CS347: Operating Systems

Problem Set 3: Solutions

1. (a) Each PTE has frame number (21 bits) and flags (10 bits) $\approx$ 4 bytes. The total number of pages per process is 2^{20}, so total size of inner page table pages is $2^{20} \times 4$ = 4MB. With heirarchical paging, we require one outer page table per process, because one page is sufficient to hold all the PTE of inner page tables. So the total size of page tables of one process is 4MB + 4KB. For 1K process, the total memory consumed by page tables is 4GB + 4MB.

   (b) Each invertaed page table entry has a page number (20 bits) and a PID (10 bits) $\approx$ 4 bytes. The number of inverted page table entries is 2^{21}, i.e., one per frame. So the size of the inverted page table is $2^{21} * 4 = 8$ MB.

2. (a) For 6 bit virtual addresses, and 4 bit page offsets (page size 16 bytes), the most significant 2 bits of a virtual address will represent the page number. So the reference string is 0, 0, 1, 0, 1, 1, 2, 1, 0, 3 (repeated again).

   (b) Page faults with FIFO = 8. Page faults on 0,1,2,3 (replaced 0), 0 (replaced 1), 1 (replaced 2), 2 (replaced 3), 3.

   (c) Page faults with LRU = 6. Page faults on 0, 1, 2, 3 (replaced 2), 2 (replaced 3), 3.

   (d) The optimum algorithm will replace the page least likely to be used in future, and would look like LRU above.

3. (a) fork+allocproc

   (b) fork+allocproc

   (c) exec+walkpgdir

   (d) exec+allocvm

   (e) exec+loaduvm

4. (a) The kernel part of the virtual address space of a process should be large enough to map all physical memory, so $V - U \geq P$. Further, the user part of the virtual address space of a process should fit within the free physical memory that is left after placing the kernel code, so $U \leq P - K$. Putting these two equations together will get you a range for P.

   (b) If there are N processes, the second equation above should be modified so that the combined user part of the N processes can fit into the free physical pages. So we will have $N * U \leq P - K$. We also have $P \leq V - U$ as before. Eliminating P (unknown), we get $U \leq \frac{V-K}{N+1}$.

5. The kernel stack cannot be reallocated during `exec`, because the kernel code is executing on the kernel stack itself, and releasing the stack on which the kernel is running would be disastrous. Small changes are made to the trap frame however, to point to the start of the new executable.

6. The memory pages shared by parent and child would be marked read-only in the page table. Any attempt to write to the memory by the parent or child would trap the OS, at which point a copy of the page can be made.

7. Memory cannot be reclaimed during the kill itself, because the victim process may actually be executing on another core. Processes are periodically checked for whether they have been killed (say, when they enter/exit kernel mode), and termination and memory reclamation happens at a time that is convenient to the kernel.

8. We have $t_x = h * t_h + (1 - h) * t_m$, so $t_h = \frac{t_m - t_x}{t_m - t_h}$

9. The `exec` system call retains the old page tables, so that it can switch back to the old image and print an error message if exec does not succeed. If exec succeeds however, the old memory image will no longer be needed, hence the old page tables are switched and freed.

10. A slab allocator is fast because memory is preallocated. Further, it avoids fragmentation of kernel memory.

11. Different operating systems may have different heuristics to allocate physical memory to a process. For example, systems may differ in the heuristics for prepaging, i.e., how to prefetch pages that may be accessed in the future. This leads to different values of RSS.

12. A,B,D

13. C