Signals and Non Local jumps

龚欣洋

Signals

- 异常控制流
- 软件行为
- 由（用户）进程或内核发给（用户）进程

Signals

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Sending Signals

发送信号给进程或进程组

- `linux> /bin/kill -n PID`
- `int kill(pid_t pid, int sig);`
- 从键盘发送信号：
  Ctrl+C（终止前台作业）
  Ctrl+Z（挂起前台作业）
- `unsigned alarm(unsigned secs);`

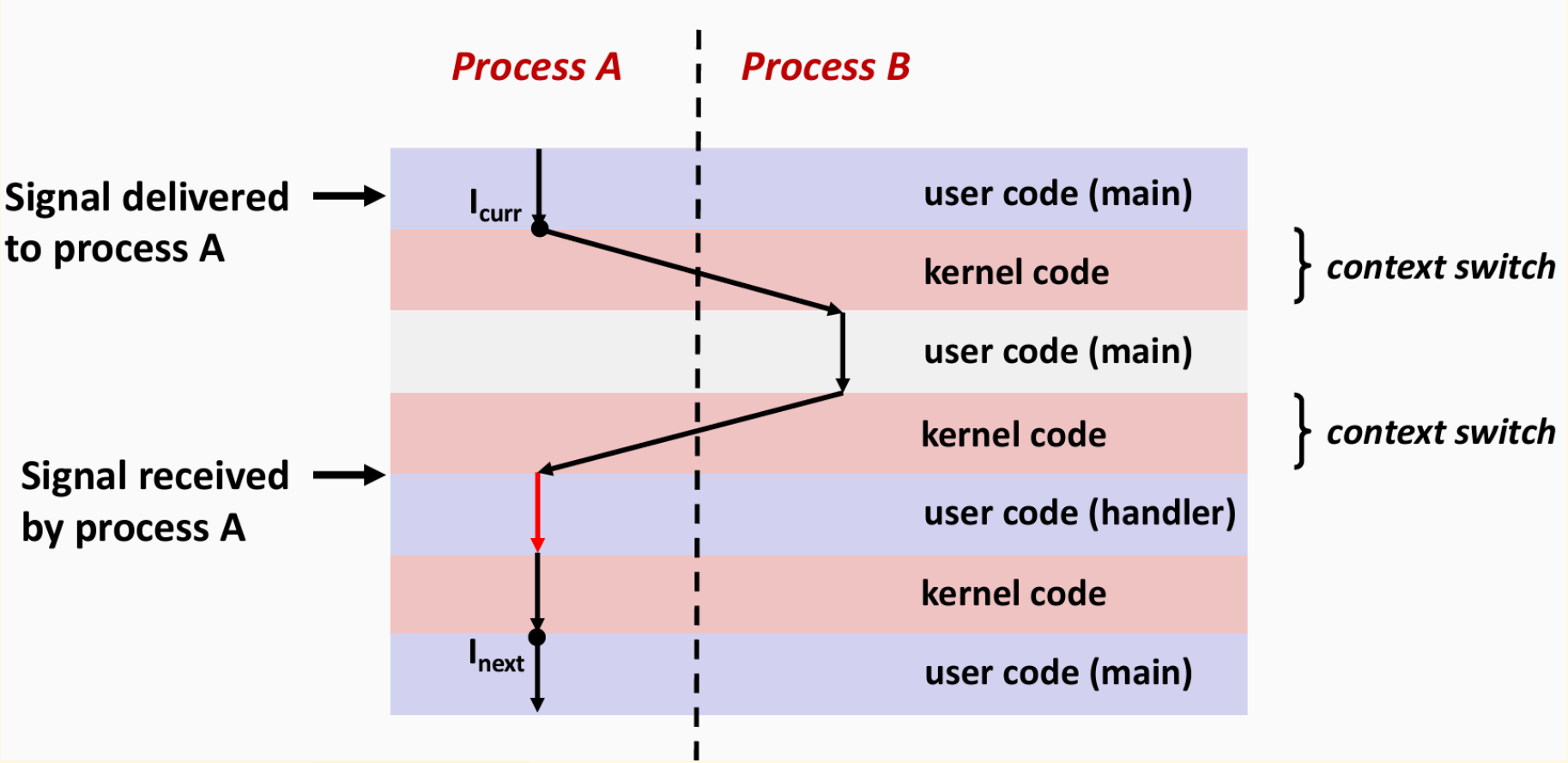
Pending/Blocked

内核为每个进程维护 `pending` 和 `blocked` 的信号集合
同一进程同一信号不能叠加

```
sigset_t set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGINT);  // 添加 SIGINT 信号到信号集
sigprocmask(SIG_BLOCK, &set, &oldset); // 阻塞 SIGINT 信号
sigprocmask(SIG_SETMASK, &oldset, NULL); // 恢复之前的信号掩码
```

```
sigsuspend(&old_set);
```

Signal handler



Signal handler in Bomb Lab

```
ubuntu@ics:~/lab2/bomb178$ ./bomb
Welcome to Mr.Gin's little bomb. You have 6 phases with
which to blow yourself up. Have a nice day! Mua ha ha ha!
^CSo you think you can stop the bomb with ctrl-c, do you?
Well...OK. :-)
```

```
int64_t sig_handler() __noreturn

    puts(str: "So you think you can stop the bo…  ")
    sleep(seconds: 3)
    __printf_chk(flag: 1, format: "Well...")
    fflush(fp: stdout)
    sleep(seconds: 1)
    puts(str: "OK. :-)")
    exit(status: 0x10)
```

Signal handler in Attack Lab

```
int64_t seghandler() __noreturn

    if (is_checker == 0)
        puts(str: "Ouch!: You caused a segmentation…  ")
        puts(str: "Better luck next time")
        notify_server(0, 0)
        exit(status: 1)
        noreturn
    puts(str: "Segmentation Fault")
    check_fail()
    noreturn
```

Attempt to blow up the system

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void handle_segv(int sig) {
    Sio_puts("Caught SIGSEGV");
    // int *ptr = NULL;
    // *ptr = 321;
}

int main() {
    signal(SIGSEGV, handle_segv);
    // Cause a segmentation fault
    int *ptr = NULL;
    *ptr = 123;  // This will cause a segmentation fault
    puts("Goodbye, World!");
    return 0;
}
```

子进程回收程序

```c
int main() {
    int N = 4;  // 子进程的数量
    const int a[] = {10,10,10,20};  // 每个子进程的休眠时间
    signal(SIGCHLD, chld_handler);  // 设置SIGCHLD信号处理函数
    // 生成N个子进程
    for (int j = 0; j < N; j++) {
        pid_t pid = Fork()
        if (pid == 0) { // 子进程
            printf("Child %d start\n", j + 1);
            sleep(a[j]);
            printf("Child %d finished\n", j + 1);
            exit(0);
        } else { // 父进程
            ccount++;
        }
    }
    while (ccount > 0) {
        puts("Daddy is working");
        sleep(4);
    }
    puts("All child processes have finished.");
    return 0;
}
```

Version 1

```c
void chld_handler(int signum) {
    pid_t pid;
    if ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_printf("Child Reaped, %d Left\n", ccount);
    }
}
```

```
Child 1 start
Child 2 start
Daddy is working...
Child 3 start
Child 4 start
Daddy is working...
Daddy is working...
Child 1 finished
Child 2 finished
Child Reaped, 3 Left
Daddy is working...
Child 3 finished
Child Reaped, 2 Left
Daddy is working...
Daddy is working...
Daddy is working...
Child 4 finished
Child Reaped, 1 Left
Daddy is working...
Daddy is working...
Daddy is working...
Daddy is working...
```

Version 2

```c
void chld_handler(int signum) {
    pid_t pid;
    while ((pid = wait(NULL)) > 0) {
        ccount--;
        Sio_printf("Child Reaped, %d Left\n", ccount);
    }
}
```

```
Child 1 start
Child 2 start
Daddy is working...
Child 3 start
Child 4 start
Daddy is working...
Daddy is working...
Child 1 finished
Child 2 finished
Child 3 finished
Child Reaped, 3 Left
Child Reaped, 2 Left
Child Reaped, 1 Left
Child 4 finished
Child Reaped, 0 Left
All child processes have finished.
```

Version 3

```c
void chld_handler(int signum) {
    pid_t pid;
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        ccount--;
        Sio_printf("Child Reaped, %d Left\n", ccount);
    }
}
```

```
Child 1 start
Daddy is working...
Child 2 start
Child 3 start
Child 4 start
Daddy is working...
Daddy is working...
Child 1 finished
Child 2 finished
Child 3 finished
Child Reaped, 3 Left
Child Reaped, 2 Left
Child Reaped, 1 Left
Daddy is working...
Daddy is working...
Daddy is working...
Child 4 finished
Child Reaped, 0 Left
All child processes have finished.
```

Nonlocal jump

```c
#include <setjmp.h>
//...
jmp_buf buf; //用于保存运行时状态
void bar() {
    if (cond2) longjmp(buf, 2);
}
void foo() {
    if (cond1) longjmp(buf, 1);
    bar();
}
int main() {
    switch(setjmp(buf)) {
        case 0:
            puts("传送门开启");
            foo();
            break;
        case 1:
            puts("传送成功 (1)");
            break;
        case 2:
            puts("传送成功 (2)");
            break;
        default:
            puts("有黑客！");
            break;
    }
}
```

Error handling in C++

```
int div() {
    int a, b;
    cin >> a >> b;
    if (b == 0)
        throw invalid_argument("Divided By Zero");
    return a / b;
 }
 void func() {
    cout << div() << endl;
 }
 int main() {
    try {
        func();
    } catch (exception& e) {
        cout<< e.what() << endl;
    }
    return 0;
 }
```

Extra - Flags for Sigaction

Macro: *int* **SA_NOCLDSTOP**

This flag is meaningful only for the `SIGCHLD` signal. When the flag is set, the system delivers the signal for a terminated child process but not for one that is stopped. By default, `SIGCHLD` is delivered for both terminated children and stopped children.

Setting this flag for a signal other than `SIGCHLD` has no effect.

Macro: *int* **SA_ONSTACK**

If this flag is set for a particular signal number, the system uses the signal stack when delivering that kind of signal. See Using a Separate Signal Stack. If a signal with this flag arrives and you have not set a signal stack, the normal user stack is used instead, as if the flag had not been set.

Macro: *int* **SA_RESTART**

This flag controls what happens when a signal is delivered during certain primitives (such as `open`, `read` or `write`), and the signal handler returns normally. There are two alternatives: the library function can resume, or it can return failure with error code `EINTR`.

The choice is controlled by the `SA_RESTART` flag for the particular kind of signal that was delivered. If the flag is set, returning from a handler resumes the library function. If the flag is clear, returning from a handler makes the function fail. See Primitives Interrupted by Signals.

https://www.gnu.org/software/libc/manual/html_node/Flags-for-Sigaction.html

thanks