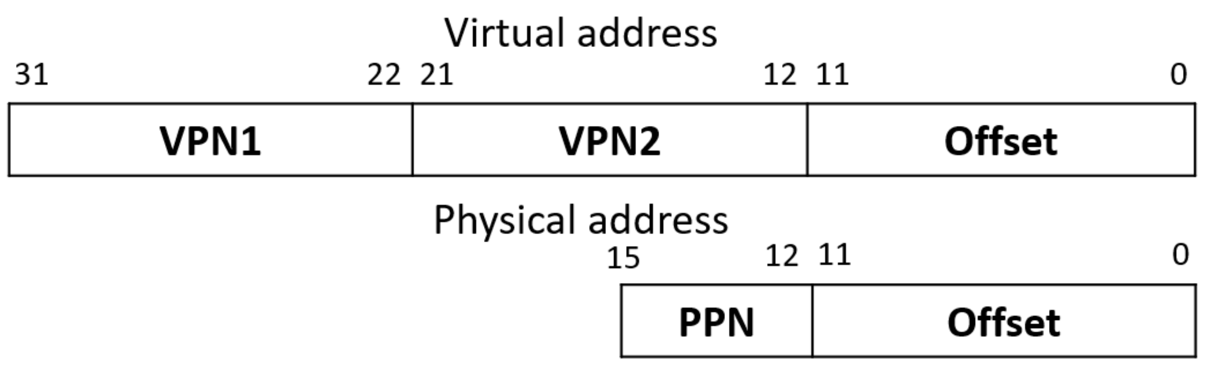
HW9.2. Page Table Walk

Walk with me.

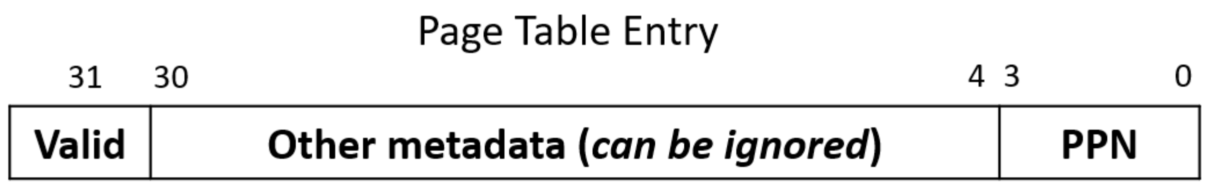
Consider a **64 KiB byte-addressable physical memory** that has the following contents physical_memory.txt. Since the physical memory is 64 KiB, there are 16 bits of memory address (ranging from 0x0000 to 0xFFFF) to uniquely identify each byte in the memory.

Consider a machine with a **page size of 4 KiB (12 offset bits)**. It provides a **32-bit virtual memory space** for each program that runs on it.

The machine utilizes a 2-level hierarchical page table that has 20 bits of VPN divided equally between L1 and L2 page tables. Note that 32 bits of virtual address is comprised of the **20-bit VPN + 12-bit offset**. Accordingly, the 16-bit physical address will be comprised of the **4-bit PPN and 12-bit offset**. Explicitly, the partitioning of the virtual and physical addresses is shown below:



Each page table entry (PTE) is 4 bytes, with the corresponding data partitioning as shown below:



The most significant bit is 1 when the entry is valid and the last 4 bits correspond to the PPN pointing to either the next level page table or the data page. Extra metadata bits are included in the PTE but are not relevant for this problem.

Now, suppose we run two different programs on this machine. Coincidentally, we observed that the two programs access the same virtual address as shown below:

```
# Program 1 code
...
lui t0, 0x00403
addi t0, t0, 0x004  # t0 = 0x00403004
lw t1, 0(t0)        # load word from address 0x00403004
...
```

```
# Program 2 code
...
lui s0, 0x00403
addi s0, s0, 0x004  # s0 = 0x00403004
lw t5, 0(s0)        # load word from address 0x00403004
...
```

As you can see, both programs are loading a word from the address `0x00403004`. However, since we now know about virtual memory, we know that even though both programs are pointing at the same address, those addresses are virtual. Thus, when we look at the physical memory, the data that will be loaded by Program 1 can be different from the data that will be loaded by Program 2. For this problem, we will try to figure out the address translation and, eventually, the data that will be loaded by each program.

For this problem, we are given the page table base register (PTBR) for the two running programs. **For Program 1, PTBR = 0x4000. For Program 2, PTBR = 0x6000**. We will need these values as we do the address translation (also known as the page table walk). The PTBR basically points to the physical memory address where we will start the translation. Follow the story guide (translation for Program 1 will be done for you) and answer the questions (for the translation for Program 2) accordingly.

We first partition the virtual address `0x00403004` into the VPN1-VPN2-offset fields. We first write out the entire 32 bit address and then partition accordingly.

For the following 4 questions, write the answers in binary without the 0b prefix and no spaces.

Q1. Write out the 32-bit equivalent of `0x00403004`.

00000000010000000011000000000100 ? ✓ 100%

Q2. Following the virtual address partitioning, what is the corresponding VPN1? Write the entire 10-bit string.

0000000001 ? ✓ 100%

Q3. What is the corresponding VPN2? Write the entire 10-bit string.

0000000011 ? ✓ 100%

Q4. What is the corresponding offset? Write the entire 12-bit string.

000000000100 ? ✓ 100%

Now, we can begin the translation. For Program 1, we are given the PTBR value of `0x4000`. This is pointing to the L1 page table for Program 1. Looking at the contents of the physical memory (from the provided text file), we navigate to the address `0x4000`. We are already looking at the L1 page table entries. Remember that page table entries are 4-bytes long. Given the VPN1 that we have acquired earlier, we look for the *nth* page table entry (where n is what VPN1 is) starting from the address pointed to by the PTBR. Remember that VPN1 indexes the L1 page table. For Program 1, we see that the corresponding L1 page table entry is `0xDEADBEEF`.

*Additional note: What we are doing mathematically is PTBR + VPN1*sizeof(PTE). PTBR is the starting point, VPN1 is the index of the array where each array element is sizeof(PTE) bytes long.*

Q5. What is the corresponding L1 page table entry for **Program 2**? There should be 8 hex digits because the page table entry is 4 bytes.

0x 88C4B640 ? ✓ 100%

Since the L1 page table entry for Program 1 is `0xDEADBEEF`, we check the most significant bit (leftmost bit) to see if the entry is valid. In this case, it is. This means that there is a valid PPN translation and no page fault will occur.

Earlier, it was illustrated that the last 4 bits of the page table entry would correspond to the PPN that will point to the next level page table. Therefore, the last 4 bits of this page table entry is `0xF`. It was also illustrated earlier that the 16-bit physical address is formatted with the 4-bit PPN as the most significant bits. Therefore, the next level page table, which is the L2 page table, for Program 1 will start at address `0xF000`.

Q6. Where is the starting address of the L2 page table for **Program 2**? There should be 4 hex digits because the physical memory address is 16 bits.

0x 0000 ? ✓ 100%

We basically repeat the same steps earlier, but this time, we are now in the L2 page table and we will be using VPN2 as the index to this page table. For Program 1, the L2 page table starts at `0xF000`. Given the VPN2 that we have acquired earlier, we look for the *nth* page table entry (where n is what VPN2 is) starting from this address. For Program 1, we see that the corresponding L2 page table entry for Program 1 is `0xB0BACAFE`.

*Additional note: What we are doing mathematically is (PPN_fromL1 << 12) + VPN2*sizeof(PTE). The PPN we acquired from the L1 page table is shifted 12 times to the left so that the PPN becomes the most significant bits of the 16-bit physical address (as drawn earlier). VPN2 is the index of the array where each array element is sizeof(PTE) bytes long.*

Q7. What is the corresponding L2 page table entry for **Program 2**? There should be 8 hex digits because the page table entry is 4 bytes.

0x AE99D5C2 ? ✓ 100%

Since the L2 page table entry for Program 1 is `0xB0BACAFE`, we check the most significant bit (leftmost bit) to see if the entry is valid. In this case, it is. This means that there is a valid PPN translation and no page fault will occur.

Now, we check the last 4 bits of this page table entry. Since we are on the last level of page table hierarchy, the last 4 bits will be the PPN pointing to the page where the actual data that the program needs is located. Since the last 4 bits of this L2 page table entry is `0xE`, then the data page for Program 1 will start at address `0xE000`. We have to shift the PPN to the upper 4 bits of the physical address again, just like how we did it earlier, because PPN comprise the most significant bits of the physical address.

Q8. Where is the starting address of the data page for **Program 2**? There should be 4 hex digits because the physical memory address is 16 bits.

0x 2000 ? ✓ 100%

Finally, we are at the starting address of the data page for Program 1, `0xE000`. Now, we use the offset bits we acquired earlier to find the actual data that we are looking for. We note that the physical memory is byte addressable. Therefore, for every increment of the page offset, we move by 1 byte. Additionally, Program 1 is doing a load word instruction. Thus, it will load a 4-byte data starting from the address pointed to by the offset. And thus, we finally found the physical address translation of the Program 1 virtual address `0x00403004`. The 16-bit physical address is `0xE004` and the word that will be loaded on register t1 (for Program 1) is `0xADE1B055`.

Additional note: Technically just copying the 12-bit offset from the virtual address at this point. The physical address `0xE004` is the concatenation of the PPN = 0xE (from the page table entry at the L2 page table) and the offset = 0x004. This is consistent to what was presented in the illustration earlier.

Q9. What is the 16-bit physical address translation of the **Program 2** virtual address `0x00403004`?

0x 2004 ? ✓ 100%

Q10. What is the word that will be loaded to register t5 (from **Program 2**)?

0x 61C061C0 ? ✓ 100%

And thus, we have shown that even though both programs tries to access the same address, that address is virtual, and it does not necessarily mean that both of these programs will load the exact same data. Since these two programs are running on a different context (i.e. they are independent of each other), they have different PTBR values. Consequently, the virtual to physical address translation for the programs will be different. That is exactly what is being demonstrated here.

Thank you for walking with me. :)

Try a new variant

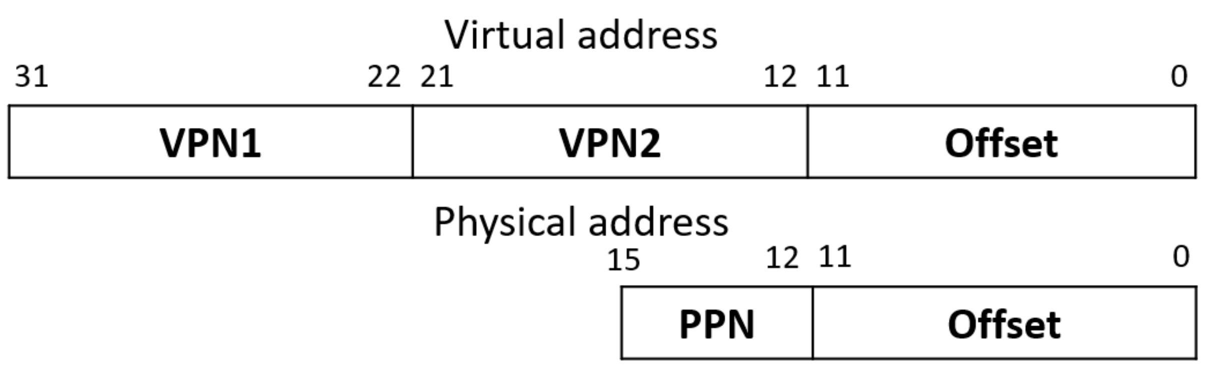
Correct answer

Walk with me.

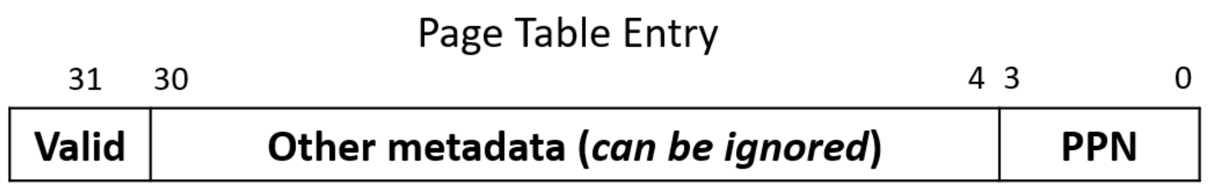
Consider a **64 KiB byte-addressable physical memory** that has the following contents `physical_memory.txt`. Since the physical memory is 64 KiB, there are 16 bits of memory address (ranging from 0x0000 to 0xFFFF) to uniquely identify each byte in the memory.

Consider a machine with a **page size of 4 KiB (12 offset bits)**. It provides a **32-bit virtual memory space** for each program that runs on it.

The machine utilizes a 2-level hierarchical page table that has 20 bits of VPN divided equally between L1 and L2 page tables. Note that 32 bits of virtual address is comprised of the **20-bit VPN + 12-bit offset**. Accordingly, the 16-bit physical address will be comprised of the **4-bit PPN and 12-bit offset**. Explicitly, the partitioning of the virtual and physical addresses is shown below:



Each page table entry (PTE) is 4 bytes, with the corresponding data partitioning as shown below:



The most significant bit is 1 when the entry is valid and the last 4 bits correspond to the PPN pointing to either the next level page table or the data page. Extra metadata bits are included in the PTE but are not relevant for this problem.

Now, suppose we run two different programs on this machine. Coincidentally, we observed that the two programs access the same virtual address as shown below:

```
# Program 1 code
...
lui t0, 0x00403
addi t0, t0, 0x004  # t0 = 0x00403004
lw t1, 0(t0)        # load word from address 0x00403004
...
```

```
# Program 2 code
...
lui s0, 0x00403
addi s0, s0, 0x004  # s0 = 0x00403004
lw t5, 0(s0)        # load word from address 0x00403004
...
```

As you can see, both programs are loading a word from the address `0x00403004`. However, since we now know about virtual memory, we know that even though both programs are pointing at the same address, those addresses are virtual. Thus, when we look at the physical memory, the data that will be loaded by Program 1 can be different from the data that will be loaded by Program 2. For this problem, we will try to figure out the address translation and, eventually, the data that will be loaded by each program.

For this problem, we are given the page table base register (PTBR) for the two running programs. **For Program 1, PTBR = 0x4000. For Program 2, PTBR = 0x6000**. We will need these values as we do the address translation (also known as the page table walk). The PTBR basically points to the physical memory address where we will start the translation. Follow the story guide (translation for Program 1 will be done for you) and answer the questions (for the translation for Program 2) accordingly.

We first partition the virtual address `0x00403004` into the VPN1-VPN2-offset fields. We first write out the entire 32 bit address and then partition accordingly.

For the following 4 questions, write the answers in binary without the 0b prefix and no spaces.

Q1. Write out the 32-bit equivalent of `0x00403004`.

`00000000010000000011000000000100`

Q2. Following the virtual address partitioning, what is the corresponding VPN1? Write the entire 10-bit string.

`0000000001`

Q3. What is the corresponding VPN2? Write the entire 10-bit string.

`0000000011`

Q4. What is the corresponding offset? Write the entire 12-bit string.

`000000000100`

Now, we can begin the translation. For Program 1, we are given the PTBR value of `0x4000`. This is pointing to the L1 page table for Program 1. Looking at the contents of the physical memory (from the provided text file), we navigate to the address `0x4000`. We are already looking at the L1 page table entries. Remember that page table entries are 4-bytes long. Given the VPN1 that we have acquired earlier, we look for the *nth* page table entry (where n is what VPN1 is) starting from the address pointed to by the PTBR. Remember that VPN1 indexes the L1 page table. For Program 1, we see that the corresponding L1 page table entry is `0xDEADBEEF`.

*Additional note: What we are doing mathematically is PTBR + VPN1*sizeof(PTE). PTBR is the starting point, VPN1 is the index of the array where each array element is sizeof(PTE) bytes long.*

Q5. What is the corresponding L1 page table entry for **Program 2**? There should be 8 hex digits because the page table entry is 4 bytes.

0x `88C4B640`

Since the L1 page table entry for Program 1 is `0xDEADBEEF`, we check the most significant bit (leftmost bit) to see if the entry is valid. In this case, it is. This means that there is a valid PPN translation and no page fault will occur.

Earlier, it was illustrated that the last 4 bits of the page table entry would correspond to the PPN that will point to the next level page table. Therefore, the last 4 bits of this page table entry is `0xF`. It was also illustrated earlier that the 16-bit physical address is formatted with the 4-bit PPN as the most significant bits. Therefore, the next level page table, which is the L2 page table, for Program 1 will start at address `0xF000`.

Q6. Where is the starting address of the L2 page table for **Program 2**? There should be 4 hex digits because the physical memory address is 16 bits.

0x `0000`

We basically repeat the same steps earlier, but this time, we are now in the L2 page table and we will be using VPN2 as the index to this page table. For Program 1, the L2 page table starts at `0xF000`. Given the VPN2 that we have acquired earlier, we look for the *nth* page table entry (where n is what VPN2 is) starting from this address. For Program 1, we see that the corresponding L2 page table entry for Program 1 is `0xB0BACAFE`.

*Additional note: What we are doing mathematically is (PPN_fromL1 << 12) + VPN2*sizeof(PTE). The PPN we acquired from the L1 page table is shifted 12 times to the left so that the PPN becomes the most significant bits of the 16-bit physical address (as drawn earlier). VPN2 is the index of the array where each array element is sizeof(PTE) bytes long.*

Q7. What is the corresponding L2 page table entry for **Program 2**? There should be 8 hex digits because the page table entry is 4 bytes.

0x `AE99D5C2`

Since the L2 page table entry for Program 1 is `0xB0BACAFE`, we check the most significant bit (leftmost bit) to see if the entry is valid. In this case, it is. This means that there is a valid PPN translation and no page fault will occur.

Now, we check the last 4 bits of this page table entry. Since we are on the last level of page table hierarchy, the last 4 bits will be the PPN pointing to the page where the actual data that the program needs is located. Since the last 4 bits of this L2 page table entry is `0xE`, then the data page for Program 1 will start at address `0xE000`. We have to shift the PPN to the upper 4 bits of the physical address again, just like how we did it earlier, because PPN comprise the most significant bits of the physical address.

Q8. Where is the starting address of the data page for **Program 2**? There should be 4 hex digits because the physical memory address is 16 bits.

0x `2000`

Finally, we are at the starting address of the data page for Program 1, `0xE000`. Now, we use the offset bits we acquired earlier to find the actual data that we are looking for. We note that the physical memory is byte addressable. Therefore, for every increment of the page offset, we move by 1 byte. Additionally, Program 1 is doing a load word instruction. Thus, it will load a 4-byte data starting from the address pointed to by the offset. And thus, we finally found the physical address translation of the Program 1 virtual address `0x00403004`. The 16-bit physical address is `0xE004` and the word that will be loaded on register t1 (for Program 1) is `0xADE1B055`.

Additional note: Technically just copying the 12-bit offset from the virtual address at this point. The physical address `0xE004` is the concatenation of the PPN = 0xE (from the page table entry at the L2 page table) and the offset = 0x004. This is consistent to what was presented in the illustration earlier.

Q9. What is the 16-bit physical address translation of the **Program 2** virtual address `0x00403004`?

0x `2004`

Q10. What is the word that will be loaded to register t5 (from **Program 2**)?

0x `61C061C0`

And thus, we have shown that even though both programs tries to access the same address, that address is virtual, and it does not necessarily mean that both of these programs will load the exact same data. Since these two programs are running on a different context (i.e. they are independent of each other), they have different PTBR values. Consequently, the virtual to physical address translation for the programs will be different. That is exactly what is being demonstrated here.

Thank you for walking with me. :)

Submitted answer 3 correct: 100%
Submitted at 2022-11-20 04:11:29 (PST)
show

Submitted answer 2 partially correct: 90%
Submitted at 2022-11-20 04:11:10 (PST)
show

Submitted answer 1 partially correct: 80%
Submitted at 2022-11-20 04:10:06 (PST)
show