操作系统

1. porcess

a process is a program in execution

进程就是一个正在运行的程序

1.1. synchronous exceptions 同步异常

一般是程序内部出现的

traps 陷阱（类似于函数，是故意设置的）

1.

2. faults 错误，大部分时间是无意触发的（比如除数为 0 divided by zero）

3. aborts 终止，无意的（不应该触发，比如 memory ECC error）

1.2. asynchronous exceptions 异步异常

一般是程序外部出现的

1. interrupts 由硬件启动的触发的叫中断,比如终端里的 ctrl+C

2. fork

会启动一个新的线程，执行一次该函数下的所有程序（最终 fork 下面的程序会执行两次），哪怕 fork 下面代码的缩进不在 fork 下一级，也会执行。

```
int main () {
    fork();
    printf("Hello world!\n");
}
```

```
Hello world!
Hello world!
```

线程间不共享 register 和 variable!

注意：即使是全局变量，fork 也会将其 copy 一次，存入另一个寄存器以供新的进程使用（也就是说这是后全局变量不共享），哪怕这个全局变量在 fork 函数上方。

fork 启动的线程是被单独维护的。

fork 函数有返回值：

```
typedef int pid_t
```

进程的唯一标识

fork 函数生成子进程会返回 0 （应该不是 pid=0

fork 创建进程失败会返回-1

3. terminate 终止程序

3.1. return

3.2. void exit(int status);

- normal termination → exit status 0
- other exit status values = error

exit 的 status 参数会返回状态码给操作系统（不是程序中的“返回值”）

3.3. int atexit(void (*fn)());

atexit 会在整个程序中 exit 函数（或 main 函数 return 时）调用前，调用函数 fn

（注意 void (*fn)() 表示一个名为 fn 的 pointer，指向的是一个无返回值且无参数的函数。在 c 语言中，想要函数传参必须使用指针）

如果使用了 atexit 注册了多个函数，会逆序调用注册的函数：

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

void first(void) {
    printf("First function called\n");
}

void second(void) {
    printf("Second function called\n");
}

int main() {
    atexit(second);  // 注册第二个函数
    atexit(first);   // 注册第一个函数

    printf("Main function ends\n");
    exit(0);  // 触发 atexit 注册的函数调用
    return 0;
}
```

返回：

```
Main function ends
First function called
Second function called
```

3.4. Zombie Process

僵尸进程（计算机科学）：在操作系统中，僵尸进程（Zombie Process）是指一个已经完成执行但尚未被其父进程回收其进程描述符的进程。僵尸进程已经释放了几乎所有的资源，但仍然保留一个很小的内核对象，以便其父进程可以读取其状态信息。

僵尸进程是指那些已经完成执行，但是其父进程还没有调用wait或waitpid来收集其状态信息的进程。僵尸进程会占用系统资源，因为它的进程描述符仍然存在。

当我们使用 fork 创建两个进程，再选一个结束，在内核查询时，被结束的进程显示“program_name defunct”，本质就是 Z（zombie），直到程序完全停止，该进程才无法被查询。

注意，只有当 fork 生成的子进程停止且 parent 继续运行，child 才会变成 zombie。

如果 parent 停止而 child 运行，parent 是不会变成 zombie 的。

child 变成 Z 之后 parent 会 reap（回收）zombie child，而 child 不会去 reap parent

当父进程停止，子进程会立即被 kernel 收养（此时子进程再停止，不会变成 zombie，会直接被回收），如果子进程之前是 zombie，则会被直接清理。

parent 进程停止后是被 shell 收养（reap）的，在命令行中可以很明显看到被停止的 parent 进程被标识“bash”。

bash（Bourne Again SHell） 标识该进程被 shell 收养。

所以所有的进程实际上都被 shell 看管，进程只有在未停止时会 reap 它已死亡的 child。

3.4.1. pid_t wait(int *stat_loc)

1. 当调用这个函数，会一直等待，直到有一个线程变成 zombie 被捕获。
2. 返回该被捕获的线程的 pid，并在指针 stat_loc 处填充状态信息（如果传入 NULL 则不填充）。这里看 process2 的 ppt-22 页。
   wait(&stat)（注意 stat 需要被初始化），当 exit 调用后，stat 自动被填充，填充内容为 exit 返回给 os 的值。
   下面的函数依据 stat 的内容会返回 T/F（相较于判断 stat 的数值，这种方式更快）：

```
/* macros */
WIFEXITED(status)    /* exited normally? */
WEXITSTATUS(status)  /*  if so, exit status */
WIFSTOPPED(status)   /* process stopped? */
WIFSIGNALED(status)  /* process signaled? */
WTERMSIG(status)     /*  if so, signal number */

/* prints information about a signal */
void psignal(unsigned sig, const char *s);
```

3. 被 wait 捕获的 zombie 线程直接死亡（不再显示）

如果有多个 zombie 线程，随机捕获一个

当没有 child，直接返回-1 并填充 errno

wait 多用于同步线程 synchronize

3.4.2. pid_t waitpid(pid_t pid, int *stat_loc, int options);

不同于上一节中普通的 wait，waitpid 可以等待并捕获固定的 child，而不是随机捕获。

如果传入的 pid 为-1，表示等待任意 child（同 wait）

下面的是 int options 中的内容：

第一个 w no hang 是指不需要等待，直接停止返回 0（这个触发的前提是在 wait 调用前没有 child 停止）；

第二个 w un traced 则是会输出 child 的信息。

```
/** Wait options **/

/* return 0 immediately if no terminated children */
#define WNOHANG     0x00000001

/* also report info about stopped children (and others) */
#define WUNTRACED   0x00000002
```

4. Running new programs

需要查看下面的函数的详细用途

跳转到一个全新的程序（每次都是一个完全新的开始）：

```
/* the "exec family" of syscalls */

int execl(const char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);
```

注意，即使跳转了程序，新的程序依然在同一条进程中运行（会被 parent 捕获）。

5. processmgmt - 3

5.1. shell 命令行的运行原理

1. 显示提示符（ `$` ），等待用户输入命令。
2. 读取用户输入的命令，并将其分割成命令和参数（比如输入“ls -l”，则被分割为"ls"和"-l"），存储在 `argv` 数组中。
3. 使用 `fork()` 创建一个新的进程（子进程）。
4. 在子进程中，使用 `execvp(argv[0],argv)` 执行用户输入的命令。
5. 如果 `execvp()` 执行失败（即命令未找到），则打印错误信息并退出子进程。
6. 在父进程中，使用 `waitpid()` 等待子进程完成。

Python

```
pid_t pid;
char buf[80], *argv[10];
while (1) {
     /* print prompt */
     printf("$ ");

     /* read command and build argv */
     fgets(buf, 80, stdin);
     for (i=0, argv[0] = strtok(buf, " \n");
         argv[i];
         argv[++i] = strtok(NULL, " \n"));

     /* fork and run command in child */
     if ((pid = fork()) == 0)
     if (execvp(argv[0], argv) < 0) {
         printf("Command not found\n");
         exit(0);
     }

     /* wait for completion in parent */
     waitpid(pid, NULL, 0);
}
```

5.2. fg & bg

`fg` 和 `bg` 是在 Unix 和类 Unix 操作系统的 shell 中使用的两个命令，它们用于控制作业（job）的运行环境。

5.2.1. `fg` 命令

`fg` 命令用于将一个在后台运行的作业（job）移到前台，并且继续执行它。这个命令常用于你想要继续在终端监视某个长时间运行的进程的情况。

使用方法:

- `fg`：将最后一个放入后台的作业带到前台。
- `fg %job_number` 或 `fg %job_name`：将指定编号或名称的作业带到前台。

例如：

```
1   fg %1
```

这会将作业号为1的作业带到前台。

> 这个 number 是指的使用 jobs 输出的 [number]
>
> 
>

5.2.2. `bg` 命令

`bg` 命令用于将一个在前台运行的作业放到后台执行，它会使作业在后台继续运行，而不会阻塞终端。

使用方法:

- `bg`：将当前作业放到后台执行。
- `bg %job_number` 或 `bg %job_name`：将指定编号或名称的作业放到后台执行。

例如：

```
1   bg %1
```

这会将作业号为1的作业放到后台执行。

> fg 恢复的 job 可被 ctrl Z 停止，但 bg 的不行。
>
> 在此基础上，shell 不需要等待 bg job 的停止，也可以继续读取用户命令

5.2.3. &符号

在 shell 输入命令的结尾加一个"&"，表示该程序会在 bg 中运行。

> 这意味着命令会立即启动，但 shell 不会等待该命令完成，而是会立即准备接受下一个命令。这对于那些需要长时间运行的命令非常有用，因为它们不会阻塞 shell 的进一步交互。

5.2.4. background zombie

bg job 也需要回收，否则会变成 zombie，在正常的 shell 中自然会被回收，但如果是要我们自己写一个 shell，很显然需要给出回收程序。

下面是一个模拟 shell 回收 bg 的示例：

```
/* ... and reap all bg zombies at once */
while (waitpid(-1, NULL, WNOHANG) > 0) ;
```

但是这个代码有个问题，就是回收太耗时：

reaping may be delayed for a long time

所以我们希望线程在完成后会发送一个提示（signal）

5.3. int kill(pid_t pid, int sig);

向进程发送信号 signal

No	Name	Default Action	Description
1	SIGHUP	terminate process	terminal line hangup
2	SIGINT	terminate process	interrupt program
3	SIGQUIT	create core image	quit program
6	SIGABRT	create core image	abort program (formerly SIGIOT)
9	SIGKILL	terminate process	kill program
10	SIGBUS	create core image	bus error
11	SIGSEGV	create core image	segmentation violation
12	SIGSYS	create core image	non-existent system call invoked
13	SIGPIPE	terminate process	write on a pipe with no reader
14	SIGALRM	terminate process	real-time timer expired
17	SIGSTOP	stop process	stop (cannot be caught or ignored)
18	SIGTSTP	stop process	stop signal generated from keyboard
19	SIGCONT	discard signal	continue after stop
20	SIGCHLD	discard signal	child status has changed
30	SIGUSR1	terminate process	User defined signal 1
31	SIGUSR2	terminate process	User defined signal 2

比如向子进程发送 kill-sigint，会使得该进程变成 zombie，直到被 wait 回收

- if kill is given a negative pid, signal is
  sent to all processes with PGID=abs(pid)

如果 传入的 pid 是一个负数，则 kill 的 signal 会被发送给 id = abs(pid) 的进程组

5.4. PGID

每一个进程都有一个进程组-pgid

5.4.1. int setpgid(pid_t pid, pid_t pgid);

- if pid=0, alter the calling process
- if pgid=0, set the process’ s PGID equal to its PID

- `pid_t pid`：这是你想要改变其进程组的进程的进程 ID。如果 `pid` 等于0，那么这个调用会改变调用它的进程（即当前进程）的进程组，而不是指定的进程。
- `pid_t pgid`：这是你想要设置的新进程组 ID。如果 `pgid` 等于0，那么进程的进程组 ID 将被设置为它的进程 ID（PID）。这意味着进程将成为一个新的进程组的领导，并且该进程组的 ID 将与进程的 PID 相同。

5.5. sig_t signal(int sig, sig_t func);
typedef void (*sig_t) (int);

- func is typically a pointer to a signal handler function — “callback” API
- some signals cannot be caught! (e.g., SIGKILL)

signal 函数中的 func 是一个回调指针，有些在 kull 中可以使用的 signal 不能在 func 中使用

- func can also take special values:
  - SIG_IGN: ignore signal ------- 表示忽略下面的信号
  - SIG_DFL: use default action

5.6. 传递信号（核心机制）- delivering a signal

ctrl C

```
Back in main
^CSignal 2 received
^CSignal 2 received
Back in main
^CSignal 2 received
^C^CSignal 2 received
Back in main
^CSignal 2 received
^C^C^CSignal 2 received
```