# Shell Lab Solution

# 信号是什么

> From WKIPEDIA
>
> 信号是发送给正在运行的程序的标准化消息，用于触发特定行为，如退出或错误处理。它们是一种有限的进程间通信（IPC）形式，通常用于 Unix、类 Unix 以及其他符合 POSIX 标准的操作系统。
>
> 信号是发送给进程或同一进程内的特定线程的异步通知，用于通知其某个事件的发生。信号的常见用途包括中断、暂停、终止或杀死进程。

一个信号就是一条“小消息”，它告知进程系统中发生了一个特定类型的事件。信号机制的一个重要特点是它们是异步的。这意味着程序可能在执行任何操作的时候都有可能接收到信号，因此它们需要被设计得能随时处理这些突如其来的信号。

# 为什么需要信号机制?

## 一个办公室的例子 

想象一下，你在一个大办公室工作，你的工作需要专注和集中。在这个办公室里，有一个“管理员”负责确保一切正常运行。这个管理员就像操作系统，而你和你的同事们就像不同的程序。

为什么操作系统需要信号机制?

1. **通知发生的事件**：假设突然停电了（一个紧急事件），管理员（操作系统）需要立刻通知每个人（程序）停止使用电脑并采取适当措施。在计算机中，这类紧急事件（如硬件错误）就通过信号来通知给程序。
2. **让员工（程序）优雅地结束工作**：如果办公室即将关闭，管理员需要告诉每个人收拾东西并离开。在计算机中，当你关闭一个程序时，操作系统通过发送一个信号来告诉程序“收拾结束，安全退出”。

为什么应用开发者需要信号机制?

1. **准备突发情况**：假如你正在处理一份重要文件，突然管理员告诉你要疏散办公室（发生了大地震）。如果你提前准备了一个计划（比如保存所有工作），那么即使在紧急情况下，你也能保证工作不会丢失。在编程中，开发者可以编写代码来处理信号，确保即使在程序意外终止时也能安全地保存数据。
2. **响应特定指令**：如果管理员告诉你休息一会儿，你可以立刻停下手头的工作，休息一下。同样，程序可以被设计为在收到特定信号（比如用户按下Ctrl+C）时执行特定操作，比如暂停运行或切换到不同的任务。

## 两个视角下的需求

操作系统视角：

1. **异步事件处理**：操作系统需要一种机制来处理外部或内部发生的异步事件，如用户输入、硬件中断或其他系统级事件。信号提供了一种方式，使得操作系统能够异步地通知进程某个事件的发生。
2. **进程控制和管理**：操作系统使用信号来控制进程的行为，例如终止不响应的进程或响应用户请求的中断。这是维护系统稳定性和响应性的关键部分。
3. **异常处理**：对于应用程序错误（如访问非法内存、除零操作等），操作系统可以通过发送信号来通知相应的进程，允许它们有机会以有序的方式响应这些异常。
4. **系统级信号**：操作系统使用信号来实现某些系统级操作，如关闭计算机时发送的 `SIGTERM`，让所有进程有机会正确关闭。

用户程序视角：

1. **自定义响应**：开发者可以自定义程序对特定信号的响应。这允许程序在接收到信号时执行清理操作、保存状态、或进行其他必要的处理，而不是简单地意外终止。
2. **增强程序的鲁棒性**：通过处理如 `SIGFPE` 这样的信号，应用程序可以在遇到严重错误时执行特定的错误处理逻辑，比如记录错误信息，防止程序崩溃导致的数据丢失。

3. **实现优雅的停止**：开发者可以编写代码来捕捉如 `SIGTERM` 这样的信号，以便在程序需要停止时执行优雅的关闭流程，例如释放资源、关闭文件等。
4. **进程间通信**：在一些情况下，信号可以作为进程间通信的一种简单形式，尽管它的能力有限，但对于某些特定场景（如通知子进程退出）可能是足够的。

一句话总结：信号机制是操作系统用于异步通知程序发生特定事件的一种方式，允许程序对这些事件作出适当反应。

# 信号的类型有哪些?

## POSIX 标准

```
# The number of possible signals is limited. The first 31 signals are standardized in LINUX; all have names starting with SIG. Some are from POSIX.
man 7 signal
```

| 信号 | 描述 |
| --- | --- |
| `SIGHUP` | 在控制终端检测到挂起或控制进程死亡时发出。 |
| `SIGINT` | 如果用户发送中断信号（`Ctrl + C`），则发出此信号。 |
| `SIGQUIT` | 如果用户发送退出信号（`Ctrl + D`），则发出此信号。 |
| `SIGFPE` | 如果尝试进行非法的数学运算，则发出此信号。 |
| `SIGKILL` | 如果进程接收到这个信号，它必须立即退出，并且不会执行任何清理操作。 |

| 信号 | 描述 |
| --- | --- |
| `SIGTERM` | 软件终止信号（默认由kill发送）。 |
| `SIGALRM` | 闹钟信号（用于计时器）。 |

## 为什么需要这么多信号类型？一个服务器程序的例子

假设有一个服务器程序，它使用多个子进程来处理客户端请求。这个服务器需要能够：

1. **优雅地关闭**：当需要停止服务时，它应该优雅地关闭，释放资源，断开所有客户端连接。

   服务器程序可以注册一个信号处理函数来捕获 `SIGTERM` 信号。当操作系统发送这个信号时（通常是由系统管理员发出的停止命令），服务器将执行该信号处理函数。在信号处理函数中，服务器将关闭所有打开的连接，释放资源，并正常退出。

2. **处理子进程退出**：当子进程完成任务并退出时，它需要通知父进程以避免产生僵尸进程来消耗系统资源。

   服务器程序可以使用 `SIGCHLD` 信号来检测子进程何时退出。当一个子进程结束运行时，操作系统会向父进程发送 `SIGCHLD` 信号。服务器在其信号处理函数中调用 `waitpid()` 来收集子进程的退出状态，这样可以防止子进程变成僵尸进程。

3. **处理超时**：如果某个子进程在执行任务时花费了太长时间（比如因为某个阻塞操作），服务器需要能够终止这个子进程。

   对于可能长时间运行的子进程，服务器可以设置一个闹钟信号（`SIGALRM`）。通过调用 `alarm()` 函数，服务器可以在指定的秒数后收到 `SIGALRM` 信号。如果在设定的时间内子进程没有完成任务，`SIGALRM` 信号的处理函数可以被用来终止子进程。

一句话总结：不同的信号类型允许操作系统和应用程序对各种不同的事件和异常情况进行细致和专门化的响应。

# 当信号出现时发生了什么?

当信号发生时，你可以把它想象成是电脑给运行中的程序发送的一个特别通知，告诉程序需要马上处理某件事。对于这个通知，程序有三种处理方式：

1. **忽略它**：就像忽略手机上的一个无关紧要的通知一样，程序有时可以选择不理会某些信号。但并不是所有信号都可以被忽略，有些信号太重要或太严重了，比如“0作除数”这样的错误，程序无法忽略。
2. **捕捉并处理它**：这就像你的手机收到通知，然后你打开一个应用来处理这个通知。程序可以预先设定一个特定的函数，当收到信号时就运行这个函数。这个函数可以选择结束程序，或者处理问题后让程序继续运行。
3. **让它的默认行为发生**：每种信号都有一个“默认的反应”，就像某些手机通知自动消失，而另一些则需要你的回应。程序对于不同的信号有不同的默认反应，比如停止运行、完全关闭或暂停。
    - 进程终止
    - 进程终止并转储内存
    - 进程停止（挂起）直到被 SIGCONT 信号重启
    - 进程忽略该信号。

进程可以通过使用 signal 函数修改和信号相关联的默认行为。唯一的例外是 `SIGSTOP` 和 `SIGKILL`，它们的默认行为是不能修改的。

思考？为什么有的信号不能被捕捉（catchable），有的默认行为无法修改呢?

在操作系统中，某些信号是设计用来处理紧急或关键情况的，如果这些信号可以被任意忽略或修改，可能会使系统变得不稳定或不安全。（病毒 修改自己被 `Kill` 时的默认行为）

**不可捕捉信号**：

- 某些信号是设计为不能被进程捕捉的，这意味着进程不能为这些信号设置自定义的处理程序。当这些信号发出时，将执行它们的默认操作，通常是终止或挂起进程。
- 例如，`KILL`信号（在大多数Unix系统上是信号编号-9）就是一个不可捕捉信号。进程无法忽略或处理这个信号，它总是导致进程立即终止。这种信号通常用于关闭“失控”的进程。

信号机制通过区分可捕捉和不可捕捉信号，为进程提供了处理不同类型的系统事件的灵活性。不可捕捉信号提供了一种确保系统稳定性和安全性的机制，而可捕捉信号则允许进程对特定事件做出更复杂的响应。

# 如何发送信号?

## 前置概念：进程组

每个进程都只属于一个进程组，进程组是由一个正整数进程组 ID 来标识的。getpgrp 函数返回当前进程的进程组 ID：

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

// 返回：调用进程的进程组 ID。
```

默认地，一个子进程和它的父进程同属于一个进程组。一个进程可以通过使用 setpgid 函数来改变自己或者其他进程的进程组：

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

// 返回：若成功则为0，若错误则为 -1。
```

setpgid 函数将进程 pid 的进程组改为 pgid。如果 pid 是 0，那么就使用当前进程的 PID。如果 pgid 是 0，那么就用 pid 指定的进程的 PID 作为进程组 ID。例如，如果进程 15213 是调用进程，那么

```c
setpgid(0, 0);
```

会创建一个新的进程组，其进程组 ID 是 15213，并且把进程 15213 加入到这个新的进程组中。

# 发送信号的方式

## Keyboard

- `Ctrl-C` 按下这个键会导致系统向运行中的进程发送一个 INT 信号（`SIGINT`）。默认情况下，这个信号会导致进程立即终止。
- `Ctrl-Z` 按下这个键会导致系统向运行中的进程发送一个 TSTP 信号（`SIGTSTP`）。默认情况下，这个信号会导致进程暂停执行。
- `Ctrl-\` 按下这个键会导致系统向运行中的进程发送一个 ABRT 信号（`SIGABRT`）。默认情况下，这个信号会导致进程立即终止 Z（冗余设计，提供了额外的灵活性）

## Terminal

```
$ kill -<signal> <PID>
```

```
$ fg - bring job to foreground
```

## Syscall

```c
#include <unistd.h>     /* 标准Unix函数，如getpid() */
#include <sys/types.h>  /* 各种类型定义，如pid_t */
#include <signal.h>     /* 信号名称宏和kill()原型 */

/* 首先，找到我的进程ID */
pid_t my_pid = getpid();

/* 现在我得到了我的PID，给自己发送STOP信号。 */
kill(my_pid, SIGSTOP);
```

思考：进程可以向其他的进程发送信号，比如停止信号，这是一个合理的设计吗?

允许一个进程向另一个进程发送信号，如停止信号或终止信号，是有其风险和争议的，但这也是一个设计决策，反映了在灵活性与安全性之间的权衡。这种设计的合理性和潜在的问题取决于多个因素：

**权限和安全性**：在Linux系统中，普通用户进程不能随意向其他用户的进程发送信号。这种能力受到严格的权限控制。通常，只有具有相同用户ID的进程或具有适当权限的进程（如root用户）才能向其他进程发送信号。这种权限检查提供了一定程度的安全性，防止恶意进程干扰其他用户的进程。

**灵活性和控制**：允许进程向其他进程发送信号提供了一定的灵活性，这对于某些应用场景是必需的。例如，守护进程可能需要向其管理的子进程发送信号以控制其行为。在一些系统管理任务中，能够从一个进程向另一个进程发送信号是一种有效的控制和通信手段。

**潜在的滥用风险**：虽然存在权限检查，但这种机制仍然可能被滥用，特别是在系统配置不当或安全漏洞存在的情况下。恶意进程或用户可能利用这一机制来干扰或破坏其他进程，尤其是在权限管理不当时。

**设计哲学**：Unix和Linux的设计哲学强调了工具和机制的通用性和灵活性，而这通常伴随着对用户或管理员的信任。信号机制的设计反映了这一哲学。

## 内核视角下的信号处理流程

**（1）信号的生成**：信号可以由多种事件触发，包括用户动作，系统调用，或硬件异常。当这些事件发生时，内核生成相应的信号。

**（2）信号的存储**：每个进程都有一个进程控制块（PCB），其中包含了关于该进程的信息，包括信号相关的信息,未处理的信号存储在进程控制块的信号队列中。这些信号被称为“待处理信号”（`pending signals`）。进程控制块还包含一个“信号掩码”（`signal mask`），指定了哪些信号被阻塞，即暂时不被该进程处理。

**（3）信号的检查和传递**

1. **从内核模式到用户模式的转换**：当内核准备将控制权从内核模式转换到用户模式时（比如从系统调用返回或完成上下文切换），它会检查进程的待处理信号集合。
2. **检查未阻塞的信号**：内核计算 `pending & ~blocked`，这是一个位运算，用于确定哪些待处理信号当前没有被阻塞。
3. **信号集合为空**：如果结果为空（即没有未阻塞的待处理信号），内核就会继续正常的流程，将控制权交还给进程的用户空间部分。
4. **处理信号**：如果有未被阻塞的待处理信号，内核则需要在将控制权交还给用户空间之前处理这些信号。

**（4）信号的处理**

1. **信号处理函数（信号处理器）**：如果为待处理信号之一注册了信号处理函数，内核将设置进程的上下文以运行该处理函数。

2. **默认行为**：如果没有为信号指定处理函数，内核将执行该信号的默认行为（如终止进程、忽略信号等）。
3. **返回到用户空间**：处理完信号后，内核将控制权交还给用户空间的进程，要么是返回到信号发生时的位置，要么是开始执行信号处理函数。

# 捕捉信号后如何处理？ `Signal Handlers`

```
When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal.
Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.
```

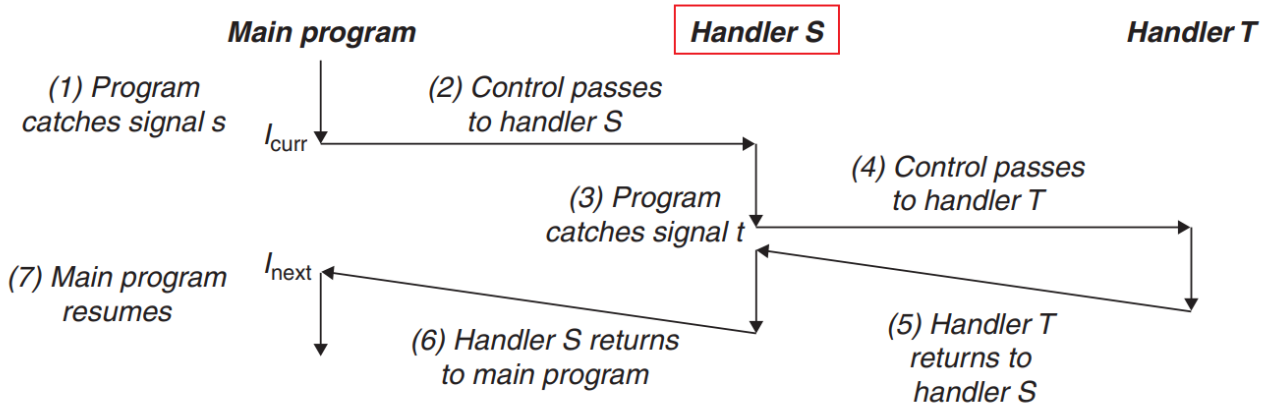


**Figure 8.31** Handlers can be interrupted by other handlers.

```
int sigaction(int S, const struct sigaction * Act, struct sigaction * OldAct)
```

```c
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

// 返回：若成功则为指向前次处理程序的指针，若出错则为 SIG_ERR（不设置 errno）。
```

signal 函数可以通过下列三种方法之一来改变和信号 signum 相关联的行为：

- 如果 handler 是 SIG_IGN，那么忽略类型为 signum 的信号。
- 如果 handler 是 SIG_DFL，那么类型为 signum 的信号行为恢复为默认行为。
- 否则，handler 就是用户定义的函数的地址，这个函数被称为**信号处理程序**，只要进程接收到一个类型为 signum 的信号，就会调用这个程序。通过把处理程序的地址传递到 signal 函数从而改变默认行为，这叫做**设置信号处理程序**（installing the handler）。调用信号处理程序被称为**捕获信号**。执行信号处理程序被称为**处理信号**。

# 如何阻塞与解除阻塞信号 `Masking Signals`

Linux 提供阻塞信号的隐式和显式的机制：

- **隐式阻塞机制**

  内核默认阻塞任何当前处理程序正在处理信号类型的待处理的信号。例如，上图 8-31 中，假设程序捕获了信号 s，当前正在运行处理程序 S。如果发送给该进程另一个信号 s，那么直到处理程序 S 返回，s 会变成待处理而没有被接收。

- **显式阻塞机制**

  应用程序可以使用 sigprocmask 函数和它的辅助函数，明确地阻塞和解除阻塞选定的信号。

```c
#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);
//返回；如果成功则为 0，若出错则为 -1。

int sigismember(const sigset_t *set, int signum);
// 返回：若 signum 是 set 的成员则为 1，如果不是则为 0，若出错则为 -1。
```

**sigprocmask** 函数改变当前阻塞的信号集合（8.5.1 节中描述的 blocked 位向量）。具体的行为依赖于 how 的值：

- `SIG_BLOCK`：把 set 中的信号添加到 blocked 中（`blocked=blocked | set`）。
- `SIG_UNBLOCK`：从 blocked 中删除 set 中的信号（`blocked=blocked &~set`）。
- `SIG_SETMASK`：`block=set`。

如果 oldset 非空，那么 blocked 位向量之前的值保存在 oldset 中。

```c
// 举例：如何用 sigprocmask 来临时阻塞接收 SIGINT 信号
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
.
.  // Code region that will not be interrupted by SIGINT
.
/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

# Shell Lab 分析

## 一些重要知识的回顾

## 回顾：回收子进程

一个终止了但是还未被回收的进程称为僵死进程。对于一个长时间运行的程序（比如Shell）来说，内核不会安排 `init` 进程去回收僵死进程，而它虽不运行却仍然消耗系统资源，因此实验要求我们回收所有的僵死进程。

`waitpid` 是一个非常重要的函数，一个进程可以调用 `waitpid` 函数来等待它的子进程终止或停止，从而回收子进程。

这个函数用来挂起调用进程的执行，直到 `pid` 对应的等待集合的一个子进程的改变才返回，包括三种状态的改变：

- 子进程终止

- 子进程收到信号停止
- 子进程收到信号重新执行

如果一个子进程在调用之前就已经终止了，那么函数就会立即返回，否则，就会阻塞，直到一个子进程改变状态。

等待集合以及监测那些状态都是用函数的参数确定的，函数定义如下：

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

各参数含义及使用

- **pid：判定等待集合成员**
    - `pid > 0` : 等待集合为 pid 对应的单独子进程
    - `pid = -1` : 等待集合为所有的子进程
    - `pid < -1` : 等待集合为一个进程组，ID 为 pid 的绝对值
    - `pid = 0` : 等待集合为一个进程组，ID 为调用进程的 pid
- **options：修改默认行为**
    - `WNOHANG` ：集合中任何子进程都未终止，立即返回 0
    - `WUNTRACED` ：阻塞，直到一个进程终止或停止，返回 PID
    - `WCONTINUED` ：阻塞，直到一个停止的进程收到 SIGCONT 信号重新开始执行
    - 也可以用或运算把 options 的选项组合起来。例如 WNOHANG | WUNTRACED 表示：立即返回，如果等待集合中的子进程都没有被停止或终止，则返回值为 0；如果有一个停止或终止，则返回值为该子进程的 PID
- **statusp：检查已回收子进程的退出状态**
    - waitpid 会在 status 中放上关于导致返回的子进程的状态信息

# 回顾：并发编程原则

1. 注意保存和恢复 errno。很多函数会在出错返回式设置 errno，在处理程序中调用这样的函数可能会干扰主程序中其他依赖于 errno 的部分，解决办法是在进入处理函数时用局部变量保存它，运行完成后再将其恢复。
2. 访问全局数据时，阻塞所有信号。
3. 不可以用信号来对其它进程中发生的事情计数。未处理的信号是不排队的，即每种类型的信号最多只能有一个待处理信号。

   **举例**：如果父进程将要接受三个相同的信号，当处理程序还在处理一个信号时，第二个信号就会加入待处理信号集合，如果此时第三个信号到达，那么它就会被简单地丢弃，从而出现问题。
4. 注意考虑条件同步错误

```
1  /* WARNING: This code is buggy! */
2  void handler(int sig)
3  {
4      int olderrno = errno;
5      sigset_t mask_all, prev_all;
6      pid_t pid;
7
8      Sigfillset(&mask_all);
9      while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap a zombie
   child */
10         Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
11         deletejob(pid); /* Delete the child from the job list */
12         Sigprocmask(SIG_SETMASK, &prev_all, NULL);
13     }
14     if (errno != ECHILD)
15         Sio_error("waitpid error");
16     errno = olderrno;
17 }
18
19 int main(int argc, char **argv)
```

```
20 {
21     int pid;
22     sigset_t mask_all, prev_all;
23 
24     Sigfillset(&mask_all);
25     Signal(SIGCHLD, handler);
26     initjobs(); /* Initialize the job list */
27 
28     while (1) {
29         if ((pid = Fork()) == 0) { /* Child process */
30             Execve("/bin/date", argv, NULL);
31         }
32         Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* Parent
   process */
33         addjob(pid);  /* Add the child to the job list */
34         Sigprocmask(SIG_SETMASK, &prev_all, NULL);
35     }
36     exit(0);
37 }
```

当父进程创建一个新的子进程后，它就把这个子进程添加到作业列表中。当父进程在 SIGCHLD 处理程序中回收一个终止的（僵死）子进程时，它就从作业列表中删除这个子进程。

信一看，这段代码是对的。不幸的是，可能发生下面这样的事件序列：

1. 父进程执行 fork 函数，内核调度新创建的子进程运行，而不是父进程。
2. 在父进程能够再次运行之前，子进程就终止，并且变成一个僵死进程，使得内核传递一个 SIGCHLD 信号给父进程。
3. 后来，当父进程再次变成可运行但又在它执行之前，内核注意到有未处理的 SIGCHLD 信号，并通过在父进程中运行处理程序接收这个信号。
4. 信号处理程序回收终止的子进程，并调用 deletejob，这个函数什么也不做，因为父进程还没有把该子进程添加到列表中。

5. 在处理程序运行完毕后，内核运行父进程，父进程从 fork 返回，通过调用 add-job 错误地把（不存在的）子进程添加到作业列表中。

因此，对于父进程的 main 程序和信号处理流的某些交错，可能会在 addjob 之前调用 deletejob。这导致作业列表中出现一个不正确的条目，对应于一个不再存在而且永远也不会被删除的作业。另一方面，也有一些交错，事件按照正确的顺序发生。例如，如果在 fork 调用返回时，内核刚好调度父进程而不是子进程运行，那么父进程就会正确地把子进程添加到作业列表中，然后子进程终止，信号处理函数把该作业从列表中删除。

这是一个称为**竞争**（race）的经典同步错误的示例。在这个情况中，main 函数中调用 addjob 和处理程序中调用 deletejob 之间存在竞争。如果 addjob 赢得进展，那么结果就是正确的。

如果它没有，那么结果就是错误的。这样的错误非常难以调试，因为几乎不可能测试所有的交错。你可能运行这段代码十亿次，也没有一次错误，但是下一次测试却导致引发竞争的交错。

```
1  void handler(int sig)
2  {
3      int olderrno = errno;
4      sigset_t mask_all, prev_all;
5      pid_t pid;
6
7      Sigfillset(&mask_all);
8      while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap a zombie
   child */
9          Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
10         deletejob(pid); /* Delete the child from the job list */
11         Sigprocmask(SIG_SETMASK, &prev_all, NULL);
12     }
13     if (errno ≠ ECHILD)
14         Sio_error("waitpid error");
15     errno = olderrno;
16 }
17
```

```c
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* Parent process */
        addjob(pid); /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
    }
    exit(0);
}
```

通过在调用 fork 之前，阻塞 SIGCHLD 信号，然后在调用 addjob 之后取消阻塞这些信号，我们保证了在子进程被添加到作业列表中之后回收该子进程。注意，子进程继承了它们父进程的被阻塞集合，所以我们必须在调用 execve 之前，小心地解除子进程中阻塞的 SIGCHLD 信号。

## eval

这个函数功能是解析命令行后判断为内置命令还是程序路径，分别执行。

如果是前台作业，则要等待其完成，如果是后台作业，则要输出其相应信息

> 注意要在执行前创建新线程组
>
> 主要是为了将子进程组与 tsh 进程组分开，防止发信号终止子进程组时也将 tsh 进程组终止了

```c
/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return.  Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
 */
void eval(char *cmdline)
{
    char buf[MAXLINE];
    char *argv[MAXARGS];
    int bg;
```

```
17      pid_t pid;
18      sigset_t mask, prev;
19
20      strcpy(buf, cmdline);
21      if (!(bg = parseline(buf, argv))) // Check if it's a
    background job and if command is not empty
22          return;
23      if (builtin_cmd(argv)) // Directly execute built-in command
24          return;
25
26      // Simplify signal set initialization and block SIGCHLD
27      sigemptyset(&mask);
28      sigaddset(&mask, SIGCHLD);
29      if (sigprocmask(SIG_BLOCK, &mask, &prev) < 0)
30      {
31          perror("sigprocmask error");
32          return;
33      }
34
35      if ((pid = fork()) < 0)
36      {
37          perror("fork error");
38          return;
39      }
40      if (pid == 0)
41      { // Child process
42          setpgid(0, 0);
43          if (sigprocmask(SIG_SETMASK, &prev, NULL) < 0)
44              perror("sigprocmask error");
45          if (execve(argv[0], argv, environ) < 0)
46          {
47              printf("%s: Command not found\n", argv[0]);
48              exit(0);
49          }
50      }
51
```

```
52      // Parent process
53      sigset_t mask_all;
54      sigfillset(&mask_all);
55      if (sigprocmask(SIG_BLOCK, &mask_all, NULL) < 0)
56          perror("sigprocmask error");
57      addjob(jobs, pid, (bg ? BG : FG), cmdline);
58      if (sigprocmask(SIG_SETMASK, &prev, NULL) < 0)
59          perror("sigprocmask error");
60      if (bg)
61          printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
62      else
63          waitfg(pid);
64      return;
65  }
```

## builtin_cmd

判断是否为内置命令

```
1  /*
2   * builtin_cmd - If the user has typed a built-in command then
    execute
3   *    it immediately.
4   */
5  int builtin_cmd(char **argv)
6  {
7      if (!strcmp(argv[0], "quit")) /* quit command */
8          exit(0);
9      if (!strcmp(argv[0], "&")) /* Ignore singleton & */
10         return 1;
11     if (!strcmp(argv[0], "jobs"))
12     {
13         listjobs(jobs);
```

```
14          return 1;
15      }
16      // fg or bg
17      if (!strcmp(argv[0], "fg") || !strcmp(argv[0], "bg"))
18      {
19          do_bgfg(argv);
20          return 1;
21      }
22      return 0; /* Not a builtin command */
23  }
24
```

## do_bgfg

这个函数要实现内置命令 `bg` 和 `fg`，这两个命令的功能如下：

- `bg <job>`：通过向 `<job>` 对应的作业发送 `SIGCONT` 信号来使它重启并放在后台运行
- `fg <job>`：通过向 `<job>` 对应的作业发送 `SIGCONT` 信号来使它重启并放在前台运行
- 输入时后面的参数有 `%` 则代表 `jid`，没有则代表 `pid`

这个函数的难点主要在于命令行参数的判断，要考虑健壮性。

注意原来的后台进程可能会变成前台进程，所以应修改 `job` 结构体（全局变量）的相应信息

```
1  /*
2   * do_bgfg - Execute the builtin bg and fg commands
3   */
4  void do_bgfg(char **argv)
5  {
```

```c
    struct job_t *job = NULL; // Pointer to the job to be handled
    int state;               // Command state (BG or FG)
    int id;                  // Stores Job ID (jid) or Process ID (pid)

    // Determine the command (bg or fg)
    if (!strcmp(argv[0], "bg"))
    {
        state = BG;
    }
    else
    {
        state = FG;
    }

    // Check for missing arguments
    if (argv[1] == NULL)
    {
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }

    // Handle job ID input
    if (argv[1][0] == '%')
    {
        if (sscanf(&argv[1][1], "%d", &id) > 0)
        {
            job = getjobjid(jobs, id); // Get the job using job ID
            if (job == NULL)
            {
                printf("%%%d: No such job\n", id);
                return;
            }
```

```
        }
    }
    // Handle invalid input
    else if (!isdigit(argv[1][0]))
    {
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    // Handle process ID input
    else
    {
        id = atoi(argv[1]);
        job = getjobpid(jobs, id); // Get the job using process ID
        if (job == NULL)
        {
            printf("(%d): No such process\n", id);
            return;
        }
    }

    // Send SIGCONT signal to the job's process group
    if (kill(-(job->pid), SIGCONT) < 0)
    {
        perror("kill (SIGCONT)");
        return;
    }

    // Update the job's state
    job->state = state;

    // Print job info if moved to background, or wait if moved to foreground
    if (state == BG)
    {
```

```
71          printf("[%d] (%d) %s", job->jid, job->pid, job-
    >cmdline);
72      }
73      else
74      {
75          waitfg(job->pid);
76      }
77  }
78
```

## `waitfg`

这个函数从要求实现阻塞父进程，直到当前的前台进程不再是前台进程了。

考虑如下代码：

```
1  while(fgpid(jobs) != 0)
2      pause();
```

这里会有一个竞争条件bug，考虑如下事件序列：

- 父进程调用 `fgpid` 函数，此时有一个子进程仍然在前台运行，所以判断条件为真，进入循环
- 假定父进程在进入循环后，而执行 `pause` 前，子进程终止
- 父进程接收到 SIGCHLD 信号，并处理结束后才调用 `pause`

由于 `pause` 仅在捕捉到信号后返回，而之后不会再有任何信号抵达，那么父进程就会永远休眠！

解决办法是用 `sleep` 函数：因为它不依赖信号来返回，通过每次循环来监测子进程状态

也可以用 `sigsuspend` 函数，这个函数相当于如下代码：

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

在调用 `sigsuspend` 之前阻塞 SIGCHLD 信号，调用时又通过 `sigprocmask` 函数，在执行 `pause` 函数之前解除对信号的阻塞，从而正常休眠。有同学可能会问了：这里并没有消除竞争啊？如果在第 1 行和第 2 行之间子进程终止不还是会发生永久休眠吗？

这就是 `sigsuspend` 与上述代码的不同之处了，它相当于上述代码的原子版本，即第 1 行和第 2 行总是一起发生的，不会被中断

```
1  void waitfg(pid_t pid)
2  {
3      sigset_t mask_all, prev_all;
4      sigfillset(&mask_all);
5      sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
6
7      while (fgpid(jobs) > 0)
8          sigsuspend(&prev_all);
9
10     sigprocmask(SIG_SETMASK, &prev_all, NULL);
11     return;
12 }
```

## handlers

- `SIGCHLD` 信号处理函数，能够回收所有僵死进程
- `SIGINT` 信号处理函数，将信号传送给前台进程

- `SIGSTOP` 信号处理函数，将信号传送给前台进程

```
1
2  /*
3   * sigchld_handler - The kernel sends a SIGCHLD to the shell
   whenever
4   *     a child job terminates (becomes a zombie), or stops
   because it
5   *     received a SIGSTOP or SIGTSTP signal. The handler reaps
   all
6   *     available zombie children, but doesn't wait for any
   other
7   *     currently running children to terminate.
8   */
9  void sigchld_handler(int sig)
10 {
11     int olderrno = errno;
12     sigset_t mask_all, prev_all;
13     sigfillset(&mask_all);
14
15     pid_t pid;
16     int status;
17
18     while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) >
   0)
19     {
20         sigprocmask(SIG_BLOCK, &mask_all, &prev_all); // Block
   signals when updating the job list
21
22         if (WIFEXITED(status))
23         {
24             // If the child exited normally
25             deletejob(jobs, pid);
26         }
27         else if (WIFSIGNALED(status))
```

```
28          {
29              // If the child was terminated by a signal
30              printf("Job [%d] (%d) terminated by signal %d\n",
    pid2jid(pid), pid, WTERMSIG(status));
31              deletejob(jobs, pid);
32          }
33          else if (WIFSTOPPED(status))
34          {
35              // If the child was stopped
36              printf("Job [%d] (%d) stopped by signal %d\n",
    pid2jid(pid), pid, WSTOPSIG(status));
37              struct job_t *jp = getjobpid(jobs, pid);
38              if (jp != NULL)
39              {
40                  jp->state = ST;
41              }
42          }
43
44          sigprocmask(SIG_SETMASK, &prev_all, NULL); // Restore
    previous signal mask
45      }
46
47      errno = olderrno;
48  }
49
50
51  /*
52   * sigint_handler - The kernel sends a SIGINT to the shell
    whenver the
53   *    user types ctrl-c at the keyboard.  Catch it and send it
    along
54   *    to the foreground job.
55   */
56  void sigint_handler(int sig)
57  {
58      int olderrno = errno;
```

```
59      int pid;
60      sigset_t mask_all, prev_all;
61      sigfillset(&mask_all);
62      sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
63
64      if ((pid = fgpid(jobs)) != 0)
65      {
66          sigprocmask(SIG_SETMASK, &prev_all, NULL);
67          if (kill(-pid, SIGINT) < 0)
68          {
69              unix_error("kill (SIGINT)");
70          }
71      }
72
73      errno = olderrno;
74      return;
75  }
76
77  /*
78   * sigtstp_handler - The kernel sends a SIGTSTP to the shell
    whenever
79   *     the user types ctrl-z at the keyboard. Catch it and
    suspend the
80   *     foreground job by sending it a SIGTSTP.
81   */
82  void sigtstp_handler(int sig)
83  {
84      int olderrno = errno;
85      int pid;
86      sigset_t mask_all, prev_all;
87      sigfillset(&mask_all);
88      sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
89
90      if ((pid = fgpid(jobs)) > 0)
91      {
92          sigprocmask(SIG_SETMASK, &prev_all, NULL);
```

```
93          if (kill(-pid, SIGSTOP) < 0)
94          {
95              unix_error("kill (SIGSTOP)");
96          }
97      }
98
99      errno = olderrno;
100     return;
101 }
```

## Take Away Message

信号机制有哪些缺陷?

1. 不可靠
2. 传输信号有限
3. 非原子操作
4. 复杂的同步需求
5. 中断正常程序处理流程
6. …

提前祝大家新春快乐~🐉

# Reference


1. https://www.cs.kent.edu/~ruttan/sysprog/lectures/signals.html
2. https://faculty.cs.niu.edu/~hutchins/csci480/signals.htm
3. https://en.wikipedia.org/wiki/Signal_(IPC)

4. https://hansimov.gitbook.io/csapp/part2/ch08-exceptional-control-flow/8.5-signals#id-8.5.7-xian-shi-di-deng-dai-xin-hao
5. https://zhuanlan.zhihu.com/p/492645370
6. man 7 signal
7. GPT-4
8. https://github.com/chenzongyao200127/CSAPP_Lab