

IT628: Systems Programming

Signal handling, SIGCHLD, non-local jumps

Signals

- **A signal is a software generated interrupt that is sent to a process by the OS because:**
 - it did something (oops)
 - the user did something (pressed ^C)
 - another process wants to tell it something (SIGUSR?)
- **There are a fixed set of signals that can be sent to a process.**

Some Signal Numbers

- Signals are identified by integers.
- Signal numbers have symbolic names. For example, SIGCHLD is the number of the signal to a parent when a child terminates.

<code>#define SIGHUP</code>	<code>1</code>	<code>/* Hangup. */</code>
<code>#define SIGINT</code>	<code>2</code>	<code>/* Interrupt. */</code>
<code>#define SIGQUIT</code>	<code>3</code>	<code>/* Quit. */</code>
<code>#define SIGILL</code>	<code>4</code>	<code>/* Illegal instruction. */</code>
<code>#define SIGTRAP</code>	<code>5</code>	<code>/* Trace trap. */</code>
<code>#define SIGABRT</code>	<code>6</code>	<code>/* Abort. */</code>

OS Structures for Signals

- For each process, the operating system maintains 2 integers with the bits corresponding to a signal numbers.
- The two integers keep track of:
 - pending signals
 - blocked signals
- With 32 bit integers, up to 32 different signals can be represented.

Example

- In the example below, the SIGINT (= 2) signal is blocked and no signals are pending.

Pending Signals

31	30	29	28	...	3	2	1	0
0	0	0	0	...	0	0	0	0

Blocked Signals

31	30	29	28	...	3	2	1	0
0	0	0	0	...	0	1	0	0

Sending, Receiving Signals

- **A signal is sent to a process setting the corresponding bit in the pending signals integer for the process.**
- **Each time the OS selects a process to be run on a processor, the pending and blocked integers are checked.**
- **If no signals are pending, the process is restarted normally and continues executing at its next instruction.**

Sending, Receiving Signals

- **If 1 or more signals are pending, but each one is blocked, the process is also restarted normally but with the signals still marked as pending.**
- **If 1 or more signals are pending and NOT blocked, the OS executes the routines in the process's code to handle the signals.**

Sending and Receiving Signals

- **Time may pass between generating and delivering signal**
- **The kernel checks for pending signals when it is about to return back to the user process**

Default Signal Handlers

- There are several default signal handler routines.
- Each signal is associated with one of these default handler routine.
- The different default handler routines typically have one of the following actions:
 - ignore the signal; i.e., do nothing, just return - Ign
 - terminate the process - Term
 - unblock a stopped process - Cont
 - block the process - Stop

User Defined Signal Handlers

- A process can replace the default signal handler for almost all signals (but not SIGKILL) by its own handler function.
- A signal handler function can have any name, but must have return type void and have one int parameter.
- Example, you might choose the name sigchld_handler for a signal handler for the SIGCHLD signal (termination of a child process). Then the declaration would be:

```
void sigchld_handler(int sig);
```

User Defined Signal Handlers

- When a signal handler executes, the parameter passed to it is the number of the signal.
- A programmer can use the same signal handler function to handle several signals. In this case the handler would need to check the parameter to see which signal was sent.
- On the other hand, if a signal handler function only handles one signal, it isn't necessary to bother examining the parameter since it will always be that signal number.

signal1.c

```
/* Handler for SIGINT, caused by Ctrl-C at keyboard. */
void handle_sigint(int sig) {
    printf("Caught signal %d\n", sig);
}

int main() {
    signal(SIGINT, handle_sigint);
    while (1) ;
    return 0;
}
```

Sending signals via kill()

■ `int kill(pid_t pid, int signal);`

- First parameter: id of destination process
- Second parameter: the type of signal to send
- Return value: 0 if signal was sent successfully

■ Better function name would be `sendsig()`

■ Example

```
pid_t iPid = getpid(); /* Process gets its id.*/  
kill(iPid, SIGINT); /* Process sends itself a  
                     SIGINT signal (commits  
                     suicide?) */
```

signal0.c

```
int main () {
    int stat;
    pid_t pid;
    if ((pid = fork()) == 0)
        while(1) ;
    else {
        kill(pid, SIGINT);
        wait(&stat);
        if (WIFSIGNALED(stat))
            psignal(WTERMSIG(stat),
                    "Child term due to");
    }
}
```

Blocking signals in handlers

- **What happens if the same signal is sent as its handler is running?**
- **Example: a process with a SIGINT handler**
 - First SIGINT received causes SIGINT handler to be called
 - Also causes SIGINT to become blocked for the process!
 - If another SIGINT occurs during handler execution, it is recorded in pending bit-vector, but it is not delivered!
 - When a signal handler returns, the blocked signal-type is automatically unblocked

Pop Quiz

```
int i = 0;
void handler(int a) {
    if (!i) {
        kill(getpid(), SIGINT);
    }
    i++;
}
int main() {
    signal(SIGINT, handler);
    kill(getpid(), SIGINT);
    printf("%d\n", i);
    return 0;
}
```

Circle possible values of i that could be printed:

- a. 0 b. 1 c. 2 d. Terminates with no output.

Background processes

- A shell program reads a command with 0 or more arguments and creates a child process to execute the command.
- The default is for the shell to wait for the command execution to finish; that is, to wait for the child process to terminate.
- However, if the command line ends with `&`, the shell doesn't wait. The shell prints a prompt and handles another command while the first command executes in the background.

Reaping Background Processes

- If the shell creates a child process, but doesn't wait for it (doesn't immediately call `waitpid`), it should call `waitpid` to reap the process when it terminates.
- How will the shell know when the child terminates in this case? The parent process (the shell) will receive a `SIGCHLD` signal when the child terminates.
- The shell needs to write a signal handler for the `SIGCHLD` signal and can call `waitpid` in that handler to reap the child process.
- Note: The default handler for `SIGCHLD` is to just ignore it.

pause function

■ **int pause(void);**

- **Suspends a process until it receives a signal**
- **Returns -1**
- **Can use alarm and pause to put a process to sleep for a specified amount of time**

signal3.c

```
/* Handle SIGCHLD signals. */
void handle_sigchld(int sig)
{
    pid_t pid;
    int status;
    pid = wait(&status);

    printf("Reaped child %d\n",pid);
    sleep(1);
}

int main() {
    int i;
    signal(SIGCHLD, handle_sigchld);
    for (i = 0; i < 3; i++) {
        if (fork() == 0) {
            /* Child-process code. */
            printf("Hello from child
                    %d\n", getpid());
            sleep(2);
            exit(0); /* Term child */
        }
    }
    /* Parent-process code. */
    while (1) /* Wait for children */
        pause(); /* to terminate. */
    return 0;
}
```

What went wrong?

- Run from the command line:

```
[user@host:~]> ./reaped
```

```
Hello from child 1099!
```

```
Hello from child 1101!
```

```
Hello from child 1100!
```

```
(2 seconds pass)
```

```
Reaped child 1101
```

```
(1 second passes)
```

```
Reaped child 1100
```

```
(...and then, nothing else...)
```

- Hmm, last child process doesn't get reaped.
 - ps reports that process 1099 is a zombie
 - 1099 ttys000 00:00:00 reaped <defunct>

Flaw in handler

```
/* Handle SIGCHLD signals. */
void handle_sigchld(int sig)
{
    pid_t pid;
    int status;
    pid = wait(&status);

    printf("Reaped child %d\n",pid);
    sleep(1);
}
```

Reaps one zombie per
SIGCHLD received

```
int main() {
    int i;
    signal(SIGCHLD, handle_sigchld);
    for (i = 0; i < 3; i++) {
        if (fork() == 0) {
            /* Child-process code. */
            printf("Hello from child
                    %d\n", getpid());
            sleep(2);
            exit(0); /* Term child */
        }
    }

    while (1)
        pause();
    return 0;
}
```

- Parent starts 3 child processes.
- Children terminate after two seconds.
- Kernel sends three **SIGCHLD** signals to the parent process.

Two zombies reaped

- **First SIGCHLD:**
 - Parent process handles the SIGCHLD signal
 - Kernel blocks other SIGCHLD signals while handler is running!
- **Second SIGCHLD:**
 - Parent can't receive it yet since it's still in its handler
 - Kernel records the pending SIGCHLD signal...
- **Third SIGCHLD:**
 - Parent is still in its SIGCHLD handler for the first signal
 - Since the process already has a pending SIGCHLD signal, the third SIGCHLD is discarded
- **When parent's SIGCHLD handler returns, kernel delivers pending SIGCHLD**
 - Third, dropped SIGCHLD is never delivered

One zombie reaped

- **First SIGCHLD:**
 - Kernel records the pending SIGCHLD signal ...
- **Second SIGCHLD:**
 - The first SIGCHLD signal is still pending
 - Since the process already has a pending SIGCHLD signal, the second SIGCHLD is discarded
- **Third SIGCHLD:**
 - The first SIGCHLD signal is still pending
 - Since the process already has a pending SIGCHLD signal, the third SIGCHLD is discarded
- **Kernel delivers the pending SIGCHLD signal**
 - Parent handles the first SIGCHLD signal

Corrected handler

- **The kernel does not queue up signals for delivery!**
 - Only has a pending bit-vector to track next signals to deliver
- **Instead, handler should reap as many zombies as it can, each time it's invoked**

```
void handle_sigchld(int sig) {
    pid_t pid;
    int status;

    while (1) {
        pid = waitpid(-1, &status, WNOHANG);
        if (pid <= 0)      /* No more zombie children to reap. */
            break;
        printf("Reaped child %d\n", pid);
    }
    sleep(1);
}
```

Pop Quiz

- **How many times can the handler execute?**
 - **0, 1, 2, 3, more than 3 times**

Signal Handlers and Non-local Jumps

- When a signal handler returns, execution resumes at the exact point in the process where it was suspended
- Signal handlers can use `siglongjmp()` to resume execution at a different point
 - ♦ Described as a “non-local goto”

Non-local jumps: sigsetjmp/siglongjmp

int sigsetjmp(sigjmp_buf env, int savesigs)

- ♦ identifies a return point for a subsequent jump via siglongjmp
- ♦ called once, but returns multiple times
 - returns 0 when sigsetjmp is first called
 - returns nonzero return value from siglongjmp call

void siglongjmp(sigjmp_buf env, int retval)

- ♦ called once but never returns
- ♦ triggers a return from the sigsetjmp call with the nonzero return value retval

setjmp/longjmp Example

```
sigjmp_buf buf;
void handler(int sig) {
    siglongjmp(buf, 1);
}

main() {
    signal(SIGINT, handler);
    if (!sigsetjmp(buf, 1))
        printf("starting\n");
    else
        printf("re-starting\n");
    while(1) {
        sleep(1);
        printf("processing...\n");
    }
}
```