Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Operating Systems and Networks	Spring 2016
Assignment 5	Due 19 April 2016

Part I: File Systems

1.1 Is the "open" system call in UNIX absolutely essential? What would be the consequences (i.e., how would you need to change other system calls, what impact on performance do you expect) of not having it?

> **Solution:**
> If there were no "open" system call, it would be necessary to specify the name of the file to be accessed for every read operation. The system would then have to fetch the i-node for it, although it could be cached. One issue that quickly arises is when to flush the inode back to disk. It could be based on a timeout, however it would be clumsy. Overall, it may work, but with much more overhead involved.

1.2 It has been suggested that the first part of each file be kept in the same disk block as its inode. What good would this do?

> **Solution:** Often, files are short. If the entire file fit in the same block as the inode, only one disk access would be needed to read the file, instead of two, as is presently the case. Even for longer files there would be a gain, since one fewer disk accesses would be needed.

1.3 An Operating System only supports a single directory, but allows that directory to have arbitrarily many files with arbitrarily long file names. Can something approximating a hierarchical file system be simulated? How?

> **Solution:** One way to simulate that is to prepend each file name with the name of directory that contains it and use a distinct character to seperate different directory names. For example: usrXstudentsXtimosXSomeFile

1.4 Systems that support sequential files always have an operation to rewind files. Do systems that support random access files need this too?

> **Solution:** No, random access of files does not need the "rewind" operation since if you want to read the file again, you can just access byte 0.

1.5 Contiguous allocation of files leads to disk fragmentation if files are deleted. Is this internal fragmentation or external fragmentation? What if the disk is accessed in blocks and we demand that each block contains at most data of one file?

> **Solution:** Contiguous allocation leads to external fragmentation (due to holes between files where a file was deleted, but newly created files are bigger than that hole). If additionally disk are divided into blocks there will also be internal fragmentation (space wasted due to partially empty blocks).

1.6 One way to use contiguous allocation of disk space and not suffer from holes is to compact the disk every time a file is removed. Since all files are contiguous, copying a file requires a seek and rotational

delay to read the file, followed by the transfer at full speed. Writing the file back requires the same work. Assuming a seek time of 5 msec, a rotational delay of 4 msec, a transfer rate of 8MB/sec and an average file size of 8KB, how long does it take to read a file into main memory then write it back to the disk at a new location? Using these numbers, how long would it take to compact half of a 16GB disk?

Solution: It takes 9msec to start the transfer (due to 5msec seek and 4msec rotation delay). To read 2^{13} bytes (8KB) at the transfer rate of 2^{23} bytes/sec (8MB/sec) requires 2^{-10} sec (0.977msec). Hence the total time to seek, rotate and transfer is 9.977msec. Writing back takes another 9.977msec. Thus copying an average file takes 19.954msec.

To compact half of a 16GB disk would involve copying 8GB of storage, which is 2^{20} files. At 19.954 msec per file, this takes 20,923 seconds, which is 5.8 hours. Clearly, compacting the disk after every file removal is not a great idea.

1.7 Consider an inode structure with 12 direct addresses, one indirect address, one double indirect address and one triple indirect address. Each block is 4KB in size. Assuming block addresses are 32 bit values, what is the maximum file size?

Solution: Let b be the block size (4KB), then the maximum file size is: $((b/4)^3+(b/4)^2+b/4+12)\cdot b$. For the chosen block size this gives us a maximum file size of 4 TB.

1.8 Free disk space can be kept track of using free list or a bit map. Disk addresses require D bits. For a disk with B blocks, F of which are free, state the condition under which the free list uses less space than the bit map. For D having the value 16 bits, express your answer as a percentage of the disk space that must be free.

Solution: The bit map requites B bits. The free list requires DF bits. The free list requires fewer bits if $DF < B$. Alternatively, the free list is shorter if $\frac{F}{B} < \frac{1}{D}$, where $\frac{F}{B}$ is the fraction of blocks free. For 16-bit disk addresses the free list is shorter if 6 percent or less of the disk is free.

1.9 The open() syscall returns a filehandle, which allows us to read() and write() to that file. In order to delete a file we use the unlink() syscall, which takes the pathname of the file to delete as its parameter. What happens if we create/open a file, and delete it right after, can we still use the file descriptors returned from open()? Write a short program to try it out. How can a user access the contents of the "deleted" file without modifying your program?

Solution: The filehandles returned by open() remain valid until they are close()'d, so we can open a file, use unlink() to delete it and still use read() and write() on the filehandle. This technique is used by some online video streaming players. That way they can store the video file temporarily, but a user without a solid understanding of operating/file systems can not easily download the movie by saving that file. However, we can access all open filehandles of a process through the /proc filesystem: in /proc/¡PID¿/fd/ we will find symlinks for each open file handle, and we can use those symlinks to e.g., copy the "hidden" file. Also see the provided code.

1.10 Implement your own version of the `ls` utility. Of course you do not need to implement all the options `ls` provides, emulating the behaviour of `ls -l --color=never` is sufficient. Hint: start by reading the man pages for opendir(), readdir() and fstat().

Solution: See the provided code.

Part II: I/O Systems

2.1 Why might a system use interrupt-driven I/O to manage a single serial port, but polling I/O to manage a front-end processor, such as a terminal concentrator?

Solution: Polling can be more efficient than interrupt-driven I/O. This is the case when the I/O is frequent and of short duration. Even though a single serial port will perform I/O relatively infrequently and should thus use interrupts, a collection of serial ports such as those in a terminal concentrator can produce a lot of short I/O operations, and interrupting for each one could create a heavy load on the system. A well-timed polling loop could alleviate that load without wasting many resources through looping with no I/O needed.

2.2 Polling for an I/O completion can waste a large number of CPU cycles if the processor iterates a busy-waiting loop many times before the I/O completes. But if the I/O device is ready for service, polling can be much more efficient than is catching and dispatching an interrupt. Describe a hybrid strategy that combines polling, sleeping and interrupts for I/O device service. For each of these three strategies (pure polling, pure interrupts, hybrid), describe a situation in which that strategy is more efficient than is either of the others.

Solution: A hybrid approach could switch between polling and interrupts depending on the length of the I/O operation wait. For example, we could poll and loop N times and if the device is still busy at N+1, we could set an interrupt and sleep. This approach would avoid long busy-waiting cycles. This method would be best for very long or very short busy times. It would be inefficient if the I/O completes at N+T (where T is a small number of cycles) due to the overhead of polling plus setting up and catching interrupts. Pure polling is best with very short wait times. Pure interrupts are best with known long wait times.

2.3 How does DMA increase system concurrency? How does it complicate hardware design?

Solution: DMA increases system concurrency by allowing the CPU to perform tasks while the DMA system transfers data via the system and memory buses. Hardware design is complicated because the DMA controller must be integrated into the system and the system must allow the DMA controller to be a bus master. Cycle stealing may also be necessary to allow the CPU and DMA controller to share use of the memory bus.