第三章：进程

多进程编程是指用多个独立的进程同时执行不同的任务，它能够充分利用多核处理器和分布式计算资源，提高程序的性能和并发能力。然而，多进程编程也面临着进程间同步和通信的挑战，需要谨慎处理共享资源和避免竞态条件。

本章主要介绍进程的基本概念，并讨论了进程控制以及进程间同步和通信的典型技术。

3.1 进程模型

进程模型是计算机科学中描述和理解进程（程序的执行实例）行为、特性和生命周期的概念框架。

3.1.1 基本概念

在操作系统中，进程是一个正在执行的程序的实例。它有自己的地址空间、内存、数据栈以及与操作系统关联的其他资源（如文件描述符、信号处理例程等）。

进程在其生命周期中会经历多种状态：在“就绪”状态，进程已准备好运行，只是在等待 CPU 分配时间片；当进程正在 CPU 上执行时，它处于“运行”状态；如果进程在等待某一事件完成，如 I/O 操作，它会进入“等待”状态；最后，当进程执行完毕或被终止时，它转到“终止”状态。进程因各种事件或调度决策在这些状态间进行转换。

进程的创建通常由现有的进程通过系统调用，如在 UNIX-like 系统中的 fork()。当 fork()被调用时，操作系统为新进程分配一个唯一的进程 ID，并为其复制父进程的地址空间、进程环境和执行上下文。然后，根据后续的系统调用，如 exec()，子进程加载并执行一个新程序，或继续执行与父进程相同的代码但具有不同的行为逻辑。

进程的终止是指一个进程完成其执行或因某种原因被中止的过程。这可以是由进程自身发出的正常退出请求，例如通过执行 exit()系统调用；或者是由其他进程通过特定的系统调用，如 kill()，来强制终止。无论哪种情况，操作系统都会回收进程使用的资源，如内存和文件描述符，并更新进程的状态为“终止”。

进程同步是确保多个并发执行的进程在访问和修改共享资源时能够有序、协调地进行，以避免数据不一致和其他并发错误的机制。常用的进程同步手段包括信号灯、

信号等，它们都提供了一种方式来控制资源的访问，确保在任何时刻只有一个进程能够访问某个特定的资源或执行某个特定的代码段。

进程间通信（IPC）是指在独立进程之间传输数据或信号的机制，使它们能够协同工作。常见的 IPC 方法包括管道、内存映射文件等。这些机制允许进程相互发送数据、同步执行或者通知事件，从而实现进程间的数据共享、同步和协调。

进程模型为操作系统提供了执行、管理和协调任务的结构化方式。理解这一模型对于掌握多任务环境中的并发和资源管理至关重要。

3.1.2 系统调用

在多任务操作系统中，为了实现资源管理和任务隔离，通常会将操作系统分为用户空间和内核空间两个运行模式。系统调用是两者之间通信的基础机制，它使用户空间的应用程序能够安全、高效地访问操作系统内核提供的各种资源和服务。

当一个用户空间程序需要执行某种特权操作（例如文件 I/O 或进程创建）时，该程序会触发一个系统调用。这一触发动作通常通过 CPU 的软件中断指令完成，导致处理器从用户模式切换到内核模式，进而执行相应的内核代码。完成系统调用后，控制权返回用户程序，处理器模式也随之切换回用户模式。

系统调用可以按照功能和用途分为多个类别，包括进程控制（如创建、调度和终止进程）、文件操作（如文件读写、打开和关闭）、设备管理（如设备初始化和控制）、信息维护（如获取系统状态和进程信息）、内存管理（如内存分配和回收）以及进程间通信（如管道和内存映射文件）。这些分类提供了操作系统内核功能和用户程序之间交互的基础框架。

在 C 语言中，系统调用是一组由操作系统内核提供的底层函数或程序接口，允许用户态应用程序执行诸如文件操作、进程控制、内存管理和进程间通信等核心操作。通过标准库或平台特定的头文件，这些系统调用被封装为 C 函数，从而允许开发者在应用程序中直接调用它们，以实现对操作系统资源和服务的访问和控制。

下面以 X64 为例，演示在 C 语言中如何使用系统调用来实现 puts()函数。

```
int puts(char *s) {
    long n = strlen(s);
    long r;
    asm(CALL(SYS_WRITE)
       "movq $1, %%rdi\n"
       "movq %1, %%rsi\n"
       "movq %2, %%rdx\n"
       "syscall\n"
       "movq %%rax, %0\n"
       : "=r"(r)
       : "r"(s), "r"(n)
       : "%rax", "%rdi", "%rsi", "%rdx");
    return (int)r;
}
```

该 puts 函数使用内联汇编在 Linux 下通过 syscall 机制直接调用 write 系统调用，将字符串 s 写入标准输出。首先，它计算字符串的长度，然后通过汇编将相关参数设置到适当的寄存器（如 rdi, rsi, 和 rdx）中。syscall 指令执行实际的系统调用，之后，函数返回实际写入的字节数。这种方法提供了一个低级别的、直接与操作系统交互的方式来输出字符串，避免了使用常规 C 库函数。

3.1.3 文件

在计算机系统中，文件是信息存储的基本单元。通常存储在某种持久性存储设备上。根据用途和内容，文件可以分为不同的类型，如文本文件、二进制文件、目录文件、设备文件等。

在 C 语言编程中，文件操作的抽象层实际上是对底层操作系统提供的文件系统 API 的封装。C 标准库（尤其是 stdio.h）中的文件操作函数，如 fopen(), fread(), fwrite() 等，通常在底层被转换为相应的系统调用，例如 UNIX 和 Linux 环境中的 open(), read(), write()等。这些转换过程是由 C 运行时库（C Runtime Library）负责的，它充当了用户空间程序与操作系统内核之间的中介。同时，在 UNIX 和 Linux 环境中，底层的文件描述符（File Descriptor）也被广泛使用，它们是一种低级别的文件操作手段，直接对接

操作系统的文件管理子系统。

文件描述符（File Descriptor）是 UNIX 和 Linux 操作系统中用于唯一标识一个打开的文件或其他 I/O 流（如套接字和管道）的非负整数。在底层，文件描述符实际上是一个索引，指向进程文件描述符表中的一个条目，该条目包含了与该 I/O 流相关的各种状态信息和控制标志。通过系统调用（如 read(), write()等），文件描述符允许程序直接与操作系统内核的 I/O 子系统进行交互，从而实现高效的数据读写和资源管理。这种机制为应用层与操作系统内核之间提供了一个低延迟、高吞吐量的接口，是现代 UNIX-like 操作系统中多任务和并发 I/O 操作的基础。

POSIX（Portable Operating System Interface）标准为文件操作定义了一系列接口函数。其头文件和主要函数的原型为：

```
#include <fcntl.h>
#include <unistd.h>
int open(const char *filename, int flags, ... /* mode_t mode */);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
int unlink(const char *filename);
```

以 open 函数为例（其他函数接口的细节可根据深入阅读进一步了解），open 函数是 POSIX 标准中用于打开或创建文件的接口。它接受一个文件路径名 filename 作为第一个参数，并使用 flags 参数指定文件的访问模式和行为（如只读、只写或读写，以及是否创建新文件等）。当在 flags 中指定了 O_CREAT 标志以创建新文件时，该函数还可以接受一个可选的 mode 参数，用于设置新文件的权限。成功调用会返回一个文件描述符，用于后续的文件操作，而失败则返回-1。

下面我们用一个例子来演示如何使用 POSIX 的文件操作接口进行基本的文件操作。

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
int main() {
    char buffer[1024];
    const char *filename = "sample.txt";
    const char *data = "Hello, POSIX!";
    int fd = open(filename, O_CREAT | O_WRONLY, 0644);  // 打开并写入文件
    write(fd, data, strlen(data));
    close(fd);
    fd = open(filename, O_RDONLY); // 重新打开并读取文件内容
    read(fd, buffer, sizeof(buffer) - 1);
    buffer[strlen(data)] = '\0';
    close(fd);
    printf("Content of %s: %s\n", filename, buffer); // 显示文件内容
    unlink(filename);          // 删除文件
    return 0;
}
```

该程序首先通过 open 函数打开一个名为"sample.txt"的文件，并为写操作设置权限。接着，利用 write 函数将字符串"Hello, POSIX!"写入该文件，并随后关闭文件描述符。然后，程序再次通过 open 函数以只读模式打开文件，使用 read 函数读取内容到缓冲区，并确保字符串以空字符终止。最后，程序通过 printf 输出读取到的文件内容，并使用 unlink 函数删除该文件。此程序重点演示了 open, write, read, close, 和 unlink 这五个 POSIX 文件操作函数的基本用法。

3.1.4 同步与通信

进程的同步与通信是操作系统中两个核心概念，用于协调并发进程的行为。同步机制确保多个进程在关键部分代码上的有序和协调执行，避免资源竞争和数据不一致性，常见的同步手段有信号、信号灯等。通信则是进程间交换信息的方式，使它们能够共享数据或协同工作，常用的通信方式有管道、内存映射文件等。这两者共同保证了并发进程的正确性和高效性。

不同的同步/通信机制适用于不同的场景和需求，正确使用这些机制可以提高并发程序的效率和稳定性。同步/通信机制可大致分为下面几种：

同步/通信技术	模型	作用	粒度	网络
semaphore	消息传递	同步	none	本地
signal	message passing	synchronization	none	本地
pipe/FIFO	message passing	data exchange	字节流	本地
shm()	shared memory	data exchange	none	本地
memory-mapped file	shared memory	data exchange	none	本地
socket	message passing	data exchange	either	均可
message queue	message passing	data exchange	结构化	本地

在 3.3 和 3.4 小节，我们会来深入探讨同步与通信的具体内容。

3.2 生命周期

3.2.1 进程创建

进程的创建是操作系统为新任务分配资源并初始化相关数据结构的过程，通常涉及复制现有进程的上下文或从特定程序加载新上下文，生成新的唯一进程标识符，并设置进程的初始状态，以便进程调度程序可以开始其执行。

POSIX 标准（Portable Operating System Interface）中用于创建新进程的是 fork()函数。当调用 fork()函数时，当前进程（称为父进程）被复制为一个新进程（称为子进程）。子进程几乎是父进程的完整复制，它继承了父进程的内存、文件描述符、程序计数器、环境变量、进程优先级等。

fork()系统调用原型如下：

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

在 fork()之后，两个进程（父进程和子进程）从 fork()调用处开始独立执行。为了区分这两个进程，fork()在父进程中返回子进程的进程 ID，而在子进程中返回 0。

示例

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid;
    pid pid = fork();            // 调用 fork()
    if (pid == 0) {          // 返回值为 0 表示子进程
        printf("I am the child process, PID = %d\n", getpid());
    }
    else if (pid > 0) {      // 返回值大于 0 表示父进程
        printf("I am the parent process, PID = %d\n", getpid());
    }
    else {
        perror("fork failed");
    }
    return 0;
}
```

3.2.2 程序加载

程序加载是操作系统的一项关键任务，包括读取可执行文件（如 ELF 格式）从持久存储（如硬盘）到系统内存，并在适当的内存地址空间中解析和放置其代码、数据段等。加载器还需处理重定位、解析符号和链接共享库。完成这些操作后，操作系统将初始化并为此程序分配一个进程或线程的上下文，并开始执行程序入口点指定的指令。

POSIX 标准（Portable Operating System Interface）中用于加载进程的是 exec()系列函数。这些函数用于替换当前进程映像为一个新的进程映像，而进程 ID 和其他属性（例如打开的文件描述符、信号处理和环境变量）则保持不变。即调用 exec 之后，原始程序不再执行。

在 UNIX-like 系统中，exec 系列函数包括以下成员：

```
int execl(const char *path, const char *arg0, ... /*, (char
*)NULL */);
int execle(const char *path, const char *arg0, ... /*, (char
*)NULL, char *const envp[] */);
int execlp(const char *file, const char *arg0, ... /*, (char
*)NULL */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const
envp[]);
```

exec 系列函数在 UNIX-like 系统中用于替换当前进程的内存空间内容，执行一个新程序。这些函数的不同版本提供了不同的方式来指定要运行的程序、其参数以及环境变量。尽管进程的代码和数据都被新程序替换，但该进程的 PID 和其他属性保持不变。不同的 exec 函数之间的差异主要体现在如何传递参数和环境变量以及如何定位要执行的程序文件。

本例创建一个子进程来执行另一个程序（./hello.out），而父进程则等待子进程完成并检查其退出状态。

```
#include <stdio.h>
#include <unistd.h>
int main(){
    printf("\tpid=%d\n", getpid());
    return 42;
}
```

该程序仅打印其进程 ID，然后返回值 42。

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

int main(){
    pid_t pid = fork();
    if(pid == 0){ // child
        execlp("./hello.out", "hello.out", NULL);
    }
    int status;   // parent
    wait(&status);
    if(WIFEXITED(status))
        printf("child exited normally"
               " with return code: %d\n", WEXITSTATUS(status));
    return 0;
}
```

该程序首先创建一个子进程，在子进程中，使用 execlp 函数执行名为 hello.out 的外部程序。父进程等待子进程完成并打印其返回代码 42。

```
      in.hello, world
child exited normally with return code: 42
```

本例是一个简单的并行计算框架，其目的是计算从 0 到 N−1（其中 N 由用户输

入）这些数字的平方和。

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

int main(int argc, char *argv[]){
    if(argc < 2){
        printf("Usage: mult-child <N>\n");
        exit(1);
    }
    int n = atoi(argv[1]);
    printf("\tpid=%d, arg=%d\n", getpid(), n);
    return n*n;
}
```

这个程序接受一个命令行参数，返回该命令行参数的平方值作为退出状态（第 13 行）。

该程序创建了指定数量（由命令行参数给出）的子进程，每个子进程运行 mult-child.out 程序并传递一个整数参数。每个子进程计算这个整数的平方值并返回。父进程等待所有子进程完成并收集它们的返回值（第 25 行），计算这些值的总和，然后与预期的总和（所有整数的平方的总和）进行比较，最后输出结果。

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

int main(int argc, char *argv[]){
    if(argc < 2){
        printf("Usage: mult <N>\n");
        exit(1);
    }
    int num_process = atoi(argv[1]);
    pid_t *pids = malloc(sizeof(pid_t) * num_process);
    for(int i=0; i<num_process; i++) {
        pid_t pid = fork();
        if (pid == 0) { // child
            char arg[1024];
            sprintf(arg, "%d", i);
            execlp("./mult-child.out", "mult-child.out", arg,
NULL);
        }
        pids[i] = pid;
    }
    int status;   // parent
    int sum = 0;
    for(int i= 0; i<num_process; i++){
        wait(&status);
        if(WIFEXITED(status)){
            int result = WEXITSTATUS(status);
            sum += result;
            printf("child exited normally"
                   " with return code: %d\n", result);
        }
    }
    printf("the sum is: %d\n", sum);
    int oracle = 0;
    for(int i=0; i<num_process; i++){
        oracle += i*i;
```

执行结果如下：

```
majn@tiger:~/C/sys-prog/code/chap03-exec$ ./mult.out 10
        pid=728333, arg=0
        pid=728334, arg=1
        pid=728335, arg=2
child exited normally with return code: 0
        pid=728336, arg=3
child exited normally with return code: 1
        pid=728337, arg=4
child exited normally with return code: 4
child exited normally with return code: 9
        pid=728338, arg=5
child exited normally with return code: 16
        pid=728339, arg=6
child exited normally with return code: 25
child exited normally with return code: 36
        pid=728340, arg=7
child exited normally with return code: 49
        pid=728341, arg=8
        pid=728342, arg=9
child exited normally with return code: 64
child exited normally with return code: 81
the sum is: 285
the oracle is: 285
```

3.2.3 进程退出

在 C 程序中，退出通常是指程序的正常或异常终止。程序可以通过调用 exit()函数来实现正常退出，此时会执行所有注册的终止函数、关闭所有 I/O 流并向操作系统返回一个退出状态。若程序中出现未处理的错误或异常，例如段错误或访问违规，这也可能导致程序的非正常终止。无论哪种退出方式，操作系统都会负责回收程序所使用

的资源。

在操作系统中，一个进程无论是正常退出还是异常结束，操作系统都会执行一系列内核级的处理过程。首先，系统会回收进程的地址空间，释放其使用的物理和虚拟内存。接着，与该进程关联的所有文件描述符和 I/O 通道都将被关闭。如果进程拥有任何系统级的信号量、共享内存段或其他内核资源，系统也会将其回收。最后，该进程的进程控制块（PCB）会被更新，标记其为“终止”状态，并将其退出码存储，以供其他进程或父进程查询。此过程确保了系统资源的高效利用和内核状态的一致性。

POSIX 标准（Portable Operating System Interface）中用于终止一个进程的是 exit()函数。它首先调用通过 atexit()注册的终止处理函数，接着刷新和关闭所有打开的 stdio 流，然后释放进程占用的资源，并最终返回一个表示退出状态的整数给操作系统。

exit()函数原型如下：

```
#include <stdio.h>
void exit(int status);
```

该函数向操作系统返回一个退出状态，并执行所有已注册的终止函数，然后结束进程。

示例：

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

void termination_func() {
    printf("Termination function executed.\n");
}
int main() {
    atexit(termination_func); // 注册终止处理函数
    printf("Starting the program.\n");
    exit(EXIT_SUCCESS); // 或 exit(0)
    printf("This line will never be printed.\n");
}
```

这段程序开始时注册了一个终止处理函数 termination_func（第 8 行）。当程序执行至 main 函数，通过调用 exit(EXIT_SUCCESS)，程序开始正常终止，并触发之前注册的 termination_func 函数。由于程序最后的输出语句位于 exit()之后，所以不会被执行。

3.3 同步

进程同步是指多个并发执行的进程或线程在访问共享资源时保持一致性和协调性的一种机制。在并发执行的环境中，当多个进程或线程同时对共享资源进行读写操作时，如果没有合适的同步机制，可能会导致数据不一致、临界区问题、死锁等并发访问问题。进程同步的目的是保证多个并发执行的进程或线程能够按照一定规则协同工作，确保共享资源的正确访问和数据的一致性。

3.3.1 信号灯（Semaphore）

信号灯（Semaphore）可以多进程环境下用于协调各个进程, 以保证它们能够正确、合理的使用公共资源。信号灯维护了一个许可集，我们在初始化 Semaphore 时需要为这个许可集传入一个数量值，该数量值代表同一时间能访问共享资源的进程数量。

按信号灯实现原理，信号灯分两种，一种是有名信号灯，一种是基于内存的信号

灯，也叫无名信号灯。有名信号灯，是根据外部名字标识，通常指代文件系统中的某个文件。而无名信号灯，它主要把信号灯放入内存。

Linux 提供了一组 API 来实现有名信号灯的功能，其头文件和主要函数的原型为：

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
unsigned int value);
int sem_wait(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_post(sem_t *sem);
int sem_unlink(const char *name);
```

其中，sem_opem 用于创建一个信号灯并设置其初值，或者打开一个已经存在的信号灯。进程需要访问公共资源或进入临界区时需要调用 sem_wait，此时若信号灯的数值为 0，进程便进入阻塞态，否则信号灯的值减 1（对应 PV 操作中的 P 操作）。进程释放资源后，就需要调用 sem_post 增加信号灯的值（V 操作）。以下面两个程序为例：

```
//sema1.c
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#define SEM_NAME "namedsem"    //Defines the name of the
semaphore
int main(){
    sem_t *p_sem = NULL;
    //Create a semaphore
    p_sem = sem_open(SEM_NAME, O_CREAT | O_EXCL, 0666, 0);
    sem_wait(p_sem);          //Wait the sempahore
    sem_unlink(SEM_NAME);    //Delet the semaphore
    return 0;
}
```

```
//sema2.c
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#define SEM_NAME "namedsem"
int main(){
    sem_t *p_sem = NULL;
    p_sem = sem_open(SEM_NAME, O_RDWR);    //Open the semaphore
    sem_post(p_sem);        //Release semaphores
    return 0;
}
```

编译运行 sema1.c 后会初始化一个值为 0 的信号灯，此后 sema1.c 再调用 sem_wait 便会直接进入阻塞。只有编译运行 sema2.c 调用 sem_post 释放信号灯后，sema1.c 的进程才能被唤醒，并正常结束。

无名信号灯和有名信号灯的差异主要体现在创建和销毁的方式上，但是其他工作都是一样的。无名信号灯只能存在于内存中，要求使用信号灯的进程必须能访问信号灯所在的这一块内存，所以无名信号灯只能应用在同一进程内的线程之间（共享进程的内存），或者不同进程中已经映射相同内存内容到它们的地址空间中的线程（即信号量所在内存被通信的进程共享）。相反，有名信号量可以通过名字访问，因此可以被任何知道它们名字的进程中的线程使用。

3.3.2 信号（Signal）

进程间信号（inter-process signals）是用于进程间通信和同步的另一种机制。一个进程可以向目标进程发送一个特定的事件或消息。当目标进程接收到信号时，它可以选择采取适当的动作来处理该信号。

在 Linux 系统中可以使用命令 kill -l 来查看信号列表：

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP  6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1
....
```

常见的 SIGINT（中断信号）通常由终端键盘输入的 Ctrl+C 触发，用于请求进程终止运行。SIGSEGV（段错误信号）表示进程访问了无效的内存地址。

进程是十分脆弱的，我们可以通过 kill -s <Signal> <PID>的命令向特定进程发送信号令其终止。例如编译运行如下的死循环：

```
//hello.c
#include <stdio.h>
#include <unistd.h>
int main(){
    while(1){
        puts("hello world");
        sleep(1);
    }
}
```

程序每 1 秒会在终端打印一次 hello world。新开一个终端使用 ps -e 查看其进程号：

```
$ ps -e
....
136440 pts/3    00:00:00 a.out
....
```

此时使用命令向进程发送 SIGSEGV 信号：

```
$ kill -s SIGSEGV 136440
```

该进程会误以为自己出现了段错误，便会停止运行并报错：

```
hello world
hello world
hello world
Segmentation fault
```

Linux 系统同样提供了一对库函数来实现信号机制，其头文件和函数原型为：

```
#include <signal.h>
int kill(pid_t pid, int sig);
sighandler_t signal(int sig, sighandler_t handler);
```

调用 signal 可以设定特定信号 sig 的处理函数 handler。进程收到另一个调用 kill 发送过来的信号 sig 时，便会开始执行 handler 函数。通过这一对函数便可以实现最基本的进程间通信，以下面的程序为例：

```
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
void sigroutine(int dunno) {
    switch (dunno) {
    case 6: {
        printf("\tI am child process, I receive signal
SIGABRT\n");
        break;
    }
   }
}
int main(){
    pid_t pid;
    int status;
    int sig;
    signal(SIGABRT, sigroutine);
    if (!(pid = fork())){
        printf("\tHi I am child process !\n");
        sleep(10);
        return 0;
    }
    else {
       sleep(1);
        kill(pid, SIGABRT); // Sends a SIGABRT (abort program)
signal
        wait(&status);
        if (WIFSIGNALED(status))
            printf("chile process receive signal % d\n",
WTERMSIG(status));
        else
            printf("child process exits normally with %d\n",
WTERMSIG(status));
    }
    return 0;
```

这个程序中使用 signal 定义了 SIGABRT 信号的处理函数，父进程向子进程发送 SIGABRT 信号后（26 行），子进程便会进入 sigroutine 函数并打印对应的语句。值得注意的是，调用完处理函数之后，子进程不再受之前终止信号的影响，退出码变回了正常的 0。

```
$ ./a.out
        Hi I am child process !
        I am child process, I receive signal SIGABRT
child process exits normally with 0
```

signal 函数的使用方法简单，但并不属于 POSIX 标准，在各类 UNIX 平台上的实现不尽相同，因此其用途受到了一定的限制。而 POSIX 标准定义的信号处理接口是 sigaction 函数，其接口头文件及原型如下：

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```

其中，struct sigaction 结构体定义为：

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

使用 sigaction 注册的信号处理程序被调用时，系统建立的新信号屏蔽字会自动包括正被递送的信号。因此保证了在处理一个给定的信号时，如果这种信号再次发生，那么它会被阻塞到对前一个信号的处理结束为止。sigaction 函数的功能比 signal 函数更为丰富，但基本通信的实现方法是类似的：

```
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
void sigroutine(int dunno) {
    switch (dunno) {
    case 6: {
        printf("\tI am child process, I receive signal
SIGABRT\n");
        sleep(2);
        break;
    }
   }
}
int main(){
    pid_t pid;
    int status;
    struct sigaction sigact;
    sigact.sa_handler = sigroutine;
    sigaction(SIGABRT, &sigact, 0);
    if (!(pid = fork())){
        printf("\tHi I am child process !\n");
        sleep(10);
        return 0;
    }
    else {
        sleep(1);
        kill(pid, SIGABRT); // Sends a SIGABRT (abort program)
signal
        wait(&status);
        if (WIFSIGNALED(status))
            printf("chile process receive signal % d\n",
WTERMSIG(status));
        else
            printf("child process exits normally with %d\n",
WTERMSIG(status));
```

同样我们也可以注册 sigroutine 为 SIGABRT 的信号处理函数，达到和 signal 函数一样的效果。

3.4 通信

3.4.1 管道（Pipe）

管道（Pile）是一种用于进程间通信（IPC）的机制，通常用于在父子进程之间传递数据，允许一个进程将数据发送到另一个进程。管道有两种类型：无名管道（Anoymous Pipe）和有名管道（Named Pipe，也称为 FIFO）。

无名管道

无名管道是在相关的进程之间传递数据的简单方式。它通常用于父子进程之间的通信，且通常是半双工的，即它只能在一个方向上传递数据，而从另一个方向上接受数据。

我们可以使用 pipe()系统调用来创建无名管道(使用时需包含头文件 unistd.h)，pipe()系统调用原型如下：

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

该管道有两个文件描述符，一个用于读取数据，另一个用于写入数据。参数 pipefd 是一个长度为 2 的整数数组，用于存储管道的文件描述符。pipefd[0]代表管道的读取端，pipefd[1]代表管道的写入端。pipe()成功时返回 0，失败时返回-1。

下面我们对给出一个使用无名管道的实例：

```
#include <unistd.h>
// Some other libraries are omitted here
int pipep2c[2];// parent write, child read
int main(){
    memset(pipep2c, 0, sizeof(pipep2c)); // init the pipe
    pipe(pipep2c);
    pid_t pid = fork();
    if(pid == 0){ // child
        close(pipep2c[1]);
        char buf[1024];
        int n = read(pipep2c[0], buf, 1024);
        buf[n] = '\0';
        printf("\tchild recieved: %s\n", buf);
        exit(42);
    }
    close(pipep2c[0]);// send message to child
    char *msg = "hello, world";
    write(pipep2c[1], msg, strlen(msg));
    int status;
    wait(&status);
    if(WIFEXITED(status)){
        printf("child exited normally"
               " with return code: %d\n", WEXITSTATUS(status));
    }
    return 0;
}
```

这段 C 代码演示了父子进程之间使用无名管道进行通信的示例，我们使用 fork()创建子进程(第 10 行)。子进程继承父进程的文件描述符，包括管道。

如果 fork()返回的 pid 值为 0，表示这是子进程，在后续代码块中会执行子进程的相关处理；如果 fork()返回的 pid 值不为 0，表示这是父进程。后续进行的是父进程的相关处理。

直观上看，这段代码像是先读后写的，且我们也并没有看到上一小节同步中所介绍的同步机制，但子进程却可以读到父进程写的内容。实际上是管道实现了父子进程的同步，管道保证进程能够实现进程的阻塞式读取和阻塞式写入，即子进程在读管道时，若管道中没有数据，则子进程会阻塞自己；类似的，写进程在写满管道时也会阻塞自己。

我们再来看一个无名管道的例子，在程序执行时父进程将创建两个子进程，left_child 和 right_child:

```
#include <unistd.h>
// Some other libraries are omitted here
int shell_pipe[2];
int main(){
    memset(shell_pipe, 0, sizeof(shell_pipe)); // init the pipe
    pipe(shell_pipe);
    pid_t left_child = fork();
    if(left_child == 0){ // child
        close(shell_pipe[0]);
        close(1);
        dup(shell_pipe[1]);
        close(shell_pipe[1]);
        char *the_argv[] = {"/usr/bin/ls", 0};
        execve("/usr/bin/ls", the_argv, (void *)0);
        exit(42);
    }
    pid_t right_child = fork();
    if(right_child == 0){ // child
        close(shell_pipe[1]);
        close(0);
        dup(shell_pipe[0]);
        close(shell_pipe[0]);
        char *the_argv2[] = {"/usr/bin/tail", "-2", 0};
        execve("/usr/bin/tail", the_argv2, (void *)0);
        exit(42);
    }
    close(shell_pipe[0]);
    close(shell_pipe[1]);
    int status;
    wait(&status);
    if(WIFEXITED(status)){ //
        printf("child exited normally"
               " with return code: %d\n", WEXITSTATUS(status));
    }
    wait(&status);
    if(WIFEXITED(status)){ //
        printf("child exited normally"
```

在这个例子中，父进程首先使用 pipe 函数创建管道，左子进程关闭标准输出后(第 13 行)，将标准输出重定向到管道的写入端(第十四行)，并关闭原始的管道写入文件描述符(第 15 行)，以实现其内部的打印结果直接写进管道里，右子进程进行相似的操作，从管道中读取最后两个文件名(分别为 shell.c 和 shell.out)，结果如下所示：

```
...
this is parent
child exited normally with return code: 0
shell.c
shell.out
child exited normally with return code: 0
```

有名管道：

有名管道是一种具有文件名的管道，它允许不相关的进程之间进行通信。有名管道是全双工的，允许双向通信。有名管道在文件系统中有一个唯一的路径，并可以像普通文件一样被打开、读取和写入。在 Linux 系统中，有名管道也被称为 FIFO（First In, First Out）。

有名管道可以使用 mkfifo()系统调用来进行创建(使用时需包含头文件 stat.h),其函数原型如下：

```
#include<stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

下面我们以一个简单的示例，展开对有名管道的讲解：

在 fifio-inc.h 中定义宏，用来指向管道：

```
#ifndef FIFO_INC_H
#define FIFO_INC_H
#define FIFO_NAME "/tmp/MY_TEST_FIFO"
#endif
```

fifo.c 程序如下，运行这段程序的进程读取管道中的数据并进行打印：

```
// fifo.c
#include <sys/stat.h>
#include "fifo-inc.h"
// Some other libraries are omitted here
int main(){
    mkfifo(FIFO_NAME, (S_IRUSR | S_IWUSR));// create a FIFO
    int fifo_fd = open(FIFO_NAME, O_RDONLY);// open the fifo
    char buf[1024];
    int n = read(fifo_fd, buf, 1024);
    buf[n] = '\0';
    close(fifo_fd);
    printf("process %d read: %s\n", getpid(), buf);
    unlink(FIFO_NAME);
    return 0;
}
```

fifo2.c 程序如下，运行这段程序的进程向管道中写入了一个字符串：

```
// fifo2.c
#include "fifo-inc.h"
// Some other libraries are omitted here
int main(){
    int fifo_fd = open(FIFO_NAME, O_WRONLY);
    printf("\tpid=%d\n", getpid());
    char *msg = "hello, fifo\n";
    write(fifo_fd, msg, strlen(msg));
    sleep(3);
    close(fifo_fd);
    printf("\tprocess:%d exited\n", getpid());
    return 42;
}
```

创建两个终端，第一个终端执行./fifo.out，第二个终端执行./fifo2.out，可以看到 fifo2 向 fifo 通过管道发送的信息：

```
--terminal 1:                          |      --terminal 2:
./myfifo.out                           |      ./myfifo2.out
process 1992434 read: hello, fifo      |      fifo_id=-1
                                       |      pid=1995264
                                       |      process:1995264 exited
```

从上述例子中我们可以发现：有名管道不再拘束于父子进程之间，其可以满足任意两个进程之间的通信。

3.4.2 内存映射文件

内存映射文件（Memory-Mapped File）是一种将文件内容映射到内存中的机制，允许程序直接访问文件数据，就好像这些数据已经被加载到了内存一样。这个机制允许文件的内容被映射到一个进程的地址空间，从而允许程序以一种更高效的方式读取或写入文件数据，同时，多个进程可以映射同一个文件，从而实现进程间的数据共享。这对于进程间通信非常有用。

Linux 提供了一组 API 来实现内存映射文件的功能，其头文件和主要函数的原型为：

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_open(const char *name, int oflag, mode_t mode);
void *mmap(void *addr, size_t length, int prot, int flags, int
fd, off_t offset);
int munmap(void *addr, size_t length);
```

其中 shm_open 用于创建或打开一个命名的共享内存对象，mmap 用于将共享内存对象映射到进程的地址空间，munmap 用于取消映射已映射的共享内存区域。

下面让我们来看一个内存映射文件的例子，首先，我们创建一个名为 smf.c 的程序，其作用时是用来读取共享内存映射中的内容：

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "smf-inc.h"
// Some other libraries are omitted here
int main() {
    sem_t *write_sem = sem_open(WRITE_SEM_NAME, O_CREAT | O_EXCL
| O_RDWR, 0666, 1);
    if (write_sem == SEM_FAILED) {
        write_sem = sem_open(WRITE_SEM_NAME,O_RDWR);
    }
    sem_t *read_sem = sem_open(READ_SEM_NAME, O_CREAT | O_EXCL |
O_RDWR, 0666, 0);
    if (read_sem == SEM_FAILED) {
        read_sem = sem_open(READ_SEM_NAME, O_RDWR);
    }
    int fd = shm_open(FILE_NAME, O_CREAT | O_RDWR, 0666);
    char *addr = mmap(0, 1024, PROT_READ, MAP_SHARED, fd, 0);
    for (int i = 0; i < 10; i++) {
        sem_wait(read_sem);
        printf("process-a: %s\n", addr);
        sem_post(write_sem);
    }
    munmap(addr, 1024);
    close(fd);
    unlink(FILE_NAME);
    sem_close(read_sem);
    sem_close(write_sem);
    sem_unlink(READ_SEM_NAME);
    sem_unlink(WRITE_SEM_NAME);
    return 0;
}
```

smf-inc.h 的内容为,其定义了一个宏 FILE_NAME，用于指向某个文件：

```
#ifndef FIFO_INC_H
#define FIFO_INC_H
#define FILE_NAME "MY_TEST_MMAP_FILE"
#define WRITE_SEM_NAME "MY_TEST_WRITE_SEM_FILE"
#define READ_SEM_NAME "MY_TEST_READ_SEM_FILE"
#endif
```

接下来，我们创建 smf2.c 的程序，用来向共享映射内存写入内容：

```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "smf-inc.h"
int main() {
    sem_t *write_sem = sem_open(WRITE_SEM_NAME, O_CREAT | O_EXCL
| O_RDWR, 0666, 1);
    if (write_sem == SEM_FAILED) {
        write_sem = sem_open(WRITE_SEM_NAME,O_RDWR);
    }
    sem_t *read_sem = sem_open(READ_SEM_NAME, O_CREAT | O_EXCL |
O_RDWR, 0666, 0);
    if (read_sem == SEM_FAILED) {
        read_sem = sem_open(READ_SEM_NAME, O_RDWR);
    }
    int fd = shm_open(FILE_NAME, O_CREAT | O_RDWR, 0666);
    ftruncate(fd, 1024);
    // map the file into memory
    char *addr = mmap(0, 1024, PROT_WRITE, MAP_SHARED, fd, 0);
    for (int i = 0; i < 10; i++) {
        sem_wait(write_sem);
        sprintf(addr, "%d\n", i * i);
        sem_post(read_sem);
    }
    munmap(addr, 1024);
    close(fd);
    sem_close(write_sem);
    sem_close(read_sem);
    return 0;
}
```

这段代码实现了写进程写入 0 到 9 的平方，读进程读出结果的效果，为了实现进程同步，在此处引入读信号量和写信号量，写进程等到读信号时，向共享内存写入数

据，并释放写信号，读进程等待写信号，读出共享内存数据后，再释放读信号。

3.5 本章小结

在本章中，我们首先介绍了进程的基本模型，并结合实例介绍了如何创建、运行和退出一个进程。此外，对于多进程编程中不可忽视的进程间同步和通信，我们分别讨论了信号灯、信号、管道以及内存映射文件这四种常见方法。

3.6 深入阅读

进程是操作系统进行资源分配和调度的基本单位，在操作系统内核会维护一个数据结构来描述进程的各种属性，并基于此对进程进行控制。此外，除了进程的创建、加载和退出，内核也需要负责进程的初始化、并对进程进行合理的调度以充分利用处理器和内存等资源。想要深入了解这部分内容的读者可以阅读《Linux 内核完全注释》。