CS2106 Operating Systems

Semester 1 AY2020/21

Tutorial 9 (Solution)

Virtual Memory Management

1. (TLB, paging, and virtual memory)

We have learnt the idea of paging, translation lookaside buffers (TLBs) and virtual memory in the last two lectures. In this question, we are going to put them all in the same scenario and study the interaction.

Here is the basic setup. Note: To make the question easier to comprehend, the scale is vastly reduced compared to real-world setups.

- Page Size = Frame Size = 4KB
- TLB = 2 entries (0..1)
- Page Table = 8 entries (i.e. 8 pages, 0..7)
- Physical Frame = 8 frames (0..7)
- Swap Page (Virtual page in hard disk) = 16 pages (0..15)

Below is a snapshot of process P at time T:

TLB (should have the same fields as the PTE, not shown for simplicity):

Page No	Frame No
3	7
0	4

Page table:

Page No	Frame No/ Swap Page No	Present?	Valid?
0	4	1	1
1	1	1	1
2	1	0	1
3	7	1	1
4	2	1	1
5	15	0	1
6	—	0	0
7	—	0	0

Although the content and layout of the physical memory frames and hard disk swap pages can be deduced from the above, you are encouraged to sketch the RAM and hard disk content to aid understanding.

a. Below is the skeleton of the access algorithm for this setup. Give the missing algorithm for the OS **page fault** handler.

Algorithm for accessing virtual address VA:
1. [HW] VA is decomposed into <Page#, Offset>
2. [HW] Search TLB for <Page#>:
    a. If TLB miss: Check the page table
3. [HW] Is <Page#> memory resident?
    a. Non-memory resident: Trap to OS { **Page Fault** }
4. [HW] Use <Frame#><Offset> to access physical memory.

b. Using (a), walkthrough the following access in sequence. For simplicity, only the page number is given. If TLB or page replacement is needed, pick the entry with the smallest page number.

i. Access page 3.
ii. Access page 1.
iii. Access page 5.

(Optional) Think about how a dynamically-allocated new page fits into this scheme.

c. Suppose we have the following hardware specifications.

- TLB access time = 1ns
- Memory access time = 30ns
- Hard disk access time (per page) = 5ms

Give the best and worst memory access scenarios and the respective memory access latencies.

d. If 2-level paging is used, is there a worse scenario compared to (c)?

Solution

a. **Algorithm for accessing virtual address VA:**

1. [HW] VA is decomposed into <Page#, Offset>
2. [HW] Search TLB for <Page#>:
    a. If TLB miss: Check Page table
3. [HW] Is <Page#> memory resident?
    a. Non-memory resident: Trap to OS { Page Fault }
4. [HW] Use <Frame#><Offset> to access physical memory.

Algorithm for Page Fault Handler, given <Page #>:

1. [OS] Global/Local replacement, Replacement algorithm applicable here
2. [OS] Write out the page to be replaced if needed.
3. [OS] Locate the page in secondary swap pages.
4. [OS] Load the page into a physical frame.
5. [OS] Update page table entries.
6. [OS] Flush TLB if needed.
7. [OS] Return from Trap

b.

i. Access page 3: TLB hit.

ii. Access page 1: TLB miss, memory resident. Hardware will replace TLB entry.

TLB:

Page No	Frame No.
3	7
1	**1**

Page table: (no change)

Page No	Frame No/ Swap Page No	Present?	Valid?
0	4	1	1
1	1	1	1
2	1	0	1
3	7	1	1
4	2	1	1
5	15	0	1
6	—	0	0
7	—	0	0

iii. Access page 5: TLB miss, page fault i.e. page fault handler executed.

TLB:

Page No	Frame No.
3	7
5	**4**

Page table:

Page No	Frame No/ Swap Page No	Present?	Valid?
0	**Any swap page**	**0**	1
1	1	1	1
2	1	0	1
3	7	1	1
4	2	1	1
5	**4**	**1**	1
6	—	0	0
7	—	0	0

(Dynamically allocated new page) Major points:

- Page table update (valid bit)
- Swap page allocation? / Frame allocation? / TLB update?

Consider the possible approaches: e.g. whether we should immediately move the new page into RAM and/or update TLB?

c. Can be easily calculated as overhead + actual memory access. The actual memory access is always 30ns.

   Overhead in best scenario:
   = TLB-Hit
   = 1ns

   Total = 31ns

   Minimal overhead in worst scenario:
   = TLB access + Full table access + Service page fault (Write out and Write in)
   = 1ns + 30ns + 5ms + 5ms
   = 10,000,031ns (10.000,031ms)

   Note that the above is a crude approximation for the overhead. The major component is the two disk accesses for swapping in/out the memory page, which dwarfs all other overheads. Also, the above ignores the overhead of re-accessing TLB and/or page table. So, it should be taken as the minimal overhead.

d. Since the smaller page tables are stored in separate pages. The even worse scenario is when a particular smaller page table is needed, it is not in physical memory, i.e. page fault. Hence, there is a possibility of servicing two page faults under the 2-level paging scheme.

2. (Page table structure)
Given the following information:

- Virtual Memory Address Space = 32 bits
- Physical memory = 512MB
- Page Size = 4 KB
- PTE Size = 4 bytes

Suppose there are 4 processes, each using 512MB virtual memory. Answer the following:

a. Direct paging is used. What is the total space needed for all page tables used?
b. 2-level paging is used. What is the total space needed for this scheme?
c. Inverted table is used. What is the total space needed? You can assume each inverted table entry takes 8 bytes.

Why do you think the inverted table entry takes up more space than a page table entry?

Solution

a. Number of page table entries
= Total Memory / Page Size
= $2^{32} / 2^{12}$
= 2^{20}

Size of page table
= # of page table entries * size of page table entries
= $2^{20} * 2^{2}$
= 2^{22} (4 MB)

Each process has its own page table, so total space needed
= Page table size * # of processes
= $2^{22} * 2^{2}$
= 2^{24} (16MB)

b. The page table will be split into smaller page tables, each fitting into a page.
Number of PTE per smaller page table
= Page Size / PTE size
= $2^{12} / 2^{2}$
= 2^{10} entries per smaller page table
Total Number of page table entries (PTE)
= Total Memory / Page Size
= $2^{32} / 2^{12}$
= 2^{20} page table entries
Total number of smaller page tables
= Total number of PTE / Number of PTE per smaller page table
= $2^{20} / 2^{10}$
= 2^{10} smaller page tables

Assuming that the same size of entries is used in the page directory,

Size of page directory
= Number of smaller page tables * Size of PTE
= $2^{10} * 2^{2}$
= 2^{12} bytes

A process uses 512 MB of memory.

Number of pages used by process
= Memory used / Size of page
= $2^{29} / 2^{12}$ (512MB / 4 KB)
= 2^{17} pages

Number of smaller page tables used
= Number of pages used / entries per smaller page table
= $2^{17} / 2^{10}$
= 2^{7} (128) smaller page tables

Overhead for a single process is equal to 1 page directory + 128 page table portions

Overhead for a single process
= Size of page directory +
size of smaller page table (page size) * number of smaller page table
= $2^{12} + 2^{12} * 2^{7}$
= $2^{12} + 2^{19}$
= 4KB + 512 KB
= 516 KB

Total Overhead
= 516 KB * 4 processes
= 2064 KB

c. Number of physical frames is = 2^{29} (512 MB) / 2^{12} = 2^{17} frames
A single frame table is used system wide, overhead = $2^{17} * 8$ B = 2^{20} = 1 MB

Q: Why do you think the inverted table entry takes up more space than a page table entry?
A: The IPT entry needs to record the PID in addition to the logical/virtual address (as two processes could have different physical pages mapped to the same logical/virtual address).

3. (Adapted from final exam, AY 2018/19 S 1)

Below is a snapshot of the memory frames in RAM on a system with virtual memory. The memory pages in the frame is shown as <Process><Page#>, e.g. B17 means Page 17 of Process B.

RAM

Frame #	Contents
0	B17
1	A31
2	A04
3	A17

Victim ➔

OS

Ref.
1
1
0
1

The OS maintains additional information shown on the right to perform second chance page replacement algorithm on the memory frames.

Suppose **process A** accesses **page 8** (i.e. **A08**) now.

a. Which memory frame will be replaced?
b. Give the steps the OS takes to update the affected page table entries (without an inverted page table).

Continuing from (a), suppose **process B** accesses **page 13** (i.e. **B13**) now.

c. Which memory frame will be replaced?
d. If the OS keeps an inverted page table, give the contents of the inverted page table after the replacements (after (a) and (c)).
e. Give the steps the OS takes to update the affected page table entries with the help of the inverted page table.

Solution

a. Frame 2 is replaced (it is the first 0 encountered in the second chance algorithm).

b. Steps taken
    1. Identify the process that owns the page in frame 2
        a. Since there is NO inverted page table
            i. Scan through every process's page table
            ii. Find the page that is memory resident and located in frame 2
    2. Mark Process A's page 4 as **not memory resident**
    3. Mark Process B's page 8 as **memory resident**

c. Frame 0 is replaced (circular wraparound to the top).

d. Frames 0 to 3 mapped to B13, A31, A08, A17 respectively.

e. The inverted page table immediately shows that Process B has page 17 in frame 0. B's page table entry for 17 will be marked as **not memory resident**, and page 13 will be marked as **memory resident**.

4. (Applications of virtual memory)

An application of virtual memory in some OSes (including Linux) is overcommit. When overcommit is enabled, the OS will not allocate pages immediately upon request, but instead only when they are actually used.

In Linux, a new memory allocation has its contents initialised to zero. Overcommit allows Linux to defer allocating pages for a memory allocation until the program writes to the memory, (if it ever does).

a. How can this be implemented, while allowing reads to be successful?

Another application of virtual memory is demand paging, which ties in closely with the idea of memory-mapped files (i.e. `mmap`). Memory-mapped files are a natural extension to the idea of a swap file; in a way, we are simply allowing programs to specify the "backing store" for a particular region of memory.

b. How can demand-paged (i.e. the file is not read until the program actually reads from the mapped region) memory-mapped files be implemented?
c. What are the advantages and disadvantages of using memory-mapped files compared to regular read/write system calls?

Solution

a. Allocate a single read-only "zero page" that is filled with zeroes. When a program requests for memory, map the new pages to the zero page as a read-only mapping. If the program writes to memory, a segmentation fault/general protection fault will be triggered; at this point, we have to actually allocate a page to the program.
b. When a program maps a file, simply record the memory region as mapped to the given file, and insert the relevant page table entries as non-resident pages. (Some other tracking structures will likely be needed.) When the program tries to read from the region, a page fault will occur, and the OS can then perform the file read for the page at that point.
c. Advantages:
    i. (For the programmer) The programmer can simply read/write from memory and not have to deal with read/write syscalls.
    ii. (For the OS) The OS can easily reclaim those pages in physical memory by simply reusing the pages (if it knows that the pages have not been modified), or by writing the changes back to the file (which can be done even if there is no swap file configured on the system), since it can simply read the pages from the file again if a program reads from the mapped region of memory.
    iii. (For the OS/system as a whole) With read/write syscalls, the data is read into memory that is private to the program. If multiple processes read from the same file, the data is duplicated in memory. With memory-mapped files, the OS can simply share the same physical pages across multiple processes, saving physical memory.
d. Disadvantages:
    i. It is generally not possible to return an error from a memory read. If there is an error when reading the mapped file to fulfil a memory read, the program gets a SIGSEGV or SIGBUS, which is much harder to recover from than a failed syscall.

ii. It is also not possible to fulfil memory reads asynchronously. Thus if the file read takes a while, the program will be blocked until the read completes (or fails). (This can be slightly mitigated with the madvise syscall.)

For your own exploration:

5. (Page table structure)

A computer system has a 36-bit virtual address space with a page size of 8 KB, and 4 bytes per page table entry.

a. How many pages are there in the virtual address space?
b. What is the maximum size of addressable physical memory in this system?
c. If the average process size is 8 GB, would you use a one-level or two-level page table? Why?

Solution

a. A 36-bit virtual address can address 2^{36} bytes in a byte addressable machine. Since the size of a page is 8 KB (2^{13}), the number of addressable pages is $2^{36} / 2^{13} = 2^{23}$.

b. Assuming that all 4 bytes in each PTE is used for the frame address, then we can address 2^{32} physical frames. Since each frame is 2^{13} bytes long, the maximum addressable physical memory size is $2^{32} * 2^{13} = 2^{45}$ bytes (32 TB).

c. 8 GB = 233 bytes. We need to analyse the memory and time requirements of the two paging schemes in order to make a decision. The average process size is considered in the calculations below.

   1-level paging: Since we have 2^{23} pages in each virtual address space, and we use 4 bytes per page table entry, the size of the page table will be $2^{23} * 2^{2} = 2^{25}$ (32 MB). This is 1 / 256 of the process's own memory space, so it is quite costly.

   2-level paging: The single page table will be broken into multiple page tables, and each can fit into a single page. Since we have 8 KB (2^{13} B) pages and 4-byte PTEs, each small table can hold 2^{11} PTEs. There is a total of $2^{23} / 2^{11} = 2^{12}$ page tables ➔ the page directory has 2^{12} entries. Hence, the virtual address would be divided up as 12 | 11 | 13.

   Since the process' size is only 8 GB (2^{33}) ➔ only $2^{33} / 2^{13} = 2^{20}$ pages. So we only need $2^{20} / 2^{11} = 2^{9}$ page tables (instead of the full range of 2^{12}).

   The total overhead is then (size of page directory) + (total size of page tables) = $(2^{12} * 4) + (2^{9} * 2^{11} * 4)$ = 4112 KB.

   As seen, 2-level paging requires much less space than level 1-paging scheme.