Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 11.1: Hardware- vs. Software-Walked Page Tables

a. What's the difference between the use of hardware-walked page tables and software-walked page tables? How does this relate to TLBs?

Solution:
Hardware-walked page tables differ from software-walked page tables in the way TLB misses are handled.

If an address (vpn, offset) needs to be translated to a physical address, the hardware first checks whether 'vpn' has a valid mapping entry (vpn, pfn) in the TLB. If such an entry is found, the physical frame number 'pfn' is taken from the TLB and address translation finishes (fast path). Otherwise, the page table must be consulted to find the correct mapping.

Hardware-walked page tables *are walked in hardware, that is, the MMU parses the page table to lookup the mapping from 'vpn' to a 'pfn'. With two-level page table, the hardware first consults the page directory and extracts a pointer (physical address) to the page table in charge of the requested virtual page. Then the hardware consults this page table to determine the frame number assigned to the requested page. If either the page directory entry or the page table entry are marked as invalid, a page fault is raised and a software routine must resolve the fault by either installing an appropriate mapping (by loading the required page table or page into a physical memory frame, updating the page directory or table, and restarting the faulting instruction) or by aborting the faulting application (e.g., if the page requested is defined to be inaccessible in this application's address space). If the hardware finds a valid entry in the page table, it updates the TLB and restarts the faulting instruction.*

Software-walked page tables *on the other hand consult a software exception handler for every TLB miss: As soon as the hardware finds no matching entry for a requested page in the TLB, it raises a TLB fault exception. The kernel routine handling this fault must then walk the page table in software and load the mapping found there into the TLB (using privileged instructions). If no (valid) mapping for the requested page is found in the page table, the TLB miss handler calls the page fault handler routine to resolve this (again either by installing a mapping or by terminating the application).*

b. What difference can you make out in the contents of software-walked vs. hardware-walked multi-level page tables?

Solution:
Besides the physical address of the next-level pages tables or the desired page (frame), page tables also provide some control bits, indicating whether the physical address is valid, whether the page has been modified, whether it may be written to and/or read from and/or executed, etc.

Although hardware-walked and software-walked page tables need to store the same information, software-walked tables can be structured/defined by the OS designer whereas hardware-walked page tables need to have a fixed layout as requested by the hardware. In the context of page fault handling, software-walked page tables are thus adaptable to the needs of the OS: You are free to put in all the information you need, structured in the best possible way.

Hardware-walked page tables on the other hand somewhat limit the OS designers to what they can store in them. For advanced features such as shared memory pages or copy-on-write, additional structures might be required.

c. Under what circumstances are TLB miss handlers or page fault handlers invoked?

Solution:
When the TLB does not contain a valid mapping for a requested page, a TLB miss exception is raised iff the hardware does not handle TLB misses by itself (i.e., with software-walked page tables; systems with hardware-walked page tables never raise TLB miss exceptions).

Similarly, a page fault handler is invoked if the page table does not contain a valid mapping for the looked up address. With hardware-walked page tables, the MMU performs the lookup and triggers an exception to signal a page fault to the OS. With software-walked page tables, the OS kernel parses the page table and performs a function call to the page fault handler (there are no page fault exceptions on such systems).

In addition to the events above, invalid accesses to mapped pages also raise exceptions: If you try to write to a page which is defined to be read-only, the hardware translation will indicate this: With software-walked page tables, the TLB might contain an entry (vpn, pfn, read-only); when a writing access to 'vpn' occurs, the hardware will signal a TLB write fault (usually a different exception than the TLB miss above), indicating that a mapping was found, but the desired access is forbidden. Similarly, a TLB read or execute fault will be raised if a page is read or executed, for which reading or executing is not allowed.

With hardware-walked page tables, you usually receive a mere page fault exception regardless of whether the hardware could not find a valid mapping at all or whether the desired access type is not allowed. In this case, the page fault handler must inspect the page table and the information about the page fault that the hardware provided to discern the two cases.

Question 11.2: Page Fault Handling

a. Explain the terms demand-paging and pre-paging. What are the respective strengths and weaknesses?

Solution:

Demand-Paging *Loads pages into memory only when a page fault occurs, that is, when the data contained in the page is actually needed. This strategy can lead to a high number of page faults when an application starts, but ensures that only data that will be needed resides in memory.*

Pre-paging *Speculatively loads pages that might be needed in the future, thus potentially reducing the number of page faults. In addition, reading large chunks of data from a hard disk is more efficient than reading only single blocks, so I/O throughput can be improved. It might however happen that pages are loaded that will never be used. In that case, not only the memory occupied by such a page is wasted, but also the time and I/O bandwidth that was used to bring the page into memory.*

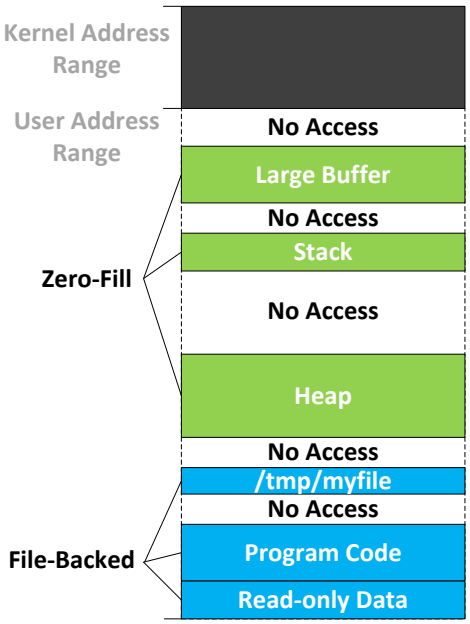
b. When a thread touches a page for the first time with demand-paging, a page fault will occur. Classify the page fault according to where the data for the unmapped page has to be fetched from by the page fault handler.

Solution:

File-backed The contents is stored in a file. This is the case with, for example, the program code (application binary, libraries), initialized data segments as well as memory-mapped files in general. The page fault handler has to read the contents for the page from disk.

Zero-Fill The page belongs to a generic memory area. This includes pages for stacks, heaps, uninitialized data sections and application-specific memory areas (e.g., for large buffers). These pages are zero-filled when first mapped. Zero-filling such pages is desirable as otherwise information from the application that 'owned' the backing frame before might leak to the faulting application (i.e., we would have a covert channel, these should be avoided).

No Access The access might be directed to a page that should not be accessed at all (neither stack nor code nor data nor heap nor . . .). In this case the page fault handler would not grab a frame at all, so there is no contents to fill in.



c. If the process has been running for some time, modifying data along its way, there is one additional case that needs to be covered on a page fault. Which?

Solution:
If the OS at some time decided to steal a frame from the process, and if this frame was modified before (e.g., the frame was occupied by a writable data segment), the frame had to be written to disk somewhere (e.g., swap file or partition), but not back into the program binary from which it might have originated. The page fault handler would then have to allocate a new frame, retrieve the previously swapped out frame from the swap area, and load it into the newly allocated frame.

d. Discuss which information is required by the page fault handler to correctly setup (or restore) the contents of accessed pages.

Solution:
For each page, you need to know:

(a) If it may be accessed at all (i.e., if there should be some data in that page) and how (readable, writable, executable, . . .).

(b) Where the most recent version of the page content can be found (in the binary, in some memory mapped file, on the swap device, or nowhere, as the page is used to hold uninitialized data such as stacks or `.bss` *ELF sections).*

(c) Precise information, where in the binary/memory-mapped file or swap area the frame is to be grabbed from (e.g., some kind of file offset).

e. Can you reuse page table entries to store some of this information? Is it a good idea?

Solution:
Yes, for example, for pages that are swapped out to disk (the valid bit in their page table entry is set to 0), you may store the offset within the swap device in the field usually used for the physical frame number. You might need an additional bit in the page table entry to discern swapped out pages from pages that are to be retrieved from the original file (i.e., if you do not simply write all to-be-replaced frames to the swap area, but recognize unmodified file-backed pages and simply drop them).

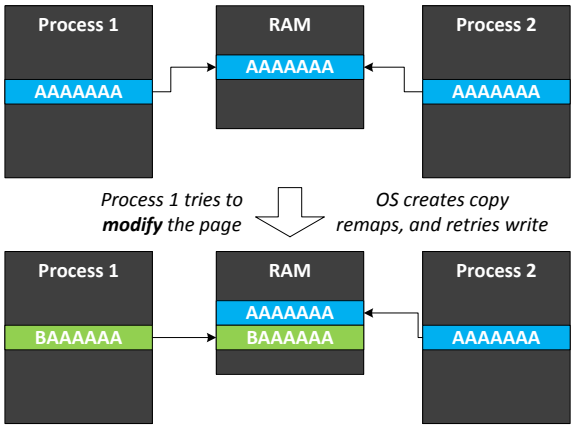
This approach is not possible when using inverted page tables, as these only store information on pages currently mapped to physical memory. Pages that are not resident in memory are not contained in the inverted page table at all.

Reusing the page table entries (PTEs) is a good idea, as you need to access the PTE anyway during page fault handling (at least to write the new mapping information, probably also to check the current access permissions). If you can locate all information to resolve a page fault in the PTE, you benefit from spatial locality (good cache usage, possibly less TLB entries wasted).If you need to parse an additional structure to find the required information, you pollute the cache and TLB, resulting in worse overall system/application performance.

f. What is Copy-on-Write? How can it be implemented?

Solution:
Copy-on-Write is a technique that avoids unnecessary copies of memory pages when a copy of some region of an address space (or the entire address space) is created. The basic idea is that the original region and the copied region can map to the same physical frames as long as they are not modified. Frames are only copied when they are modified. The probably most common example of using copy-on-write is the `fork()` *system call. When a process (the parent) calls* `fork()`*, a child process is created, which is an exact copy of the parent (apart from some exceptions). Instead of copying all the memory that the parent occupies, it suffices to copy its page table and to mark all pages as copy-on-write (this implies that the pages must be marked as read-only) both in the parent and in the child page table. From now on, both parent and child execute in separate address spaces, but both address spaces share the same physical frames.*

As soon as either parent or child tries to write to a page, a page fault will be raised (because all pages were marked as read-only) and the operating system can allocate a new physical frame, copy the contents of the page to which the write access was performed, and adapt the page table of either parent or child (but not both!) to map to the newly allocated page. Both the original page and the newly allocated page can now be granted write permissions, and the faulting instruction can afterwards be restarted. From now on, parent and child have a separate copy of that page.



g. Recap: Describe the steps necessary to handle a page fault in an application's address space.

Solution:
When a page fault occurs, the page fault handler first checks whether an access to the appropriate page is legal or not. If it is not, the application will be terminated if it does not define a handler. If the access is legal, the handler has to find and allocate a free frame. If there is none, the page replacement algorithm is invoked, which chooses a frame whose contents is written to disk and which can afterwards be reused for something else. The page fault handler then has to load the correct contents into the frame: In some cases (e.g., a new frame for a growing stack or heap) it suffices to fill the frame with zeros, in other cases, the correct contents have to be retrieved from disk (e.g., when the page was previously swapped out). Once the frame is filled with the correct data, the page fault handler needs to adapt the page tables by inserting the new mapping and setting the valid bit. In case of a SW-loaded TLB, the new mapping is also loaded into the TLB. The faulting application can now resume its execution by retrying the faulted instruction (note however that it is up to the kernel scheduler to choose the application that runs next).

Question 11.3: Page Replacement Basics

a. The pager of some systems tries to always offer a certain amount of free page frames to improve paging. What is the basic idea behind such a pager?

Solution:
The key idea is that whenever a frame must be replaced with data from a different page, its previous content should not have to be written back to disk first, which would dramatically increase the latency/duration of the page fault handling process. For this purpose, the pager periodically writes back dirty (i.e., modified) pages to disk and marks them as clean.

b. Describe the difference between a global and a local page replacement algorithm. Discuss the advantages and disadvantages of each of them.

Solution:
Global page replacement algorithms operate on all pages from all address spaces currently registered with the system. If an application requests a new page (by means of a page fault), a frame—potentially from a different application—is chosen according to the algorithm's policy, taken from its current owner and assigned to hold the requested page.

Local page replacement algorithms only operate on the frames assigned to the application currently requesting another one (via a page fault), thus each application has a certain number of frames allocated for it and fits its pages into them.

Pros and cons of local replacement strategies:

- \+ *guaranteed number of frames available per application*
- \+ *smaller set of frames to select from, thus potentially faster replacement algorithm*
- – *not easily adapting to changing memory demands of applications*
- – *difficult selection of optimal number of frames per application*

c. Does a virtual memory system implementing equal allocation require a global or a local page replacement policy? Justify your answer.

Solution:
A local page replacement policy to keep the allocated shares equal.

d. What is thrashing? When does it occur?

Solution:
Thrashing refers to an extremely high paging activity of a process (or the entire system). Thrashing occurs when the number of frames allocated to a process is too small even for its "basic" execution. For example, consider a process that requires one code page and one page for the stack. If only one frame is assigned to that process and local page replacement is used, then every access to the stack causes the code page to be written to disk and the stack page to be loaded, and the subsequent code access will cause the stack page to be written to disk an the code to be fetched from disk again. It is obvious that under these circumstances the process will hardly be able to make any progress.

e. What is the working set of a process? How can the working set be used to prevent thrashing?

Solution:
The working set of a process is the set of pages that had been accessed in the last Δ page references. If the operating system can make sure that the sum of the current working sets of all processes P_i in the system is smaller or equal to the total number of available frames, thrashing will not occur. Whenever the operating system detects that the sum of working sets is too large, it can select processes to suspend completely. All frames of a suspended process are written to disk and can be assigned to the other processes.

Question 11.4: Page Replacement Policies

a. A task has four page frames $(0, \ldots, 3)$ allocated to it. The virtual page number of each page frame, the time of the last loading of a page into each page frame, the time of the last access to the page frame, and the referenced (R) and modified (M) bits of each page frame are shown in the following table.

frame	virtual page	load time	access time	referenced	modified
0	2	60	161	0	1
1	1	130	160	0	0
2	0	26	162	1	0
3	3	20	163	1	1

A pagefault to virtual page 4 occurs. Which page frame will have its contents replaced for the *FIFO*, *LRU*, *Clock* and *Optimal* (with respect to the number of page replacements) replacement policies?

For the Clock algorithm assume that the circular buffer is ordered ascending by load time and that the next-frame pointer refers to frame 3.

For the Optimal algorithm use the following string for subsequent references:
4, 0, 0, 0, 2, 4, 2, 1, 0, 3, 2.

Explain the reason in each case.

Solution:

First-In First-Out (FIFO): *Frame 3 will be replaced since it is the frame which has been longest in memory.*

Least Recently Used (LRU): *Frame 1 will be replaced since it is the frame which has not been accessed for the longest time.*

Clock policy: *Clear R in frame 3 (oldest loaded). Clear R in frame 2 (next oldest loaded). Replace frame 0 (third oldest loaded) since it has $R = 0$.*

Optimal: *Frame 3 will be replaced since it is the frame that is accessed furthest away in the future.*

b. Evaluate *stack*, *code*, and *heap* as to how well you expect the LRU page replacement policy to perform on them. Explain your opinion for each segment.

Solution:

- *Stack best, it is used in a LIFO fashion and is dense*
- *Code is second, it is linear with loops and generally follows certain execution patterns*
- *Heap is last, it tends to be used in a more random access fashion than e.g., stacks*