Problem 1

1. Don’t think this is examinable any more but here’s a shot in the dark:
   1. + Final executable contains everything required

- The executable output is usually very large

* 1. BSS contains unititialised data, with all bytes set to 0, whereas the data segment contains the meta-data used in the program such as string values etc.
  2. Main task of the loader is to load the program into main memory and load the values into the data segment?
  3. First thread downs a, blocking the others while it decrements the count, then it ups a back to 1. It then skips over the if statement and tries to down b, making itself wait on b until it is upped.

This happens for each thread, atomically decrementing the count, until the last one which enters the if and ups b.

This unblocks first thread in the waiting queue of b, which ups b again, and starts a chain reaction of threads upping b.

The final value of count is 0, a is 1 (downed N times and upped N times) and b is 1 (downed N times, upped N+1 times).

* 1. ~~If a preemptive scheduler is used, one off the threads blocking on b may get rescheduled and when it begins running, set off the chain reaction of unblocking the remaining threads.~~

~~The preemption will not increment b’s value however so the final value of b will be < 0.~~

Same as i) except ordering of thread execution is different (we can have multiple threads blocked on semaphore a due to preemption, and any number of threads in zone after), and once count = 0, it is possible that multiple threads are still at line 6 (or before), then multiple threads will up(b) on line 8 leading to a higher value of b. b ∊ (1, N)

* 1. If lines 3 & 5 are removed, the threads can concurrently modify the count, which means their critical regions would overlap in this sense. So a decrementation could be skipped. This would mean that the final value of count is indeterminable as we could not be sure if multiple threads attempted to decrement count at the same time. This may therefore cause the value of count to not reach 0. So all threads may end up deadlocked (b = 0) on line 10, blocking b. Equally as in part ii, if we have multiple threads up(b), the value of b can range between b ∊ (1, N) Count ∊ (0, N-1), a = 1 (unused)

c)

i)

The salt has insufficient entropy (24 \* 60 possible values) so it is still feasible to make a rainbow table hashing the passwords with each salt.

ii) Possible answer: could be used to authenticate users, where hashed and salted passwords are stored on-server. when user logs in, system sends a query to the LDAP server which returns whether the credentials are valid. Advantage: password stored on a centralised system where specialised security measures can be deployed; disadvantage: authentication must be done online (but can cache passwords to perform authentication offline, but then hashed + salted passwords have to be cached locally).

Problem 2

a)

Virtual memory address is the memory address that the user programs see. Different users may have different virtual addresses for the same physical address.

Physical memory address is the memory address that actually stores the data in the physical memory.

Two adjacent virtual memory addresses are not necessarily adjacent in the physical address level.

Virtual memory address maps to the physical memory address by the MMU(Memory Management Unit). Assuming we have a TLB miss, then MMU first looks up the page table in the memory, then find the relevant entry in the page table, concatenate the frame address(i.e. physical page) with the offset to form the physical address.

b) In the address space of every process, we keep a section which contains the entire kernel. This section is also always kept in physical memory. This is still off limits to the user process.

This means, to do a system call, we can stay within the same address space. So we do not need to do a context switch for the kernel to access its memory. This improves efficiency, since context switches are expensive, and clear caches (e.g. TLB).

c) Pretty sure this must’ve got taken off

the syllabus??

<https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>

Virtual address begins at 0

frame address page offset

------------------------------------------------------------------------------------------------

| 52 bits | 12 bits |

------------------------------------------------------------------------------------------------

2^52 frames in total? so virtual address ends at frame 2^52

d)

16GB = 2^34 bytes

2^34 > 2^32 so the virtual address space is not large enough for a 16GB file.

~~A possible solution for this is to add another level to the page table structure (hierarchical page table) with k bits to index the previous level, then assuming there are 10 bits for the page offset in a 32 bit architecture, we can have 2^ (k \* (22 - k)) virtual addresses. e.g select k = 10 for 2^120.~~

Alternative

Demand Paging - Only bring into memory when the part of the file is accessed. This allows the file to be paged in and out as necessary (lazy loading). A portion of the file can be mapped into the virtual address space at a time. If the file is accessed in order (start to finish) the effect on performance should be negligible

Or make the memory mapped file word-addressable only -> the 32bit system allows for 4\* 2^32 addresses, is enough for 16 GB + use bit shifts to simulate byte addressing

32 bit table -> 32 bit table using 2 addresses

There is also this: <https://en.wikipedia.org/wiki/Physical_Address_Extension>

but this does require a bigger page table entry.

Dont Memory map 16gb on a 32 bit architecture

Wise words right there.

Alternative