USER

1. Consider a direct-mapped cache memory. Which one of the following statements is true?
(a) The cache has 1 line per set.
(b) The cache has 1 word per block.
(c) The cache has 1 set per cache.
(d) None of the above.

ASSISTANT

The correct answer is:

(d) None of the above.

In a direct-mapped cache memory, each cache line can store multiple words (or blocks), and each cache set can have multiple lines. Therefore, none of the options listed are true. Here's why:

* (a) The cache does not have 1 line per set; instead, it has multiple lines per set.
* (b) The cache does not have 1 word per block; instead, it has multiple words per block.
* (c) The cache does not have 1 set per cache; instead, it has multiple sets in the cache.

So, option (d) None of the above is the correct answer.

USER

2. Which one of the following statements about cache memories is true:
(a) Larger caches are more susceptible to capacity misses than smaller caches.
(b) Caches with lower associativity are more susceptible to conflict misses than those with higher associativy.
(c) Caches with higher associativity are more susceptile to cold misses than those with lower associativey.
(d) None of the above

ASSISTANT

The correct answer is:

(a) Larger caches are more susceptible to capacity misses than smaller caches.

Here's why:

* Capacity misses occur when a block in main memory is not found in the cache, and the cache is full. The probability of capacity misses increases as the size of the cache grows.
* Smaller caches have fewer blocks, so they are less likely to experience capacity misses.
* Conflict misses, on the other hand, occur when two or more lines in the same set need to be replaced. Larger caches with higher associativity can reduce conflict misses by storing multiple copies of a block in different sets.

The other options are false:

* (b) Caches with lower associativity are not necessarily more susceptible to conflict misses than those with higher associativity.
* (c) Higher associativity does not increase the likelihood of cold misses; it reduces the likelihood of conflict misses.

So, option (a) Larger caches are more susceptible to capacity misses than smaller caches is the correct answer.

USER

2. Which one of the following statements about cache memories is true:
(a) Larger caches are more susceptible to capacity misses than smaller caches.
(b) Caches with lower associativity are more susceptible to conflict misses than those with higher associativy.
(c) Caches with higher associativity are more susceptile to cold misses than those with lower associativey.
(d) None of the above

ASSISTANT

The correct answer is:

(b) Caches with lower associativity are more susceptible to conflict misses than those with higher associativity.

Here's why:

* Lower associativity means that a cache block can only be stored in one set, making it more likely to experience conflicts when multiple blocks need to be stored in the same set.
* Higher associativity allows a cache block to be stored in multiple sets, reducing the likelihood of conflict misses.
* Capacity misses and cold misses are not directly related to associativity. Larger caches can have both capacity misses (when the cache is full) and cold misses (when a block is not in the cache).

The other options are false:

* (a) Larger caches may experience more capacity misses, but their susceptibility to conflict misses does not change.
* (c) Higher associativity reduces the likelihood of conflict misses, not increases it.

So, option (b) Caches with lower associativity are more susceptible to conflict misses than those with higher associativity is the correct answer.

USER

4. Assuming no errors, which one of the following statements about fork is true?

(a) Called once, returns once.
(b) Called once, returns twice.
(c) Called once, returns never.
(d) Called twice, returns once.
(e) None of the above.

ASSISTANT

The correct answer is:

(b) Called once, returns twice.

Here's why:

* The `fork` system call creates a new process by duplicating the calling process. This means that when `fork` is called, it creates a new process and returns a unique process ID to the parent process.
* In the parent process, `fork` returns a value indicating the result of the fork (0 for success or -1 for failure).
* In the child process, `fork` returns 0, because the child process has just been created.

So, `fork` is called once from the parent process, but it returns twice: once in the parent process and once in the child process.

USER

5. Assuming no errors, which one of the following statements about execve is true?
(a) Called once, returns once.
(b) Called once, returns twice.
(c) Called once, returns never.
(d) Called twice, returns once.
(e) None of the above.

ASSISTANT

The correct answer is:

(c) Called once, returns never.

Here's why:

* The `execve` system call replaces the current process image with a new one. When `execve` is called, it loads a new program into memory and starts executing it.
* Once `execve` has completed its execution, the original process no longer exists, so there are no returns.
* This means that `execve` is called once from the parent process, but it does not return. The calling process is effectively replaced by the new program.

So, `execve` is called once and never returns.

USER

6. Which one of the following statements about processes is false?
(a) The operating system kernel runs as its own separate process.
(b) Each process shares the CPU with other processes.
(c) Each process has its own private address space.
(d) The environment for a process is stored on the stack.

ASSISTANT

The correct answer is:

(b) Each process shares the CPU with other processes.

Here's why:

* In modern operating systems, each process runs in parallel and gets a time slice (called a time quantum or time slice) of the CPU to execute. This means that multiple processes can run simultaneously, sharing the CPU.
* Processes do not share their private address space or resources, but they may share the CPU.

The other options are true:

* (a) The operating system kernel runs as its own separate process, often referred to as the "init" or "system" process.
* (c) Each process has its own private address space, which is a region of memory that is unique to that process and not shared with other processes.
* (d) The environment for a process is typically stored in memory regions such as the heap or stack, which are private to each process.

So, option (b) is the false statement.

USER

7. What happens if the parent of a zombie child terminates?
(a) The zombie child becomes a wraith and is never reaped.
(b) The zombie child is reaped by the init process.
(c) The zombie child is reaped by the process with the nearest PID.
(d) None of the above.

ASSISTANT

The correct answer is:

(b) The zombie child is reaped by the init process.

Here's why:

* When a parent process terminates and leaves behind one or more child processes that have finished execution, those child processes become "zombies" because they are no longer linked to their parents and cannot be terminated.
* In this case, the parent of the zombie child has already terminated, so the zombie child needs to be reaped (i.e., terminated) by another process.
* By convention, the init process is responsible for reaping zombies that have no other parent process. The init process will send a SIGCHLD signal to the zombie child and then terminate it.

The other options are false:

* (a) A "wraith" is not a standard term in Unix-like operating systems, and there is no concept of a "zombie" becoming a "wraith".
* (c) The init process reaps zombies, not some arbitrary process with the nearest PID.
* (d) None of the above means that none of the provided options are correct, but option (b) is indeed correct.

So, option (b) is the correct answer.

USER

8. Suppose that the kernel delivers two SIGCHLD signals to the parent while the parent is not scheduled. When the kernel finally schedules the parent, how many times will the SIGCHLD handler be called?
(a) None, because sending multiple signals will always crash the program.
(b) Exactly once, because signals are not queued.
(c) Exactly twice, because signals are queued.
(d) More than twice, depending on how the handler is installed.

ASSISTANT

The correct answer is:

(c) Exactly twice, because signals are queued.

Here's why:

* In Unix-like operating systems, signals are queued. This means that if a process receives multiple instances of the same signal before it has a chance to process them, those signals will be stored in a queue and processed one by one when the process is scheduled again.
* In this case, the kernel delivers two SIGCHLD signals to the parent while the parent is not scheduled. When the kernel finally schedules the parent, the SIGCHLD handler will be called twice, once for each signal that was queued.

The other options are false:

* (a) Sending multiple signals to a process does not necessarily crash the program. The program can choose how it handles multiple signals, and it may even install a signal handler to catch and handle them.
* (b) Signals are not discarded if they arrive while the process is not scheduled. Instead, they are stored in a queue and processed when the process is scheduled again.
* (d) The number of times the SIGCHLD handler is called does not depend on how the handler is installed. It depends on whether signals are queued or not.

So, option (c) is the correct answer.

USER

9. Which one of the following statements is NOT true of storage allocators?
(a) In the best case, coalescing with boundary tags is linear in the number of free blocks.
(b) Seglists typically approximate best fit search.
(c) Payloads must be aligned to some boundary.
(d) Explicit lists are typically faster than implicit lists.
(e) None of the above.

ASSISTANT

The correct answer is:

(e) None of the above.

Here's why:

* (a) Coalescing with boundary tags can have a worst-case time complexity that is linear in the number of free blocks, not just best case. In general, coalescing algorithms have a trade-off between search efficiency and update efficiency.
* (b) Seglists typically use a combination of implicit lists (e.g., bitmaps) and explicit lists (e.g., linked lists) to achieve good performance for both search and insertion/deletion operations.
* (c) In many storage allocators, payloads are aligned to some boundary (e.g., 8-byte or 16-byte boundaries) to simplify pointer arithmetic and improve cache locality.
* (d) The relative speed of explicit lists versus implicit lists depends on the specific allocator design and usage patterns. In general, explicit lists can be faster for search operations but slower for insertion/deletion operations, while implicit lists can be faster for insertion/deletion operations but slower for search operations.

So, all options except (e) are true statements about storage allocators.

USER

10. Which one of the following addresses is 8-byte aligned?
(a) 1110110101110111
(b) 1110110101110100
(c) 1110110101110000
(d) 1110110101110110

(e) None of the above

ASSISTANT

The correct answer is:

(c) 1110110101110000

Here's why:

* The address `1110110101110000` has a binary representation that ends with four zeros (`0000`). This means it is aligned to an 8-byte boundary, because any address that ends with four zeros (or fours) is divisible by 8.

The other options are not 8-byte aligned:

* (a) `1110110101110111` has a binary representation that does not end with four zeros.
* (b) `1110110101110100` also has a binary representation that does not end with four zeros.
* (d) `1110110101110110` has a binary representation that ends with two zeros, which means it is aligned to a 4-byte boundary, but not an 8-byte boundary.

So, option (c) is the only one that represents an address that is 8-byte aligned.