

Shell Lab

1 系统调用错误处理

当Unix系统级函数遇到错误时，他们通常会返回-1，并且设置全局变量 `errno` 来表示什么错了。

```
pid_t Fork(void) {
    pid_t pid;
    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

2 进程控制

```
pid_t getpid(void); // 返回当前进程的pid
pid_t getppid(void); // 返回父进程的pid
```

如何回收子进程?

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

- `pid` 参数指定要等待的子进程的进程ID。
 - 如果 `pid` 大于 0，表示等待进程ID为 `pid` 的子进程。
 - 如果 `pid` 等于 -1，表示等待任意子进程，与 `wait` 函数的行为相似。
 - 如果 `pid` 等于 0，表示等待与调用 `waitpid` 进程在同一个进程组的任意子进程。
 - 如果 `pid` 小于 -1，表示等待进程组ID等于 `pid` 的任意子进程。
- `status` 参数是一个指向整数的指针，用于存储子进程的退出状态信息。
- `options` 参数是一个用于控制 `waitpid` 行为的整数。
 - `options` 可以是 0，表示默认行为。挂起调用进程的执行，直到它的等待集合中的一个子进程终止。
 - 也可以使用 `WNOHANG` 选项，表示如果没有子进程退出或终止立即返回，而不阻塞等待。

`waitpid` 返回已终止子进程的进程ID，如果没有子进程处于终止状态并且使用了 `WNOHANG` 选项，则返回0。如果出错，返回-1。

如果 `status` 参数非空的话，就会放上关于导致返回子进程的状态信息。

3 加载并运行程序

`execve` 函数是一个用于在当前进程中执行新程序的系统调用。它会取代当前进程的内存映像，将其替换为一个新程序的内存映像。这个函数通常与 `fork` 系统调用一起使用，其中 `fork` 用于创建一个新的进程，而 `execve` 用于在新进程中执行新程序。

函数原型如下：

```
#include <unistd.h>

int execve(const char *filename, char *const argv[], char *const envp[]);
```

- `filename` 参数是要执行的可执行文件的路径。
- `argv` 参数是一个字符串数组，表示传递给新程序的命令行参数。数组的第一个元素通常是可执行文件的名称。
- `envp` 参数是一个字符串数组，表示传递给新程序的环境变量。

如果 `execve` 函数执行成功，它不会返回，因为当前进程的内存映像已经被替换为新程序的映像。如果发生错误，`execve` 函数返回 -1，并设置 `errno` 来指示错误类型。

以下是一个简单的例子，演示了 `execve` 的基本用法：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    char *const args[] = {"/bin/ls", "-l", "/tmp", NULL};
    char *const env[] = {NULL};

    // 执行 /bin/ls -l /tmp
    if (execve("/bin/ls", args, env) == -1) {
        perror("execve");
        exit(EXIT_FAILURE);
    }

    // 如果 execve 执行成功，下面的代码将不会被执行
    printf("This line will not be reached.\n");

    return 0;
}
```

在这个例子中，`execve` 函数被用来执行 `/bin/ls` 命令，显示 `/tmp` 目录的内容。如果 `execve` 执行成功，那么程序的控制流就会切换到新程序，下面的 `printf` 语句将不会被执行。

注意：`execve` 函数返回 -1 仅表示执行失败，它并不提供详细的错误信息。可以使用 `perror` 函数来打印具体的错误信息，或者通过检查 `errno` 全局变量获取错误码。

4 信号

信号是一种更高级的软件形式的异常。

1. 信号的基本概念：

- 信号是异步事件的一种形式，它可以由内核、其他进程或硬件生成。
- 每个信号都有一个唯一的标识符，通常用整数表示。例如，`SIGINT` 表示终端中断信号。

2. 信号的分类：

- **标准信号**：操作系统定义的一组常见信号，如 `SIGINT`、`SIGTERM`、`SIGSEGV` 等。
- **实时信号**：提供更多的信号，并且可以按照优先级排序，如 `SIGRTMIN`、`SIGRTMAX`。

3. 信号处理：

- 进程可以通过注册信号处理函数来捕获和处理信号。处理函数是在收到信号时由操作系统调用的用户定义的函数。
- 使用 `signal` 或 `sigaction` 函数来注册信号处理函数。

4. 常见信号：

- `SIGKILL`：强制终止进程。
- `SIGTERM`：请求进程正常终止。
- `SIGINT`：中断进程（通常由终端 Ctrl+C 生成）。
- `SIGSEGV`：段错误，表示非法内存访问。
- `SIGCHLD`：子进程状态改变。

5. 信号的发送：

- 使用 `kill` 命令或 `kill` 函数可以向进程发送信号。
- 进程可以使用 `raise` 函数向自己发送信号。

6. 信号屏蔽和阻塞：

- 进程可以使用 `sigprocmask` 函数设置信号屏蔽字，以阻塞或解除阻塞特定的信号。
- 阻塞信号可以防止在关键代码段中被中断。

7. 信号的默认处理：

- 每个信号都有一个默认的处理方式，如终止进程、忽略信号、或者停止进程等。
- 可以通过 `signal` 函数或 `sigaction` 函数来指定信号的处理方式。

8. 信号和多线程：

- 在多线程程序中，信号可能会影响整个进程，而不是特定线程。
- `pthread_kill` 函数可以发送信号给特定线程。

9. 信号和异步I/O：

- 信号可以用于处理异步事件，如异步I/O完成时发出的信号。

理解信号是系统编程中重要的一部分，因为它提供了一种处理异步事件的机制，可以用于实现进程间通信、优雅地终止进程等。在使用信号时，要注意避免竞态条件和确保信号处理函数是可重入的。

一个发出而没有被接受的信号叫做待处理信号（pending signal）。任何时刻，一种类型至多会有一种待处理信号。如果一个进程有一个类型为 k 的待处理信号，那么接下来发送到这个进程的类型为 k 的信号都不会排队等待，而只是简单地被丢弃。一个进程可以阻塞接受某种信号。可以被发送，但是不会被接受。

5 发送信号

1. 进程组

- 每个进程都只属于一个进程组。
- 一个子进程和它的父进程属于同一个进程组

```
#include <unistd.h>
pid_t getpgrp(void);
int setpgid(pid_t pid, pid_t pgid); // pid 是0, 使用当前进程PID; pgid = 0使用pid指定的进程的PID作为进程组的ID
```

2. 用 `/bin/kill` 发送信号

```
/bin/kill -9 15213
/bin/kill -9 -15213
```

一个是发送 `SIGKILL` 给15213进程，还有一个是发送 `SIGKILL` 给15213进程组。

3. 从键盘发送信号

Unix Shell用作业来表示 对一条命令行求值而创建的进程。

4. 用 `kill` 函数发送信号

`kill` 函数是用于向指定进程发送信号的系统调用。这个函数的原型如下：

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

- `pid` 参数指定目标进程的进程ID。
 - 如果 `pid` 大于 0，则信号将被发送给进程ID为 `pid` 的进程。
 - 如果 `pid` 等于 0，则信号将被发送给与调用进程属于同一进程组的所有进程。
 - 如果 `pid` 等于 -1，则信号将被发送给除了调用进程之外的所有具有权限的进程。
 - 如果 `pid` 小于 -1，则信号将被发送给进程组ID等于 `pid` 绝对值的所有进程。
- `sig` 参数是信号的编号，可以是标准信号（如 `SIGTERM`）或实时信号（如 `SIGRTMIN`）。

函数返回值为 0 表示成功，-1 表示失败，并设置 `errno` 来指示错误类型。

以下是一个简单的例子，演示了如何使用 `kill` 函数向另一个进程发送信号：

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

int main() {
    pid_t pid = 1234; // 替换为目标进程的实际进程ID

    // 向目标进程发送 SIGTERM 信号
    if (kill(pid, SIGTERM) == -1) {
```

```

        perror("kill");
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

在这个例子中，`kill` 函数向进程ID为 1234 的进程发送 `SIGTERM` 信号。这种方法常用于请求进程正常终止。

5. 用 `alarm` 函数发送信号

`alarm` 函数是一个用于在指定时间后发送 `SIGALRM` 信号的系统调用。`SIGALRM` 是一个用于定时器的信号，通常用于实现定时操作。以下是 `alarm` 函数的原型：

```

#include <unistd.h>

unsigned int alarm(unsigned int seconds);

```

- `seconds` 参数指定了定时器的定时时间，即多少秒后将触发 `SIGALRM` 信号。
- 如果 `seconds` 为 0，表示取消之前设置的任何定时器。
- 如果之前已经设置了定时器，`alarm` 返回剩余的定时时间（如果有的话），否则返回 0。

`alarm` 函数的工作方式是，调用它时，它会取消之前设置的定时器（如果有的话），然后根据传入的 `seconds` 参数设置新的定时器。

以下是一个简单的例子，演示了如何使用 `alarm` 函数设置一个定时器：

```

#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void alarm_handler(int signo) {
    printf("Alarm triggered!\n");
}

int main() {
    // 注册信号处理函数
    signal(SIGALRM, alarm_handler);

    // 设置一个定时器，5秒后触发 SIGALRM 信号
    unsigned int remaining_time = alarm(5);

    printf("Alarm set. Remaining time: %u seconds\n", remaining_time);

    // 主程序继续执行其他工作
    while (1) {
        // ...
    }

    return 0;
}

```

在这个例子中，调用 `alarm(5)` 设置了一个 5 秒的定时器，5 秒后将触发 `SIGALRM` 信号。当信号被触发时，会调用注册的 `alarm_handler` 函数。在实际应用中，定时器可以用于执行定时任务、超时处理等。

6 接收信号

当内核把进程 p 从内核模式切换到用户模式时，会检查进程 p 未被阻塞的待处理信号的集合(`pending&~blocked`)。如果集合非空的话，内核选择集合中的某个信号 k ，并且强制 p 接收信号 k 。

`signal` 函数是一个用于设置信号处理函数的库函数。它允许程序员指定在接收到特定信号时应该执行的函数。

`signal` 函数的原型如下：

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

```
void (*signal(int signum, void (*handler)(int)))(int);
```

- `signum` 参数是要设置处理函数的信号编号。
- `handler` 参数是一个函数指针，指向一个用户定义的信号处理函数。

返回值是之前与该信号相关联的信号处理函数的函数指针。如果之前没有为该信号设置过处理函数，则返回 `SIG_ERR`。

7 阻塞和解除阻塞信号

隐式阻塞机制：假设当前程序捕获了信号 s ，同时正在运行处理程序 S ，那么信号 s 直到 S 结束前都不会被接收。

显示阻塞机制：应用程序可以使用 `sigprocmask`。

`sigprocmask` 是一个 POSIX 标准定义的系统调用，用于设置和检查进程的信号屏蔽字 (signal mask)。信号屏蔽字是一个掩码，用于阻塞或解除阻塞特定的信号。通过设置信号屏蔽字，进程可以决定在某些关键代码段是否屏蔽（阻塞）某些信号，以防止它们在关键时刻中断执行。

函数原型如下：

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

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- `how` 参数指定了要执行的操作，有三种可能的值：
 - `SIG_BLOCK`：将 `set` 中的信号添加到当前信号屏蔽字中。
 - `SIG_UNBLOCK`：从当前信号屏蔽字中移除 `set` 中的信号。
 - `SIG_SETMASK`：用 `set` 中的信号集替换当前信号屏蔽字。
- `set` 参数是一个指向要设置的信号集的指针。
- `oldset` 参数是一个指向 `sigset_t` 结构的指针，用于存储调用该函数前的当前信号屏蔽字，以便之后可以还原。

这个函数允许进程在关键部分（比如临界区）阻塞或解除阻塞一些信号，以确保关键部分的执行不会被这些信号中断。在多线程环境中，也可以使用 `pthread_sigmask` 函数来设置线程特定的信号屏蔽字。

以下是一个简单的例子，演示了如何使用 `sigprocmask` 阻塞和解除阻塞信号：

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

```

#include <signal.h>

int main() {
    sigset_t new_mask, old_mask;

    // 设置要阻塞的信号集
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGINT);

    // 阻塞 SIGINT 信号
    if (sigprocmask(SIG_BLOCK, &new_mask, &old_mask) < 0) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    // 这里可以放置临界区代码, 不会被 SIGINT 中断

    // 解除阻塞 SIGINT 信号
    if (sigprocmask(SIG_UNBLOCK, &new_mask, NULL) < 0) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    return 0;
}

```

在这个例子中, 首先使用 `sigprocmask` 阻塞了 `SIGINT` 信号, 然后在临界区执行一些代码, 最后解除了对 `SIGINT` 信号的阻塞。

8 安全的信号处理程序

只调用异步信号安全的函数。产生输出的唯一安全的方法是 `write`。使用 `sprintf` 和 `printf` 是不安全的。

保存和恢复 `errno`。

阻塞所有的信号, 保护对共享全局数据结构的访问。

同时要使用循环尽可能回收僵死进程。

同步流以避免讨厌的并发性问题。在调用 `fork` 之前, 阻塞 `SIGCHLD` 信号, 在调用 `addjob` 之后取消阻塞这些信号。

`sigsuspend` 函数是一个用于挂起进程直到收到特定信号的系统调用。它与 `pause` 函数类似, 但可以在等待信号时临时修改进程的信号屏蔽字。

函数原型如下:

```

#include <signal.h>

int sigsuspend(const sigset_t *mask);

```

- `mask` 参数是一个指向 `sigset_t` 结构的指针, 指定了在等待信号时要使用的新的信号屏蔽字。一旦收到信号, `sigsuspend` 会将信号屏蔽字恢复为调用前的状态。

函数返回值为 -1, 表示发生错误。正常情况下, `sigsuspend` 不会返回, 而是在捕获到信号后继续执行。

以下是一个简单的例子，演示了如何使用 `sigsuspend` 函数：

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

void signal_handler(int signo) {
    printf("Caught signal %d\n", signo);
}

int main() {
    // 设置信号处理函数
    signal(SIGUSR1, signal_handler);

    // 设置新的信号屏蔽字
    sigset_t new_mask, old_mask;
    sigemptyset(&new_mask);
    sigaddset(&new_mask, SIGUSR1);

    // 保存当前信号屏蔽字并设置新的信号屏蔽字
    if (sigprocmask(SIG_BLOCK, &new_mask, &old_mask) == -1) {
        perror("sigprocmask");
        exit(EXIT_FAILURE);
    }

    printf("Waiting for SIGUSR1 signal...\n");

    // 使用sigsuspend挂起进程，直到收到SIGUSR1信号
    sigsuspend(&old_mask);

    printf("Resumed after receiving SIGUSR1 signal\n");

    return 0;
}
```

在这个例子中，程序阻塞 `SIGUSR1` 信号，然后使用 `sigsuspend` 函数挂起进程，直到收到 `SIGUSR1` 信号。一旦信号被捕获，程序继续执行。需要注意的是，`sigsuspend` 的使用需要谨慎，因为在等待期间可能会引入竞态条件。

9 `dup2` 函数

`dup2` 函数是一个系统调用，通常用于复制文件描述符（File Descriptor）。它允许将一个文件描述符复制到另一个文件描述符，从而实现文件描述符的重定向。

函数的原型如下：

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

其中，`oldfd` 是要被复制的文件描述符，`newfd` 是新的文件描述符。`dup2` 函数的作用是将 `oldfd` 复制到 `newfd` 上，并关闭 `newfd` 所指向的文件描述符。如果 `newfd` 已经打开，则它会首先被关闭。

这个函数在进程重定向标准输入、输出和错误时非常有用。例如，可以使用 `dup2` 将标准输出重定向到一个文件，或者将一个文件描述符复制到标准输入等。

下面是一个简单的例子，演示如何将标准输出重定向到一个文件：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }
    // 将标准输出重定向到文件描述符 fd
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("dup2");
        exit(EXIT_FAILURE);
    }
    // 现在标准输出将写入到 output.txt 文件
    printf("This will be written to output.txt\n");
    close(fd);
    return 0;
}
```

在这个例子中，程序创建了一个文件描述符 `fd`，然后使用 `dup2` 将标准输出重定向到该文件描述符。最后，通过 `printf` 输出的内容将被写入到 "output.txt" 文件。

在C语言中，你可以使用 `dup2` 函数来将标准输出（文件描述符1）重定向到一个文件。然后，你可以使用 `printf` 函数或 `write` 函数将内容写入标准输出，这将实际上写入到你指定的文件。以下是一个简单的示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main() {
    // 打开（或创建）输出文件
    int output_file = open("output.txt", O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);

    if (output_file == -1) {
        perror("Error opening output file");
        exit(EXIT_FAILURE);
    }

    // 保存原始标准输出文件描述符
    int original_stdout = dup(1);

    // 将标准输出重定向到输出文件
    if (dup2(output_file, 1) == -1) {
        perror("Error redirecting stdout");
        close(output_file);
        exit(EXIT_FAILURE);
    }

    // 现在，标准输出将写入到文件而不是终端

    // 使用printf写入内容到标准输出（实际上写入到文件）
}
```

```

printf("Hello, World!\n");

// 如果使用write函数, 也可以这样写入
// write(1, "Hello, World!\n", 13);

// 恢复原始标准输出
dup2(original_stdout, 1);

// 关闭文件描述符
close(output_file);

return 0;
}

```

在这个例子中, `dup2` 函数用于将标准输出重定向到指定的文件描述符 (在这里是 `output_file`)。写入到标准输出的内容实际上被写入到了文件 "output.txt" 中。最后, 使用 `dup2` 恢复原始的标准输出。

请注意, 这只是一个示例, 实际中你可能需要更多的错误检查和处理。

10 SIGHUP

SIGHUP 是一种 POSIX 标准定义的信号, 通常用于通知进程的终端或控制终端的断开。当用户从终端退出时, 终端关闭或控制终端发生变更时, 通常会向与终端关联的进程发送 **SIGHUP** 信号。这在以守护进程形式运行的程序中很常见。

有一些常见的场景会导致 **SIGHUP** 信号的发送:

- 终端关闭:** 如果用户从运行程序的终端退出, 终端关闭, 或者终端发生其他异常, 操作系统通常会向相关进程发送 **SIGHUP** 信号。
- 控制终端变更:** 如果一个进程与控制终端关联, 而该控制终端的会话发生变更, 可能也会导致 **SIGHUP** 信号的发送。例如, 当一个终端会话 (session) 结束时, 与该终端关联的所有进程都会收到 **SIGHUP**。

程序可以通过捕获 **SIGHUP** 信号并注册相应的信号处理函数来处理这个信号。一般情况下, 处理 **SIGHUP** 的方式可能包括清理资源、保存状态等, 因为接收到 **SIGHUP** 通常表示进程的运行环境发生了变化。

以下是一个简单的示例, 演示了如何在程序中捕获 **SIGHUP** 信号:

```

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void sighup_handler(int signo) {
    printf("Received SIGHUP signal\n");
    // 进行相应的处理, 例如清理资源或保存状态
}

int main() {
    // 注册 SIGHUP 信号的处理函数
    signal(SIGHUP, sighup_handler);

    printf("My process ID is: %d\n", getpid());
    printf("Run this process in the background and close the terminal to trigger SIGHUP.\n");
}

```

```

// 主程序继续执行其他工作
while (1) {
    // ...
}

return 0;
}

```

在这个例子中，程序通过 `signal` 函数注册了一个处理 `SIGHUP` 信号的处理函数 `sigup_handler`。当程序收到 `SIGHUP` 信号时，将调用这个处理函数执行相应的操作。

11 笔记

Linux shells also provide various built-in commands that support job control. For example:

- `jobs` : List the running and stopped background jobs.
- `bg job` : Change a stopped background job into a running background job.
- `fg job` : Change a stopped or running background job into a running foreground job.
- `kill job` : Kill a job in the job list.
- `nohup [command]`: Make the trailing command block any SIGHUP signals.

tsh should support the following built-in commands:

- The `quit` command terminates the shell.
- The `jobs` command lists all background jobs.
- The `bg job` command restarts job by sending it a SIGCONT signal, and then runs it in the background. The
- `job` argument can be either a PID or a JID.
- The `fg job` command restarts job by sending it a SIGCONT signal, and then runs it in the
- foreground. The `job` argument can be either a PID or a JID.
- The `kill job` command kills a job in the job list, or a process group by sending each relevant process a SIGTERM signal. The `job` argument can be either a PID or a JID. Note that if you get a negative job argument, such as `kill %-1` or `kill -15213`, your shell should kill the process group of the job with a JID of `%-JID`, or of the job with a PID of `-PID`. If the process group does not exist, your shell should print `"%JID: No such process group"` or `"(PID): No such process group"`, where JID and PID should be replaced by the command line argument. On the other hand, if the `job` argument is positive, your shell should kill the corresponding job. If the job does not exist, your shell should print `"%JID: No such job"` or `"(PID): No such process"`. Play with the reference shell to check the details and gain intuition. Note: The built-in command `kill` slightly differs from that of the Linux shell in semantics, since our shell always kills a job rather than a single process.
- The `nohup [command]` command makes the trailing command ignore any SIGHUP signals. For simplicity, your shell does not need to support built-in commands following this principle. Instead, you can assume `[command]` to be the path of an executable file followed by its arguments.

作业控制实际上就是维护一个jobs数组，新建一个任务时将其加入到数组之中，任务执行完毕由父进程的中断处理程序将该任务删除。另外还需要在适当的时候将任务的状态进行调整，中断处理程序。

具体到本Lab，需要做的就是eval函数中添加任务，然后在sigchld_handler处理程序中回收子进程并删除相应任务，还有sigint_handler和sigstop_handler中改变任务的状态。

值得注意的是，为了避免race，需要在fork之前阻塞SIGCHLD信号，然后完成fork，在父进程中添加该任务之后再解除SIGCHLD信号的阻塞，以免发生删除任务发生在添加任务之前的情况。另外，由于子进程会继承父进程的阻塞，所以在execve之前需要取消对SIGCHLD信号的阻塞。

本Lab对于jobs数组的各种操作的实现都已经提供，只需要调用相应api即可，无需自己实现。

[CSAPP Shell Lab 详细解答]

1 trace00 EOF

trace01要求在读取EOF信号时退出Shell，在初始代码中该功能已经实现。

```
if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
    app_error("fgets error");
if (feof(stdin)) { /* End of file (ctrl-d) */
    printf("\n");
    fflush(stdout);
    fflush(stderr);
    exit(0);
}
```

2 trace01 quit

这个是要写在 `builtin_cmd` 函数里面。

```
int builtin_command(char** argv)
{
    if (!strcmp(argv[0], "quit")) /* quit command */
        exit(0);
    .....
    return 0;                  /* Not a builtin command */
}
```

3 trace02&03&04 运行前台程序而且不是内建命令

4 trace05&06 不是内建命令+后台程序+前台程序

此外还有非常重要的一点就是，我们的shell程序本身是所有子进程的父进程，那么就会分配在同一个组里，终止子进程所在组会导致shell程序本身也被终止，这里的解决办法是给予子进程设置一个单独的组，只需要添加在fork和exec之间。

```
// 这个是写在eval函数里面的，在最外面还有一个全局变量
// int pipe_fd[2];          /* 管道的文件描述符 */
```

```

// 在调用Fork之前, 先屏蔽SIGCHLD信号
sigset_t mask_all, mask_one, prev_one;
Sigfillset(&mask_all);
Sigemptyset(&mask_one);
Sigaddset(&mask_one, SIGCHLD);

if (pipe(pipe_fd) < 0) { // 创建管道
    unix_error("pipe error");
    exit(EXIT_FAILURE);
}

if (!builtin_cmd(tok.argv, &tok)) {
    // 如果不是内建命令, 那么就fork一个子进程
    Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); // 在fork之前, 先屏蔽SIGCHLD信号
    if ((pid = Fork()) == 0) {
        // 当前是在子进程里了
        Sigprocmask(SIG_SETMASK, &prev_one, NULL); // 解除屏蔽

        // 关于重定向的部分应该写在子进程里面
        // 如果有重定向的话, 那么就需要先打开文件
        int fd_in = -1, fd_out = -1;
        if (tok.infile != NULL) {
            fd_in = open(tok.infile, O_RDONLY);
            if (fd_in < 0) {
                printf("%s: No such file or directory\n", tok.infile);
                fflush(stdout);
                exit(EXIT_FAILURE);
            }
        }
        if (tok.outfile != NULL) {
            fd_out = open(tok.outfile, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR |
S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
            if (fd_out < 0) {
                printf("%s: No such file or directory\n", tok.outfile);
                fflush(stdout);
                exit(EXIT_FAILURE);
            }
        }

        // 从重定向的文件中读取输入
        if (fd_in != -1) {
            Dup2(fd_in, STDIN_FILENO);
            close(fd_in);
        }
        // 将输出重定向到文件中
        if (fd_out != -1) {
            Dup2(fd_out, STDOUT_FILENO);
            close(fd_out);
        }

        // 设置子进程的进程组
        // After the fork, but before the execve, the child process should call
        // setpgid(0, 0), which puts the child in a new process group whose group ID is
        identical to the
        // child's PID. This ensures that there will be only one process, your shell, in
        the foreground process
    }
}

```

```

    // group.
    setpgid(0, 0);
    fflush(stdout);
    // 一定要等到父进程设置完子进程的进程组之后, 才能执行execve
    close(pipe_fd[1]); // 关闭管道的写端, 因为只读
    // 等待父进程的通知
    char dummy;
    while (read(pipe_fd[0], &dummy, 1) != 0) { }
    close(pipe_fd[0]);

    // 执行命令
    Execve(tok.argv[0], tok.argv, environ);
    _exit(0);
}
else {
    // 父进程
    // 如果是后台进程, 那么就不需要等待子进程结束
    if (bg) {
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); // 在添加到job_list之前, 先屏蔽所有信号
        // 添加到job_list
        addjob(job_list, pid, BG, cmdline);
        fflush(stdout);
        // 通知子进程可以执行execve了
        close(pipe_fd[0]); // 关闭管道的读端, 因为只写
        char dummy = 'a';
        write(pipe_fd[1], &dummy, 1);
        close(pipe_fd[1]);

        Sigprocmask(SIG_SETMASK, &mask_one, NULL); // 解除屏蔽
        // 打印信息
        printf("[%d] (%d) %s\n", pid2jid(pid), pid, cmdline);
        fflush(stdout);
    }
    else {
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); // 在添加到job_list之前, 先屏蔽所有信号
        // 如果是前台进程, 那么就需要等待子进程结束
        addjob(job_list, pid, FG, cmdline);
        // printf("[%d] (%d) %s\n", pid2jid(pid), pid, cmdline); // 确认是加入了
        job_list, 而且在收到SIGCHLD信号之前, 不会有其他进程修改job_list
        fflush(stdout);

        // 通知子进程可以执行execve了
        close(pipe_fd[0]); // 关闭管道的读端, 因为只写
        char dummy = 'a';
        write(pipe_fd[1], &dummy, 1);
        close(pipe_fd[1]);

        // 等待前台进程结束
        Sigprocmask(SIG_SETMASK, &mask_one, NULL); // 解除屏蔽
        waitfg(pid);
    }
}
Sigprocmask(SIG_SETMASK, &prev_one, NULL); // 解除屏蔽
}
return;
}
}

```

```

void waitfg(pid_t pid) {
    sigset_t mask_all;
    Sigemptyset(&mask_all);
    while(fgpid(job_list) == pid) {
        sigsuspend(&mask_all);
    }
}

```

5 trace07 内建程序 jobs

这个写在 `builtin_cmd` 里面，使用已经写好的 `list_jobs` 函数。

`STDOUT_FILENO` 是一个在 POSIX 系统上定义的常量，它表示标准输出文件描述符的整数值。具体而言，`STDOUT_FILENO` 的值通常是 1。

在 POSIX 系统中，有三个标准的文件描述符，它们是：

- **标准输入 (Standard Input)：** 文件描述符 0，通常表示为 `STDIN_FILENO`，用于输入数据，默认为键盘输入。
- **标准输出 (Standard Output)：** 文件描述符 1，通常表示为 `STDOUT_FILENO`，用于输出数据，默认为终端显示。
- **标准错误 (Standard Error)：** 文件描述符 2，通常表示为 `STDERR_FILENO`，用于输出错误信息，默认为终端显示。

这些文件描述符是由操作系统创建并与程序关联的，默认情况下，它们通常分别与终端设备关联。在 C 语言中，这些文件描述符由 `stdio.h` 头文件中的 `stdin`、`stdout`、`stderr` 指针表示。例如，`stdout` 指向标准输出流。

`STDOUT_FILENO` 可以在程序中使用，以明确表示标准输出文件描述符的整数值。例如，可以使用它在低级别的文件描述符层次上操作标准输出流。例如，将它作为参数传递给 `write` 函数，用于将数据写入标准输出：

```

#include <unistd.h>

int main() {
    const char *message = "Hello, STDOUT_FILENO!\n";
    write(STDOUT_FILENO, message, strlen(message));
    return 0;
}

```

在这个例子中，`write` 函数使用 `STDOUT_FILENO` 将消息写入标准输出。

6 trace08 中断前台程序（检查 `sigint_handler`）

```

/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenever the
 *   user types ctrl-c at the keyboard.  Catch it and send it along
 *   to the foreground job.
 */
void sigint_handler(int sig)
{
    // trace08 passed
    // 中断当前前台进程组

```

```

int olderrno = errno;
pid_t pid;
sigset_t mask_all, prev_all;
Sigfillset(&mask_all);
Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
pid = fgpid(job_list);
Sigprocmask(SIG_SETMASK, &prev_all, NULL);
if (pid != 0) {
    // 如果当前有前台进程，那么就中断它
    Kill(-pid, SIGINT);
}
errno = olderrno;
return;
}

```

7 trace 09 停止前台程序（检查 `sigtstp_handler`）

```

void sigtstp_handler(int sig)
{
    // trace09 passed
    // 停止当前前台进程组
    int olderrno = errno;
    pid_t pid;
    sigset_t mask_all, prev_all;
    Sigfillset(&mask_all);
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    pid = fgpid(job_list);
    if (pid != 0) {
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
        // 如果当前有前台进程，那么就停止它
        Kill(-pid, SIGINT);
    }
    errno = olderrno;
    return;
}

```

8 trace10 发送 `SIGTERM` 信号

`SIGTERM` 是一个 POSIX 标准定义的信号，表示终止（terminate）。通常，当使用 `kill` 命令或类似的操作时，会向进程发送 `SIGTERM` 信号。这是一种通知进程正常终止的方式，允许进程在收到信号后执行一些清理工作，保存状态等。

有些特点关于 `SIGTERM`：

- **默认行为：** 如果进程没有捕获 `SIGTERM` 信号，它将终止运行。这是与 `SIGKILL` 信号不同的地方，`SIGKILL` 信号是强制终止进程，不能被进程捕获、阻塞或忽略。
- **捕获信号：** 进程可以通过注册信号处理函数来捕获 `SIGTERM` 信号。在捕获信号的处理函数中，程序员可以执行一些清理工作，并选择是否终止进程。

以下是一个简单的例子，演示了如何在程序中捕获 `SIGTERM` 信号：

```

#include <stdio.h>

```



```

#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void sigterm_handler(int signo) {
    printf("Received SIGTERM signal. Cleaning up...\n");
    // 进行一些清理工作
    exit(EXIT_SUCCESS);
}

int main() {
    // 注册 SIGTERM 信号的处理函数
    signal(SIGTERM, sigterm_handler);

    printf("My process ID is: %d\n", getpid());
    printf("Send a SIGTERM signal to terminate this process.\n");

    // 主程序继续执行其他工作
    while (1) {
        // ...
    }

    return 0;
}

```

在这个例子中，程序通过 `signal` 函数注册了一个处理 `SIGTERM` 信号的处理函数 `sigterm_handler`。当程序收到 `SIGTERM` 信号时，将调用这个处理函数执行相应的操作。

不知道是在哪里实现的，但是过了。

9 trace11（检查孩子给自己发送 `SIGINT`）

10 trace12（检查孩子给自己发送 `SIGTSTP`）

11 trace13（孩子给前台发送 `SIGINT`）

在信号处理程序里不可以使用异步信号不安全的 `printf`，我这里使用的是 `csapp.h` 里给出的 `Sio` 包。

在 POSIX 系统上，`WTERMSIG` 是一个宏，用于从 `wait` 系列函数的返回状态中提取导致子进程终止的信号编号。`WTERMSIG` 返回的是一个整数，表示导致子进程终止的信号的编号。

它的使用形式如下：

```

#include <sys/wait.h>
int WTERMSIG(int status);

```

`status` 是 `wait` 或 `waitpid` 函数返回的状态，其中包含有关子进程终止的信息。

12 trace14 (孩子给前台发送SIGTSTP)

```
/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 * a child job terminates (becomes a zombie), or stops because it
 * received a SIGSTOP, SIGTSTP, SIGTTIN or SIGTTOU signal. The
 * handler reaps all available zombie children, but doesn't wait
 * for any other currently running children to terminate.
 */
void
sigchld_handler(int sig)
{
    // P536 要保存和恢复errno
    int olderrno = errno;
    int status; // waitpid的一个参数
    pid_t pid;
    sigset_t mask_all, prev_all;
    // 如果处理程序和主程序共享一个全局数据结构, 那么就需要在处理程序中屏蔽所有信号
    Sigfillset(&mask_all);

    while ((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0) {
        // WNOHANG | WUNTRACED: 如果没有子进程终止或者停止, 那么waitpid就会立即返回0
        // 如果有子进程终止或者停止, 那么waitpid就会返回子进程的pid
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        if (WIFEXITED(status)) {
            // 如果子进程正常终止, 那么就删除job_list中的记录
            deletejob(job_list, pid);
        }
        else if (WIFSIGNALED(status)) {
            // 如果子进程是因为信号终止的, 那么就打印信息
            // printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,
WTERMSIG(status));
            // 在信号处理程序里不可以使用异步信号不安全的函数, 比如printf
            // 所以使用sio_put来代替printf
            // WTERMSIG(status)返回导致子进程终止的信号的编号
            sio_puts("Job ["); // 因为sio_put不支持%d, 所以只能一个一个输出
            sio_putl(pid2jid(pid));
            sio_puts("] (");
            sio_putl(pid);
            sio_puts(") terminated by signal ");
            sio_putl(WTERMSIG(status));
            sio_puts("\n");
            // 然后删除job_list中的记录
            deletejob(job_list, pid);
            // trace13 passed
        }
        else if (WIFSTOPPED(status)) {
            // 如果子进程是因为信号停止的, 那么就打印信息
            // printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,
WSTOPSIG(status));
            sio_puts("Job [");
            sio_putl(pid2jid(pid));
            sio_puts("] (");
            sio_putl(pid);
            sio_puts(") stopped by signal ");
        }
    }
}
```

```

        sio_putl(WSTOPSIG(status)); // 和WTERMSIG一样，返回导致子进程停止的信号编号
        sio_puts("\n");
        // 然后修改job_list中的记录
        getjobpid(job_list, pid)->state = ST;
    }
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
}

errno = olderrno;
return;
}

```

13 trace15 bg built-in command

JIDs should be denoted on the command line by the prefix '%'. For example, "%5" denotes JID 5, and "5" denotes PID 5.

在Unix-like操作系统中，`kill`命令的主要目的是发送信号给进程。信号是一种进程间通信的机制，用于通知进程发生了某个事件。`kill`命令可以向目标进程发送不同的信号，其中包括`SIGCONT`信号。

`SIGCONT`信号用于继续（恢复）一个停止状态的进程。当进程收到`SIGSTOP`信号时会被停止，而`SIGCONT`信号则用于恢复这个进程的执行。这种机制通常用于进程的暂停和继续操作。

当你使用`kill`命令并指定`-CONT`选项时，它实际上是在向目标进程发送`SIGCONT`信号，从而使该进程从停止状态恢复到运行状态。语法通常是：

```
kill -CONT <进程ID>
```

这种机制对于控制进程的执行状态非常有用，例如，你可能希望暂停一个进程以进行调试，然后在一段时间后恢复其执行。

命令名字的选择通常是为了提供一种直观、易于记忆的方式，与命令的功能或目的有关。在Unix和类Unix系统中，`kill`命令的名字确实有点令人困惑，因为它的名字暗示着“杀死”进程，而实际上，`kill`主要是用来发送信号而不是终止进程。

历史上，`kill`命令的设计目标是发送信号，但它也可以用来终止进程。发送信号的功能对于操作系统的进程管理是非常重要的，因此`kill`命令作为一个通用工具被设计用于与进程通信。命令的名字可能有点不直观，但它已经成为Unix和类Unix系统中标准的命令之一，因此一直保留下来。

总的来说，尽管`kill`命令的名字可能让人联想到终止进程，但它实际上是一个通用的进程管理工具，主要用于发送信号，其中包括但不限于终止信号。

逆天的`kill`是用来发送信号的！！！！！！！！！！

14 trace16-18 都是关于fg和bg的

```

// 这个函数要实现内建的bg和fg命令
void conduct_bgfg(char **argv) {
    // IDs should be denoted on the command line by the prefix '%'.
    // For example, "%5" denotes JID 5, and "5" denotes PID 5
    // 同时通过发送SIGCONT信号来恢复进程组，也就是一个job，但是给的表示这个job的参数不同
    struct job_t *job;
    char *id = argv[1]; // id是一个字符串，到底是JID还是PID
}

```

```

if (id[0] == '%') {
    // JID
    int jid = atoi(id + 1);
    job = getjobjid(job_list, jid);
    if (job == NULL) {
        printf("%s: No such job\n", id);
        return;
    }
}
else {
    // PID
    pid_t pid = atoi(id);
    job = getjobpid(job_list, pid);
    if (job == NULL) {
        printf("(%s): No such process\n", id);
        return;
    }
}

// 如果是bg命令, 那么就把job的状态改为BG
if (!strcmp(argv[0], "bg")) {
    job->state = BG;
    printf("[%d] (%d) %s\n", job->jid, job->pid, job->cmdline);
    fflush(stdout);
    // 使用kill发送信号
    Kill(-(job->pid), SIGCONT); // 给当前的进程组发送SIGCONT信号
    fflush(stdout);
}
else {
    // 如果是fg命令, 那么就把job的状态改为FG
    job->state = FG;
    // 使用kill发送信号
    Kill(-(job->pid), SIGCONT); // 给当前的进程组发送SIGCONT信号
    fflush(stdout);
    waitfg(job->pid);
}
return;
}

```

15 trace19-20 继续检验 **SIGINT** 和 **SIGTSTP** 的

16 trace21 给每个进程重启

17 trace22 I/O重定向

17.1 **open** 函数

open 函数是一个用于打开文件的系统调用, 通常在Unix-like系统中提供。它的原型如下:

```

#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

```

- `pathname` 是一个指向以 null 结尾的字符串，表示要打开的文件的路径名。
- `flags` 是一个用于指定文件打开方式和行为的标志参数。
- `mode` 是一个用于指定新文件权限的参数，通常只在创建文件时使用。

`open` 函数的主要用途是打开文件并返回一个文件描述符，应用程序可以使用这个文件描述符来进行读写等操作。如果打开文件失败，`open` 将返回 -1，并且可以通过检查全局变量 `errno` 获取错误信息。

下面是一些常用的 `flags` 参数和它们的含义：

- `O_RDONLY`：只读方式打开文件。
- `O_WRONLY`：只写方式打开文件。
- `O_RDWR`：读写方式打开文件。
- `O_CREAT`：如果文件不存在，就创建它。
- `O_TRUNC`：如果文件存在，并且以写入方式打开，就截断文件。
- `O_APPEND`：打开文件并将文件指针移到文件末尾。
- `O_EXCL`：与 `O_CREAT` 一同使用，用于确保文件不存在。

`mode` 参数通常用于指定新文件的权限，例如：

- `S_IRUSR`：用户可读权限。
- `S_IWUSR`：用户可写权限。
- `S_IXUSR`：用户可执行权限。
- `S_IRGRP`：组可读权限。
- `S_IWGRP`：组可写权限。
- `S_IXGRP`：组可执行权限。
- `S_IROTH`：其他用户可读权限。
- `S_IWOTH`：其他用户可写权限。
- `S_IXOTH`：其他用户可执行权限。

以下是一个简单的使用 `open` 函数的示例：

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    int file_descriptor = open("example.txt", O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);

    if (file_descriptor == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }
}
```

```

}

// 在这里进行文件操作，例如写入数据
write(file_descriptor, "Hello, World!\n", 13);

// 关闭文件
close(file_descriptor);

return 0;
}

```

在这个例子中，`open` 函数创建了一个新文件 "example.txt"，并以只写方式打开，如果文件已存在则截断文件。接着，通过 `write` 函数写入了数据，最后使用 `close` 函数关闭了文件。

17.2 `write` 函数

`write` 函数用于将数据从缓冲区写入文件描述符所指定的文件中。它是Unix和类Unix系统中的系统调用，其声明如下：

```

#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);

```

- `fd` 是文件描述符，指定要写入的目标文件。
- `buf` 是指向包含要写入数据的缓冲区的指针。
- `count` 是要写入的字节数。

函数返回实际写入的字节数，如果出现错误则返回-1。错误信息存储在全局变量 `errno` 中，可以通过 `perror` 函数或其他手段进行打印。

下面是一个简单的例子，演示如何使用 `write` 函数向文件中写入数据：

```

#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    // 打开（或创建）文件
    int file_descriptor = open("output.txt", O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR);

    if (file_descriptor == -1) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    // 写入数据到文件
    const char *data = "Hello, World!\n";
    ssize_t bytes_written = write(file_descriptor, data, strlen(data));

    if (bytes_written == -1) {
        perror("Error writing to file");
        close(file_descriptor);
        exit(EXIT_FAILURE);
    }
}

```

```

}

// 关闭文件
close(file_descriptor);

return 0;
}

```

在这个例子中，`write` 函数被用于将字符串 "Hello, World!\n" 写入到文件 "output.txt" 中。请注意，`write` 函数通常用于低级I/O，更高级的标准库函数如 `fprintf` 和 `fwrite` 也可以用于文件写入，并提供了更多的便利和格式化选项。

17.3 写法

写在 `eval` 函数里面，而且是要写在子进程里面，昨天看了这么久全都是bullshit！！！！

```

// 如果有重定向的话，那么就需要先打开文件
int fd_in = -1, fd_out = -1;
if (tok.infile != NULL) {
    fd_in = open(tok.infile, O_RDONLY);
    if (fd_in < 0) {
        printf("%s: No such file or directory\n", tok.infile);
        return;
    }
}
if (tok.outfile != NULL) {
    fd_out = open(tok.outfile, O_WRONLY | O_CREAT | O_TRUNC);
    if (fd_out < 0) {
        printf("%s: No such file or directory\n", tok.outfile);
        return;
    }
}
if (fd_in != -1) {
    Dup2(fd_in, STDIN_FILENO);
    close(fd_in);
}
if (fd_out != -1) {
    Dup2(fd_out, STDOUT_FILENO);
    close(fd_out);
}
}

```

这个是来自于一个讲解shell如何实现重定向的网站。

[\[如何用底层I/O实现重定向?\]](#):

17.3.1 原理:

1. 实现重定向，只需要将进程的输出（这里是一种抽象的表达）写到文件就行了。
2. 创建进程时，**标准输入(stdin)**、**标准输出(stdout)**、**标准错误(stderr)**的文件描述符（默认为0、1、2）就会被打开，并连接到终端上。
3. 进程的输出总是默认为标准输出（printf等等函数的输出）

这样看来，我们要做事只有一件，**使0（或1）成为一个文件的文件描述符**。联想到最低可用文件描述符（Lowest-Available-fd）原则，我们有了下面的三种方法（摘自*Unix/Linux*编程实践教程）。

1. close->open

关闭0或1的文件描述符，再打开文件，分配给文件的文件描述符只能是0或1

2. open->close（0或1）->dup->close（被复制的）

dup是一个系统调用，只有一个参数，作用为复制参数中的文件描述符（虽控制同一个文件，被同时影响（偏移等），但两个文件描述符的数值并不相等），成功时返回新的文件描述符，失败时返回-1。

由于dup遵循最低可用文件描述符(Lowest-Available-fd)原则，因此这个方案是可行的

3. open->dup2->close（被复制的）

dup也是一个系统调用，有两个参数(oldfd、newfd)，作用为复制oldfd给newfd（同样的，两个文件描述符都控制一个文件），返回值与dup同理。

```
/*
 * redirect_to_file.c
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <wait.h>

int main(void)
{
    pid_t pid;
    int fd;

    printf("About to run who into a file\n");
    /* create a new process or quit */
    if ((pid = fork()) == -1)
    {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    /* child does the work */
    if (pid == 0)
    {
        /* 第一种方式 */
        close(1); // close
        fd = open("userlist", O_CREAT | O_WRONLY); // then open
        /* 第二种方式 */
        // fd = open("userlist", O_CREAT | O_WRONLY);
        // dup2(fd, 1);
        execlp("who", "who", NULL); // and run
        perror("execlp");
        exit(EXIT_FAILURE);
    }
    /* parent waits then reports */
    if (pid != 0)
    {
        wait(NULL);
        printf("Done running who, results in userlist\n");
    }
}
```



```

    return 0;
}

```

18 trace23 内建命令的重定向

发现 `jobs` 无法重定向输出，那么直接在做 `jobs` 的时候重定向输出就好了

```

int builtin_cmd(char **argv, struct cmdline_tokens * tok) // 书上介绍的判断是否是内建命令的函数
{
    if(!strcmp(argv[0], "quit")) // quit命令直接结束shell
        exit(0); // trace01
    else if(!strcmp(argv[0], "jobs")) {
        // 重定向到文件中
        if(tok->outfile != NULL) {
            int fd_out = open(tok->outfile, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR |
S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);
            if(fd_out < 0) {
                printf("%s: No such file or directory\n", tok->outfile);
                return 1;
            }
            else listjobs(job_list, fd_out);
            close(fd_out);
        }
        listjobs(job_list, STDOUT_FILENO); // 使用标准输出来输出所有的jobs
        // trace07 passed
        return 1;
    }
    else if(!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg")) {
        conduct_bgfg(argv);
        return 1;
    }
    else if(!strcmp(argv[0], "kill"))
        return 1;
    else if(!strcmp(argv[0], "nohup"))
        return 1;
    else
        return 0;
}

```

trace24也差不多

19 trace25 使用 `nohup`

```

else if(!strcmp(argv[0], "kill")) {
    conduct_kill(argv);
    return 1;
}
else if(!strcmp(argv[0], "nohup")) {
    // 只要对外部命令解决这个问题就好了
    // 让跟在后面的命令忽略SIGHUP信号
    Signal(SIGHUP, SIG_IGN);
    return 1;
}

```

20 trace26 使用 `kill`

```
void conduct_kill(char **argv) {
    struct job_t *job;
    char *id = argv[1]; // id是一个字符串, 到底是JID还是PID
    if (id[0] == '%') {
        // JID
        if (id[1] == '-') {
            // 要通过杀死jid为首的进程组
            int jid = atoi(id + 2);
            job = getjobjid(job_list, jid);
            if (job == NULL) {
                printf("%%s: No such process group\n", id+2);
                return;
            }
            Kill(-(job->pid), SIGTERM);
        }
        else {
            // 要通过杀死jid为首的进程
            int jid = atoi(id + 1);
            job = getjobjid(job_list, jid);
            if (job == NULL) {
                printf("%s: No such job\n", id);
                return;
            }
            Kill(job->pid, SIGTERM);
        }
    }
    else {
        // PID
        if (id[0] == '-') {
            // 要通过杀死pid为首的进程组
            pid_t pid = atoi(id + 1);
            job = getjobpid(job_list, pid);
            if (job == NULL) {
                printf("(%s): No such process group\n", id+1);
                return;
            }
            Kill(-(job->pid), SIGTERM);
        }
        else {
            // 要通过杀死pid为首的进程
            pid_t pid = atoi(id);
            job = getjobpid(job_list, pid);
            if (job == NULL) {
                printf("(%s): No such process\n", id);
                return;
            }
            Kill(job->pid, SIGTERM);
        }
    }
}
```

trace27-28也是一样的trace30 kill process group

21 **trace29** 大杂烩之前做好了应该都是对的

22 **trace31** 是过不去的