

Lecture 21: Signals

Philipp Koehn, David Hovemeyer

October 28, 2026

601.229 Computer Systems Fundamentals



Example code

Example code for today is on course website in `signals.zip`

Signals

Signals

- ▶ Software-level communication between processes
- ▶ Sending the signal from one process
- ▶ Receiving the signal by another process
 - ▶ ignore
 - ▶ terminate
 - ▶ catch signal
- ▶ Handled by kernel

Examples

Number	Name	Default	Corresponding Event
1	SIGHUP	terminate	terminal line hangup
2	SIGINT	terminate	interrupt from keyboard
3	SIGQUIT	terminate	quit from keyboard
4	SIGILL	terminate	illegal instruction
5	SIGTRAP	terminate & dump core	trace trap
9	SIGKILL	terminate*	kill process
18	SIGCONT	ignore	continue process if stopped
19	SIGSTOP	stop until SIGCONT*	stop signal not from terminal
20	SIGTSTP	stop until SIGCONT	stop signal from terminal

* = SIGKILL and SIGSTOP cannot be caught

Sending Signals

- ▶ From shell with command
`$ /bin/kill -9 2423`
- ▶ From shell with keystroke to running process
`$ start-my-process`
CTRL+C
 - ▶ CTRL+C: sends SIGINT
 - ▶ CTRL+Z: sends SIGTSTP
- ▶ There is also a C function and an Assembly syscall

Receiving Signals

- ▶ When kernel about to continue process, checks for signals
- ▶ If there is a signal, forces process to receive signal
- ▶ Each signal has a default action
 - ▶ ignore
 - ▶ terminate
 - ▶ terminate and dump core
 - ▶ stop
- ▶ Process can also set up a signal handler for customized response

Signal Handler

- ▶ Signal handler in C

```
#include "csapp.h"
```

```
void sigint_handler(int sig) {  
    printf("Caught SIGINT\n");  
    exit(0);  
}
```

```
int main() {  
    signal(SIGINT, sigint_handler);  
    pause();  
    return 0;  
}
```

- ▶ Now, process writes "Caught SIGINT" to stdout before terminating

Signal delivery, signal masks

Signal delivery

- ▶ In general, the OS kernel could deliver a signal to a process at any time
- ▶ Delivering a signal:
 - ▶ Pushing a special return address of code to restore the CPU state (so that process can continue normal execution when signal handler returns)
 - ▶ Creating stack frame for signal handler
 - ▶ Setting argument registers for signal handler
 - ▶ Jumping to signal handler
- ▶ Signals are normally delivered on the process's call stack
 - ▶ Really a *thread's* call stack, more about threads later on
- ▶ Process may designate a special area of memory to serve as a stack for received signals

Signals and asynchrony

- ▶ Signal delivery could occur before or after *any* instruction
- ▶ That means that signals are *asynchronous*
- ▶ “Asynchronous” means “could happen at any time” or “ordering is unpredictable”
- ▶ Signal handlers are asynchronous with respect to the rest of the program
- ▶ This can cause strange behavior!

A C program

```
#include "csapp.h"

#define NCOUNT 1000000000
volatile int count = 0;

int main(void) {
    // count up
    for (int i = 0; i < NCOUNT; i++) { count++; }
    printf("count=%d\n", count);
    return 0;
}
```

Note that “volatile” tells the compiler not to optimize away accesses to the count variable

Compiling and executing the program

```
$ gcc -O -Wall -c count.c  
$ gcc -o count count.o  
$ ./count  
count=100000000
```

Nothing surprising happened

Interval timers

- ▶ An *interval timer* is a means for notifying the process that an interval of time has elapsed
- ▶ Can be “one shot” or repeating
- ▶ The `setitimer` system call allows the process to create an interval timer
- ▶ When the timer elapses, OS kernel sends `SIGALRM` signal to process
- ▶ Let's change the program so that the handler for `SIGALRM` is also incrementing the global counter

Modified version of program

```
#include "csapp.h"

#define NCOUNT 100000000
volatile int stop = 0, nsigs = 0, count = 0;

void sigalrm_handler(int signo) {
    if (!stop) { nsigs++; count++; }
}

int main(void) {
    // handle SIGALRM signal
    code to set up signal handler for SIGALRM

    // arrange for SIGALRM to be delivered once every millisecond
    code to set up interval timer

    // count up
    for (int i = 0; i < NCOUNT; i++) { count++; }
    code to check final counts

    return 0;
}
```

Code to set up signal handler

```
// code to set up signal handler for SIGALRM  
struct sigaction sa;  
sigemptyset(&sa.sa_mask);  
sa.sa_flags = 0;  
sa.sa_handler = sigalrm_handler;  
sigaction(SIGALRM, &sa, NULL);
```

Note that to install a signal handler, `sigaction` is recommended over `signal`, for reasons we'll discuss soon

Using setitimer

```
// code to set up interval timer  
struct itimerval itv;  
itv.it_interval.tv_sec = 0;  
itv.it_interval.tv_usec = 1000; // 1000 microseconds = 1 millisecond  
itv.it_value = itv.it_interval;  
setitimer(ITIMER_REAL, &itv, NULL);
```

ITIMER_REAL means that the intervals are “real time” (not relative to CPU time used by the process)

Does the final count make sense?

```
// code to check final counts
stop = 1; // tell signal handler to stop incrementing count and nsigs
sleep(1); // wait a bit

printf("count=%d, NCOUNT=%d, nsigs=%d\n", count, NCOUNT, nsigs);
if (count == NCOUNT + nsigs) { printf("  count makes sense\n"); }
else                          { printf("  anomaly detected!\n"); }
```

In theory, the final value of count should be $\text{NCOUNT} + \text{nsigs}$

- ▶ NCOUNT is the number of increments (to count) in main
- ▶ nsigs is the number of calls to the signal handler (which also increments count)

Running the modified program

```
$ gcc -O -Wall -c alarm1.c
$ gcc -o alarm1 alarm1.o
$ ./alarm1
count=100000028, NCOUNT=100000000, nsigs=174
  anomaly detected!
```

What just happened?

Asynchrony and atomicity

- ▶ When a program
 - ▶ has code paths which execute asynchronously, and
 - ▶ the asynchronous paths update shared datathen anomalous behavior can be observed if either process executes code which is not *atomic*
- ▶ “Atomic” means “happens in its entirety, or not at all”
- ▶ Incrementing a variable is not (necessarily) atomic

Why increment is not atomic

- ▶ The statement `count++`; really means

```
1: tmp = count;  
2: tmp = tmp + 1;  
3: count = tmp;
```

where `tmp` is a register

- ▶ If `count` is updated by code executing asynchronously, the updated value could be overwritten by step 3
- ▶ The anomaly in our program execution shows this happening (the final value of `count` doesn't reflect all of the increments)

Clicker quiz!

Clicker quiz omitted from public slides

Synchronization, signal masks

- ▶ “Synchronization” means coordinating asynchronous accesses to shared data to avoid anomalous results
- ▶ For programs using signals we can use *signal masks* to synchronize signal handlers with the main program
- ▶ Signal mask = set of signals that are temporarily blocked
 - ▶ OS kernel will only deliver a signal if it isn't blocked
 - ▶ Note that not all signals may be blocked
 - ▶ For our example program, we can block SIGALRM to avoid the signal handler from executing at the wrong time

Modified main loop

```
sigset_t mask;
sigemptyset(&mask);
sigaddset(&mask, SIGALRM);

// count up
for (int i = 0; i < NCOUNT; i++) {
    sigprocmask(SIG_BLOCK, &mask, NULL);
    count++;
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
}
```

Running the modified program

```
$ gcc -O -Wall -c alarm2.c
$ gcc -o alarm2 alarm2.o
$ ./alarm2
count=100070462, NCOUNT=100000000, nsigs=70462
    count makes sense
```

No anomaly! However, note that the program took a very long time to run (more than 70 seconds) due to the overhead of calling `sigprocmask` in the main loop.

signal vs. sigaction

- ▶ Historically, the `signal` system call was used to register a signal handler on Unix systems
- ▶ New code should use `sigaction`
- ▶ Why?
 - ▶ Handlers registered using `signal` may get “unregistered” when the signal arrives
 - ▶ `signal` doesn't provide any mechanism for preventing signal handlers from being interrupted by other signals