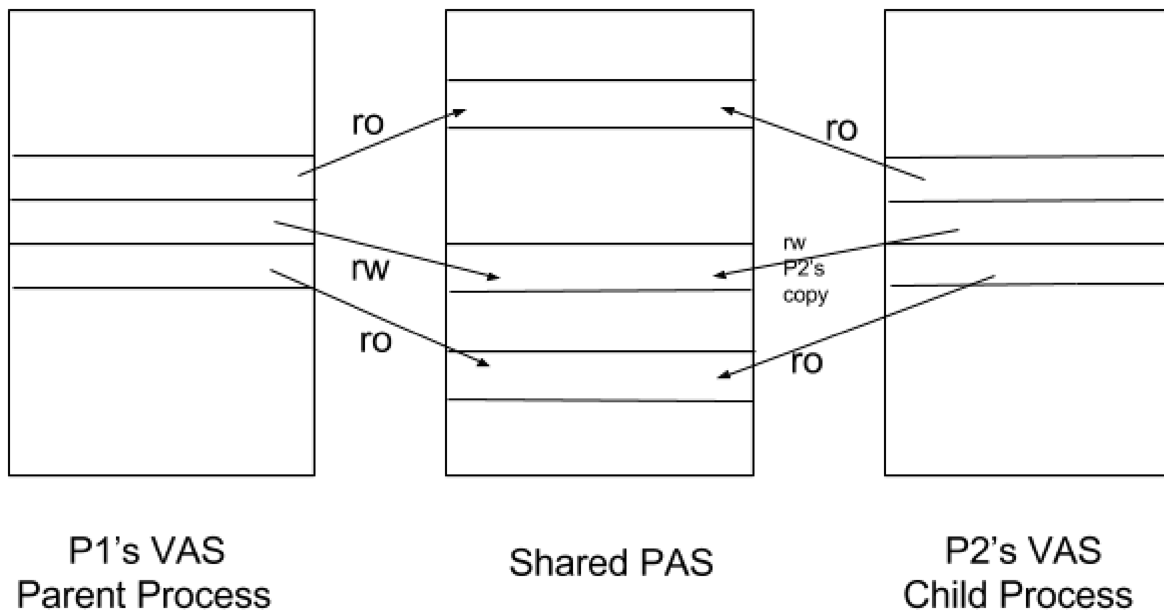
COL331/COL633 Minor2 Exam
Operating Systems
Sem II, 2016-17

Answer all 8 questions Max. Marks: 25

1. True/False. Copy-on-write allows faster implementation of UNIX's fork() system call. Briefly explain. (No marks if incorrect explanation). [2]

Answer: True

Due to copy on write optimization, only some of the pages (that are written by the child process) need to be copied from the parent to the child. All other pages can be shared between the parent and child (including pages that are never accessed, and pages that are accessed in read-only mode, e.g., code/rodata pages).



Shared pages are made read-only.

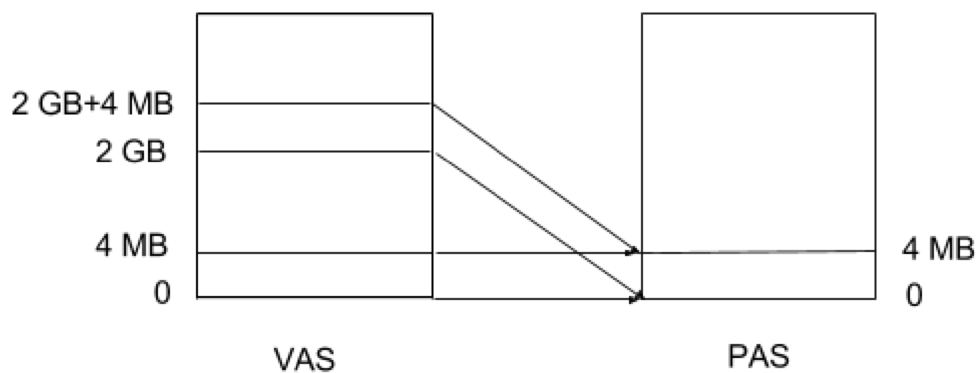
// 0.5 marks if explained COW only.
// 2 marks if explained how COW improves fork() performance

2. True/False. Different virtual addresses in the same page table (belonging to a single process) could point to the same physical memory address. In other words, is it possible for two different virtual addresses VA1 and VA2 (both in the same page table PT) to point to the same physical address PA. Briefly explain when this could happen, with example from real operating systems. (No marks if incorrect example/explanation). [3]

True. A page allocated to the user process, would be mapped both in the kernel side and the user side at the same time in xv6, Linux, Windows (all monolithic kernels). [3 marks if correctly answered]

Another answer (incorrect): In XV6, when the paging is enabled at line 1152, a temporary page table is created with both the 0th and 512th entry of the page table pointing to location 0 in the physical address space. This is because both 0 to 4 MB and 2 GB to (2 GB + 4 MB) in the virtual address space are mapped to the same memory location (0 to 4 MB) in the physical address space.

Hence, it is possible for two virtual addresses VA1 and VA2 to point to the same physical address PA.



This answer is incorrect because this page table is not for a “process”. This page table is only a temporary page table, and processes have not been created yet. Still, award 2 marks for such an answer.

3. True/False. Different virtual addresses in different page tables (belong to two different processes) could point to the the same physical memory address. In other words, is it possible for two different virtual addresses VA1 and VA2, belonging to different page tables, PT1 and PT2 respectively, to point to the same physical address PA. Notice that the virtual addresses VA1 and VA2 must be different. Briefly explain when this could happen, with example from real operating systems. (No marks if incorrect example/explanation).   [3]

Answer: True.

Multiple possible answers:

1. This is possible when two or more processes have shared libraries and hence shared pages.
2. The mapping for the user pages in one process would also be mapped on the kernel side in another process's page table in xv6, Linux, Windows

Full marks if any of these mentioned.

4. Mention two advantages of paging over segmentation, and two advantages of segmentation over paging (briefly). [2]

Answer:

Paging over segmentation -

1. No external fragmentation.
2. Non-contiguous allocation of pages solves the problem of segment outgrowth.
3. Allows demand paging and more efficient swapping.
4. Allows sharing (copy-on-write) optimizations.

Segmentation over paging -

1. Less hardware complexity (lower power consumption, less area on chip).
2. Faster VA -> PA translation.

// 0.5 marks per correct advantage for both paging and segmentation.

5. What are some important differences between the L1 cache and the Translation Lookaside Buffer (TLB)? Focus on the most important differences, starting with their purpose, their organization, and their characteristics. [3]

Answer:
Purpose:

1. TLB - To reduce the number of memory accesses required to do access a memory location using paging scheme of memory management. TLBs store virtual to physical address translation for a subset of user process' pages.
2. L1 Cache - To eliminate the side-effects of speed mismatch between processor and main memory. L1 cache stores a subset of content currently present in the main memory.

Organisation:

1. TLBs are organized as caches, if small, and are usually fully associative for high hit rates. They are generally located on the same chip as processor. They can cache VA->PA mappings end-to-end.
2. L1 cache is also located on the same chip as processor and may be split into instruction and data cache. These can be fully-associative, direct mapped or set associative.

Characteristics:

1. TLBs typically have 12 to 1024 entries, a hit rate of 99+ percent, a miss penalty of 10-100 cycles, and access time of 0.5-1 cycle. TLB hit rates need to be very high, to avoid translation overheads
2. L1 cache are typically 32 KB in size and have an access time of 1 ns, and usually much lower hit rates.
3. On Context Switch : TLB stores VA to PA address translations, hence on every context switch it needs to be flushed whereas L1 Cache(dependent on the architecture design) stores PA mapped values hence need not be flushed.

// 1.5 marks for explaining the purpose of both TLB and L1
// 1 mark for mentioning that TLBs are expected to have higher hit rates
// 0.5 marks for mentioning that TLB is usually fully associative

6. xv6 : What is the significance of lines 2521-2522 in the userinit function? The lines are reproduced below. [3]

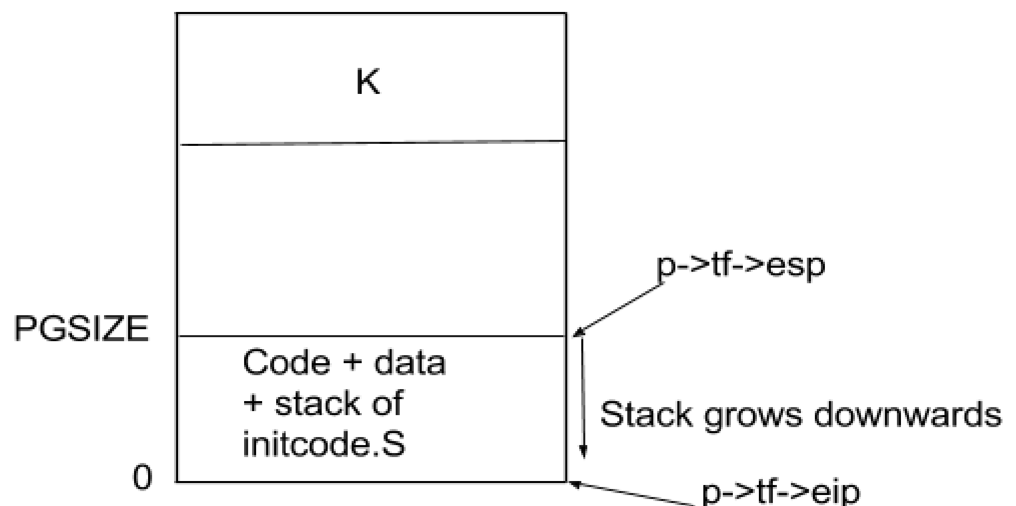
```
2521  p->tf->esp = PGSIZE
2522  p->tf->eip = 0
```

Answer:

At this location, the user-side of the address space is mapped with a single page starting at address 0.

p->tf->esp = PGSIZE sets the esp to point to location marked by the PGSIZE so the stack can now grow downwards.

p->tf->eip = 0 sets the eip to contain the first instruction of the first user process initcode.S. So then the execution of initcode.S which is the first user process to run begins from its first instruction.



// 1.5 marks per line significance explanation

7. xv6 : Explain the function argint(int n, int *ip) on line 3545.
- What is ‘n’ [0.5]
- What is ‘ip’ [0.5]
- Why is it calling fetchint(proc->tf->esp + 4 + 4*n, ip) [2]
- What does fetchint() on line 3516 do? Explain the two checks on line 3519. Why are they needed? Do we need two checks, or would one check suffice? [2]

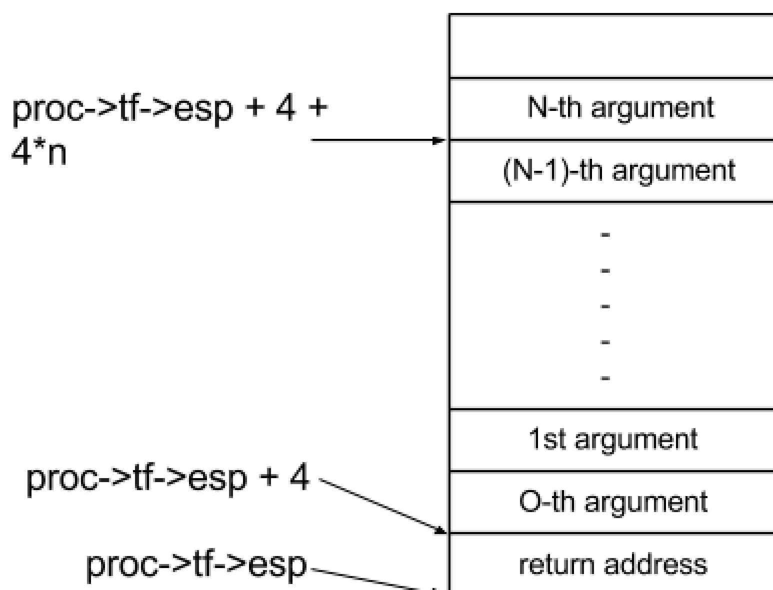
Answer:

1. ‘n’: It is n-th 32-bit system call argument to be fetched.

2. ‘ip’: It is the location where the n-th argument is to be stored, upon function return (it is the return value).

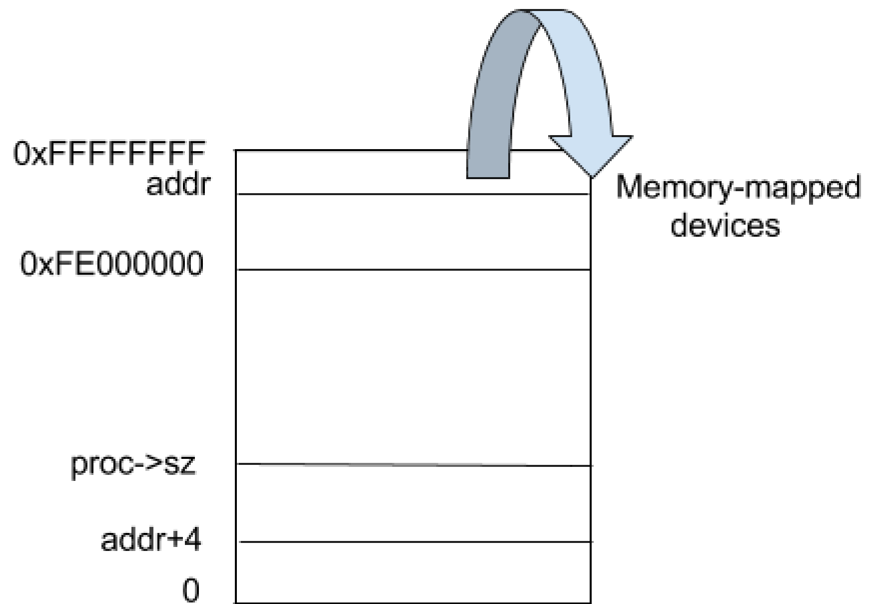
3. fetchint(proc->tf->esp + 4 + 4*n, ip)

On xv6, system call arguments are stored on stack. As we are fetching the n-th argument that was pushed onto the stack, note that arguments are pushed onto the stack in reverse order and each argument being 32-bit consumes 4 byte on the stack, hence the location of n-th argument will be
proc->tf->esp + 4 + 4*n .
Note : Syscall name is passed through proc->tf->eax.



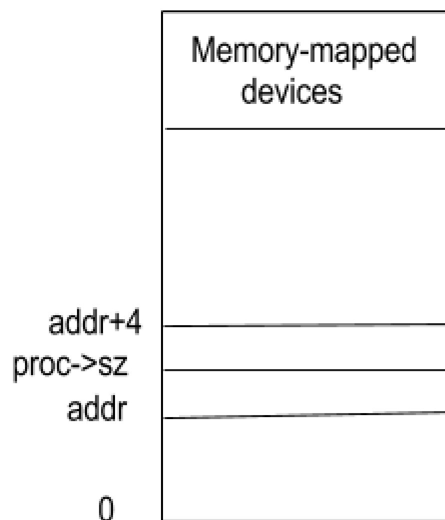
4. fetchint() at line 3516 fetches the integer at stored at address ‘addr’ and stores the content of ‘addr’ at location being pointed to by ‘ip’.

Both the checks at line 3519 are necessary and only one of them will not suffice because there is a possibility of wrap around of addresses as shown in the figure below:



Here addr+4 <= proc->sz but addr >= proc->sz, so if second check is not made wrong location will be dereferenced.

Similarly,



Here addr < proc->sz but addr+4 > proc->sz, so if only first check is made, wrong location would be dereferenced.

Hence, both the checks are necessary.

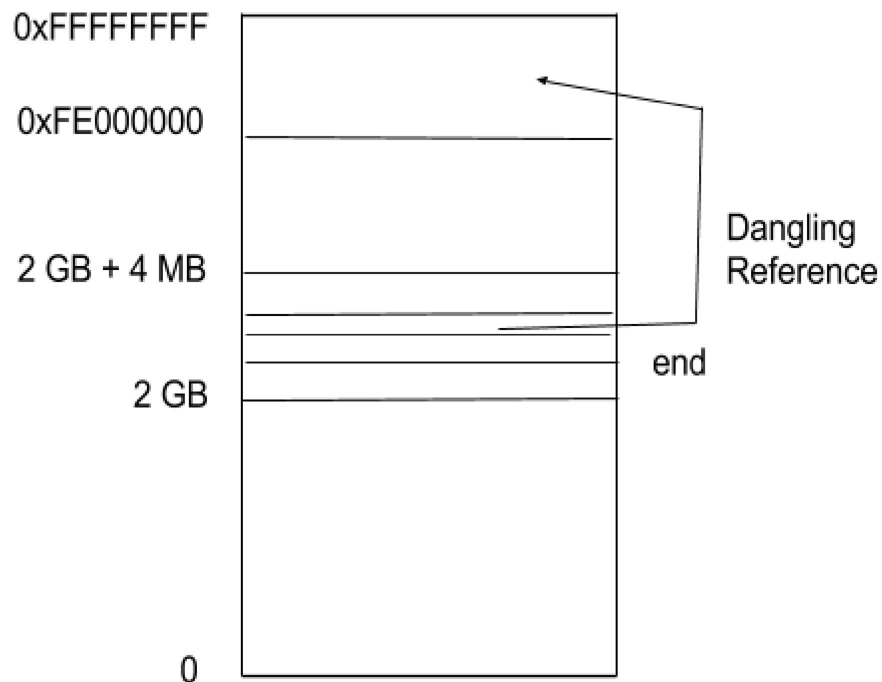
8. xv6: Look at line 3073 in the kfree() function:

```
3072  //Fill with junk to catch dangling refs
3073  memset(v, 1, PGSIZE)
```

The comment says that this call to memset() is supposed to fill the memory region with junk to catch dangling references. What is a dangling reference? Briefly give an example of how this command will help to catch a dangling reference. Will the xv6 kernel run correctly if this line (line 3073) is commented out? [4]

Answer:
A dangling reference is a reference made through an illegal address, i.e., an address that does not point to a reachable/valid memory location.

The kfree() at line 3064 frees the page of physical memory being pointed at by 'v'. This page is then returned when a call to kalloc() is made. A memory location in the page could initially be pointing to at some other memory location, say one in the DEVSPACE reserved for memory mapped devices. A reference to this memory location will then become a dangling reference. [2 marks for these two answers]



Assuming that the xv6 kernel has no bugs, xv6 kernel should work correctly even if the line at 3073 is commented out, because in a correct piece of software, a memory region that has been freed should not be used/de-referenced. [2 marks]