

ECF

13 元培数科 常欣海

Here we go! →

2024/11/20



Knowledge Review

This part is cited from Arthals

控制流

Control Flow

控制流：直观理解就是一条条指令的执行顺序。

处理器读取并执行一串指令序列，程序计数器产生一串相应的序列： a_0, a_1, \dots

每次从 a_k 到 a_{k+1} 的过渡称为 **控制转移**。

控制转移

Control Transfer

常见：跳转、分支、调用、返回（对程序状态的变化做出反应）

但是，如何对系统状态的变化做出反应？

比如：`Ctrl+C`、请求磁盘、处理异常...

异常控制流

Exceptional Control Flow

定义：程序执行过程中遇到特殊事件或条件时，改变正常指令执行顺序的机制，它发生在计算机系统的各个层次。

包括：

- 异常
- 进程控制
- 信号
- 非本地跳转

异常

Exception

异常：控制流的突变。发生异常时，控制流会转移至 **操作系统内核** 以响应某些事件（处理器状态的变化）。

内核：操作系统常驻内存的部分，负责管理计算机硬件和软件资源。

异常处理类似于过程调用，但有区别：

Diff	过程调用	异常
返回位置	返回地址	当前指令 I_{cur} / 下一条指令 I_{next}
跳转准备	压栈相关信息	压栈相关信息，但会额外压栈一些内容
运行模式	用户态	内核态

异常的类别

Exception Categories

异常可以分为四类：中断、陷阱、故障、终止。

类别	原因	异步 / 同步	返回行为
中断 (Interrupt)	来自 I/O 设备的信号	异步	总是返回到下一条指令 I_{next}
陷阱 (Trap)	有意的异常	同步	总是返回到下一条指令 I_{next}
故障 (Fault)	潜在可恢复的错误	同步	可能返回到当前指令 I_{cur}
终止 (Abort)	不可恢复的错误	同步	不会返回

- **异步异常：**是由处理器外部的 I/O 设备中的事件产生的
- **同步异常：**是一条指令的直接产物

辨析：异步异常和当前控制流无关，是来自“外界”的；同步异常和当前控制流有关，是源自当前指令。

This slide is cited from Arthals.

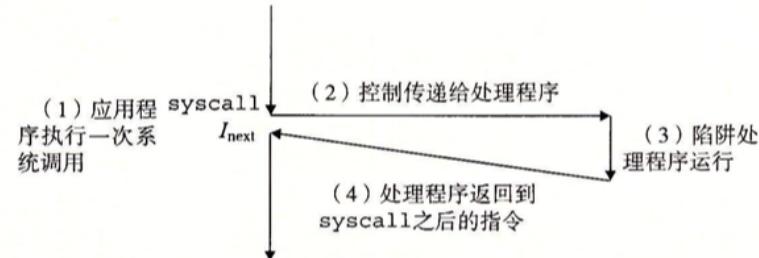
中断

Interrupt

类型：异步（来自外部）

返回行为：总是返回到下一条指令 I_{next}

常见：I/O 设备（磁盘读取完成）、定时器（周期性定时器中断）



直观理解：

1. 你正在写作业，突然你父母（不是你自己写着写着发现的）叫你吃饭（异步）
2. 你得到消息后停笔，但是已经写完的东西不需要再写 (I_{cur})
3. 等你吃完饭回来后，你再从写完的字的下一个字开始接着写 (I_{next})

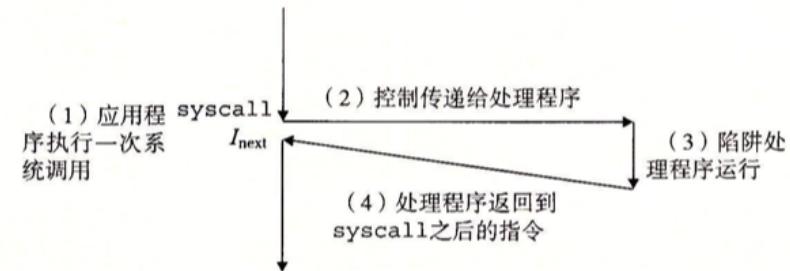
陷阱、系统调用

Trap & System Call

类型：同步（源自当前指令），陷阱是**故意的异常**

返回行为：总是返回到下一条指令 I_{next}

常见：系统调用，如 `write` `read` 等涉及文件 I/O 的指令，`fork` 等涉及进程控制的指令。



直观理解：

1. 你正在写作文，突然你意识到需要引用一段名人名言（同步，故意的）
2. 你停笔去查谷歌，但是已经写完的字不需要再写 (I_{cur})
3. 查完书回来后，你再从写完的字的下一个字开始接着写 (I_{next})

故障

Fault

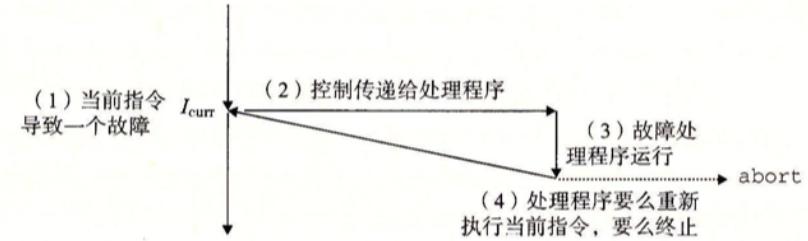
类型：同步（源自当前指令），故障是 **潜在可恢复的错误**

返回行为：可能返回到当前指令 I_{curr}

常见：缺页故障

直观理解：

1. 你正在写作业，突然你发现自己写错了一段话 (I_{curr} 导致了故障)
2. 你停笔找修正带，但是有可能找不到（尝试运行故障处理程序）
3. 如果你能找到，你可以用修正带修正后重新写 (I_{curr})
4. 如果找不到，你只能终止这次写作业（abort），出门看看能不能买到修正带，或者干脆开摆



终止

Abort

类型：同步，终止是 **不可恢复的错误**

返回行为：不会返回

常见：非法操作、地址越界、算术溢出、除零错误、硬件错误

直观理解：

1. 你正在写作业，突然发现你写错题了
2. 直接不写这道题了，拜拜了您内

常见的异常

Common Exceptions

异常号	描述	异常类别
0	除法错误	故障 \Rightarrow abort
13	一般保护故障	故障 \Rightarrow abort
14	缺页	故障 \Rightarrow core
18	机器检查	终止 DRAM, SRAM FAILURE
32~255	操作系统定义的异常	中断或陷阱

图 8-9

x86-64 系统中的异常示例
时间到时 \Rightarrow 进程并发切换

操作系统定义 interrupt trap

- 中断：外部 I/O 设备
- 陷阱（同步）：故意引发的异常，目的是进行系统调用
- 故障（同步）：出错了，有可能修复。例：**缺页**、**除法错误**、**一般保护故障（段错误）**
- 终止（同步）：致命错误，无法恢复。例：DRAM/SRAM 损坏

系统调用

System Call

在 x86-64 系统上，系统调用是通过一条称为 `syscall` 的陷阱指令来提供的。

所有 Linux 系统调用的参数都是通过 **通用寄存器** 而不是栈传递的。具体如下：

- 系统调用号：寄存器 `%rax`，**每个系统调用有唯一的整数号**
- 参数寄存器：`%rdi` `%rsi` `%rdx` `%r10` `%r8` `%r9`

(注意和过程调用有出入，第 4 个参数是 `%r10`，而不是过程调用中的 `%rcx`，回顾)

从系统调用返回时，寄存器 `%rcx` 和 `%r11` 都会被破坏，`%rax` 包含返回值 `errno`。

`errno` : Error Number, 错误码，全局变量，存储最近一次系统调用失败的原因。

返回值在 -4095 到 -1 之间的负数表示发生了错误，对应于负的 `errno`。

系统调用

System Call

编号	名字	描述	编号	名字	描述
0	read	读文件	33	pause	挂起进程直到信号到达
1	write	写文件	37	alarm	调度告警信号的传送
2	open	打开文件	39	getpid	获得进程 ID
3	close	关闭文件	57	fork	创建进程
4	stat	获得文件信息	59	execve	执行一个程序
9	mmap	将内存页映射到文件	60	_exit	终止进程
12	brk	重置堆顶	61	wait4	等待一个进程终止
32	dup2	复制文件描述符	62	kill	发送信号到一个进程

图 8-10 Linux x86-64 系统中常用的系统调用示例

```
int main()
{
    // 写入 "hello, world\n"
    write(1, "hello, world\n", 13);
    // 退出程序, 返回代码为 0
    _exit(0);
}
```

```
.section .data
string:
    .ascii "hello, world\n"      // 字符串 "hello, world\n"
string_end:
    .equ len, string_end - string // 计算字符串长度
.section .text
.globl main
main:
```

进程

Process

假象：好像我们在跑的程序是系统中唯一运行的程序，**独占 CPU 和内存**（然而 `top` 一下很容易打假）

1. **独立的逻辑控制流**：好像我们的程序独占地使用处理器（实际上由 **上下文切换** 的机制实现）
2. **私有的地址空间**：好像我们的程序独占地使用内存（实际上由 **虚拟内存** 机制实现）

进程 (Process)：一个 **执行中程序的实例**，系统中的每个程序都运行在某个进程的上下文中。

上下文 (Context)：直观理解就是程序运行时所需的各种状态信息。

- 程序的代码和数据
- 栈
- 通用目的寄存器的内容
- 程序计数器
- 环境变量
- 打开文件描述符的集合

并发流

Concurrent Flow

并发流: 一个逻辑流的执行在时间上与另一个流重叠。

也即: A 的开始 ~ A 的结束 与 B 的开始 ~ B 的结束 在时间上重叠。

右图中, 从 A 到 B, 发生了 **抢占 (Preemption)**、**中断 (Interrupt)**、**上下文切换 (Context Switch)**

鉴别

并行: 同一时刻, 多个进程在不同核上运行, 并行是并发的真子集。

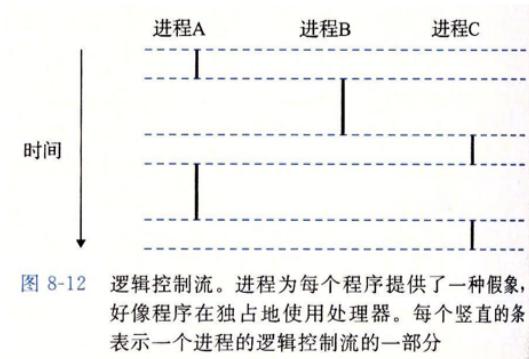


图 8-12 逻辑控制流。进程为每个程序提供了一种假象,好像程序在独占地使用处理器。每个竖直的条表示一个进程的逻辑控制流的一部分

私有地址空间

Private Address Space

- 用户区**: 地址 $0x400000$ (2^{22}) $\sim 2^{48} - 1$
- 内核区**: 地址 $\geq 2^{48}$

回顾

实际上，并没有真的分配这么多（假象，用到的很稀疏），而是通过**虚拟内存映射**实现的。

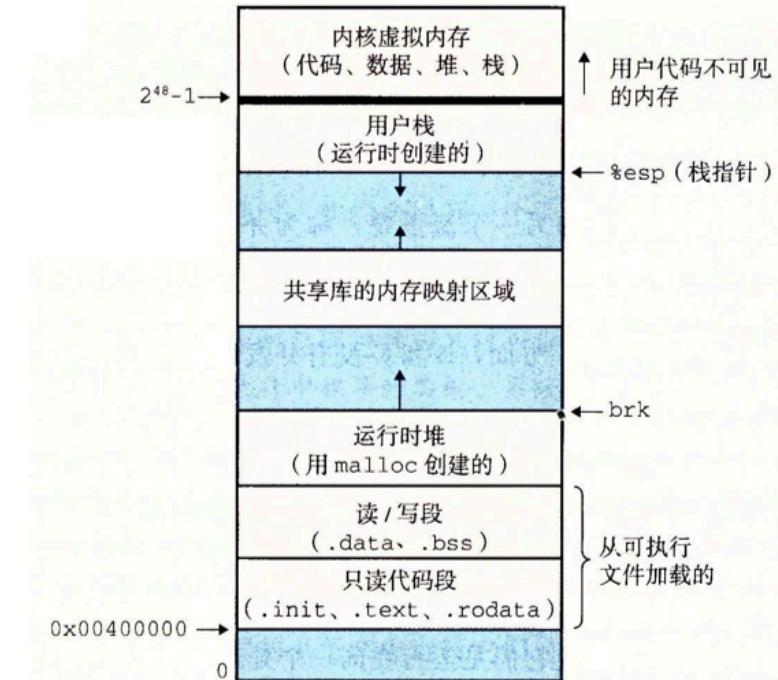


图 8-13 进程地址空间

用户模式 vs 内核模式

User Mode vs Kernel Mode

比较项	用户模式	内核模式
模式位	0	1
权限	受限	不受限
访存	仅限用户区 *	任意地址
特权指令 **	不能执行	可以执行

* 直接引用地址空间中内核区内的代码和数据会导致保护故障 (Abort) , 但是可以通过 `/proc` 文件系统访问一部分内核数据结构的内容。

** 特权指令：修改模式位、**执行 I/O 操作**、改变内存中的指令流。

上下文切换

Context Switch

定义：上下文切换是内核 **重新启动一个被抢占的进程** 所需的 **进程状态** (context) 的转换。

1. 保存当前进程的上下文
2. 恢复下一个进程的上下文
3. 将控制权转交给新进程

上下文：用户栈、状态寄存器、内核栈和各种内核数据结构（内存结构的页表、进程表、已打开文件的文件表）

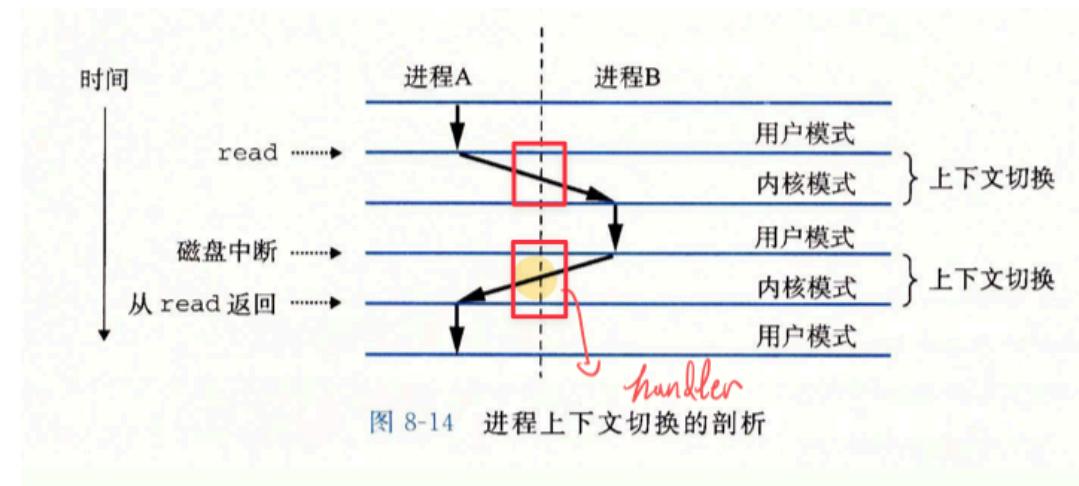
调度：内核决定抢占当前进程，并决定哪个进程来重新开始。

例子

- **DMA传输：**进程切换等待磁盘数据传输
- **无阻塞系统调用：**内核决定执行上下文切换，而不是返回用户态
- **中断：**周期性定时器中断 (1ms / 10ms)

上下文切换

Context Switch



1. **磁盘中断**: 进程 A 在执行 `read` 操作时, 由于 `read` 是特权指令, 系统调用陷入内核态 (陷阱)
2. **上下文切换**: 内核处理系统调用, 知道要等很久 (DMA 直接内存访问), 于是决定不返回到 A, 而是切换到 B (抢占、调度)
3. **中断处理程序 (handler)** : B 正在运行, 来了磁盘中断, 告知内核数据已经拿到了, 于是进程 B 需要处理中断, 进入中断处理程序 (内核态)
4. **调度**: 中断处理过程中, 内核知道 A 数据等到了, 于是决定切换回 A
5. **继续执行**: 控制流回到 A, A 继续执行

系统调用错误处理

System Call Error Handling

当 Unix 系统级函数遇到错误时，它们通常会返回 -1，并设置全局整数变量 errno 来表示什么出错了。

```
if ((pid = fork()) < 0) {
    fprintf(stderr, "fork error: %s\n", strerror(errno)); // fprintf 输出到标准错误流, strerror 返回错误描述的文本串
    exit(0);
}
```

继续包装，以首字母是否大写指示是否是包装过的函数：

```
void unix_error(char *msg) { // unix 风格的错误处理
    // errno 是全局变量，不需要传参
    fprintf(stderr, "%s: %s\n", msg, strerror(errno));
    exit(0);
}
if ((pid = fork()) < 0) {
    unix_error("fork error");
}
```

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

进程控制 - 进程 ID

Process ID

每个进程都有 **唯一的正数进程 ID (PID)**

试试在 clab 上执行 `ps -ef | head -n 5 !` (`ps` 是 process status, `-e` 是所有进程, `-f` 是完整格式)

- `getpid` (get process ID) 返回调用进程的 PID
- `getppid` (get parent process ID) 返回它的父进程的 PID

两个函数返回类型为 `pid_t` 的整数值, 在 Linux 系统上它在 `types.h` 中被定义为 `int`。

```
#include <sys/types.h> // 定义 pid_t
#include <unistd.h>    // 定义 getpid 和 getppid
pid_t getpid(void);
pid_t getppid(void);
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	Nov08 ?		00:01:28	/sbin/init
root	2	0	0	Nov08 ?		00:00:00	[kthreadd]
root	3	2	0	Nov08 ?		00:00:00	[pool_workqueue_release]
root	4	2	0	Nov08 ?		00:00:00	[kworker/R-rcu_g]

进程控制 - 创建 / 终止

Process Creation / Termination

同一时刻，操作系统中有若干个进程，每个进程都属于三种状态之一：

1. 运行

- 同一时间可以有若干个进程同时运行
- 运行 ≠ 正在 CPU 上执行（调度机制，可能是在等待调度的队列中）

2. 停止

- 进程被挂起，且 **不会** 进入等待调度的队列
- 收到以下 4 种信号会导致进程停止：
 - SIGSTOP (Signal Stop)
 - SIGTSTP (Signal Terminal Stop)
 - SIGTTIN (Signal Terminal/TTY Input for Background Process)
 - SIGTTOU (Signal Terminal/TTY Output for Background Process)
- 收到 SIGCONT (Signal Continue) 后被转为运行状态

3. 终止

- 进程永不运行，不能被转为运行状态
- 可能原因：
 1. 收到相关信号
 2. 从主程序返回，返回的整数值会被设为进程的退出状态，非 0 表示异常退出
 3. 调用 `exit` 函数，`int exit(int status)`
- 一个进程终止后必须被回收，否则会变成僵尸进程 (Zombie Process)

一些说明：

1. TTY 是 Teletypewriter 的缩写，是电传打字机，是早期的计算机外设，用于连接计算机和终端，现在一般指终端。
 2. 在后台的进程若想从终端读取输入/写入输出时，进程会停止，直到他们转为前台进程。这是为了确保终端的输入只被前台进程使用。
- This slide is cited from Arthals.

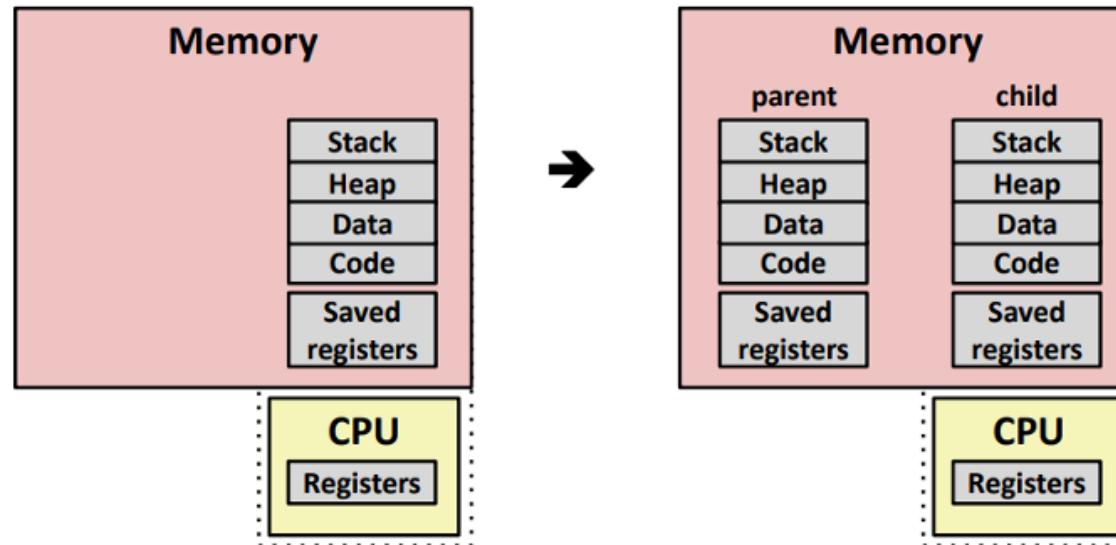
进程控制 - 分叉

Process Fork

`fork` 函数创建一个新进程，新进程是调用进程的副本。

原进程称为 **父进程**，新的进程称为 **子进程**。

此时两个进程完全相同：相同但独立的地址空间，堆栈，变量值，代码，打开的文件：



进程控制 - 分叉

Process Fork

父进程可以调用 `fork` 函数创建新的子进程，它们是 **并发的独立进程**。

`fork` 函数调用一次，返回 2 次：

- 一次返回在父进程中，返回值为 **子进程的 PID**
- 一次返回在子进程中，返回值为 **0**

因为子进程 PID 总非零，可以以此区别父子进程。

父子进程最大的区别就是 PID 不同。

```
int main(){
    pid_t pid;
    int x = 1;
    pid = Fork();
    if(pid == 0){
        /*Child */
        printf("child: x=%d\n", ++x);
        exit(0); // 子进程退出，不会继续执行后续代码
    }
    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0); // 父进程退出
}
```

```
pid = Fork();
if (pid == 0){ // 子进程
    // do something
} else { // 父进程
    // do something
}
```

进程控制 - 分叉

Process Fork

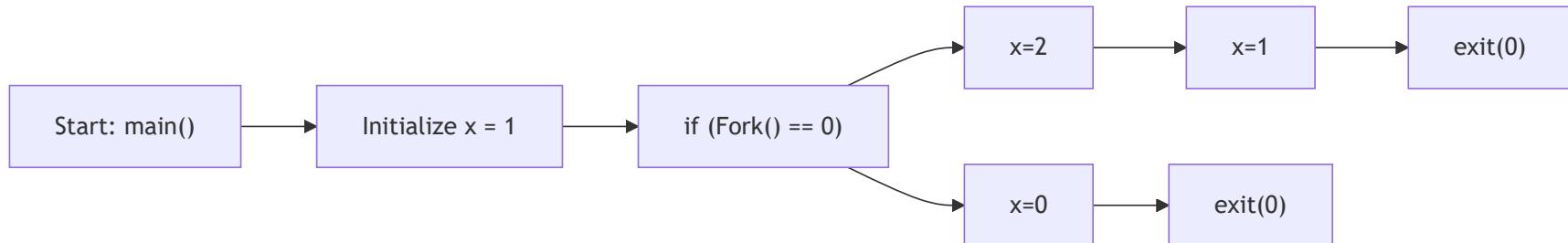
```
int main()
{
    int x = 1;

    if (Fork() == 0)
        printf("p1: x=%d\n", ++x);
    printf("p2: x=%d\n", --x);
    exit(0);
}
```

考点: `printf` 有缓冲区, 且 `fork` 后, 子进程会具有父进程缓冲区的副本, 但是后续再写入缓冲区时, 父子进程彼此独立。

清空缓冲区:

- `fflush(stdout)` `scanf()`
- `printf` 遇到换行符 `\n`、回车符 `\r` 会清空缓冲区
- 进程退出时会清空缓冲区



拓扑排序: 不违反这张图中的箭头方向。

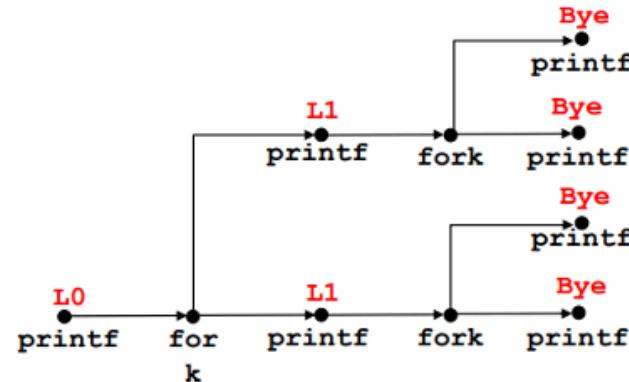
This slide is cited from Arthals.

进程控制 - 分叉

Process Fork

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

forks.c



Feasible output:

L0	L0
L1	Bye
Bye	L1
Bye	Bye
L1	L1
Bye	Bye
Bye	Bye

Infeasible output:

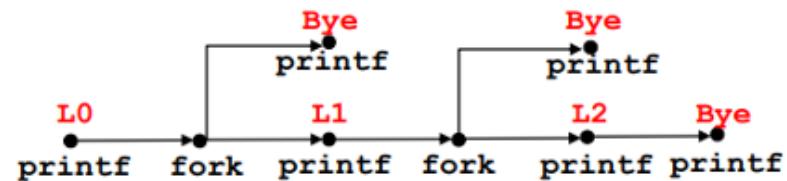
Bye
Bye
Bye

进程控制 - 分叉

Process Fork

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
        }
    }
    printf("Bye\n");
}
```

forks.c



Feasible output:

- L0
- L1
- Bye
- Bye
- L2
- Bye

Infeasible output:

- L0
- Bye
- L1
- Bye
- L2

进程控制 - 回收子进程

Process Reaping

一个进程终止后必须被其父进程回收。

如果父进程已终止，则安排 `init` 进程作为养父。

`init` 进程: `pid=1`, 系统启动由内核创建的第一个进程，所有的进程都是它衍生出来的。

```
#include <sys/wait.h>
int waitpid(pid_t pid, int *statusp, int options);
```

`waitpid()` 函数: 父进程调用 `waitpid()` 等待其子进程终止。

- `pid` : 表明等待集合包含哪个/哪些子进程
- `statusp` : status pointer, 若放入一个指针，则 `waitpid` 返回后会把子进程的相关状态信息存储在该指针指向的位置 *
- `options` : 修改等待的具体行为

* C 中有很多类似的函数，不通过返回值传递结果，而是通过传递指针参数，然后再在过程中将结果写入指针指向的内存。要习惯这种用法。

waitpid 函数详解

waitpid() Function

等待集合 pid

- pid > 0 : 等待集合包含进程 ID 为 pid 的子进程
- pid = -1 : 等待集合包含父进程的所有子进程

waitpid 函数详解

waitpid() Function

选项 options

options 是基于位向量实现的，所以可以使用 | 来组合多个选项。 $0001 \mid 0010 = 0011$

- WNOHANG : wait no hang, 如果等待集合中没有子进程 终止，则 立即返回 0
- WUNTRACED : wait untraced, 挂起 调用进程，等待集合中任一子进程 终止或停止
- WCONTINUED : wait continued, 挂起 调用进程，等待集合中任一子进程 继续

默认行为： waitpid(-1, NULL, 0) , 挂起 调用进程，等待集合中任一子进程 终止

组合例子： WNOHANG | WUNTRACED :

1. 首先必然立即返回 WNOHANG
2. 其次对于子进程的要求是终止或停止 WUNTRACED
3. 所以，若无子进程满足条件，则返回 0，否则返回子进程的 PID

waitpid 函数详解

waitpid() Function

状态信息 statusp

statusp 允许留空，若非空则要求其是一个指针，指向一个整数。

waitpid 返回后会把子进程的相关状态信息存储在该指针指向的内存中。

你可以用宏来解析 statusp 指向的内存中的状态信息。

- WIFEXITED(status) : 如果子进程通过调用 exit 或者一个返回 (return) 正常终止，就返回真。
- WEXITSTATUS(status) : 返回一个 **正常终止** 的子进程的退出状态。只有在 WIFEXITED() 返回为真时，才定义这个状态。
- WIFSIGNALED(status) : 如果子进程是因为一个 **未被捕获的信号** 终止的，那么就返回真。
- WTERMSIG(status) : 返回 **导致子进程终止** 的信号的编号。只有在 WIFSIGNALED() 返回为真时，才定义这个状态。
- WIFSTOPPED(status) : 如果引起返回的子进程当前是 **停止** 的，那么就返回真。
- WSTOPSIG(status) : 返回引起子进程 **停止** 的信号的编号。只有在 WIFSTOPPED() 返回为真时，才定义这个状态。
- WIFCONTINUED(status) : 如果子进程 **收到 SIGCONT 信号重新启动**，则返回真。

waitpid 函数详解

waitpid() Function

如果调用进程没有子进程，那么 waitpid 返回 -1，并且设置 errno 为 ECHILD (Error Child)。

如果 waitpid 函数被一个信号中断，那么它返回 -1，并设置 errno 为 EINTR (Error Interrupt)。

wait()

简化版本的 waitpid。

```
pid_t wait(int *statusp);
```

只接受一个参数，等价于 waitpid(-1, statusp, 0)

进程控制 - 回收子进程

Process Reaping

回收的乱序性

程序不会按照特定的顺序回收子进程。

如何顺序回收：指定 `waitpid` 的 `pid` 参数。

```

int status, i;
pid_t pid;

/* 父进程创建 N 个子进程 */
for (i = 0; i < N; i++)
    if ((pid = Fork()) == 0) /* 子进程 */
        exit(100 + i);

/* 父进程以任意顺序回收 N 个子进程 */
while ((pid = waitpid(-1, &status, 0)) > 0) {
    if (WIFEXITED(status))
        printf("子进程 %d 正常终止, 退出状态=%d\n", pid, WEXITSTATUS(status));
    else
        printf("子进程 %d 异常终止\n", pid);
}

/* 唯一的正常终止是没有更多的子进程 */
if (errno != ECHILD) /* EINTR */
    unix_error("waitpid 错误");

exit(0);
}

```

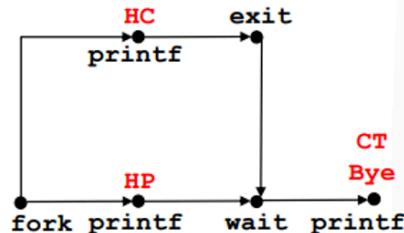
进程控制 - waitpid 函数

`waitpid` Function

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c



Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

注意：默认行为下，使用 `waitpid` `wait` 会 **挂起** 调用进程，直到子进程终止。

这会使得拓扑排序的可能性受限。

进程控制 - 休眠

sleep & pause

sleep : 将一个进程挂起一段指定的时间。

```
#include <unistd.h>

unsigned int sleep(unsigned int secs);
```

返回：还要休眠的秒数。

- 如果请求的时间量已经到了， sleep 返回 0
- 否则返回还剩下的要休眠的秒数（当 sleep 函数被一个信号中断而提前返回）

pause : 让调用进程休眠，直到该进程收到一个信号。

```
#include <unistd.h>

int pause(void);
```

总是返回 -1 。

进程控制 - 新建进程

execve Function

execve : execute vector environment, 在当前进程的上下文中加载并运行一个新程序。

```
#include <unistd.h>
int execve(const char *filename, const char *argv[], const char *envp[]);
```

- filename : 执行的目标文件名
- argv : 参数列表数组，每个指针指向一个参数字符串
- envp : 环境变量数组，每个指针指向一个环境变量字符串

返回值：

- 成功：不返回
- 失败：返回 -1，并设置 errno

如果找不到 filename，函数返回到调用程序，否则函数调用一次并且从不返回。

参数列表和环境变量

Argument & Environment

对于一个指令：

```
LD_PRELOAD=/usr/lib/libkdebug.so ls -l /usr/include
```

参数列表

参数：传递给新程序的参数。

```
argv[0] -> "ls" // 可执行文件名  
argv[1] -> "-l" // 参数 1  
argv[2] -> "/usr/include" // 参数 2  
argv[3] -> NULL
```

环境变量

环境变量：key=value 的键值对。

```
envp[0] -> "LD_PRELOAD=/usr/lib/libkdebug.so"  
envp[1] -> NULL
```

进程控制 - 新建进程

execve Function

调用 `execve` 后，程序会执行新的主函数，格式如下：

```
int main(int argc, char **argv, char **envp); // 注意，是 char ** 指针
```

- `argc`：参数个数，argument count
- `argv`：参数列表指针数组，argument vector
- `envp`：环境变量数组指针，environments array pointer

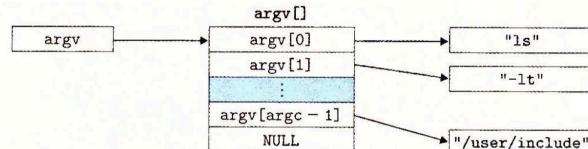


图 8-20 参数列表的组织结构

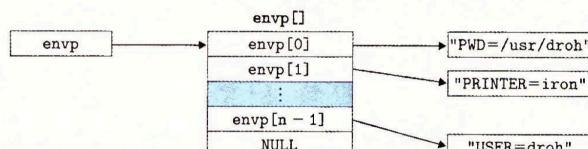


图 8-21 环境变量列表的组织结构

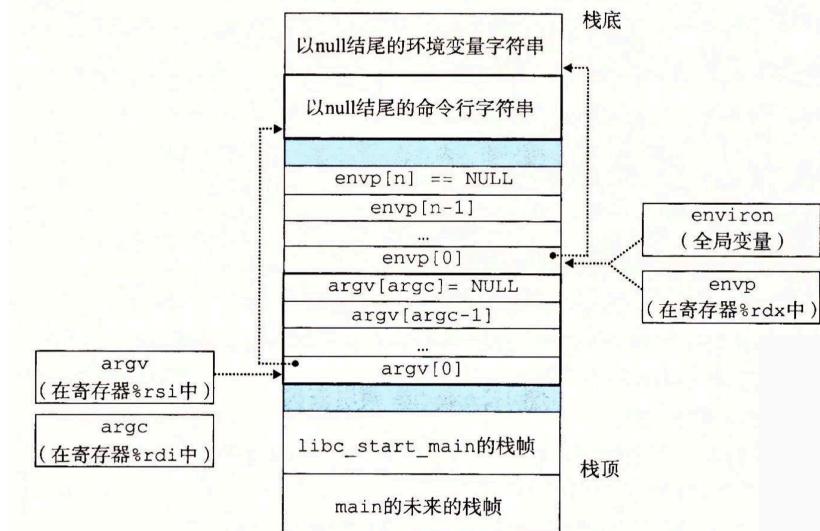


图 8-22 一个新程序开始时，用户栈的典型组织结构

This slide is cited from Arthals.

信号

Signals

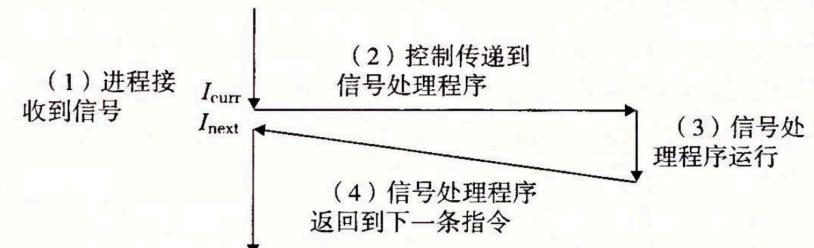
信号：是一种用于通知进程（运行中的程序）发生了某些事件的机制。（书上定义：一条小消息）

你可以把信号想象成一种“提醒”或“通知”，它会告诉进程某些事情发生了，需要做出反应。

- 发送者：内核（检测到事件）/ 进程（调用 `kill` 函数）
- 接收者：进程（行为：忽略、终止、捕获并调用信号处理函数）

当你按下键盘上的 `Ctrl+C` 组合键时，操作系统会发送一个特定的信号（通常是 `SIGINT` 信号）给正在运行的程序，通知它应该停止运行。

程序可以选择如何处理这个信号，比如立即停止，或者进行某些清理工作后再停止。



调用结束后，若返回控制流给进程，则继续运行 I_{next}

常见信号

Common Signals

信号	信号全称	默认行为	相关事件
SIGINT	Interrupt Signal	终止	来自键盘的中断 (Ctrl-C)
SIGILL	Illegal Instruction Signal	终止并转储内存	非法指令
SIGFPE	Floating Point Exception	终止并转储内存	浮点异常
SIGKILL	Kill Signal	终止	杀死进程
SIGSEGV	Segmentation Fault Signal	终止并转储内存	无效的内存引用 (段故障)
SIGUSR1	User-defined Signal 1	终止	用户定义的信号 1
SIGUSR2	User-defined Signal 2	终止	用户定义的信号 2

常见信号

Common Signals

信号	信号全称	默认行为	相关事件
SIGALRM	Alarm Clock Signal	终止	来自 <code>alarm</code> 函数的定时器信号
SIGCHLD	Child Status Changed Signal	忽略	一个子进程停止或终止
SIGCONT	Continue Signal	继续执行	继续进程，如果该进程停止
SIGSTOP	Stop Signal	停止直到下一个SIGCONT	不是来自终端的停止信号
SIGTSTP	Terminal Stop Signal	停止直到下一个SIGCONT	来自终端的停止信号 (Ctrl-Z)

重点： `SIGKILL` 和 `SIGSTOP` 无法被捕获、忽略。

这意味着它们的行为是由操作系统内核直接处理的，不依赖于用户空间的代码（反例：`SIGINT` `SIGTSTP`）。

这确保了系统管理员和操作系统能够在必要时强制控制进程的状态，而不受进程本身的干扰。

待处理信号

Pending Signals

待处理信号：进程在排队等候运行时，发送给它的信号不能马上被处理。

实现：名为 `pending` 的位向量（每个进程都有独属于自己的一个）

- 当信号被传送到进程，那么 `pending` 位向量中对应位置的值会被置为 1
- 当信号在对应进程得到接收时，`pending` 位向量中对应位置的值会被重置为 0

所以，**进程只能知道“自己收到过某类信号”，但不能知道总共收到了几次**

1. (单核) 操作系统中，多个进程轮流运行在 CPU 上
2. 如果进程 a 正在 CPU 上运行，则马上能收到信号并处理
3. 如果进程 a 在排队等候运行，则发给 a 的信号不能马上被处理，那么这个信号就称之为待处理信号，会保存在进程 a 的一个叫做 `pending` 的位向量 中
4. 位向量：每一位对应一个信号，只有 0 和 1 两种状态，0 表示未收到此信号，1 表示收到此信号
5. 轮到 a 在 CPU 上运行时，它才能开始处理这些待处理信号
6. 但是，这时可能 a 已经收到了一堆待处理信号

阻塞信号

Blocked Signals

阻塞信号: 进程可以阻塞某些信号，使其不能被处理。

实现: 名为 `blocked` 的位向量 (每个进程都有独属于自己的一个)

进程可以有选择性地阻塞接收某种信号。

当一种信号被阻塞时，它仍可以被发送，**但是产生的待处理信号不会被接收，直到进程取消对这种信号的阻塞。**

进程和进程组

Process & Process Group

进程组 (process group) : 一个或多个进程的集合，它们共享一个共同的进程组 ID。

进程可以通过 `setpgid` 函数改变 **自己 / 其他** 进程的进程组。

作业

Jobs

作业 (job) : 一个或多个进程的集合，通常由一个前台进程和若干后台进程组成，通常由 shell 创建和管理。

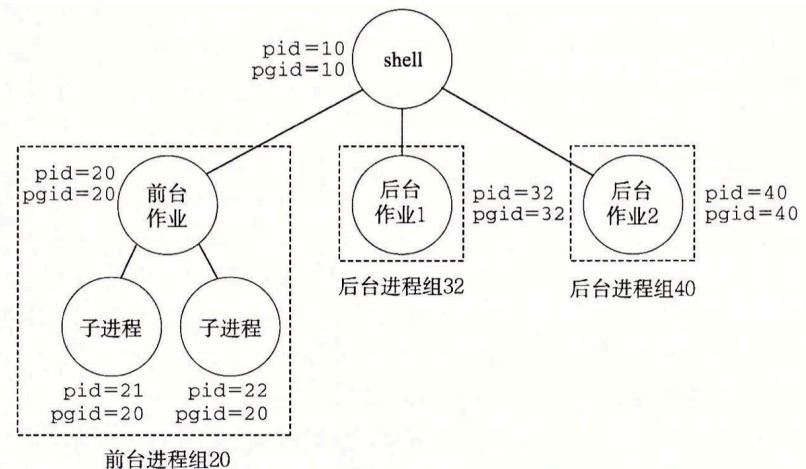
一个作业可以包含一个单独的命令或一组通过管道连接的命令，作业可以在前台运行，也可以在后台运行。

前台 (foreground) : 前台作业是当前在终端中运行的作业。

- 前台作业会占用终端，并且可以从用户那里接受输入。
- 一个后台作业通过 `fg` 命令转换为前台作业。

后台 (background) : 后台作业是在终端中启动但不占用终端的作业。

- 后台作业可以在不与用户交互的情况下运行，用户可以继续在终端中执行其他命令。
- 后台作业通常通过在命令后面加上 `&` 符号来启动，也可以通过 `bg` 命令来启动。



发送信号

Send Signals

/bin/kill 程序

```
/bin/kill -signal pid
/bin/kill -9 15213
```

发送信号 SIGKILL 给进程 15213，杀死它

kill 函数

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

- 如果 `pid > 0`，则将信号 `sig` 发送给进程 ID 为 `pid` 的进程
- 如果 `pid == 0`，则将信号 `sig` 发送给与调用 `kill` 的进程属于同一个进程组的所有进程
- 如果 `pid == -1`，则将信号 `sig` 发送进程组 `|pid|` 内的所有进程

返回值：

- 成功：0
- 失败：-1

发送信号

Send Signals

从键盘发送信号

发送信号到 **前台进程组** 中的每个进程

Ctrl+C 发送 `SIGINT` 信号，终止前台进程

Ctrl+Z 发送 `SIGTSTP` 信号，挂起前台进程

Ctrl+D 发送 `EOF` 信号，指示输入结束

alarm 函数

设置一个定时器，在指定的秒数后发送 `SIGNALRM` 信号给当前进程

```
#include <unistd.h>
unsigned int alarm(unsigned int secs);
```

对 `alarm` 的调用都将取消任何待处理的 (pending) 闹钟。

返回值：

- 如果之前有定时器，返回剩余时间
- 如果之前没有定时器，返回 0

接收信号

Receive Signals

接收时刻：内核把进程 p 从 **内核模式** 切换到 **用户模式** 时，在执行代码前处理信号。

接收信号类型：`pending & ~blocked`，若非空，强制进程接收其中之一。

进程会采取前文所述三种行为之一（忽略、终止、捕获并调用信号处理函数），处理完毕后，继续选择其他信号接收，直到集合为空后，才将控制转移回进程 p 的逻辑控制流的下一条指令 I_{next} 。

接收信号

Receive Signals

信号的默认行为 回顾：

- 进程终止，如 SIGINT SIGKILL
- 进程终止并转储内存，如 SIGILL SIGFPE SIGSEGV
- 进程停止（挂起），直到被 SIGCONT 信号重新启动，如 SIGSTOP SIGTSTP
- 忽略该信号，如 SIGCHLD

signal 函数

signal Function

我们可以使用 signal 函数修改信号的处理行为：

```
#include <signal.h>
typedef void (*sighandler_t)(int); // 函数指针类型

sighandler_t signal(int signum, sighandler_t handler);
```

参数 handler 可以有三种情况（前两种都是宏）：

- SIG_IGN : 忽略信号
- SIG_DFL : 采用默认行为
- 用户自定义的函数地址

信号处理函数

Signal Handler

```
// Bomblab/support.c
#include <signal.h>
static void sig_handler(int sig)
{
    printf("So you think you can stop the bomb with ctrl-c, do you?\n");
    sleep(3);
    printf("Well... ");
    fflush(stdout);
    sleep(1);
    printf("OK. :-)\n");
    exit(16);
}
signal(SIGINT, sig_handler);
```

这个函数会做什么？

信号处理函数

Signal Handler

信号处理程序可以被其他信号处理程序中断

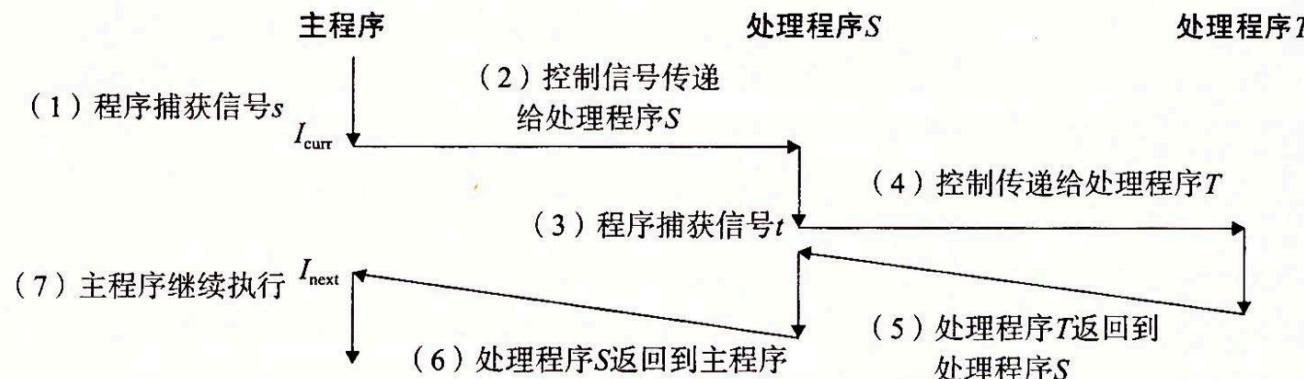


图 8-31 信号处理程序可以被其他信号处理程序中断

信号处理程序是用户态的一部分，当信号处理程序捕获信号 t 并处理时，它是在用户态运行的。

阻塞 / 解除阻塞信号

Blocked / Unblocked Signals

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信号。

显式阻塞机制：用 `sigprocmask` 函数和它的辅助函数。

阻塞 ≠ 丢弃，阻塞只是暂时不处理。

但是，若一个信号来了很多次，且已有同类型信号阻塞，则该信号会被丢弃（一直在做或运算）。

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); // 改变当前阻塞的信号集合

// 以下操作 sigset_t 的函数本质都是对位向量进行操作
int sigemptyset(sigset_t *set); // 初始化信号集合为空
int sigfillset(sigset_t *set); // 将所有信号添加到信号集合中
int sigaddset(sigset_t *set, int signum); // 将指定信号添加到信号集合中
int sigdelset(sigset_t *set, int signum); // 从信号集合中删除指定信号

int sigismember(const sigset_t *set, int signum); // 返回：若 signum 是 set 的成员则为 1，否则为 0。
```

sigprocmask 函数

sigprocmask Function

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); // 改变当前阻塞的信号集合
```

sigprocmask : 改变当前阻塞的信号集合。具体行为依赖于 how 的值：

- SIG_BLOCK : 把 set 中的信号添加到 blocked 中 ($\text{blocked} = \text{blocked} | \text{set}$)。
- SIG_UNBLOCK : 从 blocked 中删除 set 中的信号 ($\text{blocked} = \text{blocked} & \sim\text{set}$)。
- SIG_SETMASK : $\text{blocked} = \text{set}$ 。

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

信号处理函数

Signal Handler

原则：

1. 处理程序尽可能简单 因为信号可以在程序执行的任何时候异步地发生
2. 只调用 **异步信号安全 函数** 可重入 / 不能被信号处理程序中断，本质就是保证不干扰正在处理的程序，不然可能会导致死锁
3. 保存恢复 `errno`
4. 访问全局数据结构时阻塞所有信号 避免死锁
5. 全局变量使用 `volatile` 声明 避免错误的编译器优化，告诉编译器不要缓存此变量
6. 标志使用 `sig_atomic_t` 声明 保证读写操作的原子性，即读写不可被中断
7. 不可以用信号来对其他进程中发生的事件计数 因为阻塞机制的存在

信号处理函数

Signal Handler

异步信号安全函数

PRO

_exit fork

sigset sigemptyset sigfillset sigaddset
sigdelset

sigprocmask

sleep

read write

wait waitpid

非异步信号安全函数

CON

malloc

printf scanf fprintf fscanf

exit

为什么 printf 不行?

- 不可重入，有缓冲区
- 考虑信号处理函数序列

$\text{handler}_1 \rightarrow \text{handler}_2 \rightarrow \text{handler}_1$

信号处理函数

Signal Handler

以下两段代码都是 SIGCHLD 信号的信号处理函数。

```
void handler1(int sig)
{
    int olderrno = errno;

    if ((waitpid(-1, NULL, 0)) < 0)
        Sio_error("waitpid error");
    Sio_puts("Handler reaped child\n");
    Sleep(1);
    errno = olderrno;
}
```

```
void handler2(int sig)
{
    int olderrno = errno;

    while (waitpid(-1, NULL, 0) > 0) {
        Sio_puts("Handler reaped child\n");
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    Sleep(1);
    errno = olderrno;
}
```

waitpid 每次只回收一个子进程，但是存在 SIGCHLD 信号不代表只有一个子进程终止。

存在一个信号代表至少有一个信号到达。

竞争

Race Condition

错误代码

```

while (1) {
    if ((pid = Fork()) == 0) {
        Execve("/bin/date", argv, NULL);
    }
    // 关注下一行
    Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
    addjob(pid);
    Sigprocmask(SIG_SETMASK, &prev_all, NULL);
}
exit(0);

```

正确代码

```

while (1) {
    // 先阻塞 SIGCHLD
    Sigprocmask(SIG_BLOCK, &mask_one, &prev_one);
    if ((pid = Fork()) == 0) {
        // 分出子进程后解除阻塞
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);
        Execve("/bin/date", argv, NULL);
    }
    Sigprocmask(SIG_BLOCK, &mask_all, NULL);
    addjob(pid);
    Sigprocmask(SIG_SETMASK, &prev_one, NULL);
}
exit(0);

```

一旦分出子进程，则父子进程就会并发，顺序就不一定和代码顺序一致（只要满足拓扑排序即可）。

这就导致了竞争。

显式等待信号

Explicit Signal Waits

```
#include "csapp.h"
volatile sig_atomic_t pid; // 定义一个易失性的原子类型变量 pid
void sigchld_handler(int s) // 定义一个处理 SIGCHLD 信号的处理函数
{
    int olderrno = errno; // 保存当前的 errno 值
    pid = waitpid(-1, NULL, 0); // 等待任意子进程结束，并将其 pid 保存到全局变量 pid 中
    errno = olderrno; // 恢复之前的 errno 值
}
void sigint_handler(int s){};

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

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask, &prev); /* 阻塞 SIGCHLD */
        if (Fork() == 0) /* 子进程 */
            /* 子进程代码 */
    }
}
```

显式等待信号

Explicit Signal Waits

```
/* 等待接收 SIGCHLD 信号（可能引发竞争条件） */
while (!pid) /* 竞争！ */
    pause();
```

这段代码中，若 `SIGCHLD` 信号发生在 `while` 测试之后，`pause` 之前，则永远不会再次唤醒。

```
/* 等待接收 SIGCHLD 信号（速度太慢） */
while (!pid) /* 太慢！ */
    sleep(1);
```

这段代码中，`sleep` 正确，但是间隔不好设置。

sigsuspend 函数

sigsuspend Function

```
int sigsuspend(const sigset_t *mask); // 函数声明：等待信号到来，并临时替换信号掩码
```

等价于原子化版本的：

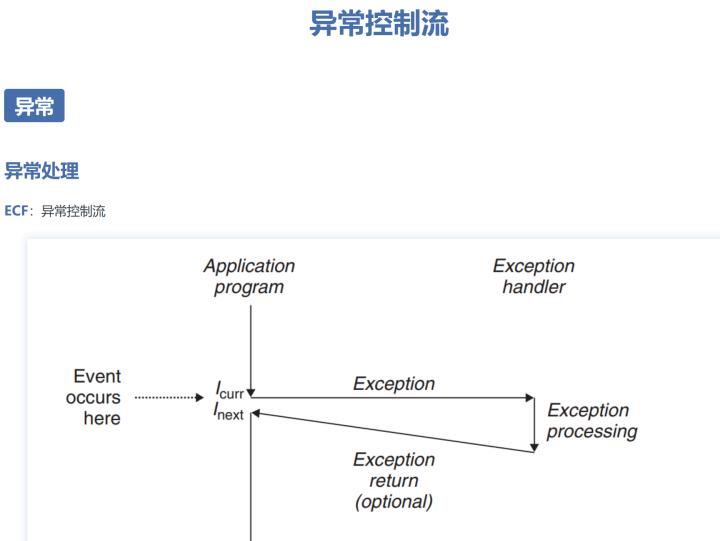
```
sigprocmask(SIG_SETMASK, &mask, &prev); // 设置新的信号掩码，保存旧的信号掩码到 prev  
pause(); // 暂停进程，直到接收到信号  
sigprocmask(SIG_SETMASK, &prev, NULL); // 恢复之前的信号掩码
```

Emphasis

Notes

■ Note Link

- 异常控制流
- 异常
- 异常处理
- 异常类别
- X86-64实例
- 进程
- 进程抽象
- 上下文切换
- 进程控制
- 进程状态
- 创建和终止进程
- 回收子进程
- 加载并运行程序
- 其它
- 进程/进程组ID
- 进程休眠
- 进程终止
- 信号
- 发送信号
- 接收信号
- 阻塞信号
- 等待信号
- 信号处理程序：并发原则
- 非本地跳转



异常表: 系统启动时分配的跳转表 (唯一非负整数异常号)

异常表基址寄存器: 特殊的CPU寄存器, 存放异常表的起始地址

Homework Review

HW7

学号	HW-分数	HW-问题
2300012932	10	
2300012935	10	
2300012950	10	
2300012951	10	
2300013083	10	
2300013115	10	
2300013158	10	
2300013201	10	
2300013222	10	
2300013230	10	
2300013272	10	
2300017788	10	
2300094610	10	

Exercises

E1

10. 当一个网络数据包到达一台主机时，会触发以下哪种异常：
- A. 系统调用
 - B. 信号
 - C. 中断
 - D. 缺页异常

E1

10. 当一个网络数据包到达一台主机时，会触发以下哪种异常：

- A. 系统调用
- B. 信号
- C. 中断
- D. 缺页异常

10. C。送分题，外部 I/O 会触发中断，CPU 执行完当前指令之后可能会去处理该中断。

E2

12. 以下关于缺页异常和系统调用的描述，不正确的是：
- A. 两个异常都是同步异常
 - B. 两个异常的触发都是由于执行某一条指令
 - C. 两个异常在正常退出时，都需要判断是否有非阻塞的待处理信号
 - D. 两个异常在正常退出后，都需要重新执行触发异常的指令

E2

12. 以下关于缺页异常和系统调用的描述，不正确的是：
- A. 两个异常都是同步异常
 - B. 两个异常的触发都是由于执行某一条指令
 - C. 两个异常在正常退出时，都需要判断是否有非阻塞的待处理信号
 - D. 两个异常在正常退出后，都需要重新执行触发异常的指令
12. D。缺页是故障，这种故障需要重新执行引发故障的指令；系统调用是陷阱。二者都是同步的异常，而且处理完毕是从内核态回到用户态，会处理信号。

E3

13. 对于 Linux 系统，下列说法错误的是：
- A. 在单核 CPU 上，所有看似并行的逻辑流实际上是并发的。
 - B. 用户模式不可访问/proc 文件系统中包含的内核数据结构的内容。
 - C. 在 execve 函数参数的 argv 数组加入”> 1.txt”，不能自动实现 IO 重定向。
 - D. 即便在信号处理程序中调用 printf 前阻塞所有信号，也不一定安全。

E3

13. 对于 Linux 系统，下列说法错误的是：
- A. 在单核 CPU 上，所有看似并行的逻辑流实际上是并发的。
 - B. 用户模式不可访问/proc 文件系统中包含的内核数据结构的内容。
 - C. 在 execve 函数参数的 argv 数组加入”> 1.txt”，不能自动实现 IO 重定向。
 - D. 即便在信号处理程序中调用 printf 前阻塞所有信号，也不一定安全。
13. B。 A 显然是正确的。 B 我们实验过，一般这种读取对于用户来说是允许的，参看书 P511。 C 一般同学不会实验过，需要推理一下，因为参数列表和环境变量实际上是程序自己处理的，而且 shell lab 中重定向是 shell 帮忙完成的，故推测 C 正确。 D 是陆老师著名的“灵魂出窍”例子，即 printf 在信号处理程序中可能导致死锁。

E4

7. 关于 x86-64 系统中的异常，下面那个判断是正确的：
- A. 除法错误是异步异常，Unix 会终止程序；
 - B. 键盘输入中断是异步异常，异常服务后会返回当前指令执行；
 - C. 缺页是同步异常，异常服务后会返回当前指令执行；
 - D. 时间片到时中断是同步异常，异常服务后会返回下一条指令执行；

E4

7. 关于 x86-64 系统中的异常，下面那个判断是正确的：
- A. 除法错误是异步异常，Unix 会终止程序；
 - B. 键盘输入中断是异步异常，异常服务后会返回当前指令执行；
 - C. 缺页是同步异常，异常服务后会返回当前指令执行；
 - D. 时间片到时中断是同步异常，异常服务后会返回下一条指令执行；
7. C。除法错误是不可恢复故障，是同步的；I/O 中断是异步的，一般会执行完当前指令，再去处理，返回就不再需要再处理了；缺页异常当然需要重新执行遇到问题的访存指令；时间片到中断属于时钟中断，属于异步异常。

E5

8. 考虑 4 个具有如下开始和结束时间并运行在不同处理器上的进程，

进程	开始时间	结束时间	运行处理器
A	5	7	P0
B	2	4	P1
C	3	6	P0
D	1	8	P1

下面那个判断是正确的。

- A. 进程 A 和 B 不是并发运行的，但是并行运行的；
- B. 进程 A 和 C 是并发运行的，并且是并行运行的；
- C. 进程 B 和 D 是并发运行的，但不是并行运行的；
- D. 进程 A 和 D 是并发运行的，并且是并行运行的；

E5

8. 考虑 4 个具有如下开始和结束时间并运行在不同处理器上的进程，

进程	开始时间	结束时间	运行处理器
A	5	7	P0
B	2	4	P1
C	3	6	P0
D	1	8	P1

下面那个判断是正确的。

- A. 进程 A 和 B 不是并发运行的，但是并行运行的；
- B. 进程 A 和 C 是并发运行的，并且是并行运行的；
- C. 进程 B 和 D 是并发运行的，但不是并行运行的；
- D. 进程 A 和 D 是并发运行的，并且是并行运行的；

8. 未解之谜，CD。并行一定并发。

E6

5. 关于进程，以下说法正确的是：

- A. 没有设置模式位时，进程运行在用户模式中，允许执行特权指令，例如发起 I/O 操作。
- B. 调用 `waitpid(-1, NULL, WNOHANG & WUNTRACED)` 会立即返回：如果调用进程的所有子进程都没有被停止或终止，则返回 0；如果有停止或终止的子进程，则返回其中一个的 ID。
- C. `execve` 函数的第三个参数 `envp` 指向一个以 `null` 结尾的指针数组，其中每一个指针指向一个形如”`name=value`”的环境变量字符串。
- D. 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 `SIGKILL`，它的默认行为是不能修改的。

E6

5. 关于进程，以下说法正确的是：

- A. 没有设置模式位时，进程运行在用户模式中，允许执行特权指令，例如发起 I/O 操作。
- B. 调用 `waitpid(-1, NULL, WNOHANG & WUNTRACED)` 会立即返回：如果调用进程的所有子进程都没有被停止或终止，则返回 0；如果有停止或终止的子进程，则返回其中一个的 ID。
- C. `execve` 函数的第三个参数 `envp` 指向一个以 `null` 结尾的指针数组，其中每一个指针指向一个形如”`name=value`”的环境变量字符串。
- D. 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 `SIGKILL`，它的默认行为是不能修改的。

答案：C

说明：C 正确见 P521。A 不正确 见 P510。B 中 option 参数应使用 | 运算结合 (P517)。D 中 `SIGKILL` 不是唯一的例外，例外共有两个 `SIGKILL`、`SIGSTOP` (P531)。

E7

11. 在键盘上输入 Ctrl+C 会导致内核发送一个（ ）信号到（ ）进程组中的每个进程
- A. SIGINT, 前台
 - B. SIGTSTP, 前台
 - C. SIGINT, 后台
 - D. SIGTSTP, 后台

E7

11. 在键盘上输入 Ctrl+C 会导致内核发送一个（ ）信号到（ ）进程组中的每个进程
- A. SIGINT, 前台
 - B. SIGTSTP, 前台
 - C. SIGINT, 后台
 - D. SIGTSTP, 后台

11. A。送分题，属于书上原话。

E8

9. 进程管理相关函数的调用和返回行为，下列那些函数都是可能返回多于一次的？
- A. longjmp 和 fork
 - B. execve 和 longjmp
 - C. fork 和 setjmp
 - D. setjmp 和 execve

E8

9. 进程管理相关函数的调用和返回行为，下列那些函数都是可能返回多于一次的？
- A. longjmp 和 fork
 - B. execve 和 longjmp
 - C. fork 和 setjmp
 - D. setjmp 和 execve
9. C。fork 返回两次，setjmp 可返回多次，longjmp 和 execve 不返回。

E9

第四题. 请结合教材第八章“异常控制流”的有关知识回答问题(10分)

PART A. Alice 想用两个进程来顺序输出奇数和偶数, 请你帮她补全代码。她的做法是: 父进程首先 fork 出两个子进程, 之后子进程之间互相通信, 顺序输出 $1, 2, \dots, N$ 。请严格按照注释描述的功能填写代码, 假设兄弟进程的 pid 是相邻的。

```

int N;
int nxt; // 表示该子进程下一个需要输出的数
int pid_1 = 0; // 第一次 fork 出的子进程 pid
int pid_2 = 0; // 第二次 fork 出的子进程 pid
// All child process should do their work in handler 1
void handler1(int sig) {
    printf("%d\n", nxt);
    nxt += 2;
    if (nxt & 1) kill(getpid() + 1, SIGUSR1);
    else {
        assert(pid_1 != 0);
        _____ A _____ // 通知兄弟进程
    }
    if (nxt > N) exit(0); // 子进程完成输出后退出
}

int main(int argc, char* argv[]) {
    _____ B _____ // 将 handler1 绑定到 SIGUSR1 上
    N = atoi(argv[1]);
    nxt = 1;
    if ((pid_1 = fork()) != 0) {
        _____ C _____; // 设置 nxt 的初值
        if ((pid_2 = fork()) != 0) { // 该进程是父进程
            kill(pid_1, SIGUSR1);
            goto wait_til_end; // 跳转并等待子进程结束
        }
    }
    while (1) { sleep(1); } // 子进程会在此循环直到输出完成
    wait_til_end:
    int status;
    while (_____ D _____) // 用 waitpid 等待子进程结束, 注意
    status
        assert(WIFEXITED(status));
    return 0;
}

```

(4分, 每空1分)

- A. _____
- B. _____
- C. _____
- D. _____

PART B. 阅读如下 C 代码, 回答问题

```

int counter = 0;
int pid = 0;
int N = 2;
void handler1(int sig) {
    counter++;
    printf("sd", counter); fflush(stdout);
    // Kill(pid, SIGUSR2);
}
void handler2(int sig) {
    printf("R"); fflush(stdout);
}
int main() {
    Signal(SIGUSR1, handler1);
    Signal(SIGUSR2, handler2);
    if ((pid = Fork()) == 0) { // child
        for (int i = 0; i < N; ++i) {
            printf("C"); fflush(stdout);
            Kill(Getppid(), SIGUSR1);
        }
    } else { // parent
        Wait(NULL);
    }
    return 0;
}

```

1. 进程在何时检查待处理信号, 并调用相应的 signal handler 处理信号?

(1分)

- A. 随时
- B. 用户态切换到内核态
- C. 内核态切换到用户态

2. 在两次检查并处理信号量的时间间隙中, 如果进程接收到 n 个相同信号, 那么

进程实际会处理几个信号? _____ (1分)

- A. 1
- B. 1 到 n 之间的随机数值
- C. n

3. 如果 $N=2$, 所有可能的输出为:

(全部选对得2分, 部分选对得1分, 选错不得分)

- A. CC
- B. CC1
- C. CC12
- D. C1C2

4. 如果 $N=2$, 并且取消 handler1 中的注释 (第7行), 所有不可能的输出为:

(全部选对得2分, 部分选对得1分, 选错不得分)

- A. CC
- B. CC1
- C. CC1R2
- D. C1RC2
- E. C1CR2

E9

PART A. ↴

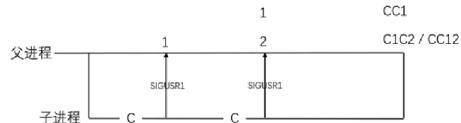
- A: `kill(pid_1, SIGUSR1);` 或 `kill(getpid()-1, SIGUSR1);`
- B: `signal(SIGUSR1, handler1);`
- C: `nxt = 2;`
- D: `waitpid(0, &status, 0) > 0` 或 `waitpid(-1, &status, 0) > 0`

PART B. ↴

1.C ↴

2.A ↴

3.BCD: CC1, C1C2, CC12 ↴

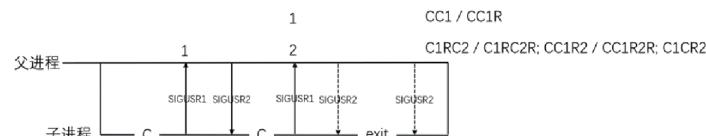


↳

4.A: CC ↴

所有可能的输出为CC1, CC1R, CC1R2, CC1R2R, C1RC2, C1RC2R, C1CR2。 ↴

比如, cc1是可能的, 因为子进程可以在接收到父进程的SIGUSR2之前退出。 ↴



解题要点: ↴

1. 进程只在内核态切换到用户态时检测并处理信号 ↴
2. 在处理信号之前相同信号接收到多次, 只按照一次计算 ↴
3. 注意子进程可以比父进程先结束, 父进程将无法发送信号给子进程 ↴

E10

第四题 (10 分)

C 语言中的代码如下:

```

int counter = 0;

void handler(int sig) {
    counter++;
}

int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    int parent;

    signal(SIGUSR1, handler);

    fd1 = open(fname, O_RDONLY);
    read(fd1, &c1, 1);
    parent = getpid();
    if (fork()) {
        wait();
        fd2 = open(fname, O_RDONLY);
    } else {
        kill(parent, SIGUSR1);
        fd2 = dup(fd1);
    }
    read(fd2, &c2, 1);

    parent = getpid();
    if (fork()) {
        wait();
        fd3 = open(fname, O_RDONLY);
    } else {
        kill(parent, SIGUSR1);
        fd3 = dup(fd2);
    }

    read(fd3, &c3, 1);
    printf("%c %c %c %d\n", c1, c2, c3, counter);
    return 0;
}

```

当 fname 文件内容为 abcde 时, 这段代码的打印结果为____行, 输出结果如下: (不考虑出错的情况)

E10

第四题 (10 分)

C 语言中的代码如下:

```

int counter = 0;

void handler(int sig) {
    counter++;
}

int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    int parent;

    signal(SIGUSR1, handler);

    fd1 = open(fname, O_RDONLY);
    read(fd1, &c1, 1);
    parent = getpid();
    if (fork()) {
        wait();
        fd2 = open(fname, O_RDONLY);
    } else {
        kill(parent, SIGUSR1);
        fd2 = dup(fd1);
    }
    read(fd2, &c2, 1);

    parent = getpid();
    if (fork()) {
        wait();
        fd3 = open(fname, O_RDONLY);
    } else {
        kill(parent, SIGUSR1);
        fd3 = dup(fd2);
    }

    read(fd3, &c3, 1);
    printf("%c %c %c %d\n", c1, c2, c3, counter);
    return 0;
}

```

当 fname 文件内容为 abcde 时, 这段代码的打印结果为____行, 输出结果如下: (不考虑出错的情况)

解答题三 4 行, 内容为

```

a b c 0
a b a 1
a a b 1
a a a 2

```

此题直接画进程树, 在结点旁边标注父进程和子进程, 并写出状态变化就可以。考点为每次父进程 open 时, 指针都重新回到文件开头; dup 则保持文件指针为同一个; count 四个进程相互独立。注意(相对的)父进程总是用 wait(NULL) 等待子进程, 所以输出的顺序一定是子子、子父、父子、父父。

Notices

THANKS

Made by WalkerCH

changxinhai@stu.pku.edu.cn

Reference: [Weicheng Lin]'s presentation.

Reference: [Arthals]'s templates and content.

