CS 390
Chapter 8 Homework Solutions

8.3 `Why are page sizes always ...`

Page sizes that are a power of two make it computationally fast for the kernel to determine the page number and offset of a logical address. The page number can be determined by right-shifting the logical address. The offset can be determined by masking out some number of the most-significant bits of the logical address.

8.4 `Consider a logical address space ...`

a. We need to calculate the total number logical addresses: There are 2^6 pages $\cdot 2^{10}$ words per page $= 2^{16}$ words in the logical address space. Thus, logical addresses need to be at least 16 bits.

b. 32KB of physical memory contains 2^{15} addresses, so there need to be a minimum of 15 bits in the physical address.

8.5 `What is the effect of allowing ...`

If two page table entries point to the same frame, then two pages will map to the same frame. If we need to copy the contents of a large number of pages, we can allocate a range of logical addresses and then make the page table entries for those addresses point to the data to be copied. To the process, it looks like the data has been copied, when in reality, both "copies" of the data refer to the same physical memory. (We use this same scheme in object-oriented programming when we avoid copying an object and instead just copy the reference to the object.)

This is the technique used to implement POSIX shared memory; when two processes want to share a segment of memory, the kernel allocates pages for the shared segment, then adjusts the page tables of the two processes so that the specified logical addresses both map to the same shared physical pages.

8.9 `Explain the difference between internal ...`

Internal fragmentation is free space *inside* of a partition that can not be allocated for another purpose. External fragmentation is free space *outside* of any partition that cannot be used because it is too small.

8.10 `Consider the following process for ...`

The linkage editor must alter every address in each module to the actual physical address of the object. Since the object code is split into modules, the easiest way to do this is to calculate the offset of each address from the beginning of the module. The physical address of each object is then calculated by finding the physical address of the module and then adding the offset of the object.

The compiler can help by maintaining a table that contains an entry for each object in the program that has an address. (Compiler-savvy students will recognize this data structure as a symbol table.)

Each object entry consists of the name of the module that contains it, and the offset of the object within the module.

8.13 `Compare the memory organization schemes...`

a. External fragmentation

Contiguous Allocation with Fixed-Size Partitions: does not suffer from external fragmentation.

Contiguous Allocation with Variable-Size Partitions: suffers from external fragmentation.

Pure Segmentation: suffers from external fragmentation.

Paging: does not suffer from external fragmentation.

b. Internal fragmentation

Contiguous Allocation with Fixed-Size Partitions: suffers from internal fragmentation.

Contiguous Allocation with Variable-Size Partitions: does not suffer from internal fragmentation.

Pure Segmentation: does not suffer from internal fragmentation.

Paging: suffers from internal fragmentation.

c. Ability to share code across processes

Contiguous Allocation with Fixed-Size Partitions: no support for code sharing across processes.

Contiguous Allocation with Variable-Size Partitions: no support for code sharing across processes.

Pure Segmentation: supports code sharing across processes. However, we must be careful to make sure that processes do not mix code and data in the same segment.

Pure Paging: supports code sharing across processes. As in pure segmentation, we must make sure that processes do not mix code and data in the same page.

8.16 `Although Android does not support ...`

Many Android applications depend on having a known, fixed amount of space for temporary files in order to work correctly. These files are stored on the boot disk. If an Android device allowed swapping to its boot disk, the disk might become full, breaking these applications.

8.19 `Program binaries in many systems ...`

a. contiguous-memory allocation: First, assume that the size of the process's memory partition is fixed at boot time; that is, the system is using fixed-size partitions. The maximum size of the stack and heap are fixed for each process, and is calculated as the size of the partition, minus the size of the code and data segments.

If the system is using variable-sized partitions, then the problem becomes more complex. A process can request a particular partition size at load time. However, if the size of the partition is too large, memory between the stack top and the end of the heap will be wasted. If the partition size is too small, the stack and the heap will collide during execution, leading to corrupted memory and a possible process crash.

b. pure segmentation If the stack and heap are placed in the same segment, memory will either be wasted, or we risk the possibility of a stack/heap collision. If the stack and heap are placed in different segments, we may be able to expand the heap, but the maximum stack size is fixed at load time, since the stack grows downward. One solution on a system that uses pure segmentation is to redefine the stack to grow upwards from low memory to high.

c. pure paging Pure paging suffers from the same problem as contiguous-memory allocation, since we need to allocate a frame for each page of the process, even if the page is unused. We can solve this problem by combining paging with swapping in a scheme called "virtual memory" which we will study in the next chapter.

8.20 `Assuming a 1-KB page size, ...`

a. page 3, offset 13

b. page 41, offset 111

c. page 210, offset 161

d. page 634, offset 784

e. page 1953, offset 129

8.25 `Consider a paging system with ...`

a. `If a memory reference takes ...`

100ns: 50ns to access the page table entry, plus another 50ns to access the desired memory location. Storing the page table in memory effectively halves the speed of memory under paging.

b. `If we add TLBs, and ...`

The effective access time is
$0.75 \cdot (2 + 50) + 0.25 \cdot (2 + 2 \cdot 50) = 64.5\text{ns}$.

8.28 `Consider the following segment table ...`

What are the physical addresses for the following logical addresses?

a. `0,430:` 649

b. `1,10:` 2310

c. `2,500:` Address error. The requested address is beyond the end of the segment.

d. `3,400:` 1727

e. `4,112:` Address error. The requested address is beyond the end of the segment.

8.24 `What is the purpose of ...`

Imagine a system that allows processes to be a maximum of 16GB in size, uses 64 bit addresses, and which has 1KB pages. A 16GB process would have a maximum page table size of 16GB/1KB pages $\cdot$ 8 bytes per page table entry $=$ 128MB. Thus, each process requires 128MB of memory just for the page table, even if the process doesn't use all 16GB of its legal address space.

For quick access, the page table must be stored in contiguous memory (like an array). Paging the page table (in combination with virtual memory, which we will study in the next chapter) allows the page table to be stored non-contiguously and allows the process to execute without the entire page table in memory.