1. OS Overview and Boot Process

    1.1. Three roles of an OS are

        1.1.1. Referee, OS manages the allocation of shared resources such as CPU, memory, etc. for each task. For example, when we play a game and look up tutorials online simultaneously. Both the game and the browser needs CPU and other resources to run at the same time, which are allocated by the OS.

        1.1.2. Illusionist, OS abstracts the workings of each computer function and present to us instead an easy-to-work with interface. For instance, file managers and similar programs allow us to easily navigate and organize our files and folders, without having to worry about all the memory allocation and complex details behind the scenes.

        1.1.3. Glue, OS shares standard workings that can be applied to any application. For example, copy and paste function will work on text in most applications, allowing for ease of use and convenience.

    1.2. Portability is the least important. The main purpose of a game is to provide entertaining experience, as such, it must be reliable (does not break easily) and high-performance (smooth and good-looking graphics to provide best experience). For portability, while hardware abstraction will definitely be needed, it is usually a one-time process at the beginning when it comes to setting up and playing a single game. Reliability and performance, which actually affects the game experience, is least important.

    1.3. At the first-stage bootloading, BIOS load up the first 512 byte area called the MBR. Within this, it checks if the code has signature of 0xAA at byte 510 and 0x55 at byte 511. If it does, the disk is bootable.

    1.4. You can edit the first bytes in the bootloader, it will print out the edited string, up until it hits the 0 byte.

2. Kernel and system calls

    2.1. Dual mode operation means that an OS is designed to switch between 2 modes, User mode and Kernel mode. This design provides defense against program error which may affect the rest of the computer. To simplify things, Kernel mode is the more privileged mode that can

do a lot more than user mode can, such as handling interrupts, manages I/O, and more, while the User mode is where user application is run and, whenever needed, ask for some functions by sending system calls to the Kernel mode.

2.2. User mode -> Kernel mode

2.2.1. Hardware interrupts

2.2.2. Processor exceptions (seg faults, divide by zero)

2.2.3. System calls

2.3. System calls used in projects

2.3.1. fork()

2.3.2. exit()

2.3.3. wait()

3. Processes and Signals

3.1. 8 processes in total. After first fork, there are 2 processes. Each of those 2 then encounter the second fork, each creating a child, leaving 4 processes. Each of those 4 then hit the last fork, creating a child of their own, leaving 8 processes.

3.2. Command exec() completely transform shell() to the program it is given to run. Anything after exec() is unreachable dead code, so fork() will not be executed.

3.3. Three ways a process may react to signals

3.3.1. Ignore the signal

3.3.2. Terminate the process

3.3.3. Catch the signal by executing signal handler and customize what to do next

3.4. kill()

3.5. A process' exit status is a value from 0 to 255 that gives an insight into its cause of termination. A parent process can use wait() or waitpid() system call to retrieve the exit code.

4. Threads, Synchronization and Deadlocks

4.1. Fill in the blank.

4.1.1. Not enough utilization of threads. Use more threads

4.1.2. Full disks. Get faster disk

4.1.3. Overwhemed CPU, Get faster CPU

4.2. Social media

4.2.1. Here we are accessing resources from a and b. By running it once, we lock both of them, access both of them, then unlock both. But problems can occur if, for instance, one thread lock a then another lock b. The access by either would be messed up.

4.2.2. We can use semaphores for this.

4.3. Kernel & User level threads

4.3.1. Kernel-level thread

4.3.2. User-level thread

4.4. Possible values of c are;

4.4.1. When it passes if, c=6, c=13

4.4.2. When it goes to else, c=4, c=14, c=7

5. CPU Scheduling

5.1. FCFS queue processes in the order of arrival, meaning it will run processes one by one until each is finished. This makes average waiting time higher, and there is also possibility of shorter processes being run last, needing more time.

5.2. One of the ways to approximate running time in advance is to use machine learning to approximate runtime based on previous process run speed.

5.3. Round Robin Scheduling

5.3.1. Quantum is basically allowance of CPU time for each process. Once it ran out, it move on to the next process, even if its not done yet, and return later.

5.3.2. Small quantum means that all processes will be executed little by little, eliminating problems of starvation.

5.3.3. Large quantum means less context switches/overhead, and each process is given ample time to run.

6. Protection and Security

6.1. Principle of least privilege means each user or each application should be given only enough access and permissions to do its task. Military army structure is one example; the lower-

ranked footsoldiers cannot be given access to the command room or resources critical to the army because they do not need it to fight individually- this reduces the risk of them sabotaging the whole army.

6.2. Access vs Capacity lists

6.2.1. Access lists. Once you receive sticker, you can access the buildings. Weakness is that once you receive sticker from one building, you can access all other. If temperature checker at the one you received your sticker is faulty, everyone is screwed.

6.2.2. Capacity lists. Only some teachers' ID card have access to the Science Division room- but that same ID might allow you into Business division instead. Weakness is that it is a damn pain for lecturers to meet, plus any time a teacher walk in/out anyone can just go through that opened door anyway. That's what I did.