time
```

This algorithm is optimal, in that it minimizes the number of page faults for any given reference string. However, it requires knowing the reference string in advance, which is generally not case.

The following algorithm requires no knowledge at all about the reference string:

```
# Algorithm 9.8: FIFO page replacement
    # Return a new victim page on a page fault
1: inputs:
2:  pq: a FIFO queue of all used PFNs in the system
3: p <- pq.pop-head()
4: pq.push_tail(p)
5: return p
```

Bélády's Anomaly is the behavior exhibited by some replacement algorithms in caching systems (notably demand paging), where increasing the size of the cache actually reduces the hit rate for some reference strings.

Example: FIFO page replacement exhibits Bélády's Anomaly. Consider the following reference string:

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

In a machine with 3 physical pages available, this reference string will result in 9 page faults. With 4 physical pages available, the result is 10-page faults.

```
# Algorithm 9.11: A Least Recently Used (LRU) page replacement implementation
    # Initialization
1: inputs:
2:  S: stack of all physical pages p_i, 0 <= i <= N
3: for all p_i do:
4:  p_i.referenced <- False
5:  S.push(p_i)
6: end for
    # When a page p_r is referenced
7: S.remove(p_r)
8: S.push(p_r)
    # When a victim page is needed
9: return S.remove_from_bottom()
```

LRU is hard to beat performance wise. It can easily be implemented using a stack, as shown above. No stack-based algorithm exhibits Bélády's Anomaly.

```
# Algorithm 9.12: 2nd-chance page replacement using reference bits
    # Initialization
1:  inputs
2:      F: FIFO queue of all phyiscal pages p_i, o <= i <= N
3:  for all p_i do:
4:      p_i.referenced <- False
5:      F.add_to_tail(p_i)
6:  end for
```

```
    # When a physical page p_r is referenced
7:  p_r.referenced <- True
    # When a victim page is needed
8:  repeat:
9:      p_h <- F.remove_from_head()
10:     if p_h.referenced = True then:
11:         p_h.referenced <- False
12:         F.add_to_tail(p_h)
13:     end if
14: until p_h.referenced = False
15: return p_h
```

This algorithm is called "2nd-chance" because each referenced virtual page gets a "second chance" before being evicted.

On x86 machines, the MMU hardware provides a "referenced" bit, in addition to a dirty bit, inside the PTE. On ARM machines, this is not the case. The workaround for lack of hardware reference bits is to mark the virtual page invalid at the point where one would otherwise clear the reference bit. The next time the virtual page is referenced, the OS will take a trap on the reference, and set the bit in software before making the page valid and continuing.

```
# Algorithm 9.13: "Clock" page replacement using reference bits
    # Initialization
1:  for all physical pages p_i, 0 <= i <= N do:
2:      p_i.referenced <- False
3:  end for
4:  Next physical page number n <- 0
    # When physical page p_r is referenced
5:  p_r.referenced <- True
    # When a victim page is needed
6:  while p_n.referenced = True do:
7:      p_n.referenced = False
8:      n <- (n + 1) mod N
9:  end while
10: return p_n
```

This classic clock algorithm is the basis for most modern page replacement policies when detailed reference information is not available. The term "clock" algorithm comes from the visual image of a clock hand (the next pointer `n`) sweeping round through the memory, clearing the referenced bits.

9.4 Allocating Physical Pages Between Processes

In **global physical page allocation,** the OS selects a replacement physical page from the set of all such pages in the system. However, this doesn't quite work: each process needs to have some minimum number of physical pages in order to be runnable at all.

Instead, with **local physical page allocation,** a replacement physical page is allocated from the set of physical pages currently held by the faulting process.

The **working set** $W(t, \tau)$ of a process at time t is the set of virtual pages referenced by the process during the previous time interval $(t - \tau, t)$. The *working set size* $w(t, \tau)$ of a process at time t is the amount $|W(t, \tau)|$. The working set size of a process is usually a good approximation to how many pages of physical memory a process needs to avoid excessively paging pages to and from storage.

The standard approach uses sampling at an interval $\sigma = \tau/K$, where K is an integer. For each page table entry, we keep a hardware-provided "accessed" bit u_0 and $K - 1$ software-maintained "use bits" u_i, $0 < i < K$.

```
# Algorithm 9.17: Estimating the working set by sampling
    # On an interval timer every sigma time unites
1:  WS <- {}
2:  for all page table entries do:
3:      for all use bits u_i, 0 < i < K do:
```

```
4:          u_i <- u_i-1 {Shift all use bits right}
5:      end for
6:      u_0 <- 0
7:      U <- {Logical-or all the use bits together}
8:      if U = 1 then:
9:          WS.add(page)
10:     end if
11: end for
12: return WS {The working set W(t, sigma K)}
```

A paging system is **thrashing** when the total working set significantly exceeds the available physical memory, resulting in almost all application memory accesses triggering a page fault. Performance falls to near zero. Working set estimation provides a basis for how many physical pages to ideally allocate to each process.

Nevertheless, this concept of *virtualization,* which we will see later in other forms, is fundamental to computer science, and is possibly unique to computers. The thing that makes a computer special is its ability to virtualize itself.

The **Filing System** is the functionality in an operating system which provides the abstractions of files in some system-wide namespace.

Remarks:

- The filing system virtualizes the collection of stable storage devices in the system, in the same way that the virtual memory system virtualizes main memory.
- As with any other virtualization function, this is a combination of *multiplexing* (sharing the storage between applications and users), *abstraction* (making the device appear as a more convenient collection of files with consistency properties), and *emulation* (creating this illusion over an arbitrary set of storage devices).

10.1 Access Control

A security **principal** or subject is the entity to which particular access *rights* are ascribed, which grant the ability to access particular objects.

Example: In a file system, for example, the principals can be system users, objects are files, and rights include `read`, `write`, `execute`, etc.

The **protection domain** of a principal is the complete set of objects that the principal has some rights over. The **access control matrix** is a table whose rows correspond to principals, and whose columns correspond to objects. Each element of the matrix is a list of the rights that the corresponding principal has over the corresponding object.

An **access control list (ACL)** is a compact list of non-zero entries for a column of the access control matrix; i.e. an encoding of the set of principals that have any rights over the object, together with what those rights are.

Example: Traditional UNIX access control is a rather simplified form of ACLs, which limits each ACL to exactly 3 principals:

1. A single user, the *owner* of the file.
2. A single group of users. Groups are defined on a system-wide basis.
3. "Everyone else", i.e. the implicit group composed of all users excluding the owner and the named group.

Moreover, for each principal, the ACL defines three rights: `read` (or list, if the file is a directory), `write` (or create a file, for directories), and `execute` (or traverse, if a directory).

10.2 Files

A **file** is an abstraction created by the operating system that corresponds to a set of data items (often bytes), with associated metadata. Files in UNIX are *unstructured:* their contents are simply vectors of bytes.

The **metadata** of a file is additional information about a file which is separate from its actual contents (data), such as its size, file type, access control information, time of creation, time of last update, etc. It also includes, critically, where the file's data is *located* on the storage devices.

10.3 The Namespace

A **filename** is a name which is bound to a file in the namespace implemented by the filing system.

Remarks:

- More than one name can refer to the same file, in general, though some file system forbid this.
- The organization of the file system namespace can vary quite a bit.

10.4 The POSIX Namespace And Directories

What follows is a lengthy description of the API of the UNIX file system, but written from a more abstract perspective, relating it to the ideas we saw back in the Naming chapter, for example. In UNIX, the file system consists of a single, system-wide namespace organized as a directed acyclic graph.

A UNIX **directory** (also called a *folder* on other systems) is a non-lead node in the naming graph. It implements a naming context. Every process has, as part of its context, a *current working directory.* Any path name which does not start with a `/` is resolved starting at this directory. There is a distinguished directory whose name is simple `/`. This is called the *root* of the file system. All path names which start with `/` are resolved relative to this root.

Remark: In the root, both `.` and `..` are bound to the root. In other words, the root is its own parent.

A **symbolic link** or symlink is a name in a directory bound not to a file or directory, but instead to another name.

There is a clear distinction between a file itself, and the directory entry referring to the file – there can be multiple instances of the latter corresponding to the same file.

10.5 Open UNIX Files

An **open file** is an object which represents the context for reading from, writing to, and performing other operations on, a file. An open file is identified in user space by means of an _open file descriptor.

An **access method** defines how the contents of a file are read and written. *Direct access* (or random access) is an access method whereby any part of the file can be accessed by specifying its offset. *Sequential access* is an access method under which the file is read (or written) strictly in sequence from start to end.

In contrast to files which are represented simply as a vector of bytes, **structured files** contain *records* which are, to some extent, understood by the file system.

10.6 Memory-mapped Files

A **memory mapped file** is an open file which is accessed through the virtual memory system.

10.7 Executable Files

An **executable file** is one which the OS can use to create a process.