7 Page Tables [14 points]

Suppose that a computer system consists of a processor that uses 50 bit virtual addresses and its paging hardware supports 16 KB page frames. The size of each page table entry is 4 bytes. Each entry has a valid bit, a referenced bit, a dirty bit and three bits for protection that are located in the high order bits of the entry. If a question below asks you to calculate a size, show the size in a human readable unit (e.g., KB, MB or GB).

Part A [3 points]

If the paging hardware used a single-level page table, calculate the total size of the page table.

Part B [5 points]

Design a multi-level page table for the paging hardware. Ensure that each page table fits within a page frame. Draw a diagram with the page tables and physical memory that clearly shows how address translation would work in your design.

Question continues ...

Part C [2 points]

Calculate the maximum size of physical memory that can be supported by this system.

Part D [4 points]

Suppose that there are currently 8 processes that are executing in the system. Each process is using 2 MB of memory and no pages are swapped to disk. Calculate the approximate maximum amount of memory that your multi-level page table could be using at this time? Would the multi-level page table fit in memory?

8 Page Replacement Mechanism [6 points]

Part A [3 points]

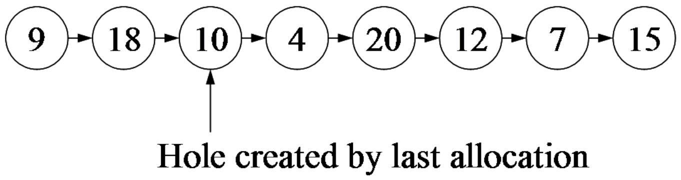
Many processers maintain valid, reference and dirty (modified) bits in their page table entries. Explain the purpose of each of these bits for virtual memory management. Describe when these bits are set or reset.

Part B [3 points]

Some older processors do not provide a reference bit in the page table entry. Explain how the operating system software could simulate a reference bit using the valid bit.

5 Memory Management [10 points]

Consider a dynamic partitioning system in which memory consists of the following holes (all sizes are in KB):



Part A [8 points]

Suppose a new process arrives requiring 13 KB of memory, followed by a process requiring 11 KB of memory. Show the list of holes after both these processes are placed in memory for each of the following algorithms (start with the original list of holes for each algorithm).

1. First Fit:

   9, 5, 10, 4, 9, 12, 7, 15

2. Worst Fit:

   9, 7, 10, 4, 7, 12, 7, 15

3. Best Fit:

   9, 18, 10, 4, 20, 1, 7, 2

4. Next Fit:

   9, 18, 10, 4, 7, 1, 7, 15

Part B [2 points]

The above list is not sorted in any order. State one benefit if the list is kept sorted by memory address.

It is easier to coallesce holes because the holes will be adjacent in the list.

6 Page Replacement [10 points]

The memory reference string consists of the sequence of memory addresses accessed by a program in a run. This reference string can be used to simulate various page replacement algorithms to evaluate their performance. However, it is expensive to log the memory reference string, unless additional hardware is present. Instead, page replacement algorithms can be simulated using a page reference string, which consists of the sequence of pages accessed by a program in a run. Alternatively, the page reference string consists of a memory reference string in which all consecutive memory addresses that access the same page are replaced with a single page number. For example, suppose x is an arbitrary page offset, then if the memory reference string is 1x, 1x, 2x, 3x, 3x, then the page reference string is 1, 2, 3.

Part A [8 points]

Describe how you would implement a method by which the OS can accurately log the page reference string of a process. Justify why your method will work. Hint: use the page table entry.

The key idea is that the OS must ensure that only one page table entry is valid at a time. When a page fault occurs, the OS should make the page valid (assuming that the page is mapped and has the correct permissions), and make the previous page invalid. As long as the process accesses the current page, there will be no page faults. When a new page is accessed, a page fault is generated. The sequence of page faults determines the page reference string.

Part B [2 points]

Will your method above work if other processes are running? Explain your answer.

The above method will work even if other processes are running because it logs the pages (virtual addresses) that are accessed by the process, which is mostly independent of which other processes are running. This method will work even with shared memory across processes because each process has its own page table.

It is possible that the program behavior will change because other processes are running, but then again, the above method will track the pages referenced by the process.

5. Virtual Memory [20 marks]

You need to port the virtual memory system for 353OS to a new architecture. The processor has a 16-bit address space, and each address accesses a single 8-bit byte of memory. A two-level page table scheme is used, with 16 entries in the top-level (primary) page table and 64 entries in the second-level (secondary) page table. Each page table is stored page aligned. Each page table entry is two bytes wide and has the following format:

1 bit	1 bit	1 bit	1 bit	12 bits
Valid	Readable	Writeable	Executable	Frame number

Page table entries are stored in memory in big endian order, i.e., lowest address stores the highest-order byte.

For top-level page tables, the R, W, and X bits are unused and should be set to zero. For second-level page tables, the readable (R) bit indicates whether the page can be read by the thread; the writeable (W) bit indicates whether the page can be written by the thread; and the executable (X) bit indicates whether the page can be used to fetch instructions for execution.

A partial listing of the machine's physical memory is shown below. (For convenience we are showing the contents of memory four bytes at a time, although the machine is byte addressed. The first byte in each entry is the lowest-order byte. For example, the value of physical address 0x01 in memory is 0xf7.)

0x00	6b f7 30 7b	**0xa0**	8a be 9b 09	**0x140**	7c a7 00 80
0x04	da 0b 6a ff	**0xa4**	53 f0 13 31	**0x144**	e6 30 d2 71
0x08	9f 07 7a 8e	**0xa8**	b4 38 60 85	**0x148**	0f 1e 21 9e
0x0c	42 47 b8 2e	**0xac**	e3 8e 2b 36	**0x14c**	f1 98 97 6e
0x10	c6 17 d2 41	**0xb0**	34 bc e0 07	**0x150**	c0 12 d0 04
0x14	50 11 f0 00	**0xb4**	0b 3f 30 1a	**0x154**	11 81 d0 72
0x18	b6 ff f2 4c	**0xb8**	16 31 61 7c	**0x158**	96 7b d6 5b
0x1c	f3 ce a4 28	**0xbc**	38 ab 42 06	**0x15c**	e3 bb ef e6
0x20	c0 5a 18 25	**0xc0**	64 0f aa ac	**0x160**	0b a1 c6 d4
0x24	95 95 b9 be	**0xc4**	72 d2 07 b6	**0x164**	c7 1a a3 ca
0x28	0b c2 01 f1	**0xc8**	cc 12 c2 50	**0x168**	6b f7 30 7b
0x2c	c9 f7 f8 88	**0xcc**	05 ca 78 2b	**0x16c**	ad 6c 01 22
0x30	44 16 6e 48	**0xd0**	02 ce 70 3e	**0x170**	5b fd ab 63
0x34	3a d1 2b ca	**0xd4**	d0 df 45 e6	**0x174**	b0 d7 14 06
0x38	2d 15 31 1a	**0xd8**	6b d5 1f 4c	**0x178**	93 12 4c 1f
0x3c	72 be bc b4	**0xdc**	00 a7 80 02	**0x17c**	8b 42 47 f3
0x40	f2 bc 4e 89	**0xe0**	80 03 00 41	**0x180**	ac 3e e7 89
0x44	58 19 a0 2d	**0xe4**	00 16 80 06	**0x184**	78 49 5d ba
0x48	7f f2 ab 03	**0xe8**	00 03 80 00	**0x188**	34 43 4d 8e
0x4c	1a 01 d2 e1	**0xec**	80 05 00 09	**0x18c**	c8 96 d8 40
0x50	84 01 4b 80	**0xf0**	89 2d c6 d5	**0x190**	56 78 69 ea
0x54	19 ba 67 3f	**0xf4**	b2 d3 53 58	**0x194**	90 37 16 89
0x58	78 82 75 95	**0xf8**	8a be 9b 09	**0x198**	f2 89 34 f1
0x5c	84 52 00 08	**0xfc**	e1 7a d9 12	**0x19c**	19 e4 26 16
0x60	30 b5 17 dd	**0x100**	22 9c 18 67	**0x1a0**	17 90 91 a0
0x64	f2 ba 51 0a	**0x104**	b5 37 23 8f	**0x1a4**	1d ad 95 de
0x68	bc 04 ed 19	**0x108**	b6 94 c6 d8	**0x1a8**	af 87 3d d7
0x6c	57 84 dd 3f	**0x10c**	cc f7 3f e8	**0x1ac**	fe 42 dd 41
0x70	5a 2a 3d 4e	**0x110**	13 ef 40 4e	**0x1b0**	52 d2 0b 3a
0x74	b9 a7 89 d8	**0x114**	f4 a1 2c 6a	**0x1b4**	44 06 28 55
0x78	6c d2 d8 65	**0x118**	ed 6e 6e 5c	**0x1b8**	31 f3 b4 ee
0x7c	3a b3 87 81	**0x11c**	ce a2 8b 8f	**0x1bc**	31 56 3d 23
0x80	00 10 00 00	**0x120**	58 ae fe b5	**0x1c0**	8c d1 19 c6
0x84	00 03 80 08	**0x124**	1f a3 0d 82	**0x1c4**	fa 69 8b 2f
0x88	80 f4 80 cc	**0x128**	1a af 41 78	**0x1c8**	75 16 cb f7
0x8c	00 a7 80 0a	**0x12c**	e5 9e cb 97	**0x1cc**	32 5a 6d 2c
0x90	80 04 00 41	**0x130**	fd cd ee f6	**0x1d0**	73 23 20 04
0x94	00 03 80 00	**0x134**	1d e0 0e 18	**0x1d4**	02 55 bb 8d
0x98	80 07 00 09	**0x138**	da 4c 36 a2	**0x1d8**	62 ee 09 20
0x9c	00 03 80 0c	**0x13c**	49 0c 20 9d	**0x1dc**	62 db d1 5c

a) [2 marks] What is the size of each page in bytes?

Address space is 16 bits.
4 bits for top page table, 6 bits for secondary page table.
6 bits are left for each page = 2^6 bytes (each bit allows accessing one byte) = 64 bytes per page.

b) [2 marks] What is the maximum size of the physical memory in bytes?

12 bits for number of frames.
6 bits per frame.
Total memory = 18 bits = 2^18 bytes (each bit allows accessing one byte) = 256KB.

c) [4 marks] Assume that the current thread's top-level page table starts at physical address 0x80. List the contents of the top-level page table here. How can you tell that you are decoding memory correctly?

Index	Valid	Page Frame Number
0		10
1		0
2		3
3	V	8
4	V	f4
5	V	cc
6		a7
7	V	0a
8	V	4
9		41
10		3
11	V	0
12	V	7
13		9
14		3
15	V	c

d) [12 marks] Translate each of the following virtual addresses into a physical address. The V, R, W and E are the Valid, Readable, Writeable and Executable bits.

Virtual address	Primary page table entry value	Secondary page table entry value	V?	R?	W?	E?	Physical address
0x808d	8004	b537	yes	no	yes	yes	0x14dcd
0x2142	0003	N/A					
0xc112	8007	7516	no	yes	yes	yes	N/A

5. Virtual Memory [8 marks]

A processor uses 24 bit virtual addresses. The memory management unit uses a two-level page table scheme and the page size is 2^{10} bytes (1KB). The page table entries are 32 bits wide. The top-level page table stores 16 page table entries. The memory management unit is designed to be able to address a maximum of 4GB of physical memory. The frame number is stored in the lowest order bits in the page table entry as shown below:

A page table entry:

Control bits	Frame number

a) [1 mark] How many page-table entries are stored in each second level page table?

There are 10 bits for page offset (1KB page size), 4 bits for first level page table. So there are 10 bits for second level page table (virtual address is 24 bits).

b) [1 mark] At maximum, how many entries are in a process's page table?

First level page table can store 16 PTE. Second level table can store 1024 entries (10 bits for second level page table). So maximum number of entries is 16 + 16 * 1024.

c) [1 mark] A process P uses at most 12 pages in its address space. What is the minimum number of pages that need to be allocated for the two-level page table for this process?

minimum number of pages = 1 (for top-level page table) + 1 * 4 (for second-level page table) = 5
Note that a second-level page table needs 4 pages.

d) [1 mark] What is the maximum number of pages that need to be allocated for the two-level page table for the process P above?

maximum number of pages = 1 (for top-level page table) + 12 *4 (for second-level page table) = 49

e) [4 marks] A partial second level page table, with page table entries on each row, is shown below. The start of the table is shown at the top and memory addresses grow downwards. The MMU will use this table to translate the virtual address 0x200b4c. What is the physical address corresponding to this virtual address?

Note that the MMU can access 4GB (32 bits) of memory. The page/frame size is 10 bits. So the number of bits for the frame number is 22. The control bits in the page table entry are stored in the upper 10 bits (see diagram in previous page).
virtual address = (0010, 0000, 0000, 1011, 0100, 1100) (0x200b4c in binary)
Top 4 bits are used to access the first page table. That page table points to the second-level page table shown below (according to the question above). The next 10 bits are used to access the second level page (0000, 0000, 10) -> index to the page table below is 2 -> PTE has value (00 03 80 00), or (0000, 0000, 0000, 0011, 1000, 0000, 0000, 0000) in binary. The frame number consists of the low 22 bits or (11, 1000, 0000, 0000, 0000). Combine that with the page offset (low 10 bits from virtual address or (11, 0100, 1100)) to get the physical address, (1110, 0000, 0000, 0000, 0011, 0100, 1100) or 0xe00034c.

Increasing memory addresses

80 03 00 40
00 16 80 06
00 03 80 00
80 05 00 09
89 2d c6 d5
b2 d3 53 58
8a be 9b 09
e1 7a d9 12
22 9c 18 67
b5 37 23 8f
b6 94 c6 d8
cc f7 3f e8

Question 7. Page Replacement [8 MARKS]

Two processes, P1 and P2 are executing on a system with 8 pages of physical memory. The OS attempts to maintain a pool of free page frames. The page replacement algorithm will be triggered when the number of free frames drops below the low water mark (1 frame) and it will reclaim pages until the number of free frames reaches the high water mark (3 frames). The last free frame has just been allocated.

P1 page table:

Virt Pg	Valid bit			Ref bit			Frame
	(a)	(b)	(c)	(a)	(b)	(c)	(a)
0	1		0	1	0	0	3
1	1			1	0	0	7
2	1			1	0	0	4
3	1			1	0	0	6
4	0			0			—
5	1	0	0	0			0

P2 page table:

Virt Pg	Valid bit			Ref bit			Frame
	(a)	(b)	(c)	(a)	(b)	(c)	(a)
0	0			0			—
1	1			1	0	0	2
2	1	0	0	0			5
3	0			0			—
4	1	0		1	0	0	1
5	0			0			—

Coremap:

Page Frame	In Use			PID	Virt Pg
	(a)	(b)	(c)	(a)	(a)
0	1	0	0	1	5
1	1	0		2	4
2	1			2	1
3	1		0	1	0
4	1			1	2
5	1	0	0	2	2
6	1			1	3
7	1			1	1

Part (a) [2 MARKS] Fill in the initial coremap data using the page tables for P1 and P2.

Part (b) [3 MARKS] Show the changes that would result (by filling in column (b) in the tables) if the global clock replacement algorithm is used, and the clock hand is initially pointing at Frame 0.

Part (c) [3 MARKS] Consider the following modification to the global clock algorithm: select victim page frames such that each process loses pages proportional to the amount of memory it is currently allocated. For example, if a process is using half of the total physical memory, then half of the victim frames should come from that process (fractions are rounded to the nearest integer number of pages). Show the changes that would result (by filling in column(c) in the tables above) using clock with this modification. Start with the initial state and assume the clock hand begins at Frame 0 again.

For (c), how many pages should be stolen from each process? P1: ______ P2: _______

P1: 3*5/7 = 15/7 = 2

P2: 3*3/7 = 9/7 = 1

Question 5. Memory Management [10 MARKS]

Part (a) [4 MARKS] You are updating the operating system for an older CPU architecture that provides base and limit registers for address translation and memory protection. You have been asked to implement paged memory allocation for this system. Is this possible? If so, give a high-level description of how it would work. If not, explain why it would not work.

It is not possible to implement paging with a single base and limit register.

A single base and limit register could be setup to allow access to one page at a time, similar to having a TLB with one entry. When the page is accessed, it would be allowed. When any other page is accessed, a fault would be generated. This could be handled by modifying the base and limit register to allow access to the faulting page.

However, this approach does not work because a single MIPS instruction can access the code region and the data region, e.g., a load or store instruction. Such an instruction will cause an infinite number of faults and not make any progress with the above scheme.

Part (b) [4 MARKS] A particular system has 17-bit virtual addresses, 20-bit physical addresses, and a 1024 byte page size.

a) How many bits are needed for the offset? 10 ______

b) How many virtual pages can a process have? $2^{17-10} = 128$ ______

c) How many physical page frames can the system have? $2^{20-10} = 1024$ ______

d) If page table entries include Valid, Dirty, Referenced, Read, Write and Execute bits, what is the minimum size of a page table entry? 10+6 = 16 ______

Part (c) [2 MARKS] For the system in part 2 above, should we implement a two-level page table? Why or why not?

With a one-level page table, the total page table size is 128 (nr. of pages) * 2 bytes (size of page table entry) = 256 bytes, which is less than one page (1024 bytes). So we don't need a two-level page table, because it would increase the memory requirements for the page table and be slower.

Question 6. Virtual Memory Page Replacement [8 MARKS]

Given the following stream of page references by an application, calculate the number of page faults the application would incur with the following page replacement algorithms. Assume that all pages are initially free.

Reference Stream: A B C D A B E A B C D E B A B

Part (a) [2 MARKS] FIFO page replacement with 3 physical pages available.

11 faults, see below. We will accept 8 as the answer if you assumed that loading the initial three pages didn't cause page faults.

```
a  d  e b
 b  a  c  a
  c  b    d
```

Part (b) [2 MARKS] LRU page replacement with 3 physical pages available.

12 faults, see below. We will accept 9 as the answer if you assumed that loading the initial three pages didn't cause page faults.

```
a  d  e  c  b .
 b  a  .  d  a
  c  b  .  e
```

Part (c) [2 MARKS] Optimal page replacement with 3 physical pages available.

8 faults, see below. We will accept 5 as the answer if you assumed that loading the initial three pages didn't cause page faults.

```
a    cd
 b
  cde  a
```

Part (d) [2 MARKS] If we increase the number of physical pages from 3 to 4, will the number of page faults always be equal to or decrease using FIFO page replacement for any reference stream? If so, briefly explain your answer. Otherwise, show an example of a reference stream for which the number of page faults with 3 physical pages is less than the number of page faults with 4 physical pages. State the number of page faults in both cases.

Counter example, using the reference stream shown above: A B C D A B E A B C D E B A B.
With 3 physical pages, the number of page faults is 11.
With 4 physical pages, the number of page faults is 12.

Question 5. TLB [18 MARKS]

Part (a) [4 MARKS] An early RISC processor uses a software-managed TLB with a 4 KB page size, and 4 byte integer size. It only provides a single control bit in the TLB entry. This control bit is the valid bit. You wish to implement swapping on this processor and would like to track referenced and dirty pages. Can you simulate each of these bits in the OS? If so, explain how. If not, explain why. If you make any assumptions, state it clearly below.

Bits	Yes/No	How/Why
Referenced	Yes	On a TLB fault, mark the page table entry as referenced, and load the TLB entry. Periodically (e.g., timer interrupt) mark the page table entry as unreferenced and evict the TLB entry, so that next time the page is accessed, a TLB fault will occur, and the page can be marked referenced again.
Dirty	Yes	The TLB does not have a writeable bit, so we cannot use a TLB fault to determine that a page that was initially read has been dirtied. Instead, when a page is read, we will need to mark it referenced and not load the TLB entry. When a page is written, mark the page dirty and load the TLB. This approach does not take advantage of the TLB for read-only accesses, and hence is very inefficient.

If the answer mentioned disabling/not using the TLB for both read and write, we gave full marks.

Part (b) [4 MARKS] For the processor described above, consider the following code snippet:

```
int a[4096];
int b[4096];

int simple() {
    int c = 0; int i = 0;
cswitch:    /* <- context switch takes place */
    for (i = 0; i < 4096; i++) {
        c = a[i] + b[i];
    }
    return c;
}
```

How many TLB misses will take place when the `simple()` function is run for the first time? Ignore the context switch comment in the code. Provide a justification for each TLB miss. State any assumptions that you make.

page size = 4KB, size of integer is 4 bytes.

Total number of misses = 10.

1 TLB miss for code section (assuming all code fits in 4 KB).
1 TLB miss for stack section (for variables i and c).
8 TLB misses for data section (Array 'a' takes 4 pages (4096 * 4), similarly 'b').

Part (c) [2 MARKS] Now suppose a context switch occurs at the `cswitch:` label in **Part (b)**. At this context switch, the TLB is flushed completely. Will there be additional TLB misses when this code is run? If so, how many? Clearly explain why.

There will be 2 additional TLB misses for the code and stack sections. These entries would have been loaded previously but would have been evicted on a context switch. The data section TLB entries (for the two arrays) would not have been loaded at the time of the context switch, and so there will be no additional TLB misses for the data section.

Part (d) [2 MARKS] Suppose you decide to use a linear (single-level) page table in your OS for the processor described earlier. How many memory accesses would occur to the page table when the code shown in **Part (b)** is run without any context switches?

There would be 10 memory accesses to the linear page table for the 10 TLB misses.

Part (e) [4 MARKS] Now let's imagine that the processor didn't have a TLB and accessed your linear page table directly when translating addresses. How many memory accesses would occur to the page table when the code shown in **Part (b)** is run without any context switches? Circle the best answer below and provide a justification.

1. Same as the answer to **Part (d)**
2. Somewhere between 4K-8K accesses
3. Somewhere between 8K-12K accesses
4. Somewhere between 12K-16K accesses
5. Somewhere between 16K-20K accesses
6. [✓ More than 20K accesses]

The for loop runs 4096 (4K) times. In the loop, there are memory accesses to a[i], b[i], c, and i. In addition there are at least two instructions that are executed in the loop, e.g., the load and store instructions for loading a, b, i, and storing c and i, and a jump instruction. So the total number of accesses to the page table are 4K * (4 data accesses + 2 or more instructions) > 20K accesses.

Part (f) [2 MARKS] Now suppose a context switch occurs at the `cswitch:` label. Would your answer in **Part (e)** be the same or not? Why?

The context switch does not change the number above because the context switch affects the TLB accesses, and in Part (e), there is no TLB.

Question 6. Page Tables [20 MARKS]

Suppose that a computer system consists of a processor that uses 50 bit virtual addresses and its paging hardware supports 16 KB page frames. The size of each page table entry is 4 bytes. Each entry has a valid bit, a referenced bit, a dirty bit and two bits for protection that are located in the high order bits of the entry. Please answer the questions shown below. If a question below asks you to calculate a size, show the size in a human readable unit (e.g., KB=2^{10} bytes, MB=2^{20} bytes, GB=2^{30} bytes or TB=2^{40} bytes), unless indicated otherwise.

Part (a) [3 MARKS] If the paging hardware uses a single-level page table, calculate the total size of the page table.

page size = 16 KB, so page offset has 14 bits
# of pages = 2^(50-14) = 2^36
page table size = 2^36 * 4 bytes per entry = 2^38 = 2^8 GB = 256 GB

Part (b) [3 MARKS] Calculate the maximum size of physical memory that can be supported by this system.

frame offset = 14 bits (see above)
PTE has 32 bits. It has 5 control bits, so the maximum frame number can have 27 bits.
So maximum physical memory supported = 2^(27 + 14) = 2^41 = 2 TB

Part (c) [4 MARKS] You realize that a single-level page table is not the best design for this processor. Design a multi-level page table for the paging hardware by answering the following questions: 1) how many levels does your page table design need, 2) how many page table entries will each level have? Ensure that each page table fits within a page frame.

The number of pages is 2^36 (see Part a). So we need 36 bits to index a page number. The page size is 16 KB. The number of page table entries that can fit in a page is 16 KB/4B= 4K = 2^12. So each level can support 12 bits of the page index. So we need a three-level page table, with each level indexing 12 bits of the page number (this accounts for the 36 bits in the page number).

Part (d) [2 MARKS] Suppose a process is using 4 MB of memory and no pages are swapped to disk. What is the minimum amount of memory that your multi-level page table could be using at this time? Show your answer in pages.

A page is 16 KB. Nr.of pages needed for 4 MB is 2^22/2^14 = 2^8 = 256 pages.
A page table has 2^12 = 4096 entries. So a single third-level page table can support all the 256 pages above. The minimum amount of memory needed is a page for each level of the page table or 3 pages.

Part (e) [4 MARKS] Suppose a process is using 4 MB of memory and no pages are swapped to disk. What is the maximum amount of memory that your multi-level page table could be using at this time? Show your answer in pages.

The maximum amount of memory needed for page tables would be as follows:
1. Top-level page table: 1 page
2. Second-level page table: 256 pages, each page pointing to one third-level page
3. Third-level page table: 256 pages, with a single page table entry in each page pointing to a physical frame.
So total number of pages = 513 pages.

Part (f) [4 MARKS] Given the pages being accessed by the process in **Part (e)**, how would you redesign your page table to reduce the memory needed for page tables?

The main idea is to increase the number of levels in the page table and have smaller size page table at each level.

For example, we could have a 4 level page table. Each table would index 9 bits (9 * 4 = 36 bits for indexing all pages). Each page table would be 2^(9 + 2 bits per entry) = 2 ^11 = 2 KB. In this scheme, the total memory requirement is:

1 + 256 + 256 + 256 = 769 pages (but the page size is 1/8 that of Part e).

Another option might be to use an inverted page table. Nr of frames = 2^27. Page table entry size = 8 bytes (4 bytes for next field). So total size of the page table is 2^30 bytes = 2^30 bytes/2^14 bytes/page = 2^16 pages = 65536 pages. So this doesn't really work unless there were many (> 65536 / 513) processes in the system.

Question 4. **Paging** [6 MARKS]

The figure below shows the address translation mechanism used by a 64-bit processor. Starting with a 48-bit virtual addresses, it uses 4 levels of page table to translate the address to a 52-bit physical address. Each page table entry is 8 bytes long.

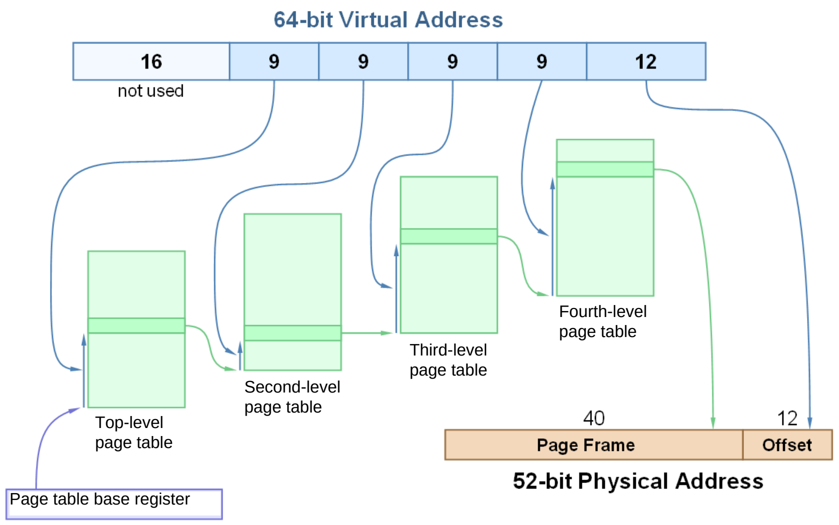


Figure 1: 4-level Page Table

If the page size is increased, the number of levels of page table can be reduced. How large must pages be in order to translate 48-bit virtual addresses with only a 2 level page table? Draw the address translation mechanism, similar to the figure above, corresponding to this page size.

Notice first that the page size is 2^12 in Figure 1. The page table size is also 2^12 (2^9 * 8 bytes per page table entry, for a 64 bit processor).

For a two level page table, say page size is 2^n. The page table size is also 2^n bytes in this multi-level page table. So the number of PTE in a page table is 2^(n-3).

So the virtual address is broken into (n-3), (n-3), and n bits.

(n-3) + (n-3) + n = 48

n = 18, so page size is 2^18 = 256KB.

The 48-bit virtual address is divided into 15, 15, 18 bits. The first 15 bits index into the first page table, the second 15 bits index in the second page table, and the rest of the 18 bits is the page offset.

Question 5. TLB [8 MARKS]

A 32-bit processor uses a two-level page table, with a page size of 4KB. It has two TLBs, an I-TLB (instruction TLB) that stores mappings for code addresses, and a D-TLB (data TLB) that stores mappings for data addresses. Both TLBs are fully associative (all TLB entries are looked up in parallel), and both have 64 entries. Consider the following code snippet:

```
char array[4096 * 64];

void simple() {
  int i = 0;
  int or = 0;
  int and = 1;

  for (i = 0; i < 4096 * 64; i++) {
    or = or | array[i];
  }
  for (i = 0; i < 4096 * 64; i++) {
    and = and & array[i];
  }
}
```

Part (a) [1 MARK] How many I-TLB misses will take place when the `simple()` function is run? Provide a justification for each TLB miss. State any assumptions that you make.

One I-TLB miss, assuming all the code above fits in 4KB.

Part (b) [3 MARKS] What is the **minimum** number of D-TLB misses that will take place when the `simple()` function is run? Provide a justification for each TLB miss. State any assumptions that you make.

Minimum number of D-TLB misses = 1 + 63 + 1 + 1 (see reasons below) = 66

Notice that array is stored in 64 pages

- 1 D-TLB miss for a stack page (which stores the local variables of simple()).

Loop 1:
- 63 D-TLB misses when the first 63 pages of the array are accessed. Now the D-TLB is full.
- 1 D-TLB miss when the last element of array is accessed. This access will require invalidating a TLB entry. Let's invalidate the last TLB entry (for the 63th page), and replace it with the TLB entry for the 64th page.

Loop 2:
- 0 D-TLB misses for the first 62 page accesses (these entries are already present).
- 1 D-TLB miss for page 63. Let's invalidate TLB entry for page 62, and replace it with TLB entry for page 63.
- 0 D-TLB miss for page 64.

Part (c) [2 MARKS] Let us estimate the performance overhead of using the paging system. Assume that we have disabled the processor cache, so that every load and store instruction accesses memory (RAM). Assume also that there is no cost to accessing the TLB, and the compiler has performed no optimizations. Also, we will only consider accesses to `array` in the `simple()` function. How many memory accesses are performed to read `array`? How many memory accesses are performed to read the page table? Now estimate the overhead of the paging system. State your answer as a single number, possibly in terms of a power of 2.

Total number of memory access for reading array: 4096 * 64 * 2
Number of memory accesses to read the page table = 2 (nr. of levels of page table) * nr. of TLB misses = 2 * 66 (from previous answer)

Overhead = 2* 66/(2*66 + 2 * 64 * 4096)
~ (2 * 64)/(2 * 64 * 4096)
~ 1/4096
= 2^(-12)

If you said 2^(-11), we accepted the answer. This is reasonable if you simply assume that there is one TLB miss for each page. So you require two additional memory accesses for the page table for accessing each page (4096 memory accesses).

Part (d) [2 MARKS] Suggest a way of changing the code in the `simple()` function to reduce the number of D-TLB misses. What is the number of misses with your code change? Provide a justification.

Use one loop to do both calculations.

This reduces the one additional D-TLB miss that occurred in Part(b) when loop 2 executed.

Total number of D-TLB misses = 65

Question 6. **Paging** [10 MARKS]

You have been hired by the MobARM Corporation to develop the virtual memory architecture on their next generation mobile processor. The virtual memory design has the following parameters:

- Virtual addresses are 54 bits.
- The page size is 64K byte.
- The architecture allows a maximum of 128 Terabytes (TB) of physical memory.
- The first- and second-level page tables are stored in physical memory.
- All page tables can start only on a page boundary.
- Each second-level page table fits exactly in a single page frame, but the first-level page table may be larger than a page frame.
- There are only valid bits and no other extra permission, or dirty bits.

Draw a figure showing how a virtual address gets mapped into a physical address. Your design should be as efficient as possible. You should list how the various fields of the virtual and the physical address are interpreted, including the size of each field (in bits). Show the format of the page table entry, the total number of entries each table holds, and the total size of each table (in bytes).

page size 64KB => offset is 16 bits.

max physical memory = 128 TB => 47 bits.

number of bits in the frame = 47 - 16 = 31

with 1 valid bit, the page table entry can nicely fit in 4 bytes (31 + 1) bits.

second level page table is 64KB (one page, 16 bits), so it can have 2^14 entries (each entry is 4 bytes).

virtual address size = 54 bits

so top level page table should be able to address (54 - 14 - 16) = 24 bits

top-level page table size = 2^24 * 4 (bytes per entry) = 2^26 = 64MB.

Question 5. Missing the TLB [12 MARKS]

TLB misses can be nasty. The following code can cause a lot of TLB misses, depending on the values of STRIDE and MAX. Assume that your system has a 32-entry TLB with a 8KB (2^{13} bytes) page size. Assume that the code runs on a 32-bit machine architecture. Also, assume that `malloc` below returns a page-aligned address.

```
int value = 0;
int *data = malloc(sizeof(int) * MAX); // an array of MAX integers
for (int j = 0; j < 1000; j++) {
    for (int i = 0; i < MAX; i += STRIDE) {
        value = value + data[i];
    }
}
```

Part (a) [6 MARKS] Choose the minimum value for `MAX` and for `STRIDE` so that every access to the `data` array causes a TLB miss (100% TLB miss rate). Use the space below to show any calculations. If you make any assumptions, state them below.

Assuming that the TLB entries are replaced using LRU (or randomly, or any reasonably replacement algorithm), if the array is 33 pages long, and a new page is accessed when data[i] is accessed, then each such access will likely cause a TLB miss. This will not cause a page fault because 33 pages is just 33 * 8KB = 264KB of memory, which is pretty small on modern machines.

`STRIDE =` STRIDE >= 8KB/sizeof(int) = 8KB/4 = 2048

We divide by sizeof(int) above because data is an int array. With a stride value of 2048, each read of data[i] will access a new page.

`MAX =` MAX >= 33 * STRIDE = 33 * 2048 = 67584

If the answer mentioned that the local variables and code will require 1 page each, we also accepted 31 * STRIDE.

Part (b) [2 MARKS] Would the *page fault* rate 1) increase, 2) remain the same, or 3) decrease, if the MAX value is made 10 times the value of MAX you have chosen in Part(a) (STRIDE is unchanged)? Explain your answer.

Assuming any contention (other programs are running), page fault will increase.
Page fault will remain the same, only if a lot of physical memory is available.

Part (c) [2 MARKS] Would the *page fault* rate 1) increase, 2) remain the same, or 3) decrease, if the MAX and STRIDE values are both made 10 times the values of MAX and STRIDE you have chosen in Part(a)? Explain.

Page fault will remain the same because the number of unique pages that are accessed compared to Part(a) is the same.

Part (d) [2 MARKS] Say we doubled the TLB size (64 entries). With the MAX and STRIDE values you have chosen in Part(a), would the *TLB miss* rate be 1) less than 10%, 2) around 50%, or 3) greater than 90%?

The TLB miss rate will be less than 10%, likely close to 0, because the number of TLB entries is larger than 33, and we are accessing 33 pages repeatedly.

Question 6. Paging Redone [12 MARKS]

An 16-bit machine with a paging MMU has a virtual address space size of 256 bytes, and a page size of 16 bytes. The machine uses a linear page table and the page table entry (PTE) uses 8 bits, with the format shown below. Note that the valid bit is the least significant bit in the PTE.

	7 bits	1 bit
PTE:	frame number	valid bit

Part (a) [4 MARKS] The memory dump of the page table, located at physical address 0, shows the following:

```
00: 0xa9 0xb4 0xbf 0xc2 0xff 0xd7 0xfe 0x03 0xfd 0x09 0x10 0x87 0x39 0x7f 0x6e 0x05
```

For the following virtual addresses, say whether it is a *valid* virtual address or will cause a *fault*. For those that are valid, write the physical address (in hex) that results from the translation.

	Fault/Valid Physical Address (in hex)
virtual address 0x65:	0x65: '6' maps to 6th PTE = 0xfe. Lowest bit is unset => invalid => fault.
virtual address 0x0c:	0x0c: '0' maps to 0th PTE = 0xa9. Lowest bit is set => valid frame nr = 1010100 = 0x54, paddr = 0x54c
virtual address 0x26:	0x26: '2' maps to 2nd PTE = 0xbf. Lowest bit is set => valid frame nr = 1011111 = 0x5f, paddr = 0x5f6
virtual address 0xa8:	0xa8: 'a' maps to 10th PTE = 0x10. Lowest bit is unset => invalid => fault.

This question continues on the following page.

Question 6. Paging Redone (CONTINUED)

Part (b) [8 MARKS] The designers of the 16-bit machine wanted to run larger programs. They decided to update the paging MMU to use a two-level page table, supporting a virtual address space of 4096 bytes. The designers carefully ensured that the top and the second-level page tables have the same size, and the format of the page table entry (PTE) in both of them is the same as shown in the previous page. No other changes were made to the architecture.

Consider the partial memory dump of the page table, located at physical address 0:

```
00:0xa9 0xb4 0xbf 0xc2 0xff 0xd7 0xfe 0x03 0xfd 0x09 0x10 0x87 0x39 0x7f 0x6e 0x05
16:0xb2 0xb7 0x7e 0xbe 0x93 0xeb 0xa2 0xd2 0x97 0x96 0xd9 0xc7 0xb5 0xa7 0xea 0x9d
32:0x15 0x0e 0x14 0x17 0x31 0x0b 0xa0 0xdb 0x07 0x04 0x13 0x0f 0x1d 0x4f 0x1e 0x47
```

The page table register has the value 0. For the following virtual addresses, say whether 1) it is a *valid* virtual address, 2) it will cause a *top fault*, i.e., a fault in the top-level page table, 3) it will cause a *second fault*, i.e., a fault in a second-level page table, or 4) you *can't tell* because the memory dump doesn't provide enough information. For valid addresses, write the corresponding physical address (in hex).

Top Fault/Second Fault/Valid/Can't Tell Physical Address (in hex)

virtual address 0x143:

0x143: '1' maps to 1st PTE in top-level page table = 0xb4. Lowest bit is unset => invalid => top fault.

virtual address 0x70c:

0x70c: '7' maps to 7th PTE in top-level page table = 0x03 = 1|1. Lowest bit is set => valid second level page table at frame 1. Since each frame is 16 bytes, this page table starts at address 16. '0' maps to 0th PTE = 0xb2. Lowest bit is unset => invalid => second fault.

virtual address 0xf43:

0xf43: 'f' maps to 15th PTE in top-level page table = 0x05 = 10|1. Lowest bit is set => valid second level page table at frame 2. Since each frame is 16 bytes, this page table starts at address 32. '4' maps to 4th PTE = 0x31 = 11000|1. Lowest bit is set => valid. frame number = 11000, paddr = 11000|0011 = 0x183.

virtual address 0x9a8:

0x9a8: '9' maps to 9th PTE in top-level page table = 0x9 = 100|1. Lowest bit is set => valid second level page table at frame 4. Frame 4 is not shown. Can't tell.