

CSIS0230A Principles of Operating Systems (Class A)

Notes for Tutorial 3

Signals

In the last tutorial we examined one form of inter-process communication (IPC): shared memory. In this tutorial we will try another form of IPC, signals.

1. What are signals

When we write programs, we usually have scenarios when we want to wait for events that will happen only after quite some time, or simply may not happen at all depending on circumstances. For example, a program performing I/O will usually need to wait until the I/O is done (the event being “I/O completion”). It poses a problem if we also have something else to do: we don’t want to dedicate the CPU for waiting for the event (i.e., “poll” the event), because it would drain our CPU completely. On the other hand, if we only check for the event once a while, the program becomes unresponsive (because the event will be handled only at the next event check). Having to write our program in such a way that it performs event check periodically would also make our program more complicated.

For the real machine, the “interrupt” mechanism resolves this problem. The hardware that triggers the event have a wire connected to the CPU called the “interrupt line”. When the event happens, the hardware activates that interrupt line, the CPU notices it, and change the instruction pointer to execute an “interrupt handler” temporarily. Once the interrupt handler finishes, the original program continue to execute. In this way, the program (probably the kernel program) that needs to wait for the event will simply not wait for it. Instead, it continues to work on other tasks. When the event happens the interrupt handler would perform the needed actions to respond to the event.¹

Signal is a similar mechanism in the virtual world of processes. Upon seeing an event relevant to a process, the OS kernel stops the virtual machine, changes the instruction pointer to the address of a “signal handler” of the virtual machine, and restart the virtual machine in order to execute the signal handler. Once the signal handler returns, the OS resumes the execution of the original program.

Just like there are different types of interrupts, there are different types of signals. Each type of signal is allocated a distinctive number, each can be associated with a different signal handler.

2. Sending signals

A process can send a signal to another by calling *kill()* (or using the `kill` command). It is defined in `<signal.h>` as `int kill(pid_t pid, int signum)`. The first argument specifies the process to receive the signal. Unless you are the `root` user, you can only send signals to processes of your own. The second argument specifies the “signal number” to be used. If succeeded, it returns 0. To send a signal to oneself, `int raise(int sig)` may also be used.

Signal numbers are small integers with predefined meaning. You can find their symbolic names in the man page `signal(7)`. E.g., `SIGSEGV` is defined to be 11. The OS kernel sometimes

¹In a sense, the actual polling is done by the interrupt hardware. So instead of dedicating the whole CPU to wait for the event, only the interrupt hardware is used.

generate signals with predefined numbers. E.g., when a process performs a restricted operation, the kernel sends it *SIGSEGV*. Other signal numbers are only generated by user processes. See the last section for a description of all the common signal numbers.

3. What to do on signal reception?

When a process receives a signal, some **action** is taken. The default action depends on the signal number. E.g., by default *SIGSEGV* terminates the process. But you can configure the process to do things different from the default action. In particular, there is a “signal actions” for each signal number, used for specifying what to do when a signal of that number is received. The actual signal action is a property of the process managed by the kernel (and thus cannot be seen by the process), although you can copy it from and to the process memory using the *sigaction()* system call. The *sigaction()* system call is defined like this:

```
int sigaction(int signum, const struct sigaction *act, const struct sigaction *oact);
```

This retrieves the previous signal action to **oact*, and sets the new signal action to **act*. If either argument is *NULL*, the corresponding retrieval or setting is not performed. See the lecture notes for a full example.

The signal action structure is defined like this:

```
struct sigaction { /* defined in <signal.h> */  
    void (*sa_handler)(int);  
    void (*sa_sigaction)(int, siginfo_t *, void *);  
    sigset_t sa_mask;  
    int sa_flags;  
};
```

Within the structure, *sa_flags* is the bitwise-or of some flags modifying the behaviour of the action. Depending on whether or not *sa_flags* contains *SA_SIGINFO*, either *sa_sigaction* or *sa_handler* points to the function to be called upon receiving the signal. The former is more informative than the latter: it has more arguments passed. Most programs do not need to use the informative form, so we will only discuss the *sa_handler*.

Other flags can be specified in *sa_flags*, full information can be found in the man page `sigaction(2)`. The more important ones include:

SA_RESTART

Normally, if a signal occurs during the time of a “long” system call¹, and if the signal is not ignored (using *SIG_IGN*, see below), the system call will be interrupted, returning an error (i.e., -1), with the *errno* set to *EINTR* (Interrupted system call). If this flag is included, the system call will instead be restarted.

SA_ONESHOT

Normally, a signal handler remains valid until the program calls *execve()*, change the signal handler again, or terminates. If this flag is included, the signal handler is also reset to the default after the first execution of that handler.

¹A “long” system call is one which may wait for arbitrarily long, usually because the system call needs to do I/O or need to wait for time to pass.

4. The signal handler

A signal handler is a function of the following prototype:

```
void handler(int signum);
```

Here *signum* is the signal number which triggered the signal handler. There are two special handlers defined: *SIG_DFL* specifies that the default handler should be used, while *SIG_IGN* specifies that the signal should be ignored. A simple signal handler looks like this:

```
void my_sigchld_handler(int signum) { /* Call wait to avoid zombie */  
    int old_errno = errno;  
    while (waitpid(0, 0, WNOHANG)>0); /* repeatedly call waitpid */  
    errno = old_errno;  
}
```

It can be tricky to write correct signal handlers. The problem is that they **occur at any time**. E.g., suppose your signal handler calls *malloc()*. What will happen if, when the signal occurs, the process is calling *malloc()* itself? The result is chaotic: the *malloc()* of the main program is probably in the middle of modifying a linked list, e.g., of free memory regions, and the *malloc()* within the signal handler will see an inconsistent linked list. The result in general is a segmentation fault which is very difficult to locate.¹

So a signal handler should have minimal effect to the remainder of the code. Functions like *malloc()* should not be used in signal handler at all. Many functions modify global variables, and if those global variables are used outside the signal handler, the handler must either refrain from calling them, or be careful to save and restore the variables. The above example shows how this is done to system calls, which modify the global variable *errno*.

5. Signal numbers

Each signal number represents a different type of events. They are documented in the `signal(7)` man page, so type `man 7 signal` to view it. (For solaris, they are documented in `signal(3HEAD)`, and you have to type `man -s 3HEAD signal` to read it.) We end this note by a list of such signal numbers. Here are the most important signal numbers, including their usual meaning. In the description, there is a name, which you can use when using the “kill” command (e.g., `kill -KILL 12345` kills the process 12345 unconditionally). In programs you have to add *SIG* before it (like *SIGKILL*) to refer to the signal number (*SIGKILL*==9). The list is rather long, so you might want to skim it during the first reading. But they are also important, and all Unix programmers should understand them.

HUP (Hangup)

The control terminal “hangs up”. The word is coined when the typical terminal comes from a modem, when it means the serial line (usually, phone line) is broken. However, it also happens for virtual terminals when you logs out a text-mode terminal, or when you close a console window. This signal is responsible for terminating processes that are still running when the console dies, so the default action is naturally to terminate the process.

¹This is a “synchronization problem”, requiring the coordination between two independent “threads of execution”—the main program and the signal handler. One can solve the *malloc()* problem by temporarily preventing signal handlers from being called, during the time when the main program calls *malloc()*, and reenabling signal handling immediately afterwards. This can be achieved using *sigprocmask()*.

INT (Interrupt)

The user presses the *interrupt* character (usually Control-C) to terminate the process. The interrupt character of a terminal (and other special characters that appear later in this list) can be set through *ioctl()* of the terminal device, or by the command `stty`. This is done by programs like `emacs` which treat Control-C as a normal character. On the other hand, programs which simply ignore Control-C from terminals (like `less`) probably just setup a signal handler to ignore it. Again, the default action is to terminate.

QUIT (Quit)

The user types the *quit* character (usually Control-\) to “quit” the process. This provides an alternative way to break the program. Unlike the INT signal, the default action is to terminate *and make a core dump*. Of course, a core dump is a file containing the virtual memory of the process, allowing a programmer to use a debugger like `gdb` to find out where the program was executing, and what values were in its variables, before the program terminates.

ILL (Illegal instruction)

The program has attempted to execute an instruction that is not in the instruction set of the CPU. The compiler will not generate such instructions normally, so this happens rarely. If it does happen, it is probably because your program corrupted the memory of a return address in the stack, so that instruction pointer point to some memory that is not code at all. The default action is to terminate with a core.

ABRT (Abort)

The program has executed the *abort()* function, which causes an “abnormal program termination”. Again, the default action is to terminate with a core. The C library function *assert()* is the more convenient form of *abort()*, which stops the program if the argument is zero. For example, if you think that a variable *x* should be less than 1024 but worry that it is not (due to programming error), you can add the line `assert(x<1024);` to your program. If the program aborts, you know your worry is true. The nice thing is that it prints out a message telling you which line caused the error, and you get a core afterwards for you to check the program status using a debugger.

FPE (Floating Point Exception)

A “floating point exception” occurs. Despite whatever suggested by the name, this does not occur on floating point numbers operations: if you divide a floating point number by zero you will get a floating point number representing “infinity” instead of getting an exception. And if you take the square root of a negative number you get a floating point number representing “Not a Number” rather than an FPE. On the other hand, dividing an integer by zero will give you this signal. The default action is to terminate with a core.

KILL (Kill)

The user sends a kill signal (using, e.g., `kill -9`), wanting to unconditionally terminate the process. The action is always to terminate. Unlike most other signals described here, the signal cannot be trapped and handled by the program. This means that (1) the user can always terminate the program, and (2) the program has no chance to do any cleanup. This is sort of a last resort.

SEGV (Segmentation Violation)

The program executes a CPU instruction that it is not permitted to execute, usually because it tries to access wrong memory. The name is rather misleading: the signal is raised even on non-memory issues, e.g., trying to halt the computer completely using `asm("hlt");` gives you SEGV. The default action is to terminate with a core.

BUS (Bus error)

The process tries to output to a misaligned address. In some architectures (e.g., SPARC), it is an error to access multi-byte data at an address that is not a multiple of its size. For example, since `int` has a size of 4, it is an error to access an integer at a memory addresses which is not a multiple of 4. If you does generate such an address and dereference it (e.g., `*((int *)1)`), a bus error signal is generated. This does not happen to our computers of the 80x86 architecture, however. The default action is to terminate with a core.

PIPE (Broken pipe)

The program writes to a file descriptor referring to a pipe for which the read end is closed. E.g., if you execute `yes | true` (the `yes` program repeatedly prints `y` forever, while the `true` program exits immediately ignoring the input), the `yes` program will receive the PIPE signal. The default action, naturally, is to terminate.

ALRM (Alarm)

If you ask the OS to send you a signal at some later time, the signal you will receive is ALRM. This can be arranged through the `alarm()` system call if the amount of time is an integral number of seconds, and through `setitimer()` if you want finer granularity. If you use such system calls, you will naturally arrange a signal handler to deal with it. If you don't, the default is to terminate.

TERM (Terminate)

This is the default signal that you will send with the `kill` command (that is, if you don't specify a different signal to send), and the default action is to terminate.

USR1 (User defined 1) and USR2 (User defined 2)

These signals do not have default meaning, and is usually used for programs to provide users with a simple means of controlling it. The default action is again to terminate.

CHLD (Child terminated)

A child of the process is terminated. Its signal handle is a convenient place to call `waitpid()` if it the program wants to do something else. The default action is to ignore the signal.

STOP (Stop) and CONT (Continue)

These signals are used to temporarily stop a process and resume it later (usually by the user selecting the process to execute in the foreground using the `fg` shell command). Like KILL, STOP cannot be trapped.

TSTP (Terminal stop)

This is the signal sent when you press the *suspend* character, normally Control-Z. The default action is the same as STOP, to temporarily stop the process until a CONT signal is received. But unlike STOP, this signal can be trapped by the program. Note that the program is still there in the memory, waiting for resumption when it receives `fg` command. If the user wants to terminate the program instead, he should use Control-C or Control-\.

TTIN (Terminal Input) and TTOU (Terminal output)

These signals are sent to background processes which attempts to input or output to the terminal (output of background process triggers the TTOU signal only if the terminal *tostop* flag is set through `ioctl()`). The default action is to stop the process temporarily, until a CONT signal is received. For example, if you type `pine` (execute the `pine` program in the background), you will soon see a message saying it is stopped.