Chapter 18

Demand Paging

Today, the prevailing view is that if you’re paging to storage, either in a server or a mobile phone, you’ve already lost the game performance-wise.

It is difficult these days to imagine how transformative the idea of demand-paged virtual memory was at the time [KELS62]: it became possible to consider programs whose space requirements were not limited by a machine’s memory. For the first time, a machine could precisely emulate a much more powerful machine (memory-wise).

However, the basic ideas may be returning: persistent main memory (byte-addressable flash, phase-change, memristor, etc.) may be the only option to scale to very large sizes (for power reasons), whereas traditional DRAM is the only thing that can endure lots of read/write cycles. We may end up paging between main memory addresses using \texttt{memcpy}.

The other important reason to cover demand paging in this course is that it is almost the canonical example of caching as a general system design principle. Many of the concepts below carry over into other caching scenarios which crop up again and again in systems.

18.1 Basic mechanism

\textbf{Definition 18.1 (Demand paging).} \textit{Demand paging} uses page faults to exchange virtual pages on demand between physical pages in main memory, and locations in a larger \texttt{page file} held on cheaper persistent storage.

\textbf{Remarks:}

- This is the traditional use of virtual memory: making a small (and expensive) main memory plus a large (and relatively cheaper) disk look like a large main memory.

- Ideally, this larger virtual memory would not be too much slower than a real one of the same size, for a typical workload.

- This is, essentially, caching. However, since the access time to fetch a virtual page from disk is much higher than in a processor cache, and the transfer unit (a page) is much larger than a typical cache line,
software can be much smarter about page replacement (though not as smart as in wide-area network caches).

- Demand paging is lazy: it only loads a page into memory when a virtual address in the page has been touched by the processor (or at least its cache) – a demand pager was sometimes called a lazy swapper for this reason.

- A note on terminology: I’ve removed any reference to frames from this script, but you’ll hear me talk about them. A frame is the same as a physical page (think of it as something that contains virtual page).

The process works roughly as follows, but note that this algorithm assumes that all accesses are actually valid (the check that throws a fatal error on an invalid access is omitted), and we also ignore concurrency issues entirely for now:

**Algorithm 18.2 Demand paging: page fault handling**

On a page fault with faulting VPN $v_{fault}$:

1: if there are free physical pages then
2:   $p \leftarrow \text{get\_new\_pfn}()$
3: else
4:   $p \leftarrow \text{get\_victim\_pfn}()$
5:   $v_{old} \leftarrow$ 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

### 18.2 Paging performance

Remarks:

- The performance of a demand paging system is critically dependent on how many page faults are generated for a workload (see below). The goal is to minimize these page faults.

- The critical part of Algorithm 18.3 for performance is what happens in $\text{get\_victim\_pfn}()$, in other words what the page replacement algorithm is.

**Definition 18.3 (Page replacement policy).** The page replacement policy of a demand paging system is the algorithm which determines which physical page (the victim page) will be used when paging a virtual page in from storage.
Page replacement algorithms matter because the performance of a computer when it is paging can be completely dominated by the overhead of paging.

How do we measure the performance or efficiency of a paging system?

**Definition 18.4 (Effective Access time).** 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 (unlikely, but not unheard of).

Then the Effective Access Time is:

\[
EAT = ((1 - p) \times m) + (p \times (o + m))
\]

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

**Remarks:**

- In practice, \( o \) is the sum of a number of factors: page fault overhead itself, the cost of swapping a dirty virtual page out if required, the cost of swapping the required virtual page in, and the cost of restarting the instruction.

**Example 18.5.** Suppose \( m = 50\)ns, and on average \( o \) is 4ms (these are plausible figures for a disk-based system).

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. \( p = 0.001 \), then \( EAT = 4\)µs, and we have a slowdown over main memory of a factor of 80.

Analyzing the performance of a paging system is a rather under-specified problem. In general, it’s good to fix some things in advance.

**Definition 18.6 (Reference String).** 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.

**Remarks:**

- Reference strings capture the essential part of a workload, without having to look at what a program is actually doing.

- Generally, the best paging replacement policy minimizes the number of page faults for a given reference string.

- Reference strings are useful in analysing page replacement algorithms, as we will see below.

### 18.3 Page replacement policies

What is the optimal page replacement strategy?
Algorithm 18.7 Optimal page replacement

\begin{algorithm}
\begin{algorithmic}
\State \textbf{return} The virtual page that will not be referenced again for the longest period of time.
\end{algorithmic}
\end{algorithm}

Remarks:

- This algorithm is optimal, in that it minimizes the number of page faults for any given reference string. The proof is left as an exercise.

- It requires knowing the reference string in advance, which is generally not the case (except for some deterministic real-time systems). As a result, it’s almost never used, but is useful to provide a baseline for performance comparisons.

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

Algorithm 18.8 FIFO page replacement

\begin{algorithm}
\begin{algorithmic}
\State \textbf{inputs}
\State \textbf{pq}: a FIFO queue of all used PFNs in the system.
\State \textbf{p} ← \textbf{pq.pop}\_head()
\State \textbf{pq.push}\_tail(\textbf{p})
\State \textbf{return} \textbf{p}
\end{algorithmic}
\end{algorithm}

Remarks:

- Intuitively, FIFO is not the best cache replacement policy in most cases (though you might be able to think of a case where it is), but it is simple to implement.

- You might also think, intuitively, that increasing the size of memory (i.e. increasing the number of physical pages available to hold virtual pages) would increase the efficiency of the paging algorithm by decreasing the number of times a victim page had to be evicted. Surprisingly, this is not the case for all page replacement algorithms, and is famously not the case for FIFO.

Definition 18.9 (Bélády’s Anomaly). \textbf{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 18.10. \textit{FIFO page replacement exhibits Bélády’s Anomaly. Consider the following reference string (from the original paper by Bélády, Nelson, and Shedler [BNS69]):}

\begin{equation*}
1 \ 2 \ 3 \ 4 \ 1 \ 2 \ 5 \ 1 \ 2 \ 3 \ 4 \ 5
\end{equation*}
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.

In practice, we need a dynamic, online page replacement algorithm. Consider Least Recently Used:

**Algorithm 18.11** A Least Recently Used (LRU) page replacement implementation

1: **inputs**
2: S: Stack of all physical pages \( p_i \), \( 0 \leq i < N \)
3: for all \( p_i \), do
4: \( p_i \).referenced \( \leftarrow \) 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 Return S.remove_from_bottom()

**Remarks:**

- LRU is hard to beat performance-wise, though not impossible: see e.g. ARC [MM03].

- It can be easily implemented using a stack, as shown above. The general class of algorithms with this property are called *stack algorithms*. No stack-based algorithm exhibits Belády’s Anomaly.

- LRU needs detailed information about page references. The OS sees every page fault, but in general does not get informed about every memory access a program makes. This makes LRU impractical for a general-purpose OS on typical hardware, but it is the usual baseline choice for other types of cache.

- What a modern machine and OS can do is track which virtual pages have been referenced at some unspecified time in the recent past (and which ones have been modified). This is done by setting a flag associated with a page whenever the page is accessed, and then periodically clearing it after reading it.
Algorithm 18.12 2nd-chance page replacement using reference bits

Initialization

1: inputs
2: $F$: FIFO queue of all physical pages $p_i, 0 \leq i < N$
3: for all $p_i$ do
4: $p_i$.referenced $\leftarrow$ False
5: $F$.add_to_tail($p_i$)
6: end for

When a physical page $p_r$ is referenced

7: $p_r$.referenced $\leftarrow$ True

What a victim page is needed

8: repeat
9: $p_h \leftarrow F$.remove_from_head()
10: if $p_h$.referenced = True then
11: $p_h$.referenced $\leftarrow$ False
12: $F$.add_to_tail($p_h$)
13: end if
14: until $p_h$.referenced = False
15: return $p_h$

Remarks:

- On x86 machines, the MMU hardware provides a “referenced” (or “accessed” 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 can set the bit in software before marking the page valid and continuing. This works: you only pay the overhead of a trap for the first reference; after that it’s free.

- Since the x86 hardware keeps the reference bit in the PTE, it is associated with the virtual page not the physical page (which is what you might want), so extra bookkeeping is needed anyway.

- This algorithm is so-called because each referenced virtual page gets a “second chance” before being evicted.

- This is a rough approximation to LRU, but we can get the same result more efficiently with conventional hardware.
18.4. ALLOCATING PHYSICAL PAGES BETWEEN PROCESSES

Algorithm 18.13 “Clock” page replacement using reference bits

*Initialization*
1: for all physical pages $p_i, 0 \leq i < N$ do
2: \hspace{1em} $p_i$.referenced ← False
3: end for
4: Next physical page number $n \leftarrow 0$

*When physical $p_r$ is referenced*
5: $p_r$.referenced ← True

*When a victim page is needed*
6: while $p_n$.referenced = True do
7: \hspace{1em} $p_n$.referenced = False
8: \hspace{1em} $n \leftarrow (n + 1) \mod N$
9: end while
10: return page $p_n$.

Remarks:

- This classic “clock” algorithm is the basis for most modern page replacement policies when detailed reference information is not available (note that in software caches, such as web caches, such reference information is easy to obtain since the software sees all web requests, and so LRU is practical).

- The term “clock” algorithm comes from the visual image of a clock hand (the next pointer $n$) sweeping round through memory clearing the referenced bits.

18.4 Allocating physical pages between processes

All the schemes we’ve seen so far operate on a single pool of physical pages, and don’t differentiate between different processes or applications. What about trying to allocate different physical memory to different programs?

**Definition 18.14** (Global physical page allocation). *In global physical page allocation, the OS selects a replacement physical page from the set of all such pages in the system.*

Remarks:

- This doesn’t quite work: each process needs to have some minimum number of physical pages in order to be runnable at all. This can vary depending on the OS, but it’s never zero, so you need to reserve some memory for each process.

- Also, we need to adjust for physical pages which are shared between multiple processes.
Definition 18.15 (Local physical page allocation). Instead, with *local physical page allocation* a replacement physical page is allocated from the set of physical pages currently held by the faulting process.

Remarks:

- Local physical page allocation doesn’t need to be fixed. The simplest scheme is indeed to allocate an equal share to all processes, but an alternative is to vary the proportion of physical pages dynamically, much as we have seen CPU schedulers vary to priority of a thread or process dynamically.

- Indeed, we can use priorities rather than quantitative measures of process “size” to give some processes more memory. However, this runs the risk of starving an important process which only needs a few physical pages, but suffers disproportionately when one of them is paged out.

- We could adopt a “rate-based” approach: we measure the number of page faults each process takes over some fixed time period, and reassign physical pages from processes with low fault rates to those with high rates.

- The problem with the rate-based approach is that processes with very large memory requirements can end up starving those with small requirements.

- This naturally leads to the question: how can we *define*, and then *measure*, the “size” of a running process?

Definition 18.16 (Working set). 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 $|W(t, \tau)|$.

Remarks:

- 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.

- It goes without saying that $\tau$ should be chosen carefully. However, it is often the case that above some reasonable value, the working set of a given execution typically doesn’t increase with increasing window size $\tau$ beyond a certain point.

- Accurately measuring the working set of a process at any time $t$ requires a complete list of virtual page references since time $t - \tau$. As with LRU page replacement, this information may be impractical to obtain.

The standard approach, listed in the original paper on the Working Set model [Den68], uses sampling at an interval $\sigma = \tau/K$, where $K$ is an integer.
18.4. ALLOCATING PHYSICAL PAGES BETWEEN PROCESSES

For each page table entry we keep a hardware-provided “accessed” bit \( u_0 \) and \( K - 1 \) software-maintained “use bits” \( u_i, 0 < i < K \).

An interval timer is programmed to raise an interrupt every \( \sigma \) time units.

**Algorithm 18.17** Estimating the working set by sampling

```
On an interval timer every \( \sigma \) time units
1: \( WS \leftarrow \{ \} \)
2: for all page table entries do
3:   for all use bits \( u_i, 0 < i < K \) do
4:     \( u_i \leftarrow u_{i-1} \) \{Shift all use bits right\}
5:   end for
6: \( u_0 \leftarrow 0 \)
7: \( U \leftarrow \bigoplus_{i=0}^{K-1} u_i \) \{Logical-or all the use bits together\}
8: if \( U = 1 \) then
9:   \( WS.add( \text{ page } ) \)
10: end if
11: end for
12: return \( WS \) \{The working set \( W(t, \sigma K) \}\)
```

Remarks:

- This principle, while developed for demand paging, is actually a general approach to any caching scenario.

- We find that, for the most part, when a program is allocated more physical pages than its working set, performance improves but not by much.

- When a program is allocated fewer physical pages than its working set, it starts to page excessively.

**Definition 18.18** (Thrashing). A paging system is **thrasing** 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.

Remarks:

- The term “thrashing” comes from old, large hard disk drives. When a process thrashes, almost all the available time is spent paging pages in and out of memory to and from disk, and very little in actually executing the program. The load on the disk drive causes the disk arm to thrash back and forth, causing a lot of noise and vibration.

Working set estimation provides a basis for how many physical pages to ideally allocate to each process. After that, the problem is to come up with a good policy, much as with CPU scheduling. Indeed, as [Den68] points out, there is a close connection between CPU and memory allocation in a demand paging system.
This should not be surprising: the trick that makes a small machine look like a large machine can work because the (apparently) larger machine runs more slowly.

Nevertheless, this concept of \textit{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.

\section*{Bibliography}


