Started on	Thursday, 4 February 2021, 9:09 AM PST
State	Finished
Completed on	Thursday, 4 February 2021, 11:09 AM PST
Time taken	2 hours
Grade	**91.00** out of 100.00

Question **1**

Complete

15.00 points out of 15.00

Segmentation is an approach that uses some number of base and bounds registers to help virtualize memory. Assume that a system uses segmentation only (without paging) to virtualize memory.

(a) What is a base register for? (3 points)

(b) What type of address is in the base register: physical, or virtual? (3 points)

(c) What is a bounds (limit) register for? (3 points)

(d) Why do we need more than one base/bounds register pair? (3 points)

(e) How many segments should the hardware support and why? (3 points)

a) a base register is used to get the segment base. it is the start address of the segment in PHYSICAL memory.

b) it is a physical address.

c) the bounds (limit) register is used to hold the length of the segment. if we go past this length, we get a segmentation fault. we must check whether the offset is within range of this limit register.

d) we have multiple segments 1 for stack 2 for heap, and so on. thus, we need these descriptors for every single segment

e) the hardware should support 4 segments at least, because each of the segments in the program (stack, heap, and code) require one each. However i believe x86-64 uses 6 code segments including the extra FS and GS as well as code, stack, data, and extra. it should support all of these segments because stack and heap and code should not be together. they have different privileges and keeping them apart helps security. for example, code segment should not be written to, while heap segment should be (or the computer will be kind of useless).

Question **2**

Complete

24.00 points out of 25.00

Assume we have a system with a 32-bit virtual address space, a 4-entry TLB, and a two-level page table structure. The size of each page table entry is 4B, and the size of each page is 1KB. It takes 0 ns to access the page-table base register (PTBR), and 10ns to access TLB. Time for each memory access is 100 ns.

(a) How many address bits are needed for the page offset? (2 points)

(b) How much memory in bytes is required to store the outer page table entirely in main memory? (5 points)

(c) The CPU has just been context-switched to a new process whose page tables are entirely stored in main memory. If there is a one-to-one mapping between logical and physical addresses, what is the average effective access time for sequentially accessing the following set of logical addresses: 2, 3, 4,

average effective access time for sequentially accessing the following set of logical addresses: 2, 3, 4, 5, 5, 2010, 3, 2?

(c1) Assuming that there is no preloading for TBL and TLB is flushed upon each context switch. (3 points)

(c2) Assuming that TLB performs sequential preloading (i.e., when one virtual-physical mapping is loaded, mappings for the next three logical addresses are loaded as well), what would be the total access time? (3 points)

(c3) If the TLB contained only 1 entry and the page-table base register access time was 10 ns, what would be the average effective access time for part c1? (3 points)

(d) Suppose the outer page table is stored in main memory starting at frame 0 and the inner page tables are stored in memory sequentially immediately after the outer page table. If logical address 80386 translates to physical address 1073742338, what are the values stored in the corresponding outer page table entry and inner page table entry? Assume that no permission bits are stored in each table entry. (9 points)

a) since the size of each page is 1KB or 1024 bytes, the number of bits needed for page offset of log2(1024) = 10.

b) since 10 bits are reserved for offset, and we would like to keep the second-level table within 1 page, there should be 1 KB/4B = 2^8 entries in the second level page table. this means that 8 bits should be reserved for the second level table. 32 - 8 - 10 = 14 so 14 bits should be reserved for the first level page table. this is 2^14 entries, or 2^16 bytes because each entry is 4 bytes. that is 65536 bytes.

c)

c1) the first 4 accesses are all faults, as there is nothing in the TLB. since it will still check the TLB, each of these accesses will take 10 + 100 ns, or 110 ns. next, when the second 5 is accessed, it will be in the TLB so its access time will be just 10 ms. with 2010, it will have to eject the oldest element in the TLB (assuming use of LRU), so its access time will be 10 + 100ns or 110 ns. Next, 3 is already in the TLB so its access time is 10ms, and 2 was kicked out and not in the TLB anymore, so its access time is 110 ns. Thus, the total access times for these 8 accesses is 110 * 6 + 10 * 2 = 680. and the average effective access time for this set of accesses is 680/8 = 85 ms

c2) the first access will still be a fault, which will take 10 + 100ns. However, 3, 4, and 5 will be loaded as the TLB performs sequential preloading. Thus, it will take 10ns each to do those. 2010 will be a fault, which will take 10 + 100ns. However, the next two accesses of 2 and 3 will be hits, so 10ns each. Thus, the total accesses will take 110 + 10 + 10 + 10 + 10 + 110 + 10 + 10 = 280 ns. The average effective access time is 280 / 8 = 35ns

c3) Each of the first 4 would take 10 + 10 + 100 ns = 120 ns. The second 5 would only take 10ns however. the last 3 accesses would still take 10 + 10 + 100ns = 120ns because one must first access the TLB, then when it faults, access the PTBR, then access memory. 120 * 4 + 10 + 3 * 120 = 850 ns. the average access time would then be 106.25ns

d) im assuming that outer page table means lvl 1 and inner means lvl 2. since each page is 1024 bytes, and the logical address is 80386 that means that the logical address is 78 pages + an offset of 514.
Similarly 1073742338 is 2^30 + 514. Or, 2^20 pages plus an offset of 514. Thus, the PPN is 2^20 and is what is stored in the page table level 2. each page in lvl 2 stores 2^8 entries, so we need the 2^12th entry in page table level 1. 2^12th entry means the 2^12th level 2 page, which is store 1KB * (2^12 + 1) down from address 0. (+1 because the first level page table is stored there). Thus, the address stored in page table level 1 is at about 4 MB down, or exactly 4195328.

Question 3

Complete

15.00 points out of 20.00

Consider a system where there are two data segments A and B, both of which can be accessed by user programs. The logical address space is 1KB, and the amount of physical memory is 16KB. Assume that segment A's base register has the value 1KB, and its limit (size) is set to 300 bytes; this segment grows upward. Assume segment B's base register has the value 5KB, and its limit is also 300; this segment grows downward (the negative direction).

Assume we have the following program:

```
void *ptr = 20;
  while (ptr <= 1024) {
    int x = (int *) *ptr; // LINE 1: read what is at address 'ptr'
    ptr = ptr + 20; // LINE 2: increment 'ptr' to a new address
}
```

(a) What virtual addresses are generated by this program at LINE 1 by dereferencing ptr ? Assume the program runs to completion; please just list the addresses as generated by the loads from memory via the dereferencing of ptr; don't worry about instruction fetches or the store into x, for example) (5 points)

(b) How long will this program run before crashing (due to a segmentation violation)? (5 points)

(c) What physical addresses will be generated by dereferencing ptr before the program crashes? Assume the system does not use paging, and the base register and the bus have the same width in terms of # bits. (5 points)

(d) What legal physical addresses could have been generated by dereferencing ptr, if the programmer had been more careful to avoid crashing? Hint: consider both segments. (5 points)

a) ptr is an address of value 20, and dereferencing it we end up with the integer that the address 20 points to in memory. we end up with 20, 40, 60, 80, ... all the way to 1020 or until the program segfaults.

b) it will loop since the limit is 300 bytes, the loop will loop about 300/20 = 15 times before it crashes.

c) WRONG >>> since the base register is at address 5000, and the limit is set to 300 bytes, assuming it is a one to one mapping, the physical addresses will look like 4980, 4960, 4940, etc. until 4700 right before the program crashes. the reason we are looking at segment B is that while dynamically allocated data has memory stored in segment A (since it grows upwards it is the heap), the actual address of the temporary variable ptr is stored in segment B. this is the value we are looking at.

WAIT I AM WRONG. they want physical address generated by dereferencing ptr. everything i said was the opposite. we are looking at segment A because that is the heap segment (pointing upward is heap/data segment) and so the physical addresses look like 1024, 1044, 1064, all the way until 1324 where it crashes.

d) i know that technically everything works until the stack segment and heap segment meet, but the problem is that the limit is 300 bytes which means that the stack can only grow down to 5KB - 300 bytes, and the heap can only grow up to 1KB + 300 bytes. the address that ptr points to can therefore only go up to 1324. i feel like im missing a part of this because the concept that the stack/heap meet = seg fault is fundamental, but the limit is there :(.

if we were to ignore limit (maybe i understood the concept of limits wrong), they would meet halfway at 3KB? in which case 2048 iterations would have been allowed, but i dont think that is the case as the end of the loop is only 1024, or 51 increments of ptr.

Question **4**

Complete

10.00 points out of 10.00

Modern ISAs all provide an instruction called "return from interrupt" (reti). This instruction switches the mode of operation from kernel to user mode. Usually, this instruction can be used only in kernel mode. Explain

(1) Where in the OS this instruction would be used? (5 points)

(2) What would happen if a user application executes this instruction? (5 points)

(1) the OS will call this instruction whenever it wants to return control back to the user. this will happen after an interrupt (which the name implies haha) or at the end of a context switch. the OS switches from kernel mode to user mode in this instruction, so it will only be able to be used in kernel mode.

(2) if a user application were to try to execute this instruction, the standard procedure for when a user tries to execute privileged instructions will go through. Namely, there will be a hardware exception, and then depending on the OS different things happen. The OS could just terminate the process, or it could go into an exception handler, or other things.

Question 5

Complete

14.00 points out of 15.00

Here is a table of processes and their associated arrival and running times.

Process ID	Arrival Time	Expected Running Time
1	0	4
2	2	3
3	4	5
4	6	2

(a) Show the scheduling order for these processes under First-In-First-Out (FIFO), Shortest-Job First (SJF), and Round-Robin (RR) with a 1 time unit. Assume that the context switch overhead is 0 and new processes are added to the head of the queue except for FIFO. (5 points)

Time	FIFO	SJF	RR
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			

(b) For each process in each schedule above, indicate the queue wait time and turnaround time (TRT). Queue wait time is the total time a thread spends in the wait queue. Turnaround time is defined as the time a process takes to complete after it arrives. (5 points)

Scheduler	Process 1	Process 2	Process 3	Process 4
FIFO queue wait				

FIFO TRT

SJF queue wait

SJF TRT

RR queue wait

RR TRT

(c) A FIFO page replacement algorithm replaces the page that was first referenced the longest time ago. Under a workload with temporal reference locality, choosing to replace a page accessed a long time ago seems like a good idea, since it is not likely to be within the current locality being accessed. Does this mean that a FIFO replacement algorithm should approximate a LRU algorithm for workloads with strong locality? Justify your answer with an example memory reference pattern or string. (5 points)

a)

FIFO: 11112223333344

SJF: 11112224433333

RR: 11213241324333 (new processes are added to the head of the queue)

b)

FIFO queue wait: 0, 2, 3, 6

FIFO TRT: 4, 5, 8, 8

SJF queue wait: 0, 2, 5, 1

SJF TRT: 4, 5, 10, 3

RR queue wait: 0, 0, 0, 0

RR TRT: 8, 8, 10, 5

c) FIFO replacement algorithm could approximate an LRU algorithm for workloads with strong locality because if the locality is strong enough and the varying sections of the address space that are accessed are far apart, replacing the least recently used versus the first one accessed is pretty close. For example, if you access a set like (1, 2, 4, 3, 4, 2, 1, 3), and then later you access (200, 201, 203, 201, 204, 202), replacing the 1 through the FIFO algorithm, and replacing the 4 with the LRU algorithm makes little difference.

some might argue the opposite. they might say that using FIFO could also be a bad approximation for LRU if the sections are long and close to each other. if a memory access string went like 1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5. like when accessing an array multiple times, in a two dimension for loop for an algorithm, say DP, FIFO would be a bad approximation even though this shows a degree of locality. 1 will appear again every couple of iterations, and that using FIFO would kick it out after every long segment. also if the process is frequently accessing a page, you don't want it to be paged to disk even if it was the first process that you accessed. assuming the process accesses lots of memory, then many different accesses can be done within the same workspace. think array traversal (lots of memory accesses). i think i just convinced myself that LRU is better than FIFO after writing this paragraph oops.

Question **6**

Complete

13.00 points out of 15.00

Consider the following C program. Assume that all system calls succeed when possible.

```
1 void* rem(void *args) {
2    printf("Blue: %d\n", *((int*) args));
3    exit(0);
4 }
5 void* ram(void *args) {
6    printf("Pink: %d\n", ((int*) args)[0]);
```

```
7    return NULL;
8 }
9 int main(void) {
10    pid_t pid; pthread_t pthread; int status; //declaring vars
11
12    int fd = open("emilia.txt", O_CREAT|O_TRUNC|O_WRONLY, 0666);
13    int *subaru = (int*) calloc(1, sizeof(int));
14    printf("Original: %d\n", *subaru);
15
16    if(pid = fork()) {
17       *subaru = 1337;
18       pid = fork();
19    }
20   if(!pid) {
21     pthread_create(&pthread, NULL, ram, (void*) subaru);
22   } else {
23     for(int i = 0; i < 2; i++)
24          waitpid(-1, &status, 0);
25    pthread_create(&pthread, NULL, rem, (void*) subaru);
26  }
27  pthread_join(pthread, NULL);
28  if(*subaru == 1337)
29    dup2(fd, fileno(stdout));
30  printf("All done!\n");
31 return 0;
32 }
```

(a) Including the original process, how many processes are created? Including the original thread, how many threads are created? (5 points)

(b) Provide all possible outputs in standard output. If there are multiple possibilities, put each in its own box. You may not need all the boxes. (5 points)

(c) Provide all possible contents of emilia.txt. If there are multiple possibilities, put each in its own box. You may not need all the boxes. (5 points)

a) fork is called, a child process is created. in the parent process, another fork is called, so another process is created. total number of processes at the very end is [3]. the original thread creates a process so that running thread is duplicated. the child process creates another thread by itself so that child process has 2 threads total. the parent process forks and creates two new threads, the child one creates a thread so that child has 2 threads total. the parent process also creates a thread so the parent process has 2 threads. Thus, the total number of threads is [6].
b)

1)

Original: 0

Pink: 0

All done!

Pink: 1337

Blue: 1337

2)

Original: 0

Pink: 1337

Pink: 0

All done!

Blue: 1337

c)

1)

All done!

◀ Discussion 1E Week 3

Jump to...

Discussion1B Week4 ▶