**Question 3**

3.a

===

"32bit addressing capability" => virtual address on 32bits

=> 2^32 addressable bytes = 4GB of virtual memory

the pages are 4kb => 1 million pages in these 4Gb

virtual addresses on 32bits => 4 bytes to store an address

3.b

===

each table entry (PTE) contains 48+3+1+1+11 bits = 64bits = 8 bytes

one ETP per page => 1 million in total in 8Mo in total

in a linear table, you have to allocate everything because you need

something... to tell if a virtual page is valid for each of the 1

million virtual pages

3.c

===

valid page ? => the VALID bit

used in the future? => locality principle says it's the pages

recently used, and the 11bits of AGE give us this info

(hardware puts last access times on it)

present page ? => the physical frame number (PFN)

as there is no bit present, it has to be encoded somewhere.

we reserve a special NFP value to mean "not present".

e.g. 0xffffffff (the max NFP)

that means we'll only be able to use 248-1 physical pages instead of

of 2^48, it's no big deal.

or we could remove a bit of PFN and use it as a present bit.

that would be 247 usable physical pages.

physical address = 48bits physical frame number + 12 bits in the page

= 60 bits => 10^18 bytes of RAM

@phys > @virt => will take a lot of process to use all the RAM

this case is rare these days because we've moved to 64bits @virt but we're still

is still only 1-100GB on most machines.

on the other hand, around the years 2000-2005, we had @virt = 32bits

and sometimes more than 1GB of physical RAM, @virt ~= @phys

(see @phys > @virt if we had an EAP processor: @phys = 36bits)

3.d

===

The mmu doesn't have much of a memory, so no. In physical memory yes,

but 8Mb per process is a lot. For 200 processes, that's 1.6GB

already...

3.e

===

in an array of 4kb at levels 1 and 2, 1024 pointers (4 bytes each) can

be placed 1024 = 2^10, so we use 10 bits of the address to choose the

pointer to follow in these levels

on level 3, 512 PTEs (8 bytes each) are put in

512 = 2^9, so use 9 bits of the address to choose the PTE to use.

at the end of the address, there is the offset in a page

4kb = 2^12 => 12 bits

so the address consists of

10 bits for level 1 to choose the pointer to follow to find the level 2 displayboard

10 bits for level 2 to choose the pointer to follow to find the level 3 displayboard

9 bits for level 3 to select the PTE corresponding to the target page

12 bits to add the offset in the target page

=> 41 bits total

but the archi limits virtual addresses to 32 bits,

so we can't store 41 in there

so we have to reduce the number of bits somewhere,

i.e. reducing the number of pointers or PTEs in a level

(and therefore a painting that will actually be 4kb smaller)

the simplest way is to reduce level 1 because that's one level of only

table to be reduced

in short, we remove 9 bits at the first level to reduce from 41 to 32

in total. so 2^1 pointers instead of 2^10 in the first array

3.f

===

in a linear table, we had to allocate everything. in a 3-level table,

the first level should be allocated, but not necessarily the following

ones the 1st level can say that a whole sub-part is invalid, it allows

not to allocate the sub-tables for this subpart

in the end, the necessary level 3 tables are allocated for the ETPs

describing the valid virtual pages, then the level 2 tables which

contain the pointers to these valid level 3 arrays, and the entire

level 1 array

That's 1 table (1 page) for level 1, and >=1 for levels 2 and 3.

if we use all the virtual pages of the process, we need all the PTEs

possible to describe them, so all level 3 and level 2 tables. in this

case, the set of level 3 tables is identical to the linear table to

the previous questions.

Now, the details of the specific cases:

1kb:

in general, it's in a single page of 4kb.

so you need 1 PTE for this page

so you need 1 level 3 table to store this PTE

so you need 1 level 2 pointer

so you need 1 level 2 array to store this pointer.

so you need a level 1 pointer

so you need 1 level 1 array to store this pointer

=> so 3 paintings of 4kb in total = 12kb

if the 1kb straddles 2 pages, 2 PTE are needed, etc...

100kb:

25 pages, so 25 PTE

if the pages are contiguous, the TEPs are probably in the same level 3 table (each table can contain 512 TEPs)

so a level 3 board and then the same as above.

=> 12kb also

10Mo:

2500 pages, so 2500 PTE

there are 512 ETPs per level 3 board, so 5 level 3 boards are required

so you need 5 level 2 pointers, so at least one level 2 array to store them (each array can contain 1024 pointers)

so 1 level 1 table

=> (5+1+1)4kb = 28kb

1GB:

250000 pages, so 250000 PTE

so 500 level 3 tables

so 1 level 2 table

so 1 level 1 table

=> (500+1+1)\*4kb = 2MB and qs

All (4GB):

1000000 pages, 1000000 PTE

so 2000 table 3

so 2 table 2

so 1 table 1

=> (2000+2+1)\*4kb = 8MB and qs

if the table is linear (only level), it's always 8MB exactly, so any

case... so hierarchical always better, unless very full (a little

less well)

1,000 little bits scattered around:

1000 PTE

if they are scattered, you may need a different level 1 table for each page (so 1000 in total)

if the level 1 panels are scattered, many level 2 panels may also be needed (i.e. max. 2 in total)

and always a level one board

=> (1000+2+1)\*4kb = 4MB and qs

the worst would be 2048 songs every 2Mb

to verify the calculations we can use the fact that

\* a level 3 table covers 512 PTE x 4ko = 2Mo

\* then level 2 covers 1024 pointers to level 3 = 1024\*2MB

**Question 4**

4.a

===

!VALID

VALID + RO + frame number

VALID + WO + physical frame number + DIRTY

VALID + RO (instead of RW) + frame number

VALID + not valid number (like 0 or -1 instead, depending on the spec of the proc)

we can detail the cases after COW as well:

when the son writes, his table goes to VALID + RW + dirty.

when the father also changes later, he gets a COW exception. but

there's nothing you can do about it because he's the only one using

that copy. so it just switches back to RW+dirty without duplicating.

4.b

===

handling pagefault exception in case of default page VALID but not

present (invalid PFN) without frame number the information is stored

in the page table, but not all of it. the exception handler looks at

the VMA structures (the equivalent of /proc/self/maps) for find out in

which file the data to be loaded is located, and at which location it

allocates a page (if there is no free page, he may have to free a page

by swapping something else) it asks the file system to load the data

in the allocated page he's putting the process to sleep in the

meantime because it's a long process... (all this is in the

privileged mode exception handler in the kernel)

when the hard drive's done, it does a shutdown... the interrupt

handler fills the page table and then restarts the process. (this is

done in the kernel's privileged mode interrupt handler)

when the process sleeps during this I/O, it cannot be woken up by a signal.

(like if alarm() or setitimer() were called before)

because the processor could access the page that is being loaded,

that would be fucked up:

\* or the table of pages was updated before loading and the loading is not finished.

=> invalid data

\* or the table of pages is not yet updated, it triggers another pagefault.

=> It could get boring to deal with in the OS if two ends of the same process are

asleep at the same time while waiting for the disk writing

in short, the signal won't wake up right away, it's said that the process is in uninterrupted sleep.

if there is no free page just before filling, one is freed from another process.

you may have to record it on the disk first (if it is dirty or anonymous)

so there'll be another I/O before the first one.

**Question 5**

5.a

===

only valid entries can be set, so you don't need the valid bit.

Putting in invalid entries would waste space (already small and

expensive) in the BLT. if we did, it would trigger an "invalid

address" exception to the translation, so might as well trigger a "tlb

miss" exception instead.

you need @virt, guard bits and @phys Basically, there's only what the

hardware needs to translate and verify access. the rest (used by the

OS) doesn't need to be there since the BLT won't know how to use it.

(Saving space)

***The BLT is fast because it is small but expensive, so it saves space.***

when there's no mmu, the dirty bit is handled by the OS: the first

time a page is accessed in W, the BLT makes an exception, the OS

handler sets the dirty bit. when the OS saves the changes, it removes

the dirty bit and hop some archis like mips have a dirty bit in the

tlb because they copy the entire TEP. but the stuff never touches it.

exception "TLB miss", the OS translated by hand if translation OK, it

puts in BLT and restarts the instruction if translation not OK, stain

killed (segfault) si cow, translation OK, duplicate page, updated BLT

with new protection

if MMU, the TLB complains to the MMU and eventually the MMU complains

to the OS if translation or access error (default of normal page) if

no MMU, the BLT complains directly to the OS, which will browse the

page table to fill the BLT, and while browsing, it verifies the

protections, detects invalid accesses, duplicates the copy-on-write

pages, etc.

summary of the translation process:

if TLB finds and valid and rights OK, it's finished.

if BLT doesn't find

if MMU, it scrolls across the table

if she finds

so straight OK, it's over

otherwise exception (rights not OK)

otherwise exception (invalid)

if no MMU, exception TLB miss

if BLT sees a rights issue

exception rights issue

5.b

===

instruction to empty it at context change (if not automatic)

invalidation of an entry (swap-out, mmap) add an entry if no mmu

on intel (with mmu), there is invldg which invalidates a virtual

address and global flashes if written in cr3 (or cr4) to change

context between two processes (or two VMs)

on mips (without mmu), there is an instruction to add an entry to the

BLT at the end of the BLT miss exception.

outside of context changes, invalidation is necessary to the munmap

because a valid page becomes invalid. we take the list of virtual

pages and we tell the BLT which virtual pages are fired some procs can

invalidate an interval. Usually they just know how to invalidate a

page or something. We get by with that.

during an mmap, there is no BLT modification because it is invalid

pages (so not in the BLT) that become valid. We will put them in the

BLT automatically at the first access.

5.c

===

Usually there is ***one*** BLT by heart. BLTs ***do not interact*** with each

other at all because the BLT is a ***read-only*** cache, only the OS really

modifies it (it The process changes its mapping when a process changes

its mapping. So no hardware synchro between BLTs.

if a process does munmap(), we have to invalidate the BLT of all

cores. executing this process. If not multithreaded, it's only the

local core. So easy. If multi-threaded, you have to warn the other

hearts, it's expensive. so don't do a lot of mmap/munmap in big

multithreaded thing.

MMUs are also generally independent. There is a particular case where

the BLT is shared, it's hyperthreading. of intel. There's a tag on

each line to tell which thread it's on. matches. So it's like two

independent BLTs, except they're both are fighting a little bit to

share the space in there.