Lecture Notes - Signals, Shell operations, IPC

Debadatta Mishra
Indian Institute of Technology Kanpur

Signals

- In many scenarios, either the OS or some other processes may want to send some notifications to a particular user process. For example, if a process is executing an infinite loop because of programming errors, the user may want to kill the process (from a shell). Further, the OS may want to notify the process if it performs a division-by-zero so that the process can perform cleanup activities before exiting. There are other situations where a process may require a notification to realize certain functionalities. For example, a parent process may want a notification from the OS when one of its child process terminates.
- UNIX OSs provide the signal system call API to enable user processes register notification handlers to different events. The signal system call allows a user process to register handlers for different events (signals). The signature of signal system call is as follows,

  signal (signum, sighandler) where **signum** is an integer specifying the signal (event) for which the user process wants notification from the OS. Some examples of signal numbers are, SIGINT, SIGHUP, SIGALRM, SIGCHLD etc. **sighandler** is the function pointer of the handler which is invoked when the signal event occurs. For example, if **signal (SIGINT, sigint_handler)** is invoked from a process, the **sigint_handler** function is executed every time CTRL+C is pressed on the console where the process is executing. The value of **sighandler** can be SIG_DFL to specify default handler (inside OS) or SIG_IGN to ignore the signal. Note that, for some signals (like SIGKILL, SIGQUIT), a user process is not allowed to ignore or register a custom handler. For more details, refer to the man pages (man signal).
- A signal can be sent from a process to another process using the **kill** system call. **kill(pid, SIGSEGV)** sends the SEGV (segfault) signal to the process with PID=pid. If pid is zero, the signal will be delivered to all processes in the calling process group. Note that, all child processes belong to the same process group of the parent unless the child executes setpgrp( ) system call to set its own process group.
- The OS maintains a per-process vector of handlers corresponding to all signals in the process control block (PCB). The value of handler can be one of the following: Default handler, no handler (ignore) or custom handler (registered by the process using **signal( )** system call). Further, the OS also maintains a pending signal bit vector where each bit correspond to a signal which is either set by the OS to indicate an event like SEGFAULT or by another process through the **kill( )** system call.
- Signals can be delivered in a synchronous manner when the event occurs because of any misbehavior of the executing process. For example, if a process causes an exception (by performing division by zero), the signal may be delivered when returning to the user space after the exception handling in the OS. In many other cases, if the process is executing in user mode when the signal event occurs, the

signal has to be delivered in a deferred (asynchronous) manner. For example, if process A wants to send a signal to process B which is executing in user mode, the signal pending bit is set to one (in the PCB of B) and the signal is sent to B when the OS gains the execution control.

- The OS can invoke the signal handler of a process when it is returning from an exception handler, system call handler or an interrupt handler. Therefore, the maximum time the signal remains undelivered depends on two factors---(i) the time till the OS gains control from the process and, (ii) the time till the process gets scheduled again.
- In case of a custom signal handlers, the OS returns to the user process resuming execution of the handler function after which the control comes back to the point where the process was executing before. To realize this behavior, the OS must mimic a function call in the user space. This requires careful manipulation of the user space stack and the instruction pointers by the OS. One possible design can be to create a new stack frame with return address set to the original user execution point (value of RIP before the entry into OS through a syscall, exception or an interrupt). On X86 systems, this is possible as the user process execution details (user RSP, RIP etc.) are pushed to the OS stack and the OS can access and appropriately augment them.
- An important design consideration for the OS is to design signal delivery mechanisms in presence of nested signals. More specifically, if a signal event occurs while one instance of the same signal is being handled, what should the OS do? If the signals are delivered during signal handling, the OS needs to carefully manage the user stack and further, the user stack may overflow in case of multiple nested invocation of signal handlers. Another alternate can be to disable signals during the execution of signal handlers and the user process re-registers the signal after handling the signal. There are multiple issues with this approach---lost signals, burden on the programmer to re-enable etc.
- Modern days OSs like Linux, insert system calls in the execution flow in a user transparent manner where the signal gets re-enabled through a special system call like **sigreturn( )** when the signal handler finishes**.** This can be done using techniques similar to signal handler call mimicking through user stack manipulations.
- Signal handlers are inherited by the child processes. However, the pending signals (if any) bit vector is all cleared for the child and are not delivered to the child. This makes sense as the pending signals are meant for the parent because they are set before the child creation. For multi-threaded programs, signal delivery behavior is undefined and the programmer should use pthread APIs like pthread_kill( ) and pthread_sigmask( ).

Files and Shell operations

- Most OSs provide file abstraction to the user processes to persist data across the life time of processes. Files also provide a method of sharing data across user processes. Standard input/output libraries are built on top of the OS system call API for file operations. For example, library calls like fopen( ), fread( ), fwrite( ) and fclose( ) internally use system calls like open( ), read( ), write( ) and close( ), respectively.
- **open( )** system call (e.g., fd = open(path_to_file, flags, mode)) opens the file provided in the path_to_file argument with specified permissions (READ, WRITE, EXECUTE etc.) in the flags field. The mode argument is used when O_CREAT is

passed as part of the flags to specify the file permissions for a newly created file. A successful open( ) system call returns a file descriptor (an integer) which can be used as the handle to access the file. Open file descriptors of parent process are inherited by the child process and remain open across exec( ) calls.

- Three default open file descriptors (0,1 and 2) represent the standard input, standard output and standard error, respectively.
- The OS maintains a per-process file descriptor table where each file descriptor point to a file object. The file object maintains per-process information (e.g., file position) related to the file. More than one file descriptors can point to the same file object. Every regular file object points to an inode structure which is unique for every file. So, file objects from more than one process, can point to the same inode if the file is opened in multiple processes. In short, mapping between file descriptors to file objects and the mapping from file objects to inodes can be many to one.
- **read(fd, buf, size)** reads size number of bytes from the current file position into **buf** parameter and returns the number of bytes read. Note that, read( ) may return reading less than the size parameter even when it did not reach end of file (EOF). To write to the file, a write( ) system call can be used whose usage semantic is similar to read( ) system call.
- For some interesting features like redirection, the dup system call is particularly useful. A dup(fd) system call creates a duplicate file handle by selecting the lowest available file handle. For example, if file descriptors 0, 1, 2 and 4 are currently used, dup(4) will return 3 which is a duplicate of 4 (internally 3 and 4 both point to the same file object).
- To implement typical functionalities like redirection (e.g., ls > tmp.txt), the shell can make use of fork( ) and exec( ) along with dup( ) system call. For example, to implement 'ls > tmp.txt', the shell can open "tmp.txt" file, fork( ) a child, close the standard output (fd=1) before calling a dup(fd) (fd → tmp.txt) in the child. At this point, fd = 1 represents the tmp.txt file and if **exec (ls)** is called in the child process, the output of **ls** is put into tmp.txt (as it is the stdout now). Refer to the code examples used in class for more details.
- To implement pipe functionality of a shell (e.g., ls | wc -l), UNIX provides the pipe system call. The pipe( ) system call takes an array of two file descriptors and creates a FIFO buffer where fd[0] can be used to read the content of the buffer and fd[1] can be used to put (write) data into the buffer. A fork( ) call after creation of a pipe, both parent and child have their own descriptors for both read and write end. Closing one end in the parent (say fd[0]) and closing the other end in child (say fd[1]) will establish an one-way communication channel between the parent and the child. To implement shell pipe functionality, the same one way communication through the pipe file descriptors can be used to dup( ) the standard input and the standard output. Refer the code examples and slides used in class for more details.

Shared memory using SysV IPC

- To share memory across two unrelated processes, UNIX IPC mechanisms provide shared memory regions which can be mapped to virtual address of processes wishing to share the memory. For example, processes A and B can map the shared region into their respective virtual addresses and access the shared memory area. Note that, the OS does not provide any ordering semantics and it is upto the

programmers to implement consistency, if required. For example, if A writes value 10 and B writes value 15 at the same time, the content of the shared memory location is non-deterministic (can be either 10 or 15).

- Shared memory regions are identified by a global ID i.e., key, which is assigned by the creator of the shared memory region and maintained by the OS. The creator process gets an handle to the shared memory region by using the following system call, **handle = shmget (key, size, flags = IPC_CREAT|0666),** where key is an identifier for the shared memory region. Any other process that wants to get an handle to the shared memory region, invokes the shmget system call as follows, **handle = shmget (key, size, flags = 0666).** To get the virtual memory mapping to the shared region, a process should call **shmat(handle, address_hint, flags),** where handle is previously obtained by shmget( ) system call, address_hint is the virtual address to which the OS may map the shared memory region. **shmat( )** returns the virtual address (pointer) to the shared memory. Internally, the OS maintains a list of shared memory regions along with the key, the currently attached processes and other information. This information regarding a shared memory region can be queried using **shmctl( )** system call. Please refer to the example code used in class for more details.
- The child inherits all the attached shared memory region. On exec( ) and exit( ), the shared memory regions are detached from the process.