Grenoble Universities (GrenobleINP & Université Joseph Fourier)
M1 MoSIG
Academic year: 2014-2015

Operating System Design — Midterm exam

October 22, 2014 — Duration: 90 minutes

Important instructions: All printed documents are allowed. All electronic devices are forbidden (calculators, computers, mobile phones, etc.).
The number of points indicated for each exercise is only provided to give you an idea of its weight. We reserve the right to change the exact number of points.

Problem 1 (2.5 points)

For each of the questions of this problem, simply answer by indicating its number and one or several letters corresponding to the correct statement(s) (you are not required to justify your answers). Note that there are between 1 and 5 correct statements per question. A question without any answer is considered empty (no points). A single mistake in an answer (i.e., exactly one missing correct statement or one incorrect statement) voids half of the points allocated for the question. A number of mistakes greater than one voids all the points allocated for the question (there are no negative points).

Question 1.1

Which of the following statements are true ?

(a) The best-fit allocation strategy chooses the largest free block into which the requested block fits.

(b) Using the first-fit allocation strategy on a free list that is ordered according to increasing block sizes is equivalent to using the best-fit strategy.

(c) A garbage collector for C programs may move allocated blocks to fight external fragmentation.

(d) A program that never frees dynamically allocated memory blocks may run out of memory (if the capacity of the heap is insufficient) but will not be impacted by external fragmentation.

Question 1.2

Which of the following statements are true ?

(a) For many operating systems (like Linux and Windows), the most part of the kernel source code (95% or more) is written in C and the rest is written in assembly language.

(b) Invoking a system call from user-level code always triggers a mode switch.

(c) Invoking a system call from user-level code may trigger a context switch to another process but this is not always the case (it depends on the chosen system call and the precise situation).

(d) The typical duration of 10 ms for the time quantum of an operating system scheduler is motivated by the fact that, on most machines, the hardware cannot generate timer interrupts with a higher frequency.

Question 1.3

Which of the following statements are true ?

(a) Each process has its own virtual address space, with its own logical copy of each byte. However, the operating system may try to transparently share the same physical copy of some data between different processes (for example, for the code regions of applications and libraries).

(b) In the case of a virtual memory implementation based on paging, two bytes with adjacent virtual addresses may not necessarily be stored at two adjacent physical addresses.

(c) The ABI (Application Binary Interface) of an operating system defines, among other things, conventions regarding the implementation details for the stack management and the function calls.

(d) Local variables are allocated on the stack for fast allocation/deallocation, due to their Last-In-First-Out life cycle.

Question 1.4

Which of the following statements are true ?

(a) A TLB hit brings a performance gain but only if the data corresponding to the requested virtual address is found in the processor (L1 or L2) cache.

(b) Using a multi-level paging structure (instead of a flat table) increases the time of a TLB hit.

(c) On processors with an 'architected page table" (e.g., Intel x86), swapping pages between the main memory and the disk is automatically managed by the processor hardware.

(d) In the context of paging/swapping to disk, the theoretically optimal page replacement policy consists in evicting the page that will not be used for the longest period of time.

Question 1.5

Among the following actions, which of them are forbidden/impossible for code running in user mode ?

(a) modifying the system call handler or the trap handlers ;

(b) modifying the register that points to the current paging structure ;

(c) disabling the timer device ;

(d) blocking hardware interrupts ;

(e) triggering the execution of the process scheduler.

Problem 2 (3.5 points)

Question 2.1

Some memory allocators are based on a design with the following features (summarized in Figures 1 and 2) :

- the allocator uses a linked list to keep track of free blocks ;
- each block is associated with a 32-bit header and a 32-bit footer ;
- the header contains the block size (including the bytes used for the header, padding and footer) stored on 31 bits and a 1-bit boolean flag indicating the current status of the block (allocated/free) ;
- the footer has the same contents as the header.

Describe one reason (related to the performance of the allocator) that motivates the use of such a header and such a footer. (Expected answer length : about 10 lines).

Question 2.2

We consider the allocator design from the previous question and we make the following additional assumptions :

- pointers are stored on 32 bits ;

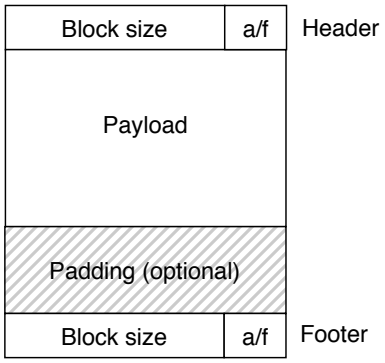

FIGURE 1 – Structure of an allocated block

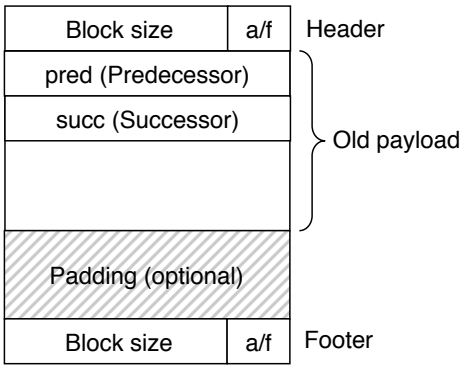

FIGURE 2 – Structure of a free block

- the allocator always enforces a "double word alignment constraint" (i.e., within an allocated block, the first address of the payload must always be a multiple of 8) ;
- initially, the heap contains only one large free block, for which the first address of the payload is a multiple of 8.

Given the above assumptions, what is the minimum possible size for a free block ? And what is the minimum possible size for an allocated block ? Justify your answers.

Question 2.3

Note : in this question, for simplification, we ignore headers, footers and alignments constraints.

We consider a new memory allocator design, which works as follows : one half of the heap memory is divided into fixed-sized units of 4 kilobytes, and the other half is managed by a best-fit free list. If an allocation request is less than or equal to 4 kB and there is space in the fixed-sized half, a 4 kB unit is allocated from the fixed-sized half ; otherwise, the best-fit algorithm is used over the other half of memory, and the requested size is returned (if enough space is available).

1. Assuming that a total memory capacity of 32 kB is initially available for the heap, what series of allocation requests will most quickly lead to all of memory getting allocated, all while requesting the least total amount of memory ?
2. What type(s) of fragmentation may occur with this new allocator ?

Problem 3 (3 points)

In this exercise, we consider a simple machine with an MMU that implements virtual memory based on segmentation. The main specifications of this machine are the following :

- The MMU hardware has two pairs of (base, bounds) registers (i.e., a process can at most have two segments).
- Virtual addresses (including the explicit segment ID) are stored on 10 bits and physical addresses are stored on 14 bits.
- The machine is equipped with a capacity of 16 kB of RAM.

We consider a process with two segments configured as follows (the values are given in decimal notation) :

- segment 0 : base = 1024, bounds (size) = 300 bytes
- segment 1 : base = 5000 , bounds (size) = 200 bytes

The process executes the following piece of code :

```
1   int x;
2   void *ptr = 20;
3   while (ptr <= 1023) {
4       x = *((int *)ptr); /* read what is at address 'ptr' */
5       ptr = ptr + 20; /* make ptr point 20 bytes further up in the addr. space */
6   }
```

Note : for the whole problem, we assume that memory accesses related to instruction fetches or to the modification of the `ptr` variable do not trigger any segmentation violation.

Question 3.1

How long will the program run before crashing (due to a segmentation violation triggered by line 4) ? More precisely, you are asked to provide (and explain) :

a) the number of successive iterations of the loop that work correctly ;

b) for each successful iteration, the virtual address stored in `ptr` and the corresponding physical address ;

c) the first iteration of the loop that triggers a segmentation violation and the corresponding virtual address stored in `ptr`.

Question 3.2

We now assume that the body of the `while` loop is modified to avoid dereferencing `ptr` when it points to an invalid address (note : you are not asked to provide the corresponding code). Describe the list of physical addresses that will be generated by the dereferencing of `ptr` during the whole execution of the loop. Justify your answer.

Problem 4 (7 points)

In this problem, we consider a machine with a 32-bit Intel x86 processor (with virtual addresses and physical addresses both stored on 32 bits). An overview of the paging structures is provided in Figures 3 and 4, respectively for a page size of 4 kilobytes (2^{12} bytes) and 4 Megabytes (2^{22} bytes). Each page directory entry (PDE) contains a bit that indicates whether the address range covered by the PDE is managed with 4-kB pages or with a 4-MB page. Note that a page directory (or a page table) is always stored on one 4-kilobyte page.

Question 4.1

The goal of this question is to determine the two following numbers regarding a process address space : (i) the number of valid pages and (ii) the amount of space required to store the paging data structures. For each of the following cases, indicate the corresponding numbers and justify your answer.

(a) There is only one valid address range (0x00000000 to 0xFFFFFFFF) in the process address space, whose mapping uses exclusively 4-kilobyte pages.

(b) There is only one valid address range (0x00000000 to 0xFFFFFFFF) in the process address space, whose mapping uses exclusively 4-Megabyte pages.

(c) There is only one valid address range (0x00000000 to 0x003FFFFF) in the process address space, whose mapping uses exclusively 4-kilobyte pages.

(d) There is only one valid address range (0x00000000 to 0x003FFFFF) in the process address space, whose mapping uses exclusively 4-Megabyte pages.

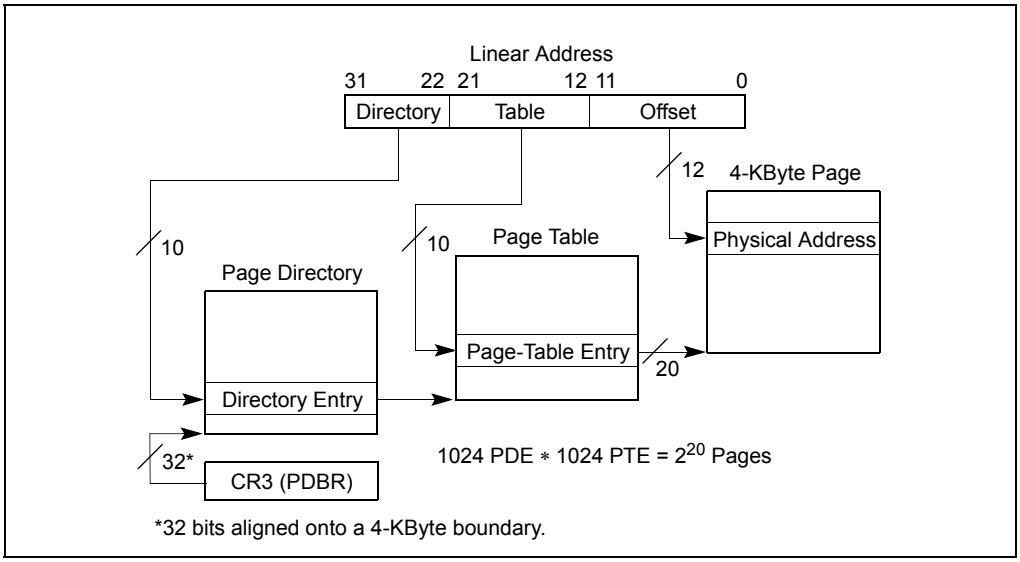

FIGURE 3 – x86 address translation with 4-kilobyte pages (source : Intel documentation)

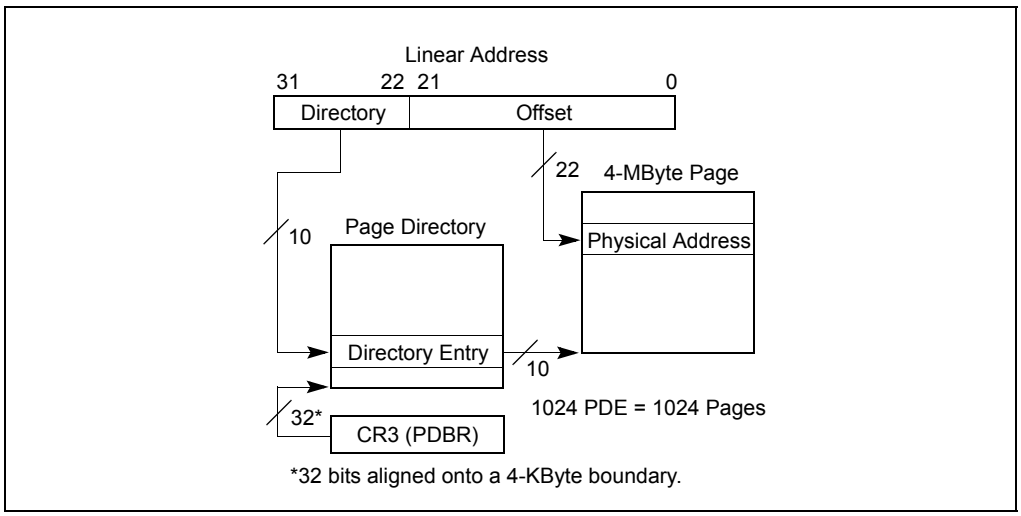

FIGURE 4 – x86 address translation with 4-Megabyte pages (source : Intel documentation)

Question 4.2

We consider a process whose address space is configured as depicted on Figure 5 (note that, for simplification purposes, some memory regions such as `.bss` and the dynamic shared libraries are not present). For instance, the process stack is located between addresses `0xCFC00000` and `0xCFFFFFFF` (included). We assume that the operating system is configured to use 4-Megabyte pages for the data region and 4-kilobyte pages for the other regions.

Justify your answers to the following questions :

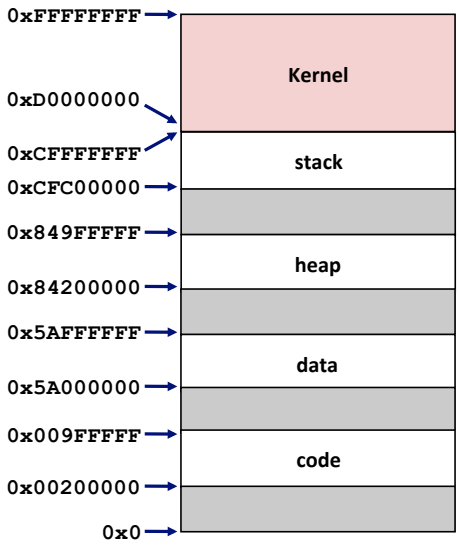

FIGURE 5 – Process address space. Note that the figure is not drawn to scale.

a) What is the available capacity (in bytes) for the kernel-level part of the address space ?

b) What is the size (in bytes and in number of pages) of the space reserved for each user-level region ?

c) What is the amount of space required by the paging structures for the code and the data regions ?

Question 4.3

One of your friends reads the following paragraph in a textbook about operating systems : *"An OS kernel for a modern processor can possibly use different page sizes for different memory regions. However, under some circumstances, this may introduce a problem of external fragmentation. For example, consider a system using two different page sizes (4 kB and 4 MB) in the following situation : a page fault occurs and this requires to swap in a 4-MB page"*. Unfortunately, the next page of the book is damaged and unreadable. How would you complete the example of the textbook for your friend ? (Expected answer length : about 10 lines)

Problem 5 (4 points)

Question 5.1

In practice, is the LRU page replacement policy always efficient ? If so, explain why. If not, describe a workload for which LRU behaves poorly. (Expected answer length : about 5-10 lines.)

Question 5.2

On Intel processors that do not have a tagged TLB (i.e. the TLB entries do not contain a process address space identifier), the contents of the TLB are automatically flushed upon every context switch. However, the specification of the contents of a Page Table Entry (PTE) includes a field named G ("Global"). This field acts as a boolean flag. When this flag is set, the corresponding PTE remains cached in the TLB even if the TLB is flushed. Note however that such an entry may still get evicted from the TLB in order to load another PTE for a more recently accessed page. Give an example to illustrate why the G flag may be useful. (Expected answer length : about 10 lines.)