PDS/SDP OS internals 11/07/2024

PDS: L'esame è comune ai corsi PDS e SDP (in inglese). Si è predisposta un'unica proposta di soluzione in inglese. Qualora ci fossero errori nelle soluzioni (possibili, specie tenendo conto di modifiche e aggiustamenti fatti nel preparare l'esame), si pubblicheranno aggiornamenti.
SDP ON/OFF: As the ON/OFF exam is a subset of the standard one, the ON/OFF solution can be easily extracted from the standard one. If there are errors in the solutions (possible, especially taking into account last minute changes and adjustments made in preparing the exam), updates will be published.

Question 1.

A computer system has the following characteristics:

- Physical memory size: 512 MB
- Virtual address space: 32-bit addresses
- Page size: 8 KB
- Uses a two-level page table
- Implements a Least Recently Used (LRU) page replacement algorithm

1. Can a page table be designed to dynamically adjust its size based on the number of pages a process is currently using, thus minimizing the memory overhead? (YES/NO, Briefly explain/motivate)

   Yes, a page table can be designed to adjust dynamically. Techniques like hierarchical (multi-level) page tables or hashed page tables can help reduce memory overhead by allocating page table entries only as needed.

2. A process is allocated more physical frames than its working set size (for a given delta time): will this always reduce the page fault rate to zero? Briefly explain/motivate the answers. (YES/NO, Briefly explain/motivate)

   No, even if a process has more frames than its working set size, it doesn't guarantee zero page faults. Other factors such as changes in the working set and system-wide memory contention can still cause page faults.

3. Can two different virtual addresses map to the same physical address in a system using paging? (YES/NO, Briefly explain/motivate)

   Yes, this is possible through the use of shared memory. Different processes can have different virtual addresses that map to the same physical address for inter-process communication.

4. Can using a larger page size result in more efficient TLB utilization despite potential increases in internal fragmentation? (YES/NO, Briefly explain/motivate)

   Yes, larger page sizes can improve TLB efficiency because fewer TLB entries are needed to cover the same amount of virtual address space, reducing TLB misses. However, this comes at the cost of increased internal fragmentation.

5. Does the presence of a TLB always guarantee a faster memory access time compared to a system without a TLB? (YES/NO, Briefly explain/motivate)

   No, while a TLB generally improves memory access time by reducing page table lookups, in cases of high TLB miss rates or small working sets that fit in the page table, the benefit might not be significant.

6. If the LRU page replacement algorithm is used, does it guarantee the lowest possible number of page faults?

   No, LRU is an approximation and does not always yield the optimal number of page faults, especially in cases where the reference pattern does not favor LRU.

7. Given the following memory reference string for a process (byte addressing, with addresses expressed in hexadecimal code) and their respective read(R)/write(W) operations:
R 1F40, W 3E20, R 5D18, W 7C9E, R 2AF3, W 6B2C, R 13B2, W 24AB, R 54BC, W 12AF

   Assume the system has 3 available frames and uses the Least Recently Used (LRU) page replacement algorithm.

   Compute the string of page references and indicate the corresponding page number for each memory reference. Explicitly indicate for each allocated frame and time/reference. the page number, and whether a page fault occurs. Calculate the total number of page faults that occur during this sequence of memory references.

	R 1f40	W 3E20	R 5D18	W 7C9E	R 2AF3	W 6B2C	R 13B2	W 24AB	R 54BC	W 12AF
P #	0	1	2	3	1	3	0	1	2	0
F 1	0	0	0	3	3	3	3	3	2	2
F 2		1	1	1	1	1	1	1	1	1
F 3			2	2	2	2	0	0	0	0
PF	X	X	X	X			X		X	

   Total Page Fault = 6

Question 2.

Consider a Unix-like file system based on inodes, with 15 pointers (12 direct, 1 single indirect, 1 double indirect, and 1 triple indirect). Pointers have a size of 32 bits, and disk blocks have a size of 4 KB. The file system resides on a disk partition where 1 TB is reserved for data blocks. The extra space reserved for metadata (including index blocks) can be neglected for the purposes of this exercise.

A) Calculate the maximum file size supported by this file system.

   Direct Pointers: 12 * 4 KB = 48 KB
   Single indirect pointer: One block can hold 4KB/4B = 1,024 pointer, 1,024 * 4 KB = 4 MB
   Double indirect pointer: 1,024 * 1,024 * 4 KB = 4 GB
   Triple indirect pointer: 1,024 * 1,024 * 1,024 * 4 KB = 4 TB
   Total Size: 48 KB + 4 MB + 4 GB + 4 TB = 4.004048 TB

B) Given a binary file of size 25,600 KB, compute exactly how many index blocks and data blocks the file is using. Also compute the internal fragmentation (for data blocks).

   Index Block:
   Data Block: Number of Data Block = Total file Size / Block Size = 25,600 KB / 4 KB = 6,400 Blocks

   Index Block: The file uses the first 12 direct blocks. Remaining blocks= 6,388
   1 single indirect blocks (1,024 block is used). Remaining blocks = 6,388 – 1,024
   Double indirect blocks = 5,364 /1,024 = 6 blocks
   Index Block = 1 + 1 + 6 = 8

C) Suppose an operation lseek(fd, offset, SEEK_END) is called to position the file offset for a subsequent read/write operation. Given a binary file of size 10 GB and an offset of -1,048,576 bytes (1 MB) from the end of the file, compute the logical block number (relative to the file blocks, numbered starting at 0) to which the position is moved.

   Total Blocks for 10 GB = 10GB /4KB = 2,621,440 blocks
   Offset = -1,048,576 bytes = -1,048,576 bytes / 4 KB = -256 blocks
   Block number = 2,621,440 + (-256) = 2,621,184
   The logical block number to which the position is moved is 2,621,184.

Question 3.

Consider a computer system that uses a combination of hard disk drives (HDDs) and solid-state drives (SSDs) for mass storage. The system also incorporates various I/O devices managed by an I/O subsystem.

1. Can a RAID 0 (striping) configuration improve data reliability compared to a single HDD setup?

   NO: RAID 0 (striping) improves performance but not data reliability. In fact, it decreases reliability because the failure of any one drive results in the loss of all data.

2. Is it possible for an SSD to suffer from fragmentation issues similar to those experienced by HDDs?

   NO: SSDs do not suffer from fragmentation in the same way as HDDs because they do not have moving parts. However, logical fragmentation can still affect performance, though it is handled differently.

3. Is it always more efficient to use Direct Memory Access (DMA) for all types of I/O operations compared to interrupt-driven I/O?

   NO: DMA is generally more efficient for large data transfers, but for small, quick operations, interrupt-driven I/O can be more efficient due to the overhead associated with setting up DMA.

4. Is polling a more efficient method for handling I/O operations compared to using interrupts in a high-speed network environment?

   NO: Interrupts are generally more efficient for high-speed network environments because they reduce CPU usage compared to polling, which continuously checks the status of an I/O device.

5. Can a well-designed I/O scheduler significantly improve the performance of a RAID 5 setup compared to a poorly designed I/O scheduler?

   NO: A well-designed I/O scheduler can optimize the order of read/write operations, reduce seek time, and improve overall performance, which is particularly beneficial in a RAID 5 setup where parity calculations and distributed data need efficient handling.

6. Is it possible for an I/O operation on an SSD to always have a lower latency compared to the same operation on an HDD?

   YES: SSDs generally have lower latency than HDDs due to the lack of moving parts, providing faster access times.

Question 4.

An oS161 system is given, running on a sys161 MIPS simulator with 8MB of RAM memory. It is known that process P1 has an address space with as->as_pbase1, as->as_pbase2, as->as_vbase1, as->as_vbase2, as->as_npages1, as->as_npages2 having the following values: 0x200000, 0x100000, 0x4000, 0x8000, 3, 4.

For each of the following physical addresses, convert to a kernel logical address. In case the address is allocated to P1, convert it to a logical user address:

- 0x100A00: K -> 0x80100A00, U -> 0x8A00
- 0x10A0F0: K -> 0x8010A0F0, U -> NO
- 0x206500: K -> 0x80206500, U -> NO
- 0x202D00: K -> 0x80202D00, U -> 0x6D00

Note: while user addresses need to be mapped into an address space, all RAM addresses can be seen as kernel logical addresses (regardless of being within/outside a user process address space): in other words, the kernel can access physical addresses mapped to a user process as logical kernel addresses: see for instance function as_zero_region(), called in as_prepare_load(). This is the reason why all 4 RAM addresses have a logical kernel address.

The OS161 function load_elf sets up the address space of a new user process. In different phases, load_elf calls load_segment, as_prepare_load, as_complete_load, as_define_region. Answer the following questions:

- Which function is called in order to allocate the address space in physical memory? as_prepare_load
- Which function is called in order to set the logical addresses and sizes of the code and data segments? as_define_region
- Which function reads the data segment from the elf file? load_segment
- Which function reads the code segment from the elf file? load_segment

Question 5.

Consider functions sys__exit and sys_waitpid implemented in lab4. Answer the following questions

- Are they already present in the base version of OS161? NO
- Are they system calls which can be called by user processes? NO

In lab4 a process has to be identified by a pid

- Is a pid a pointer? NO
- Is the pid associated to a process at bootstrap time? NO

Does function common_prog crash without implementing and using waitpid (or related functions)? NO

Is proc_destroy called within

- system call exit? NO
- system call waitpid? YES
- it is called by function thread_exit NO