C Declarations

char** var[];

Declare var as array of pointer to a pointer to a characters.

int* a, var;

Var is an integer, since a is the pointer to an int.

int var(int);

Var is a function which takes 1 integer and returns an integer.

int (*var)(int);

Var is a pointer to a function which takes 1 integer and returns an integer.

void (*var)();

Var is a pointer to a function which takes anything and returns nothing

double *(*var)(double);

Var is a pointer to a function which takes 1 double and returns a pointer to a double

double var[4];

Var is an array of 4 double precision floating point numbers

long (*var)(int, int);

Var is a pointer to a function which takes two integer arguments and returns a long integer.

void (*var)(void*, int);

Var is a pointer to a function which takes 2 arguments, 1 is an integer and the other is pointer to anything or a generic pointer and returns nothing

int *(*(*var)(int*))(void);

Var is a pointer to a function which takes 1 argument which a pointer to an integer and it returns a pointer to a function which takes no arguments and returns a pointer to an integer.

void *(*var)(void *(*)(void*), void*(*)(void*),void*);

Var is a pointer to a function which returns a generic pointer which takes three arguments, the first and the second are both a pointer to a function which takes a generic pointer and returns a generic pointer and the third one is a generic pointer.

int (*(*var[3])())(void (*)());

Var is an array of size 3 which contains function pointers which takes anything and returns a pointer to a function which 1 argument which is a pointer to a function which takes anything and returns nothing, and returns an integer.

char *(*var)(void*(*)(int), int(*)(void))

Var is a pointer to a function which returns a pointer to a character and takes two arguments. The first is a pointer to a function which takes an integer arg and returns a generic pointer. The second is a pointer which takes no arguments and returns an integer int (*var) ();

Var is a pointer to a function which returns an integer and takes anything.

int* (*var)(void)

Var is a pointer to a function which returns a pointer to an integer and takes no arguments.

int* (*var)(int, int(*)(int))

Var is a pointer to a function which returns a pointer to an integer which takes two arguments. The first is an integer, the second is a pointer to a function which returns an integer and takes one argument which is an integer.

Var is a pointer to an array of strings

char *var[]

Var is a pointer to a function, this function should return a function pointer to a function which takes two ints and returns an int. var should take two parameters each of which are the same type as its return value. (Hint use a typedef)

```
typedef int (*funct)(int, int);
funct (*var)(funct, funct);
```

Var is a pointer to a function which takes a single argument (which is an integer) and returns a pointer to a function which takes no arguments and returns an integer int (*(*var)(int))(void);

OR

```
typedef int (*FnType)(void);
FnType (*var)(int);
```

Var is an array of 10 pointers to functions, each of which takes a single argument (which is a pointer to a function which takes an integer argument and returns a generic pointer) and returns a pointer to a function (of the same type as its argument

void*(*(*var[10])(void*(*)(int)))(int)

For memory reads Q:

1. Work out which page address falls in: Divide address by page size.
2. Reads to find per page level physical address, 1 to write to TLB, 1 if already in TLB.

For largest amount of memory page table for a single process can occupy Q:

1. Work out total number of pages. Number of addresses divide by page size. Eg. 2^32 / 2^12 = 2^20 for a 32 bit address size and 4KB page size.
2. Work out how much each page can hold. Page size divide by entry size. Eg. 4KB/ 4B = 1024 = 2^10.
3. Number of page tables is Number of pages, divide by entries in each page. Eg. 2^20 / 2^10 = 2^10 = 1024.
4. Second level table size is then 1024 x 4KB = 4096KB
5. Add size of first level table = 4KB
6. Answer is 4100KB maximum amount of a memory a single process can occupy in memory.

Mapping virtual to physical address questions.

1. Note what page frame, page 0 maps to, this is the physical ram address of the page frame.
2. Split each virtual address into pagenumber and offset. The page number is mapped to a frame number, Multiplied by page size gives physical address, then add offset of virtual address to gain physical address of virtual address.
3. Eg. Virtual address 500 = 500/4096 ; 500%4096 = 0 ; 500 = page; offset
4. So page is 0, offset 500, so address is page frame * page size + page offset.
5. Eg. If page 0 is mapped to frame 200, then 0; 500 maps to 200 * 4096 (4KB page size) + 500 = 819700. For virtual address 8200 = 2 ; 8. If page 2 maps to frame 22, then 8200 maps to 22 * 4096 (4KB page size) + 8 = 90120.
6. If page is ‘invalid’ a seg fault will occur.

Memory Accesses

Submitted by joeladdison on Tue, 09/09/2014 - 14:15

There are a few things to consider when calculating how many memory accesses are required for the CPU to access virtual addresses. First, what type of caching is happening during this memory access? How is the TLB being used? You should assume that the TLB is in use, but if other caching is not specified, then assume that there is none.

Q: The CPU accesses the following (base 10) virtual addresses in sequence: 45055, 45056, 8392710, 45090, 8392714, 45091. How many accesses to memory are required?
(2011 Mid semester exam, Q13)

Relevant information: Page size is 4KB (4096B). Use a 2 level page table. L1 and L2 cache are considered part of memory.

We are going to assume that the TLB is in use, and is currently empty. Also assume that there is no caching apart from the TLB.

The first step is to work out which pages the addresses fall in. From the exam, we are told that a page is 4KB (4096B). Therefore, to get the page that contains a virtual address, we should do:

$$page = \frac{virtual\ address}{page\ size}$$

Note: This must use integer division, as we want the page. The fraction would correspond to the offset from the start of the page.

Pages of Virtual Memory addresses

Virtual Memory Address	Page
45055	10
45056	11
8392710	2049
45090	11
8392714	2049
45091	11

Following this, we need to find the physical address of each virtual address. While we are doing this, we will calculate how many memory accesses are required.

1. Virtual address 45055 (page 10)
   1. There are two levels to the page table, so we need to go to the L1 page first. This tells us which L2 page contains the address for page 10. **Memory accesses: 1**
   2. We go to the L2 page to find the physical address of our virtual address. **Memory accesses: 2**
   3. Now that we have the physical address, we access it. This map of page to frame is now stored in the TLB. **Memory accesses: 3**
2. Virtual address 45056 (page 11)
   1. Go to L1 page to find which L2 page contains the address for page 11. **Memory accesses: 4**
   2. Go to the L2 page to find the physical address of our virtual address. **Memory accesses: 5**
   3. Access the frame containing our physical address. This map of page to frame is now stored in the TLB. **Memory accesses: 6**
3. Virtual address 8392710 (page 2049)
   1. Go to L1 page to find which L2 page contains the address for page 11. **Memory accesses: 7**
   2. Go to L2 page to find the physical address of our virtual address. **Memory accesses: 8**
   3. Access the frame containing our physical address. This map of page to frame is now stored in the TLB. **Memory accesses: 9**
4. Virtual address 45090 (page 11)
   1. We have already found where page 11 is stored, so the map of page 11 to physical frame is stored in the TLB. We can go straight to this frame to access the physical address, skipping the L1 and L2 page table memory accesses. **Memory accesses: 10**
5. Virtual address 8392714 (page 2049)
   1. We have already found where page 2049 is stored, so the map of page 2049 to physical frame is stored in the TLB. We can go straight to this frame to access the physical address, skipping the L1 and L2 page table memory accesses. **Memory accesses: 11**
6. Virtual address 45091 (page 11)
   1. We have already found where page 11 is stored, so the map of page 11 to physical frame is stored in the TLB. We can go straight to this frame to access the physical address, skipping the L1 and L2 page table memory accesses. **Memory accesses: 12**

From this working, we can see that we would need a total of 12 memory accesses to access the given virtual memory addresses in sequence.

Subnets

Submitted by sshaw on Wed, 09/10/2014 - 23:04
Note: this deals entirely with IPv4, IPv6 isn't covered.

Subnets (sub-networks) are the divisions of a network into smaller networks. A subnet has a size, the number of hosts the subnet can address. The size of a subnet is determined by its **subnet mask**, which is also how we work out what the network address is of any given IP address.

Broadcast and network addresses

There are two special addresses in every subnet. The highest address in a subnet is the broadcast address. Any message sent to this address will go to all hosts on the subnet. The lowest address is the address of the network itself. Neither of these addresses will (in this course) be assigned to hosts on the network.

CIDR notation for subnets

$\mathrm{a.b.c.d/x}$ where $a, b, c, d \in 0..255; x \in 0..32$
$\mathrm{a.b.c.d}$ is the **network address** and x is the number of leading 1s in the **subnet mask**
100.123.42.0/24 indicates a network address of 100.123.42.0 and a mask of 255.255.255.0, the mask being equivalent to the binary value 1111 1111 1111 1111 1111 1111 0000 0000 (note the 24 ones in the mask matches the CIDR notation's /24)

Using mask and IP to generate network and host address

Bob has an IP of $192.168.1.43$ and subnet mask of $255.255.255.240$, what is his network address and what is his host address? First, convert the IP address and mask into binary:

$$i = 192.168.1.43 \implies 1100\ 0000\ 1010\ 1000\ 0000\ 0001\ 0010\ 1011$$

$$m = 255.255.255.240 \implies 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0000$$

The network address is the result of a bitwise and ($\&$) of IP and mask:

$$i \;\&\; m \implies 1100\ 0000\ 1010\ 1000\ 0000\ 0001\ 0010\ 0000 \implies 192.168.1.32$$

The host address is the result of a bitwise and of IP and the negation ($\neg$) of the mask (negating flips the bits, remember):

$$i \;\&\; \neg m \implies 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011 \implies 0.0.0.11$$

Finding the smallest subnet containing specific addresses

If two IP addresses with the same subnet mask do not have the same network address, they are on different networks. E.g. Using the subnet mask $255.255.255.240$, the IPs $192.168.0.7$ and $192.168.0.17$ aren't on matching subnets (the first is on the network $192.168.0.0$ and the second is on the network $192.168.0.16$). You can see from this (and the mask) that there are 16 addresses in each subnet of this size and each of the subnets has its network address as a multiple of 16 ($\ldots 0.16, \ldots 0.32$, etc).

From that fact we can find the smallest subnet that any two (or more) hosts belong to. Lets use $10.174.168.54$ and $10.174.163.129$ (if using more hosts, just take the lowest address and the highest address). Our subnet mask is $???.???.???.???$

Reading from left octet to right octet, find the **first** they differ in. In this case, that's the third octet (163 vs 168). We can now fill three of our subnet octets since the mask is always a block of 1s followed by a block of 0s. Our subnet mask is now $255.255.???.000$

Find the distance between the two values in the octet that changes. For a subnet to contain both 163 and 168, it must cover a range of **at least** 6 $(163, 164, 165, 166, 167, 168)$. Since we can tell from above that the size of a network is always a power of 2, that means that the range is **at least** 8 and the initial guess of our missing octet will use a size of 8, thus our mask becomes $255.255.248.0$ (the third octect being 11111000 to allow for a range of 8 in those zeroed bits).

Using that initial guess, check the addresses have the same network address. If they are the same, you have found the smallest subnet containing the addresses. If they are not the same, you need to go a step further. in this case they will be $10.174.160.0$ and $10.174.168.0$, which are different networks. Until they are on the same network, change the last 1 bit in your mask to 0, doubling the size of the subnet. This changes the mask to $255.255.240.0$, which when used to calculate the network addresses of those IPs results in both the network addresses being $10.174.160.0$.

NOTE: If an IP given is also the broadcast or network address of your smallest subnet, you'll need to make it larger. For the purposes of the course, assume that those addresses will never be assigned to hosts.

Map Virtual to Physical

Submitted by joeladdison on Tue, 09/09/2014 - 16:12

Mapping a virtual address to a physical address is relatively easy once the correct formulas are known. There are also some pieces of information that you will need, such as the page size and a page table.

There are four steps to converting a virtual address to a physical address:

1. Calculate the page (Note: integer division is used for this):

$$page = \frac{virtual\,address}{page\;size}$$

2. Calculate the page offset (Note: $\%$ is the modulus operator):

$$offset = virtual\,address\;\%\;page\;size$$

3. Find the frame by mapping the page to a frame via the page table
4. Calculate the physical address:

$$physical\,address = frame * page\;size + offset$$

It is not always possible to get a physical address from a virtual address. If the page table has a page listed as *invalid*, then a segmentation fault would occur, instead of accessing the physical address.

Q: Given the following page table, map the (base 10) virtual addresses to physical addresses.
Virtual Addresses: 500, 8200, 102400
(2011 Mid semester exam, Q15)

Relevant information: Page size is 4KB (4096B)

Page	Frame
0	200
1	201
2	22
3	invalid
...	invalid
25	24
...	invalid

For each of these virtual addresses, we need to calculate the physical address using the process outlined above.

Virtual address 500:

$$page = 500/4096 = 0$$

$$offset = 500\;\%\;4096 = 500$$

From our page table, we can see that page 0 maps to frame 200.

$$physical\,address = 200 * 4096 + 500 = 819700$$

Virtual address 8200:

$$page = 8200/4096 = 2$$

$$offset = 8200\;\%\;4096 = 8$$

From our page table, we can see that page 2 maps to frame 22.

$$physical\,address = 22 * 4096 + 8 = 90120$$

Virtual address 102400:

$$page = 102400/4096 = 25$$

$$offset = 102400\;\%\;4096 = 0$$

From our page table, we can see that page 25 maps to frame 24.

$$physical\,address = 24 * 4096 + 0 = 98304$$

From this, we can see that it is possible to get a physical address for every virtual address we were asked to convert. This means that there were no segmentation faults.

It is possible for a page fault to occur, but we have not been given enough information to determine whether this is the case.