Quizlet Operating Systems: Principles and Practice Chapter 3

Study online at quizlet.com/_3polo3

1. **child process**	the process being created
2. **client-server**	an alternative model to producer-consumer relationships to allow two-way communications between processes. For example, in client-server computing, the clients send requests to the server to do some task, and when the operation completes, the server replies back to the client.
3. **client-server communication**	Instead of a single pipe, we create two, one for each direction. To make a request, the client writes data into one pipe, and reads the response from the other. The server does the opposite: it reads from the first pipe, and writes to the second.
4. **details of implementing a shell**	A program can be a file of commands for the shell to interpret. A program can send its output to a file by redirecting the child's output to a file in the standard UNIX shell. For example "ls > tmp" lists the contents of the current directory into the file "tmp". A program can read its input from a file by using "dup2" to change the stdin file descriptor: "zork < solution" plays the game "zork" with a list of instructions stored in the file "solution."
5. **dynamically loadable device driver**	software to manage a specific device, interface, or chipset, added to the operating system kernel after the kernel starts running.
6. **early batch processing systems**	the kernel was in control by necessity. Users submitted jobs, and the operating system took control to instantiate the process and run the job.
7. **explicit close**	when an application is done with the device or file, it calls close so that the OS can decrement the reference-count on the device, and garbage collect any unused kernel data structures.
8. **file-system**	programs can connect together through reading and writing files. A text editor for example, may import an image from a drawing program, and the editor can then write an HTML file that knows how to display a web page.
9. **flexibility**	Refers to one of the key ideas in UNIX where the UNIX system call interface was highly portable. The operating system can be ported to new hardware without needing to rewrite application code.
10. **hardware abstraction layer**	a portable interface to machine configuration and process-specific operations within the kernel. Thus, porting an operating system to a new computer is a matter of creating new implementations of the low-level hardware abstraction layer routines and re-linking.
11. **input/output**	One of the primary innovations in UNIX was to regularize all device input and output behind a single common interface. It even uses the same interface for reading and writing files and for interprocess communication.
12. **I/O byte-oriented**	all devices, even those that transfer fixed-size blocks of data, are accessed with byte arrays.
13. **I/O, open before use**	Before an application does i/o, it must first call open on the device, file, or communication channel.
14. **I/O replace file descriptor**	The shell does redirection of a file descriptor to change the read input from a file to from the keyboard via the special system called name "dup2(from, to)"
15. **I/O uniformity**	All device I/O, file operations and interprocess communicate use the same set of system calls: open, close, read, and write.
16. **I/O wait for multiple reads**	The UNIX system call "select(fd[], number)" allows the server to wait for input from any of a set of file descriptors; it returns the file descriptor that has data, but it does not read the data.
17. **kernel-buffered reads**	Stream data, such as the network or keyboard, is stored in a kernel buffer and returned to the application on request.
18. **kernel-buffered writes**	outgoing data is stored in a kernel buffer for transmission when the device becomes available.
19. **microkernel**	An alternative to the monolithic kernel approach to run as much of the operating system as possible in one or more user-level servers. A microkernel design offers benefit to the developer as its easier to modularize and debug user-level services than kernel code. Microkernels offer little benefit in performance as it may slow it down. Thus most systems adopt a hybrid model where some OS services are run at user-level and some are in the kernel, depending on the specific tradeoff between code complexity and performance.

20. **microkernel**	Isolate privileged, but less critical, parts of the operating system such as the file system or the window system, from the rest of the kernel. The kernel itself is kept small, and instead most of the functionality of the traditional operating system kernel is put into a set of user-level processes, or servers.
21. **monolithic kernel**	most of the operating system functionality runs inside the operating system kernel. This hopes to improve performance by making it easier to arrange tight integration between kernel modules.
22. **parent process**	the process creator
23. **performance**	transferring control into the kernel is more expensive than a procedure call to a library, and transferring control to a user-level file system server via the kernel is still even more costly.
24. **preventing bugs from device drivers**	OS developers deal with bugs from device drivers through code inspection, bug tracking on the stack, making sure that device drivers run at user-level rather than in the kernel, running the device driver code in a guest operating system on a virtual machine, and driver sandboxing in an their own execution environment inside the kernel.
25. **process**	an instance of a program
26. **process control block**	It contains many pieces of information associated with a specific process: process state, program counter, CPU registers, CPU scheduling, memory management
27. **producer-consumer**	a model where programs are structured to accept as input the output of other programs.
28. **producer-consumer communication**	The shell can use a pipe to connect two programs together so that the output of one program is the input of another. Establish a pipe between the producer and the consumer. As one process computes and produces a stream of output data, it issues a sequence of write system calls on the pipe into the kernel.
29. **reliability**	The operating system kernel remains minimal to improve reliability. Kernel modules are typically not protected from one another, so a kernel bug may corrupt user or kernel data.
30. **Safety**	Resource management and protection are the responsibility of the operating system kernel. Checks cannot be implemented in a user-level library because they may skip necessary checks that prevent malicious code.
31. **shell**	a job control system that allows you to write down the sequence of steps to run various programs.
32. **Steps for Unix exec**	Load the program "prog" into the current address space. Copy arguments "args" into memory in the address space. Initialize the hardware context to start execution at "start."
33. **Steps for UNIX fork**	Create and initialize the process control block in the kernel. Create a new address space. Initialize the address space with a copy of the entire contents of the address space of the parent. Inherit the execution context of the parent (e.g., any open files). Inform the scheduler that the new process is ready to run.
34. **UNIX exec**	brings the new executable image into memory and starts it running. Takes two arguments.
35. **UNIX file descriptor**	Open returns a handle to be used in later calls to read, write, and close to identify the file, device, or channel.
36. **UNIX fork**	creates a complete copy of the parent process. The child process sets up privileges, priorities, and I/O for the program that is about to be started, by closing some files, opening others, reducing its priority if it is run in the background. Takes no arguments and returns an integer.
37. **UNIX fork returns twice**	Once in the child, with a return value of zero, and once in the parent with a return value of the child's process ID.
38. **UNIX pipe**	a kernel buffer with two file descriptors, one for writing (to put data into the pipe) and one for reading (to pull data out of the pipe).
39. **UNIX Process Management**	Splits the creation of a process into two steps: "fork" and "exec"
40. **UNIX signal**	Provides a facility for one process to send another an instant notification or upcall. Signals are used for terminating an application, suspending it temporarily for debugging, resuming after a suspension, timer expiration, and a host of other reasons.
41. **UNIX stdin**	reading commands from input (for example, the keyboard)
42. **UNIX stdout**	writing output (for example, to the display)

43. **UNIX wait**	A system call that pauses the parent until the child finishes, crashes, or is terminated. Parameterized with the process of the child. With wait, a shell can create a new process to perform some of its instructions, and then pause for that step to complete before proceeding to the next step. An optional call.
44. **upcall**	Where the kernel notifies the application about an event
45. **user-initiated processes**	Today, programs create and manage processes. These include window managers, web servers, web browsers, shell command line interpreters, source code control systems, databases, compilers, and document preparation systems.
46. **Windows CreateProcess**	Create and initialize the process control block (PCB) in the kernel. Create and initialize a new address space. Load the program "prog" into the address space. Copy arguments "args" into memory in the address space. Initialize the hardware context to start execution at "start." Inform the scheduler that the new process is ready to run.