Exceptional Control Flow III
(Signal)

Outline

- Signal Terminology
- Sending Signals
- Receiving Signals
- Signal Handling Issues
- Portable Signal Handling
- Explicitly Blocking Signals
- Synchronizing Flows to Avoid Nasty Concurrency Bugs

Signals

- Linux/Unix signal
  - 用于进程之间的通信
    - A small message that notifies a process that an event has occurred in the system
  - A higher-level software form of exception
    - That allows processes and the kernel to interrupt other processes

之前讲的 4 种 Exception 需要由硬件触发或程序自己触发

Example: The kill Program

- Kill program
  - is located in directory **`/bin/`**
  - sends an arbitrary signal to another process
    （不一定是杀死进程，可以是其他类型的信号）

`e.g. unix> kill -9 15213`

- sends signal 9 (SIGKILL) to process 15213.

Signals Type

- Signal 是通知一个进程的一个消息：
  - 系统中发生了某个类型的事件
- 每种 signal type 对应系统事件的某个类型
  - 低层次的硬件异常
    - 一般由 kernel 的 exception handler 处理，正常用户进程看不到
    - Signal 提供了一种机制，把这种 exception 的发生告知用户进程
  - 高层次的软件事件
    - 来自 kernel
    - 或来自其他用户进程

Hardware Events

- SIGFPE signal (number 8)
  - If a process attempts to divide by zero, then the kernel sends it a SIGFPE signal
- SIGILL signal (number 4)
  - If a process executes an illegal instruction, the kernel sends it a SIGILL signal.
- SIGSEGV signal (number 11)
  - If a process makes an illegal memory reference, the kernel sends it a SIGSEGV signal.

Software Events

- SIGINT signal (number 2)
  - While a process is running in the foreground
  - if you type a `ctrl-c`
  - then the kernel sends a SIGINT signal to the process
- SIGKILL signal (number 9)
  - A process can forcibly terminate another process
  - by sending it a SIGKILL signal
- SIGCHLD signal (number 17)
  - When a child process terminates or stops,
  - the kernel sends a SIGCHLD signal to the parent.

Linux Signals

`> man signal`

Number	Name	Default action	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 and dump core (1)	Trace trap
6	SIGABRT	Terminate and dump core (1)	Abort signal from `abort` function
7	SIGBUS	Terminate	Bus error
8	SIGFPE	Terminate and dump core (1)	Floating point exception
9	SIGKILL	Terminate (2)	Kill program
10	SIGUSR1	Terminate	User-defined signal 1
11	SIGSEGV	Terminate and dump core (1)	Invalid memory reference (seg fault)
12	SIGUSR2	Terminate	User-defined signal 2
13	SIGPIPE	Terminate	Wrote to a pipe with no reader
14	SIGALRM	Terminate	Timer signal from `alarm` function
15	SIGTERM	Terminate	Software termination signal
16	SIGSTKFLT	Terminate	Stack fault on coprocessor
17	SIGCHLD	Ignore	A child process has stopped or terminated
18	SIGCONT	Ignore	Continue process if stopped
19	SIGSTOP	Stop until next SIGCONT (2)	Stop signal not from terminal
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal

Linux Signals

- Linux 系统共定义了 64 种信号
  - 不可靠信号： 也称为非实时信号， standard signal
    - 不支持排队，信号可能会丢失，比如发送多次相同的信号，进程只能收到一次
    - signal number 为 1~31
  - 可靠信号： 也称为实时信号， real-time signal
    - 支持排队，信号不会丢失，发多少次，就可以收到多少次
    - signal number 为 32~64
    - 用户可以自定义这些信号

https://www.man7.org/linux/man-pages/man7/signal.7.html

Linux Signals   kill –l 可以查看所有信号

取值	名称	解释	默认动作
1	SIGHUP	挂起	
2	SIGINT	中断	
3	SIGQUIT	退出	
4	SIGILL	非法指令	
5	SIGTRAP	断点或陷阱指令	
6	SIGABRT	abort 发出的信号	
7	SIGBUS	非法内存访问	
8	SIGFPE	浮点异常	
9	SIGKILL	kill 信号	不能被忽略、处理和阻塞
10	SIGUSR1	用户信号 1	
11	SIGSEGV	无效内存访问	
12	SIGUSR2	用户信号 2	
13	SIGPIPE	管道破损，没有读端的管道写数据	
14	SIGALRM	alarm 发出的信号	
15	SIGTERM	终止信号	
16	SIGSTKFLT	栈溢出	

Linux Signals

取值	名称	解释	默认动作
17	SIGCHLD	子进程退出	默认忽略
18	SIGCONT	进程继续	
19	SIGSTOP	进程停止	不能被忽略、处理和阻塞
20	SIGTSTP	进程停止	
21	SIGTTIN	进程停止，后台进程从终端读数据时	
22	SIGTTOU	进程停止，后台进程想终端写数据时	
23	SIGURG	I/O 有紧急数据到达当前进程	默认忽略
24	SIGXCPU	进程的 CPU 时间片到期	
25	SIGXFSZ	文件大小的超出上限	
26	SIGVTALRM	虚拟时钟超时	
27	SIGPROF	profile 时钟超时	
28	SIGWINCH	窗口大小改变	默认忽略
29	SIGIO	I/O 相关	
30	SIGPWR	关机	默认忽略
31	SIGSYS	系统调用异常	

后续课程中，我们主要讨论 standard signal ，即非实时信号

Signal Terminology

- Two steps to transfer a signal to a destination process
  - Sending a signal
  - Receiving a signal

Sending a signal

- Kernel
  - 发送信号到目标进程
    - 通过修改目标进程上下文中的某些状态进行设置
- 信号产生的原因:
  - 用户操作
    - 如在 shell 中按 Ctrl+C
  - 内核检测到某个系统事件
    - 例如除以 0，子进程终止等
  - 进程调用了某些系统调用（如 kill）
    - kill()， raise()， sigqueue()， alarm()， setitimer()， abort()
    - 让内核给目标进程发信号
    - 进程可以发送信号给自己

Sending a signal

- kill()：用于向进程或进程组发送信号;
- sigqueue()：只能向一个进程发送信号，不能向进程组发送信号；主要针对实时信号提出，与 sigaction() 组合使用，当然也支持非实时信号的发送;
- alarm()：用于调用进程指定时间后发出 SIGALARM 信号;
- setitimer()：设置定时器，计时达到后给进程发送 SIGALRM 信号，功能比 alarm 更强大;
- abort()：向进程发送 SIGABORT 信号，默认进程会异常退出。
- raise()：用于向进程自身发送信号;

Receiving a signal

- 目标进程接收信号
  - 当进程从内核态刚切换到用户态时会检查并接受信号;
    - （稍后讲接收 signal 的机制）
  - 在内核态，信号不起作用;
  - 在用户态，所有未被屏蔽的信号都处理完毕;

Receiving a signal

- 对于收到的信号，进程可以有 4 种处理方式：
  - Ignore the signal (e.g., SIGCHILD)
  - Terminate (e.g., SIGKILL)
  - Stop (e.g., SIGSTOP) or restart (SIGCONT)
  - Catch the signal
    - by executing a **user-level** function called **signal handler**

Pending Signal

- Pending signal
  - 已经发送，但还没有接受的信号，叫做 pending signal
- Only one
  - 某一时刻，某种类型最多有一个 pending signal
- Not queued
  - 如果一个进程已经有一个类型 k 的 pending signal
  - 之后发送的 signal k 不会排队，而是直接被丢弃 .

以上讨论的是 standard signal，不是 real-time signal

Blocking a Signal

- 进程可以选择性 block 特定信号的接收
- 当一个信号被 block 时
  - 可以发送
  - 但是 pending signal 不会被接受
  - 直到该进程 unblock 这种 signal

Internal Data Structures

- For each process, the kernel maintains
  - the set of pending signals in the **`pending bit vector`**
  - the set of blocked signals in the **`blocked bit vector`**
- The kernel sets bit k in **`pending`**
  - whenever a signal of type k is delivered
- The kernel clears bit k in **`pending`**
  - whenever a signal of type k is received

Process Groups

- 可以发送信号给多个进程
  - Unix 系统提供一些发送信号给多个进程的机制
  - 这些机制都依赖于进程组（ process group ， pg ）
- 每个进程仅属于一个进程组
  - 进程组由一个非负数标记（进程组 ID ， pgid ）
- 默认情况下，自己和自己的父进程属于同一个进程组

Linux 中的线程也有线程组，类似于进程组

Process Groups

```
#include <unistd.h>
pid_t getpgrp(void);
```

returns: process group ID of the calling process

```
#include <unistd.h>
pid_t setpgid(pid_t pid, pid_t pgid);
```

returns: 0 on success, -1 on error

- setpgid
  - 把进程 pid 的进程组设置为 pgid
  - 如果 pid=0，那就是指当前进程
  - 如果 pgid=0，那就是用 pid 作为进程组 ID
    - 相当于“自立门户”

setgpid

`e.g., setpgid(0, 0);`

- Suppose process 15213 is the calling process,
- then this function call
  - creates a new process group whose process group ID is 15213
  - adds process 15213 to this new group

The kill Program

- Kill 是一个 Linux/Unix 自带的程序
  - **位于 `/bin/`**
  - 用于向其他进程发送 signal

`e.g. unix> kill -9 15213`

  - sends signal 9 (SIGKILL) to process 15213

The kill Program

- 可以用 kill 向 process group 发送 signal
  - negative PID:
  - sends the signal to every process in process group PID

e.g. `unix> kill -9 -15213`

  - sends a SIGKILL signal to every process in process group 15213

```
linux> ./forks 16
Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> /bin/kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Sending Signals from the Keyboard

- Job
  - Linux/Unix 中一个 shell 命令所创建的进程集合
  - 某一时刻最多有有一个 foreground job，以及 0 个或多个 background job

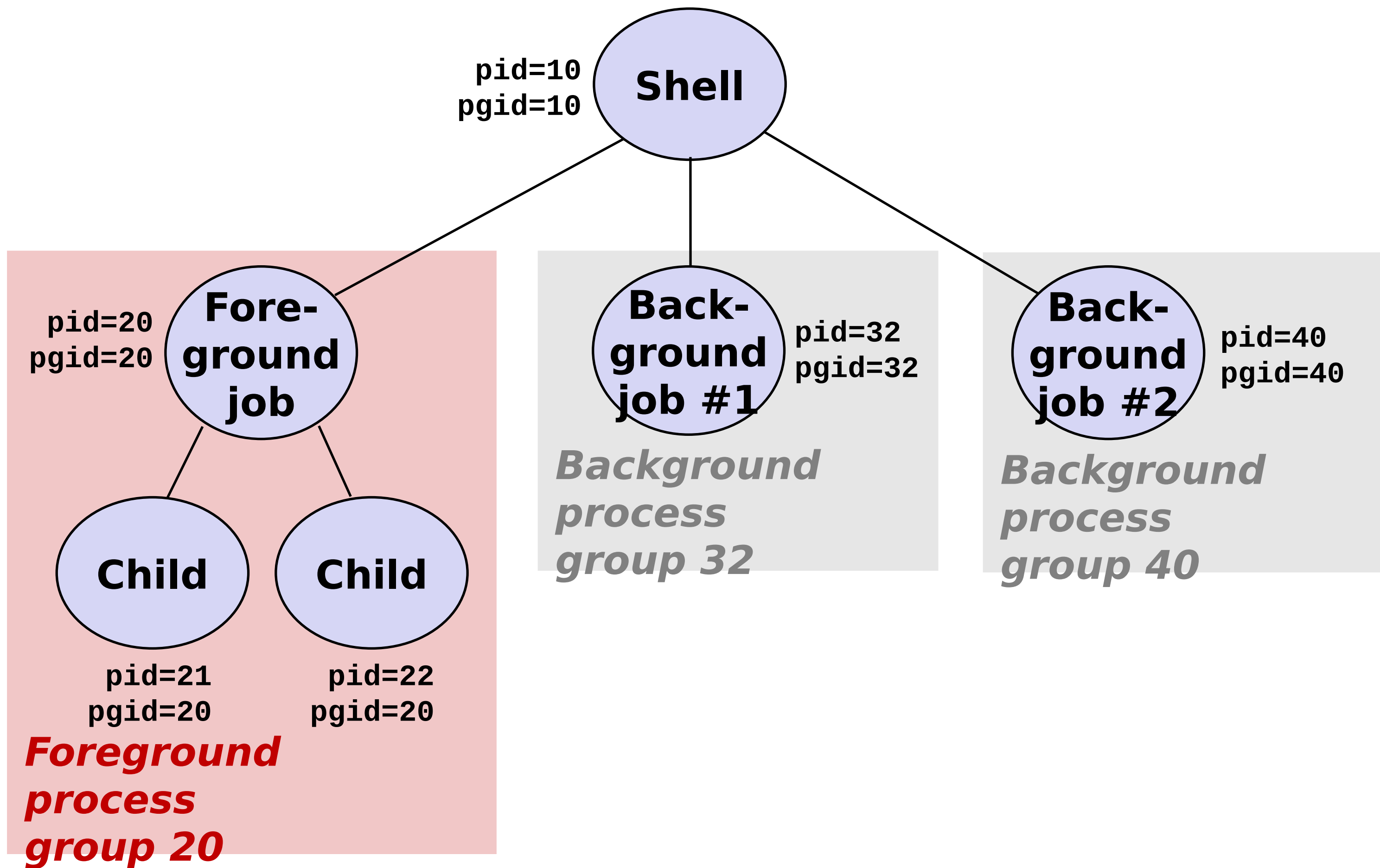
e.g. `unix> ls | sort`

  - a foreground job consisting of two processes connected by a pipe

Sending Signals from the Keyboard

- Shell 会为每个 job 创建一个 process group
  - 通常， process group ID 来自 job 中的某个 parent process

Sending Signals from the Keyboard



Sending Signals from the Keyboard

- 在键盘上输入 **`ctrl-c`**
  - causes a SIGINT signal to be sent to the shell
  - The shell catches the signal
  - Then sends a SIGINT to every process in the foreground process group
  - The result is to terminate the foreground job (default)

Sending Signals from the Keyboard

- 在键盘上输入 **`crtl-z`**
  - sends a SIGTSTP signal to the shell
  - The shell catches the signal
  - Sends a SIGTSTP signal to every process in the foreground process group
  - The result is to stop (suspend) the foreground job (default)

Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<types ctrl-z>
Suspended
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28107 pts/8    T      0:01 ./forks 17
28108 pts/8    T      0:01 ./forks 17
28109 pts/8    R+     0:00 ps w
bluefish> fg
./forks 17
<types ctrl-c>
bluefish> ps w
  PID TTY      STAT   TIME COMMAND
27699 pts/8    Ss     0:00 -tcsh
28110 pts/8    R+     0:00 ps w
```

STAT (process state) Legend:

First letter:
S: sleeping
T: stopped
R: running or ready

Second letter:
s: session leader
+: foreground proc group

See "man ps" for more details

fg: resume the most recently suspended or backgrounded job

Sending Signals with kill Function

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

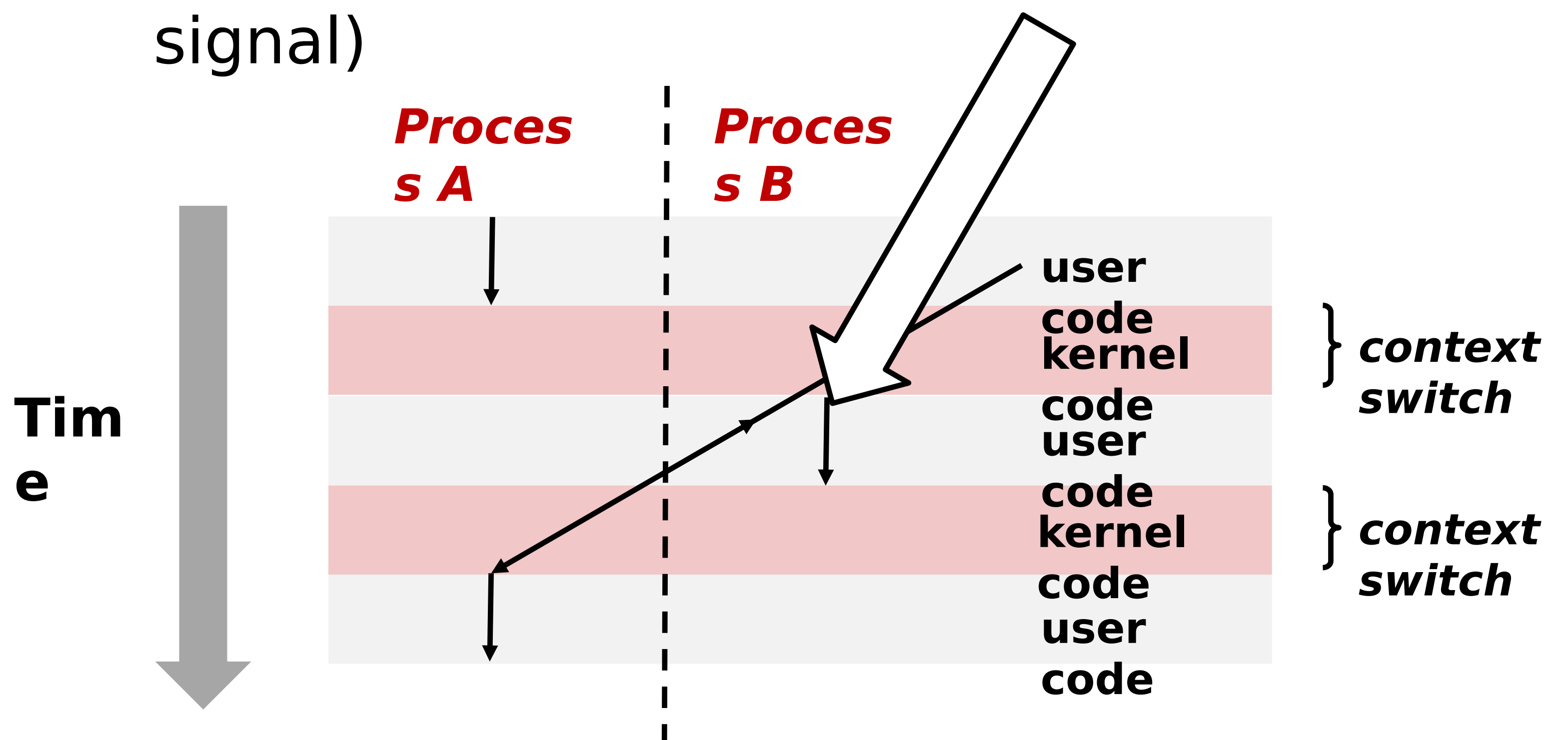
returns: 0 if OK, -1 on error

- If **pid** is greater than zero,
  - then the **kill** function sends signal number **sig** to process **pid**
- If **pid** is less than zero
  - then **kill** sends signal **sig** to every process in process group **abs(pid)**

```
1  #include "csapp.h"
2
3  int main()
4  {
5    pid_t pid;
6
7    /* child sleeps until SIGKILL signal received */
8    if ((pid = fork()) == 0) {
9      pause(); /* wait for a signal to arrive */
10       printf("control should never reach here!
  \n");
11       exit(0);
12   }
13
14   /* parent sends a SIGKILL signal to a child */
15   kill(pid, SIGKILL);
16   exit(0);
17 }
```

Receiving a Signal

- 先决条件：
  - 当内核从一个 exception handler 中返回，
  - 而且正准备将控制权传给进程 B 时 (加塞处理 signal)



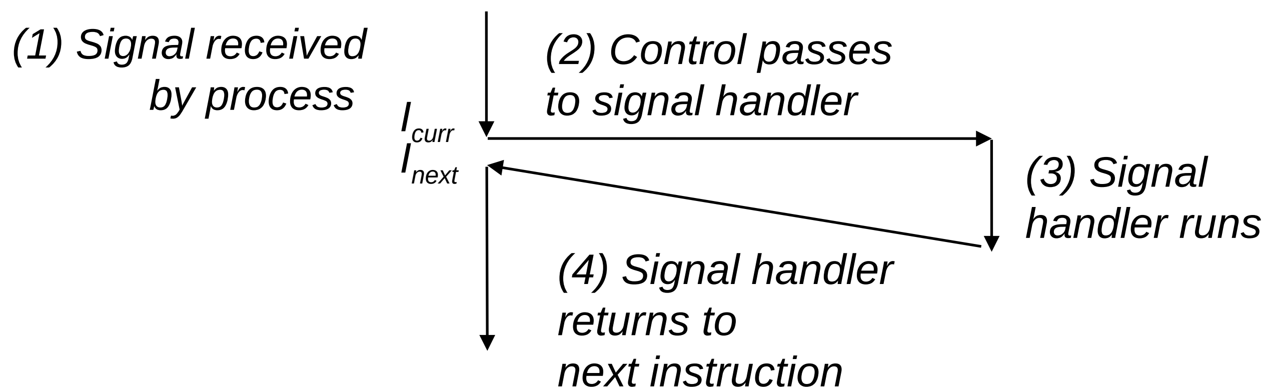
Receiving a Signal

- Actions
  - 内核检查 unblocked pending signals (**`pending & ~blocked`**)
  - If this set is empty (the usual case)
    - 内核将控制流转到进程 B 中的下一条指令 (I_{next})
  - However, if the set is nonempty
    - 内核选择其中一个 signal k (通常是最小的 signal)
    - 强制进程 B 接收 signal k.

注意： signal 由内核发送，接收进程从内核态转为用户态时，就触发上述检测和信号处理行为

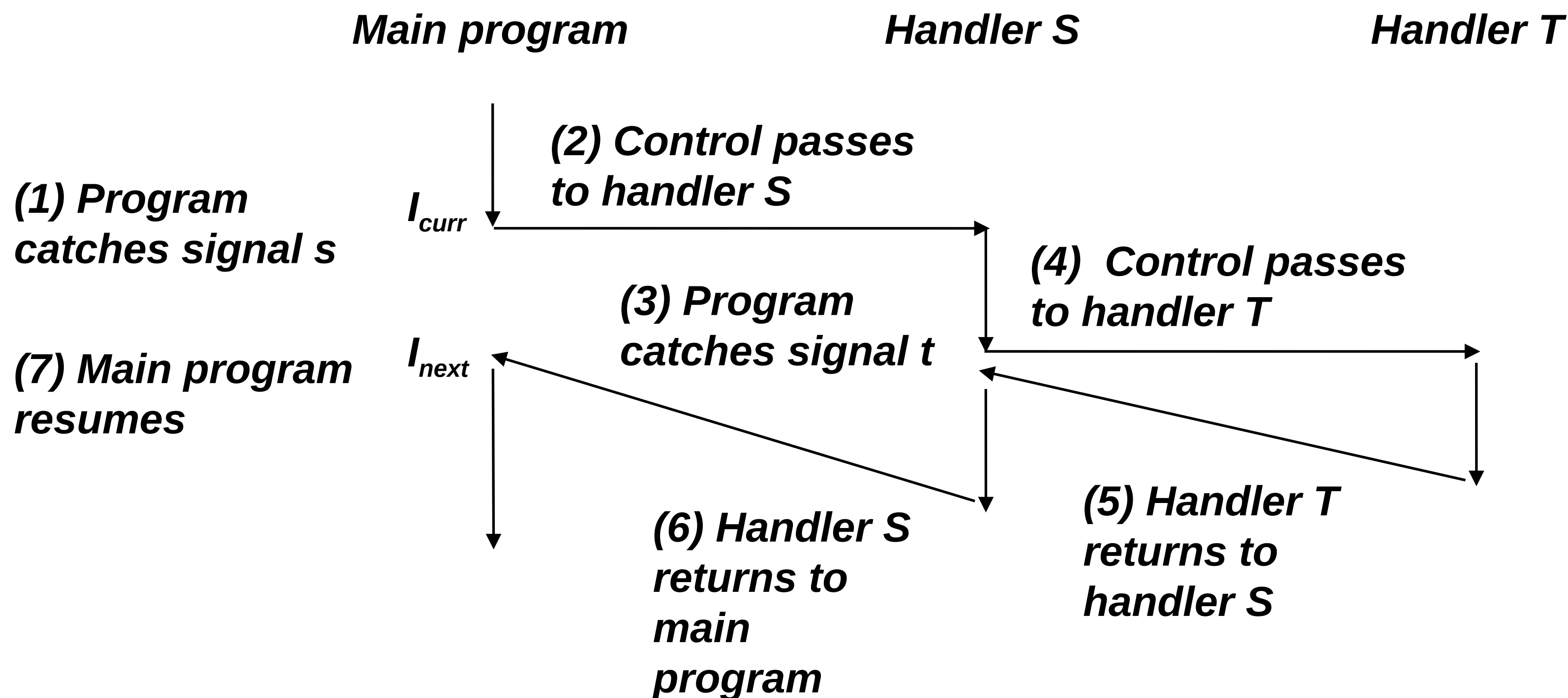
Receiving a Signal

- 进程 B 处理该 signal
- 处理完成后，控制流转向进程 B 的下一条指令（I_{next}）



Nested Signal Handlers

- 宏观：不同 signal 的 handler 是可以嵌套的 (相同 signal 的不行)
- 微观：一定是从 kernel->usr space 时进入信号处理程序



Receiving a Signal

- Each signal type has a predefined default action, which is one of the following:
  - The process terminates.
  - The process terminates and dumps core.
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal

Receiving a Signal

- SIGKILL
  - The default action is to terminate the receiving process
- SIGCHLD
  - The default action is to ignore the signal.
  - 如果父进程调用 wait()，则会监听和处理这个信号

Receiving a Signal

- A process can modify the default action associated with a signal
  - by using the **`signal`** function
  - The only exceptions are SIGSTOP and SIGKILL, whose default actions cannot be changed.

Signal Function

```
#include <signal.h>
typedef void (*signalhandler_t)(int);
signalhandler_t signal(int signum, signalhandler_t
handler)
```

returns: ptr to previous handler if OK,
SIG_ERR on error (does not set errno)

- typedef 作用是定义一种函数指针类型 signalhandler_t
- Three ways to change default actions:
  - If **handler** is SIG_IGN, then signals of type **signum** are ignored
  - If **handler** is SIG_DFL, then the action for signals of type **signum** reverts to the default action
  - Otherwise, change action to **handler** (called signal handler)

Signal Function

- Installing the handler
  - Changing the default action by passing the address of a handler to the **`signal`** function
- Catching the signal
  - The invocation of the handler
- Handling the signal
  - The execution of the handler

```c
1  #include "csapp.h"
2
3  void handler(int sig) /* SIGINT handler */
4  {
5    printf("Caught SIGINT\n");
6    exit(1);
7  }
8
9  int main()
10 {
11   /* Install the SIGINT handler */
12   if (signal(SIGINT, handler) == SIG_ERR)
13     unix_error("signal error");
14
15   pause(); /* wait for the receipt of a signal */
16
17   exit(0);
18 }
```

思考题：这个程序退出时的 status 是多少?

Signal Function

- Handling the signal
  - 当进程捕获了一个类型为 k 的 signal
  - K 对应的 handler 被激活
    - K 作为 handler 的参数传进去
  - 当 handler 执行到它的 return 语句时
    - 控制权传递回正常的程序控制流
      - 通常会返回被 signal 接收中断的位置

- 同一个 handler 可以用于捕获不同类型的信号
  - 所以才需要 signal 类型作为参数传进去

Sending Signals With the `alarm` Function

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

returns: remaining secs of previous alarm, or 0 if no previous alarm

- Arranges for the kernel to send a SIGALRM signal to the calling process **in secs seconds**（给自己设个闹钟）
- If **secs** is zero
  - any pending alarm is canceled（取消所有闹钟）
- The call to **alarm**
  - **取消掉任何待处理的（pending）闹钟**
  - 返回前一个 pending alarm 剩余的秒数
  - 如果之前没有 pending alarm，返回 0

定时 5 秒的炸弹

```
1  #include "csapp.h"
2  void handler(int sig)
3  {
4    static int beeps = 0;
5    printf("BEEP\n");
6    if (++beeps < 5)
7      alarm(1); /* next SIGALRM will be delivered in 1s */
8    else {
9      printf("BOOM!\n");
10     exit(0);
11   }
12 }

13 int main()
14 {
15   signal(SIGALRM, handler); /* install SIGALRM handler */
16   alarm(1); /* next SIGALRM will be delivered in 1s */
17
18   /* signal handler returns control here each time */
19   while (1);
20   exit(0);
21 }
```

```
linux> ./alarm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

code/ecf/counterprob.c

你的一个同事想要使用信号来让一个父进程对发生在一个子进程中的事件计数。其思想是每次发生一个事件时，通过向父进程发送一个信号来通知它，并且让父进程的信号处理程序对一个全局变量 counter 加一，在子进程终止之后，父进程就可以检查这个变量。然而，当他在系统上运行图 8-41 中的测试程序时，发现当父进程调用 printf 时，counter 的值总是 2，即使子进程向父进程发送了 5 个信号。他很困惑，向你寻求帮助。你能解释这个程序有什么错误吗？

```
1   #include "csapp.h"
2
3   int counter = 0;
4
5   void handler(int sig)
6   {
7       counter++;
8       sleep(1); /* Do some work in the handler */
9       return;
10  }
11
12  int main()
13  {
14      int i;
15
16      Signal(SIGUSR2, handler);
17
18      if (Fork() == 0) {  /* Child */
19          for (i = 0; i < 5; i++) {
20              Kill(getppid(), SIGUSR2);
21              printf("sent SIGUSR2 to parent\n");
22          }
23          exit(0);
24      }
25
26      Wait(NULL);
27      printf("counter=%d\n", counter);
28      exit(0);
29  }
```

code/ecf/counterprob.c

课堂练习

- 写一个不断寻找素数的程序，给信号 SIGINT 一个处理函数 ---- 报告当前找到的最大的素数
- 程序运行时， kill -2 pid ，程序就会打印出来目前已找到的最大的素数
- 寻找素数用暴力尝试的方式即可。程序只需要完成功能，不考虑效率

练习答案

```
#include        <stdio.h>
#include        <math.h>
#include        <signal.h>

void printmaxprime(int signum);
static long int maxprime;
void main()
{
        long int x=100,i,a,d=2000000000;
        signal(SIGINT, printmaxprime);
        for(x;x<=d;x++)
        {
                a=sqrt(x);
                for(i=2;i<=a;i++)
                        if(x%i==0) break;
                if(i>=a)
                {
                        maxprime = x;
                        //printf("maxprime is %d\n", maxprime);
                }
        }
}

void printmaxprime(int signum)
{
        printf("maxprime is %d\n", maxprime);
}
```

Signal Handling Issues

- Pending signals can be blocked
  - Unix signal handler 一般会屏蔽当前正在处理信号同一类型的 pending signals
- Pending signals are not queued
  - 在数据结构上，每一种信号类型只对应一个 bit，所以只能记录有 / 无信号，无法记录来了几个信号
  - 第二个及以后到达的 pending signal 只是被简单地丢弃了

Signal Handling Issues

- Signal handlers 和 main program 是并发的，并且可以共享全局变量
- 进程接收 signal 的方式和时机有时违反直觉
- 不同系统对 signal 的处理方式可能不一样

```
14 int main()
15 {
16   int i, n;  char buf[MAXBUF];
17   if (signal(SIGCHLD, handler1) == SIG_ERR)
18     unix_error("signal error");
19
20   /* parent creates children */
21   for (i = 0; i < 3; i++) {
22     if (fork() == 0) {
23       printf("Hello from child %d\n", (int)getpid());
24       sleep(1);
25       exit(0);
26     }
27   }
28   /* parent waits for input and then processes it */
29   if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
30     unix_error("read");
31
32   printf("Parent processing input\n");
33   while (1) ;
34
35   exit(0);
36 }
```

```
1  #include "csapp.h"
2
3  void handler1(int sig)
4  {
5    pid_t pid;
6
7    if ((pid = waitpid(-1, NULL, 0)) < 0)
8      unix_error("waitpid error");
9    printf("Handler reaped child %d\n", (int)pid);
10   Sleep(2);
11   return;
12 }
13
```

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
<cr>
Parent processing input
<ctrl-z>
Suspended

linux> ps
PID TTY STAT TIME COMMAND
...
10319 p5 T 0:03 signal1
10321 p5 Z 0:00 signal1 <zombie>
10323 p5 R 0:00 ps
```

```
1  #include "csapp.h"
2
3  void handler2(int sig)
4  {
5    pid_t pid;
6
7    while ((pid = waitpid(-1, NULL, 0)) > 0)
8      printf("Handler reaped child %d\n", (int)pid);
9    if (errno != ECHILD)
10     unix_error("waitpid error");
11   sleep(2);
12   return;
13 }
```

```
linux> ./signal2
Hello from child 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
<cr>
Parent processing input
```

Signal Handling Issues

- 编写简短、逻辑简单的 signal handler
- 在 signal handler 中只调用 async-signal-safe functions
  - 可重入 (reentrant) 或者不能被 signal 中断的函数
- 在 handler 中保存 errno，并在返回前恢复 errno
  - 避免 handler 中对 errno 的修改影响其他程序
- 如要访问全局数据结构，应 block 其他 signal
- 将全局变量声明为 volatile 和原子变量

ASF functions

很多常用的系统函数，如 printf、sprintf、 malloc、exit 等，都不 async-signal-safe 的

可以用 write 或者 sio package 中的函数输出，如 sio_puts

_Exit	fexecve	poll	sigqueue
_exit	fork	posix_trace_event	sigset
abort	fstat	pselect	sigsuspend
accept	fstatat	raise	sleep
access	fsync	read	sockatmark
aio_error	ftruncate	readlink	socket
aio_return	futimens	readlinkat	socketpair
aio_suspend	getegid	recv	stat
alarm	geteuid	recvfrom	symlink
bind	getgid	recvmsg	symlinkat
cfgetispeed	getgroups	rename	tcdrain
cfgetospeed	getpeername	renameat	tcflow
cfsetispeed	getpgrp	rmdir	tcflush
cfsetospeed	getpid	select	tcgetattr
chdir	getppid	sem_post	tcgetpgrp
chmod	getsockname	send	tcsendbreak
chown	getsockopt	sendmsg	tcsetattr
clock_gettime	getuid	sendto	tcsetpgrp
close	kill	setgid	time
connect	link	setpgid	timer_getoverrun
creat	linkat	setsid	timer_gettime
dup	listen	setsockopt	timer_settime
dup2	lseek	setuid	times
execl	lstat	shutdown	umask
execle	mkdir	sigaction	uname
execv	mkdirat	sigaddset	unlink
execve	mkfifo	sigdelset	unlinkat
faccessat	mkfifoat	sigemptyset	utime
fchmod	mknod	sigfillset	utimensat
fchmodat	mknodat	sigismember	utimes
fchown	open	signal	wait
fchownat	openat	sigpause	waitpid
fcntl	pause	sigpending	write
fdatasync	pipe	sigprocmask	

Safely Generating Formatted Output

- Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers.
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)   /* Put long */`
  - `void sio_error(char s[])   /* Put msg & exit */`

```
void sigint_handler(int sig) /* Safe SIGINT handler */
{
   Sio_puts("So you think you can stop the bomb with ctrl-c, do you?\n");
   sleep(2);
   Sio_puts("Well...");
   sleep(1);
   Sio_puts("OK. :-)\n");
   _exit(0);
}
```

课堂练习

- 之前例程的 signal handler 中，有哪些地方不符合以上的 guideline?

```
1  #include "csapp.h"
2
3  void handler2(int sig)
4  {
5    pid_t pid;
6
7    while ((pid = waitpid(-1, NULL, 0)) > 0)
8      printf("Handler reaped child %d\n", (int)pid);
9    if (errno != ECHILD)
10     unix_error("waitpid error");
11   sleep(2);
12   return;
13 }
```

课堂练习 - 答案

```
1  #include "csapp.h"
2
3  void handler2(int sig)
4  {
5    int olderrno = errno;
6
7    while (waitpid(-1, NULL, 0) > 0)
8      sio_puts("Handler reaped child\n");
9    if (errno != ECHILD)
10     sio_error("waitpid error");
11   sleep(2);
12   errno = olderrno;
13 }
```

Signal Handling Issues

- System calls can be interrupted
  - Slow system calls
    - **`read`**, **`wait`**, and **`accept`**
  - On some old systems
    - 当一些 slow system call 被信号中断后， signal handler 捕获了一个信号并进行处理和返回后， system call 就不继续执行了
    - 而是立刻返回用户程序，并设置全局变量 errno 来报错
    - 例如前面提到的 sleep，可能提前返回

sigaction Function

```
#include <signal.h>
int sigaction(int signum,
              struct sigaction *act,
              struct sigaction *oldact);
                         returns: 0 if OK, -1 on error

struct sigaction {
  void (*sa_handler)();       /* addr of signal
handler,
                               or SIG_IGN, or SIG_DFL */
  sigset_t sa_mask; /* additional signals to block */
  int sa_flags;                   /* signal options */
} ;
```

`sigaction` Function

- int **sigaction**(int signum, struct sigaction *act, struct sigaction *oldact);
  - signum： 要操作的 signal 信号
  - act： 设置对 signal 信号的新处理方式
  - oldact： 原来对信号的处理方式
  - 返回值： 0 表示成功， -1 表示有错误发生

```
1  handler_t *Signal(int signum, handler_t *handler)
2  {
3      struct sigaction action, old_action;
4
5      action.sa_handler = handler;
       /* only block sigs of type being handled */
6      sigemptyset(&action.sa_mask);
       /* restart syscalls if possible */
7      action.sa_flags = SA_RESTART;
8
9      if (sigaction(signum, &action, &old_action) < 0)
10         unix_error("Signal error");
11     return (old_action.sa_handler);
12 }
```

signal() vs. sigaction()

- signal()： 不支持信号传递信息，主要用于非实时信号安装;
- sigaction(): 支持信号传递信息，可用于所有信号安装（**struct sigaction 中可设置 3 参数的扩展 handler**）；

Signal Function

- 只有正在被处理的信号类型被屏蔽
  - 同类型信号无法嵌套
- 在所有版本的 signal 实现中，信号都不支持排队
- 被中断的系统调用，会在可以的时候自动重新执行

Signal Function

- 一旦通过 signal 函数注册信号对应的处理函数，只有当再次调用 signal 函数，并将 handler 指定为 SIG_IGN 或 SIG_DFL 才能取消
  - SIG_IGN： 忽略信号
  - SIG_DFL： 恢复默认处理方式

Explicitly Blocking Signals

```
#include <signal.h>

int  sigprocmask(int how, const sigset_t *set,
                     sigset_t *oldset)
int  sigemptyset(sigset_t *set) ;
int  sigfillset(sigset_t *set);
int  sigaddset(sigset_t *set, int signum);
int  sigdelset(sigset_t *set, int signum);
                         Return: 0 if OK, -1 on error
Int  sigismember(const sigset_t *set, int signum);
          Returns: 1 if member, 0 if not, -1 on error
```

Explicitly Blocking Signals

- sigprocmask
  - 改变当前屏蔽信号集合（blocked bit vector）的状态
  - SIG_BLOCK: add the signals in set to blocked (blocked = blocked | set)
  - SIG_UNBLOCK: Remove the signals in set from blocked (blocked = blocked & ~set)
  - SIG_SETMASK: blocked = set
  - If oldset is non-NULL, the previous value of the blocked bit vector is stored in oldset

Explicitly Blocking Signals

- Signal set
  - Sigemptyset：将 set 每一个 bit 设置为 0
  - Sigfillset ：将 set 每一个 bit 设置为 1
  - Sigaddset ：在 set 中增加一个屏蔽信号
  - Sigdelset：在 set 中删除一个屏蔽信号

Explicitly Blocking Signals

```
1     sigset_t mask, prev_mask;
2
3     Sigemptyset(&mask);
4     Sigaddset(&mask, SIGINT);
5
6     /* Block SIGINT and save previous blocked set */
7     Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
      .
8     :   // Code region that will not be interrupted by SIGINT
      .
9     /* Restore previous blocked set, unblocking SIGINT */
10    Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
11
```

Linux 信号处理程序的难点

- 处理程序与主程序并发，共享同样的全局变量，因此可能互相干扰；
- 如何以及何时接收信号的规则常常有违人的直觉；
- 不同的系统有不同的信号处理语义

课堂练习

- 下列程序的输出是什么?

A child created via fork(2) inherits a copy of its parent's signal dispositions.

```
1   volatile long counter = 2;
2
3   void handler1(int sig)
4   {
5       sigset_t mask, prev_mask;
6
7       Sigfillset(&mask);
8       Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
9       Sio_putl(--counter);
10      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
11
12      _exit(0);
13  }
14
15  int main()
16  {
17      pid_t pid;
18      sigset_t mask, prev_mask;
19
20      printf("%ld", counter);
21      fflush(stdout);
22
23      signal(SIGUSR1, handler1);
24      if ((pid = Fork()) == 0) {
25          while(1) {};
26      }
27      Kill(pid, SIGUSR1);
28      Waitpid(-1, NULL, 0);
29
30      Sigfillset(&mask);
31      Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
32      printf("%ld", ++counter);
33      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
34
35      exit(0);
36  }
```

课堂练习 - 答案

- 下列程序的输出是什么?
- 213

```
1   volatile long counter = 2;
2
3   void handler1(int sig)
4   {
5       sigset_t mask, prev_mask;
6
7       Sigfillset(&mask);
8       Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
9   1   Sio_putl(--counter);
10      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
11
12      _exit(0);
13  }
14
15  int main()
16  {
17      pid_t pid;
18      sigset_t mask, prev_mask;
19
20  2   printf("%ld", counter);
21      fflush(stdout);
22
23      signal(SIGUSR1, handler1);
24      if ((pid = Fork()) == 0) {
25          while(1) {};
26      }
27      Kill(pid, SIGUSR1);
28      Waitpid(-1, NULL, 0);
29
30      Sigfillset(&mask);
31      Sigprocmask(SIG_BLOCK, &mask, &prev_mask);  /* Block sigs */
32  3   printf("%ld", ++counter);
33      Sigprocmask(SIG_SETMASK, &prev_mask, NULL); /* Restore sigs */
34
35      exit(0);
36  }
```

Nasty Concurrency Bugs

```
10 int main(int argc, char **argv)
11 {
12     int pid;
14     signal(SIGCHLD, handler);
15     initjobs(); /* initialize the job list */
16
17     while(1) {
18         /*child process */
19         if ((pid = fork()) == 0)
20             execve(“/bin/ls”, argv, NULL);
21
22         /* parent process */
23         addjob(pid) ; /* Add the child to the job list */
24     }
25     exit(0)
26 }
```

Nasty Concurrency Bugs

```
1  void handler(int sig)
2  {
3     pid_t pid ;
      /*reap a zombie child */
4     while (pid = waitpid(-1, NULL, WNOHANG) > 0)
        /* delete the child from the job list */
5       deletejob(pid);
6
7     if (errno != ECHILD)
8       unix_error(“waitpid error”) ;
9  }
```

该程序有什么bug？

Nasty Concurrency Bugs

- Possibly incorrect sequence of events
  - Parent executes **`fork`** and kernel schedules the newly created child
  - Child terminates and kernel delivers a SIGCHLD signal to parent
  - Parent receives the SIGCHLD signal and runs the signal handler
  - Signal handler calls **`deletejob`**
  - Parent return from **`fork`** and calls **`addjob`**

对还有没有添加的 job 进行删除!

```
1  void handler(int sig)
2  {
3     pid_t pid ;
      /*reap a zombie child */
4     while (pid = waitpid(-1, NULL, WNOHANG) > 0)
5       deletejob(pid); /* delete the child from the job list */
6     if (errno != ECHILD)
7       unix_error("waitpid error") ;
8  }
9
10 int main(int argc, char **argv)
11 {
12    int pid;
13    sigset_t mask;
14
15    Signal(SIGCHLD, handler);
16    initjobs(); /* initialize the job list */
17
```

```
18      while(1) {
19          Sigemptyset(&mask);
20          Sigaddset(&mask, SIGCHLD);
21          Sigprocmask(SIG_BLOCK, &mask, NULL);
22
23          /*child process */
24          if ((pid = fork()) == 0) {
25              Sigprocmask(SIG_UNBLOCK, &mask, NULL);
26              execve("/bin/ls", argv, NULL);
27          }
28
29          /* parent process */
30          addjob(pid); /* Add the child to the job list */
31          Sigprocmask(SIG_UNBLOCK,&mask, NULL);
32      }
33      exit(0)
34 }
```

Block SIGCHLD

Unblock SIGCHLD

Explicitly Waiting for Signals

- Handlers for program explicitly waiting for SIGCHLD to arrive.

```
volatile sig_atomic_t pid;

void sigchld_handler(int s)
{
    int olderrno = errno;
    pid = Waitpid(-1, NULL, 0); /* Main is waiting for nonzero pid */
    errno = olderrno;
}

void sigint_handler(int s)
{
}
```

Explicitly Waiting for Signals

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);
        /* Parent */
        pid = 0;
        Sigprocmask(SIG_SETMASK, &prev, NULL); /* Unblock SIGCHLD */

        /* Wait for SIGCHLD to be received (wasteful!) */
        while (!pid);
        /* Do some work after receiving SIGCHLD */
            printf(".");
    }
    exit(0);
}
```

Explicitly Waiting for Signals

- Program is correct, but very wasteful
- Other options:

```
while (!pid)  /* Race! */
    pause();
```

```
while (!pid) /* Too slow! */
   sleep(1);
```

- Pause/sleep 可能被其他信号中断，所以需要一个循环
- Race: 在 while 条件达成后、 pause 开始前收到 SIGCHLD 信号，则 pause 可能会永远睡眠（没有原子性保障）

- Solution: `sigsuspend`

Waiting for Signals with `sigsuspend`

- `int sigsuspend(const sigset_t *mask)`

- Equivalent to atomic (uninterruptable) version of:

```
sigprocmask(SIG_SETMASK, &mask, &prev);
pause();
sigprocmask(SIG_SETMASK, &prev, NULL);
```

Waiting for Signals with sigsuspend

```
int main(int argc, char **argv) {
    sigset_t mask, prev;
    Signal(SIGCHLD, sigchld_handler);
    Signal(SIGINT, sigint_handler);
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* Block SIGCHLD */
        if (Fork() == 0) /* Child */
            exit(0);

       /* Wait for SIGCHLD to be received */
        pid = 0;
        while (!pid)
            Sigsuspend(&prev);

       /* Optionally unblock SIGCHLD */
        Sigprocmask(SIG_SETMASK, &prev, NULL);
        /* Do some work after receiving SIGCHLD */
        printf(".");
    }
    exit(0);
}
```