基于 Linux 的 C++

第十三讲　进程编程

提　纲

进程基本概念

信　号

进程管理

进程间通信

- 管道、进程信号量、共享内存、映射内存、消息队列、套接字

进程池

进程基本概念

进程（process）的定义

- 进程是描述程序执行过程和资源共享的基本单位
- 主要目的：控制和协调程序的执行

进程相关函数

- 用户与组ID函数：参阅上一讲
- 创建进程：system()、fork()、exec()
- 终止进程：kill()
- 等待进程终止：wait()、waitpid()

进程基本概念

进程组

- 定义：由一个或多个相关联的进程组成，目的是为了进行作业控制
- 进程组的主要特征：信号可以发送给进程组中的所有进程，并使该进程组中的所有进程终止、停止或继续运行
- 每个进程都属于某个进程组

进程组函数

- 获取进程组ID：pid_t getpgid(pid_t pid);
  - 返回pid进程的进程组ID；若pid为0，则返回当前进程的进程组ID；出错时返回-1，并设errno值
- 设置进程组ID：int setpgid(pid_t pid, pig_t pgid);
  - 若pid为0，则使用调用者PID；若pgid为0，则将pid进程的进程PID设为进程组ID；成功时返回0，出错时返回-1，并设errno值

进程基本概念

会话（session）

- 会话为一个或多个进程组的集合，包括登录用户的全部活动，并具有一个控制终端
- 登录进程为每个用户创建一个会话，用户登录shell进程成为会话首领，其PID设为会话ID
- 非会话首领进程通过调用setsid()函数创建新会话，并成为首领

进程组函数

- 获取会话ID：pid_t getsid(pid_t pid);
  - 返回pid进程的会话ID；若pid为0，则返回当前进程的会话ID；成功时返回会话ID，出错时返回-1，并设errno值
- 设置会话ID：pid_t setsid();
  - 成功时返回新创建的会话ID，出错时返回-1，并设errno值

信　号

信号（signal）：进程通讯机制

- 信号是发送给进程的特殊异步消息
- 当进程接收到信息时立即处理，此时并不需要完成当前函数调用甚至当前代码行
- Linux系统中有多种信号，各具有不同的意义；系统以数字标识不同的信号，程序一般以名称引用之

系统信号

- 缺省处理逻辑：终止进程，生成内核转储文件
- 使用“kill –l”命令可查看操作系统支持的信号列表，不同的系统可能有所不同

系统信号表

信　号	值	缺省动作	含　义
SIGHUP	1	终止进程	终端的挂断或进程死亡
SIGINT	2	终止进程	来自键盘的中断信号，通常为Ctrl+C
SIGQUIT	3	内核转储	来自键盘的离开信号
SIGILL	4	内核转储	非法指令
SIGTRAP	5	内核转储	断点或其他陷阱指令，用于调试器
SIGABRT	6	内核转储	来自abort的异常信号
SIGBUS	7	内核转储	总线错误（内存访问错误）
SIGFPE	8	内核转储	浮点异常
SIGKILL	9	终止进程	杀死进程
SIGUSR1	10	终止进程	用户自定义信号1

系统信号表

信　号	值	缺省动作	含　义
SIGSEGV	11	内核转储	段非法错误（内存访问无效）
SIGUSR2	12	终止进程	用户自定义信号2
SIGPIPE	13	终止进程	管道损坏：向一个没有读进程的管道写数据
SIGALRM	14	终止进程	计时器定时信号
SIGTERM	15	终止进程	进程终止信号
SIGSTKFLT	16	终止进程	协处理器堆栈错误（不使用）
SIGCHLD	17	忽略	子进程停止或终止
SIGCONT	18	忽略	如果停止，继续执行
SIGSTOP	19	停止进程	非来自终端的停止信号
SIGTSTP	20	停止进程	来自终端的停止信号，通常为Ctrl+Z

系统信号表

信　号	值	缺省动作	含　义
SIGTTIN	21	停止进程	后台进程读终端
SIGTTOU	22	停止进程	后台进程写终端
SIGURG	23	忽略	有紧急数据到达套接字信号
SIGXCPU	24	内核转储	超过CPU时限
SIGXFSZ	25	内核转储	超过文件长度限制
SIGVTALRM	26	终止进程	虚拟计时器定时信号（进程占用CPU时间）
SIGPROF	27	终止进程	计时器定时信号（程序占用CPU时间和系统调度时间）
SIGWINCH	28	忽略	窗口大小改变
SIGIO	29	终止进程	描述符上可以进行I/O操作
SIGPWR	30	终止进程	电力故障
SIGSYS	31	内核转储	非法系统调用

信号处理

进程间发送的信号

- SIGTERM、SIGKILL：终止进程信号，前者是请求（接收信号的进程可以忽略之），后者是强制
- SIGUSR1、SIGUSR2：用户自定义信号，可用于向进程发送命令

信号处理

- 进程接收到信号后，根据信号配置进行处理
- 缺省配置：在程序没有处理时，确定信号该如何处理
- 程序处理信号的方式：按照信号处理例程的函数指针类型定义一个函数，然后调用

信号处理

sigaction()函数：设置信号配置

- 原型：int sigaction(int signum, const struct sigaction * act, struct sigaction * oldact);
- signum为信号编号，act和oldact分别为指向信号结构体struct sigaction的指针，前者为新配置，后者为需要保存的老配置

信号结构体struct sigaction

- 最重要的成员为sa_handler，其取值为SIG_DFL（使用信号缺省配置）、SIG_IGN（忽略该信号）或指向信号处理例程的函数指针（以信号编号为参数，无返回值）

信号处理

处理信号时的注意事项

- 信号是异步操作，当处理信号时，主程序非常脆弱
- 信号处理例程应尽可能短小，它甚至有可能会被新信号所中断
- 尽量不要在信号处理例程中实施I/O操作，也不要频繁调用系统函数或库函数
- 在信号处理例程中进行复杂的赋值操作也是危险的，它可能不是原子操作，因而有可能在执行期间被中断
- 如果需要赋值，使用sig_atomic_t类型的全局变量（在Linux中等价于int，亦即允许整数或指针赋值，更大尺寸数据不允许）

信号处理

```
#include <signal.h>          // 处理信号的头文件
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <iostream>
sig_atomic_t  sigusr1_count = 0;
extern "C"  {  void OnSigUsr1( int signal_number )  {  ++sigusr1_count;  }  }
int main ()
{
  std::cout  << "pid: " << (int)getpid() << std::endl;
  struct sigaction sa;
  memset( &sa, 0, sizeof(sa) );
  sa.sa_handler = &OnSigUsr1;
  sigaction( SIGUSR1, &sa, NULL );
  sleep( 100 );    // 在终端中输入kill –s SIGUSR1 pid，信号计数器将递增
  std::cout << "SIGUSR1 counts: " << sigusr1_count << std::endl;
  return 0;
}
```

进程管理

- 进程创建
- 进程调度
- 进程终止
- 僵尸进程
- 子进程异步清除
- 守护进程

进程创建

system()函数：用于在程序中执行一条命令

- 原型：int system(const char * cmd);
- 在Bourne shell中，系统会创建一个子进程运行被调命令；返回值为shell的退出状态；如果shell不能运行，返回127；如果发生其他错误，返回-1
- 示例：int ret_val = system("ls -l /");

fork()函数：创建当前进程的副本作为子进程

- 原型： pid_t fork();
- 返回值为0（新创建的子进程）和子进程的PID（父进程）

进程创建

使用fork()函数创建进程副本

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
using namespace std;
int main ()
{
  cout << "the main program process ID is " << (int)getpid() << endl;
  pid_t  child_pid = fork();
  if( child_pid != 0 )
  {
    cout << "this is the parent process, with id " << (int)getpid() << endl;
    cout << "the child’ s process ID is " << (int)child_pid << endl;
  }
  else
    cout << "this is the child process, with id " << (int)getpid() <<endl;
  return 0;
}
```

执行命令

exec()函数簇原型

- int execl(const char * path, const char * arg, ...);
- int execlp(const char * file, const char * arg, ...);
- int execle(const char * path, const char * arg, ..., char * const envp[]);
- int execv(const char * path, char * const argv[]);
- int execvp(const char * file, char * const argv[]);
- int execvpe(const char * file, char * const argv[], char * const envp[]);

执行命令

exec()函数说明

- 函数名称中包含字母“p”（execvp、execlp）：接受程序名作为参数，在当前执行路径中按程序名查找；不包含字母“p”的，必须提供程序的完整路径
- 函数名称中包含字母“v”（execv、 execvp、execve）：接受以NULL结尾的字符串数组格式的参数列表
- 函数名称中包含字母“l”（execl、 execlp、execle）：接受C格式的可变参数列表
- 函数名称中包含字母“e”（execve、execle）：接受一个附加的环境参数列表，参数格式为NULL结尾的字符串数组，且字符串的格式为“VARIABLE=value”

执行命令

基本模式：在程序中调用fork()创建一个子进程，然后调用exec()在子进程中执行命令

```
#include <iostream>
#include <cstdlib>
#include <sys/types.h>
#include <unistd.h>
int spawn( char * program, char ** args );
int main ()
{
  char * args[] = { "ls", "-l", "/", NULL };
  spawn( "ls", args );
  cout << "Done!\n";
  return 0;
}
```

执行命令

```
// 创建一个子进程运行新程序
// program为程序名，arg_list为程序的参数列表；返回值为子进程id
int spawn( char * program, char ** args )
{
  pid_t child_pid = fork();          // 复制进程
  if( child_pid != 0 )               // 此为父进程
    return child_pid;
  else                               // 此为子进程
  {
    execvp( program, args );         // 执行程序，按路径查找
    // 只有发生错误时，该函数才返回
    std::cerr << "Error occurred when executing execvp.\n";
    abort ();
  }
}
```

进程调度

进程调度策略：先进先出，时间片轮转，普通调度，批调度，高优先级抢先

- **子进程与父进程的调度没有固定顺序；不能假设子进程一定会在父进程之后执行，也不能假设子进程一定会在父进程之前结束**

进程调度策略函数：头文件“sched.h”

- **获取进程调度策略：int sched_getscheduler(pid_t pid);**
- **设置进程调度策略：int sched_setscheduler(pid_t pid, int policy, const struct sched_param * sp);**
- **获取进程调度参数：int sched_getparam(pid_t pid, struct sched_param * sp);**
- **设置进程调度参数：int sched_setparam(pid_t pid, const struct sched_param * sp);**

进程调度

进程优先级调整：头文件“sys/time.h”和“sys/resource.h”

- 改变进程优先级：int nice(int inc);（头文件“unistd.h”）
- 获取进程优先级：int getpriority(int which, int who);
- 设置进程优先级：int setpriority(int which, int who, int prio);

处理器亲和性：头文件“sched.h”

- 获取进程的处理器亲和性：int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t * mask);
- 设置进程的处理器亲和性：int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t * mask);

进程终止

终止进程函数：kill()

- 头文件“sys/types.h”和“signal.h”
- 原型： int kill(pid_t pid, int sig);
- 函数参数：pid为子进程ID，sig应为进程终止信号SIGTERM

等待进程结束函数：wait()

- 原型：pid_t wait(int * status); pid_t waitpid(pid_t pid, int * status, int options);
- 阻塞主调进程，直到一个子进程结束
- WEXITSTATUS宏：查看子进程的退出码
- WIFEXITED宏：确定子进程的退出状态是正常退出，还是未处理信号导致的意外死亡

进程终止

```
#include <iostream>
#include <cstdlib>
#include <sys/types.h>
#include <sys/wait.h>    //  必须包含此头文件，否则与wait共用体冲突
#include <unistd.h>
int spawn( char * program, char ** arg_list );
int main ()
{
  char * arg_list[] = { "ls", "-l", "/", NULL };
  spawn( "ls", arg_list );
  int child_status;
  wait( &child_status );    //  等待子进程结束
  if( WIFEXITED( child_status ) )    //  判断子进程是否正常退出
    cout << "Exited normally with " << WEXITSTATUS(child_status) <<endl;
  else
    cout << "Exited abnormally." << endl;
  cout << "Done!\n";
  return 0;
}
```

僵尸进程

子进程已结束，但父进程未调用wait()函数等待

- 子进程已终止，但没有被正确清除，成为僵尸进程

清除子进程的手段

- 父进程调用wait()函数可确保子进程被清除
- 即使子进程在父进程调用wait()函数前已死亡（成为僵尸），其退出状态也可以被抽取出来，然后被清除
- 未清除的子进程自动被init进程收养

僵尸进程

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{
  pid_t child_pid;
  child_pid = fork();
  if( child_pid > 0 )     //  父进程，速度睡眠六十秒
    sleep( 60 );
  else                    //  子进程，立即退出
    exit( 0 );
  return 0;
}
```

子进程异步清除

SIGCHLD信号：子进程终止时，向父进程自动发送，编写此信号处理例程，异步清除子进程

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
sig_atomic_t child_exit_status;
extern "C"  {
void CleanUp( int sig_num )
{
  int status;
  wait( &status );                    //  清除子进程
  child_exit_status = status;         //  存储子进程的状态
}
}
```

子进程异步清除

```
int main ()
{
  //  处理SIGCHLD信号
  struct sigaction sa;
  memset( &sa, 0, sizeof(sa) );
  sa.sa_handler = &CleanUp;
  sigaction( SIGCHLD, &sa, NULL );

  //  正常处理代码在此，例如调用fork()创建子进程

  return 0;
}
```

守护进程

创建守护进程的步骤

- 创建新进程：新进程将成为未来的守护进程
- 守护进程的父进程退出：保证祖父进程确认父进程已结束，且守护进程不是组长进程
- 守护进程创建新进程组和新会话：并成为两者的首进程，此时刚创建的新会话还没有关联控制终端
- 改变工作目录：守护进程一般随系统启动，工作目录不应继续使用继承的工作目录
- 重设文件权限掩码：不需要继承文件权限掩码
- 关闭所有文件描述符：不需要继承任何打开的文件描述符
- 标准流重定向到/dev/null

守护进程

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>

int main()
{
  pid_t pid = fork();
  if( pid == -1 )                    return -1;
  else if( pid != 0 )                exit( EXIT_SUCCESS );
```

守护进程

```
    // 子进程
    if( setsid() == -1 )              return -2;
    // 设置工作目录
    if( chdir( "/" ) == -1 )          return -3;
    // 重设文件权限掩码
    umask( 0 );
    // 关闭文件描述符
    for( int i = 0; i < 3; i++ )      close( i );
    // 重定向标准流
    open( "/dev/null", O_RDWR );      // stdin
    dup( 0 );                         // stdout
    dup( 0 );                         // stderr
    // 守护进程的实际工作代码在此
    return 0;
}
```

守护进程

守护进程创建函数daemon()

- 实现了前述功能，减轻编写守护进程的负担
- 原型：int daemon(int nochdir, int noclose);
- 参数：若nochdir非0，不更改工作目录；若noclose非0，不关闭所有打开的文件描述符；一般均设为0
- 返回值：成功时返回0，失败时返回-1，并设置errno值

进程间通信

管道：相关进程间的顺序通信

进程信号量：进程间通信的同步控制机制

共享内存：允许多个进程读写同一片内存区域

映射内存：与共享内存意义相同，但与文件相关联

消息队列：在进程间传递二进制块数据

套接字：支持无关进程，甚至不同计算机进行通信

管　道

管道（pipe）的性质与意义

- 管道是允许单向通信的自动同步设备（半双工）
- 数据在写入端写入，在读取端读取
- 管道为串行设备，数据的读取顺序与写入顺序完全相同

管道的用途

- 只能用于有亲缘关系的进程，例如父进程和子进程之间通信

注意事项

- 管道的数据容量有限，一般为一个内存页面
- 如果写入速度超过读取速度，写入进程将阻塞，直到容量有空闲
- 如果读取速度超过写入速度，读取进程将阻塞，直到管道有数据

管　道

pipe函数：创建管道

- 头文件：“unistd.h”和“fcntl.h”
- 原型：int pipe(int pipefd[2]);
- 参数：一个包含两个元素的整数数组，元素类型为文件描述符，0号元为读取文件描述符，1号元为写入文件描述符
- 返回值：成功时返回0，不成功时返回-1，并设置errno值

```
int pipe_fds[2];
int read_fd;
int write_fd;
pipe( pipe_fds );
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

管道通信

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

const int buf_size = 4096;

//  向stream中写入count次msg
void Write( const char * msg, int count, FILE * stream )
{
  for( ; count > 0; --count )
  {
    fprintf( stream, "%s\n", msg );
    fflush( stream );
    sleep (1);
  }
}
```

管道通信

```
// 从stream中读取数据
void Read( FILE * stream )
{
  char buf[buf_size];
  // 一直读取到流的尾部
  while( !feof(stream) && !ferror(stream) && fgets(buf, sizeof(buf), stream) != NULL )
  {
    fprintf( stdout, "Data received: \n" );
    fputs( buf, stdout );
  }
}

int main()
{
  int fds[2];
  pipe( fds );                    // 创建管道
  pid_t pid = fork();             // 创建子进程
```

管道通信

```
  if( pid == 0 ) {                  //  子进程
    close( fds[1] );                //  只读取，关闭管道写入端
    //  将文件描述符转换为FILE *，以方便C/C++标准库函数处理
    FILE * stream = fdopen( fds[0], "r" );
    Read( stream );                 //  从流中读取数据
    close( fds[0] );                //  关闭管道读取端
  }
  else if( pid > 0 ) {              //  父进程
    char buf[buf_size];             //  数据缓冲区，末尾封装两个 ‘\0’
    for( int i = 0; i < buf_size-2; i++ )    buf[i] = 'A' + i % 26;
    buf[buf_size-1] = buf[buf_size-2] = '\0';
    close( fds[0] );                //  只写入，关闭管道读取端
    FILE * stream = fdopen( fds[1], "w" );
    Write( buf, 3, stream );
    close( fds[1] );                //  关闭管道写入端
  }
  return 0;
}
```

管道重定向

等位文件描述符

- 共享相同的文件位置和状态标志设置

dup()函数：将两个文件描述符等位处理

- 原型：int dup(int oldfd); int dup2(int oldfd, int newfd);
- 参数：创建oldfd的一份拷贝，单参数版本选择数值最小的未用文件描述符作为新的文件描述符；双参数版本使用newfd作为新的文件描述符，拷贝前尝试关闭newfd
- 返回值：成功时返回新文件描述符，失败时返回-1，并设errno值
- 示例：dup2(fd, STDIN_FILENO)关闭标准输入流，然后作为fd的副本重新打开

管道重定向

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
const int buf_size = 4096;
int main ()
{
  int fds[2];
  pipe( fds );                          //  创建管道
  pid_t pid = fork();
  if( pid == (pid_t)0 )                 //  子进程
  {
    close( fds[0] );                    //  关闭管道读取端
    dup2( fds[1], STDOUT_FILENO );              //  管道挂接到标准输出流
    char * args[] = { "ls", "-l", "/", NULL };  //  使用“ls”命令替换子进程
    execvp( args[0], args );
  }
```

管道重定向

```
  else                                  //  父进程
  {
    close( fds[1] );                    //  关闭管道写入端
    char buf[buf_size];
    FILE * stream = fdopen( fds[0], "r" );          //  以读模式打开管道读取端，返回文件指针
    fprintf( stdout, "Data received: \n" );
    //  在流未结束，未发生读取错误，且能从流中正常读取字符串时，输出读取到的字符串
    while( !feof(stream) && !ferror(stream) && fgets(buf, sizeof(buf), stream) != NULL )
    {
      fputs( buf, stdout );
    }
    close( fds[0] );                    //  关闭管道读取端
    waitpid( pid, NULL, 0 );  //  等待子进程结束
  }
  return 0;
}
```

进程信号量

进程信号量： System V信号量

- 可以使用同步机制确定进程的先后执行关系
- 头文件：“sys/types.h”、“sys/ipc.h”和“sys/sem.h”

信号量的定义

- 信号量是一类特殊的计数器，其值为非负整数，用于进程或线程同步

信号量的操作

- 等待（wait）操作（P）：信号量的值递减1后返回；如果值为0，则阻塞操作，直到信号量值为正（其他进程或线程释放了信号量），然后递减1后返回
- 发布（post）操作（V）：信号量的值递增1后返回；如果信号量值原为0，则其中一个等待该信号量的进程或线程取消阻塞

进程信号量

Linux信号量实现：两个版本

- 进程信号量多用于进程同步，POSIX标准实现多用于线程同步

使用进程信号量时的注意事项

- 每次创建和管理的进程信号量不是一个，而是一个集合（数组），该集合可能包含多个进程信号量
- 使用键值key关联进程信号量集，但进程信号量集本身由进程信号量的标识符semid标识，函数调用时几乎总是使用semid——可以这么理解：semid对内，key对外

获取进程信号量

semget()函数：创建或获取进程信号量集

- 原型：int semget(key_t key, int nsems, int semflg);
- 参数：key为键值，多个进程可以通过此键值访问同一进程信号量；nsems为需要创建的进程信号量集的进程信号量元素个数（不是进程信号量的信号数），semflg为访问标志
- 返回值：成功时返回进程信号量集的标识符，失败时返回-1，并设置errno值
- 要获取已分配的进程信号量集，使用原先键值查询，此时进程信号量集的元素个数可设置为0
- 键值IPC_PRIVATE用于创建当前进程的私有进程信号量集
- 使用IPC_CREAT和IPC_EXCL创建进程信号量集，后者要求创建新的唯一的进程信号量集，若其已存在，则出错

控制进程信号量

控制和管理进程信号量集

- 原型：int semctl(int semid, int semnum, int cmd, …);
- 参数：semid为进程信号量集的标识符，semnum为进程信号量集的元素下标，cmd为指定操作，第四个参数arg可有可无，与cmd有关
- 返回值：成功时与cmd有关，失败时返回-1，并设置errno值
- 调用semctl()函数的进程的有效用户ID必须与分配进程信号量集的用户权限匹配

清除进程信号量

释放（删除）进程信号量集：IPC_RMID

- 最后一个使用进程信号量的进程负责清除进程信号量集
- 进程信号量集释放后，内存自动释放
- 调用说明：cmd为IPC_RMID，semnum被忽略，arg不需要；如果需要arg，定义union semun类型的变量并作为参数，部分系统可能未定义union semun类型，需按如下格式补充定义：

```
union semun
{
  int                   val;    //  供SETVAL使用的值
  struct semid_ds *     buf;    //  供IPC_STAT、IPC_SET使用的缓冲区
  unsigned short int *  array;  //  供GETALL、SETALL使用的数组
  struct seminfo *      __buf;  //  供IPC_INFO使用的缓冲区
};
```

初始化进程信号量

初始化进程信号量集：SETALL

- 第一个参数semid为进程信号量集的标识符，第二个参数semnum为0，第三个参数cmd为SETALL，第四个参数arg必须为union semun类型的数据对象
- union semun的array字段：指向无符号短整型数组首元素的指针，该数组保存进程信号量集的所有信号量的信号数

其他常用命令参数

- IPC_STAT/IPC_SET（获取或设置进程信号量信息）、GETALL（获取全部信号量的信号数）、 GETVAL/SETVAL（获取或设置单个信号量的信号数）等

获取与释放

```
// 获取与key关联的二元信号量集，必要时会分配之
int AcquireBinarySemaphore( key_t key, int sem_flags )
{
  return semget( key, 1, sem_flags );
}
// 释放二元信号量集，单一元素
int ReleaseBinarySemaphore( int semid )
{
  union semun  ignored;
  return semctl( semid, 1, IPC_RMID, ignored );
}
// 初始化二元信号量集，单一元素，信号量初始值为1
int InitializeBinarySemaphore( int semid )
{
  unsigned short int values[1] = { 1 };
  union semun needed = { .array = values };
  return semctl( semid, 0, SETALL, needed );
}
```

等待与发布

等待与发布进程信号量函数semop()

- 原型：int semop(int semid, struct sembuf *sops, size_t nsops);
- 参数：semid为待操作的进程信号量集的标识符；sops为操作数组，nsops为操作数组的元素个数
- 返回值：成功时为进程信号量集的标识符，失败时返回-1，并设置errno值

等待与发布

struct sembuf类型的成员

- sem_num：需要操作的进程信号量集中的信号量元素下标
- sem_op：指定信号量操作的整数（递增或递减信号量的信号数）
  - 如果sem_op为正数，则立即加到信号量上（V操作）
  - 如果sem_op为负数，则从信号量上减去（P操作）
  - 如果会使结果为负数，则阻塞进程，直到信号量的信号数不小于sem_op的绝对值
  - 如果sem_op为0，则阻塞进程，直到信号量的信号数为0
- sem_flg：指定IPC_NOWAIT则不阻塞进程，指定SEM_UNDO则在进程退出时取消操作

等待与发布

```
// P原语：等待二元信号量，信号数非正时阻塞
int WaitBinarySemaphore( int semid )
{
  struct sembuf ops[1];
  ops[0].sem_num = 0;
  ops[0].sem_op = -1;
  ops[0].sem_flg = SEM_UNDO;
  return semop( semid, ops, 1 );
}
// V原语：发布二元信号量，增加信号数后立即返回
int PostBinarySemaphore( int semid )
{
  struct sembuf ops[1];
  ops[0].sem_num = 0;
  ops[0].sem_op = 1;
  ops[0].sem_flg = SEM_UNDO;
  return semop( semid, ops, 1 );
}
```

共享内存

共享内存的意义：快捷方便的本地通信机制

- 头文件：“sys/ipc.h”和“sys/shm.h”

共享内存编程原则

- 系统没有对共享内存操作提供任何缺省同步行为
- 如果需要，程序员自主设计同步策略：使用进程信号量

共享内存使用过程

- 某个进程分配一个内存段，其他需要访问该内存段的进程连接（attach）该内存段
- 完成访问后，进程拆卸（detach）该内存段
- 某个时刻，一个进程释放该内存段

共享内存

Linux内存模型

- 每个进程的虚拟地址空间按页（page）编址，页缺省为4096字节（不同硬件架构和操作系统可能不同，使用getpagesize()函数获取系统值）
- 每个进程维持从内存地址到虚拟页面地址的映射
- 多个进程可能使用同一虚拟页面，同样的数据在不同进程中的地址并不需要相同
- 分配新的共享内存段将创建虚拟内存页面，其他进程连接该共享内存段即可访问
- 共享内存段的分配只有由一个进程负责，释放也同样

获取共享内存

shmget()函数：获取或分配一段共享内存

- 原型：int shmget(key_t key, size_t size, int shmflg);
- 参数：key为内存段整数键值，size为内存段分配的字节数（圆整至4096字节整数倍），shmflg为创建共享内存段的位标志

键值参数key

- 其他进程通过键值key访问该内存段，任意设定的键值可能和其他进程的共享内存段键值冲突，使用IPC_PRIVATE以确保无冲突

创建标志：IPC_CREAT（创建）、IPC_EXCL（独占）

- 后者与前者合并使用，如果键值已使用，则创建失败
- 如果未设IPC_EXCL，则在键值已经存在时，返回其代表的共享内存段，而不是创建一个新的共享内存段

获取共享内存

位标志参数

- 模式标志：以9位数字表示宿主、组用户和其他人的访问控制权
- 常数位于头文件“sys/stat.h”

返回值：共享内存段的标识符

常用模式常数

- S_IRUSR和S_IWUSR分别表示共享内存段宿主的读写权限
- S_IRGRP和S_IWGRP分别表示共享内存段组用户的读写权限
- S_IROTH和S_IWOTH分别表示共享内存段其他人的读写权限

调用示例

- int seg_id = shmget(shm_key, getpagesize(), IPC_CREAT | S_IRUSR | S_IWUSER);

连接与拆卸共享内存

shmat()函数：连接共享内存

- 原型：void * shmat(int shmid, const void * shmaddr, int shmflg);
- 参数： shmid为共享内存段标识符（shmget()的返回值）， shmaddr为指针，指向共享内存段的映射地址，如果传递NULL，Linux自动选择合适地址， shmflg为连接标志
- 返回值：成功时返回所连接的共享内存段的地址

连接标志

- SHM_RND：shmaddr指定的映射地址向下圆整到页面尺寸的整数倍；如果未指定，则传递shmaddr时必须手工对齐页面地址
- SHM_RDONLY：共享内存段组只读

shmdt()函数：拆卸共享内存段

- 原型：int shmdt(const void * shmaddr);

使用共享内存

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main ()
{
  struct shmid_ds  shmbuf;
  int seg_size;
  const int shared_size = 0x6400;
  //  分配共享内存段
  int seg_id = shmget( IPC_PRIVATE, shared_size, IPC_CREAT | IPC_EXCL | S_IRUSR |
     S_IWUSR );
  //  连接共享内存段
  char * shared_mem = ( char * )shmat( seg_id, 0, 0 );
  printf( "Shared memory attached at %p\n", shared_mem );
  //  获取段尺寸信息
  shmctl( seg_id, IPC_STAT, &shmbuf );
  seg_size = shmbuf.shm_segsz;
  printf( "Segment size: %d\n", seg_size );
```

使用共享内存

```
    // 向共享内存区段写入字符串
    sprintf( shared_mem, "Hello, world." );
    // 拆卸共享内存区段
    shmdt( shared_mem );
    // 在不同的地址处重新连接共享内存区段
    shared_mem = ( char * )shmat( seg_id, ( void * )0x5000000, 0 );
    printf( "Shared memory reattached at %p\n", shared_mem );
    // 获取共享内存区段中的信息并打印
    printf( "%s\n", shared_mem );
    // 拆卸共享内存区段
    shmdt( shared_mem );
    // 释放共享内存区段，与semctl类似
    shmctl( seg_id, IPC_RMID, 0 );
    return 0;
}
```

映射内存

mmap()函数：头文件“sys/mman.h”

- 映射共享文件到内存；文件被分割成页面大小装载；使用内存读写操作访问文件，速度更快；对映射内存的写入自动反映到文件中
- 原型：void * mmap(void * addr, size_t length, int prot, int flags, int fd, off_t offset);

函数参数

- addr：映射目的地的内存地址，NULL表示由Linux自动选择合适的内存地址
- length：映射内存的大小，以字节为单位
- prot：指定映射内存的保护权限，为PROT_READ（允许读取）、PROT_WRITE（允许写入）、 PROT_EXEC（允许执行）或以上三者的组合

映射内存

函数参数（续）

- flags：附加选项标志；为MAP_FIXED（如果指定此标志，则Linux使用用户指定的地址映射文件，地址本身必须页对齐）、MAP_PRIVATE（内存写入不回写至外部文件，本进程保留写入的文件副本）、 MAP_SHARED（内存写入立即反映到映射文件中）； MAP_PRIVATE与MAP_SHARED不能混用
- fd：待映射文件的文件描述符
- offset：指定映射数据在文件中的偏移量

函数返回值

- 成功调用时返回映射内存的基地址，失败时返回MAP_FAILED

映射内存

munmap()函数：释放映射内存

- 原型：int * munmap(void * addr, size_t length);
- 参数：addr为映射内存的基地址；length为映射内存的大小
- 返回值：成功时返回0，失败时返回-1并设errno值

msync()函数：映射内存同步

- 原型：int msync(void * addr, size_t length, int flags);
- 参数：addr为映射内存基地址，length为映射内存大小，flags为同步标志，
  MS_ASYNC（数据更新被调度，但函数返回前并不一定会被执行）；
  MS_SYNC（数据更新立即执行，在完成前调用进程被阻塞）；
  MS_INVALIDATE（通知其他进程数据已无效，并自动提供新数据）；
  MS_ASYNC与MS_SYNC不能混用
- 返回值：成功时返回0，失败时返回-1并设errno值

读写映射内存

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <wait.h>
#include <iostream>
#include <iomanip>
const int mapped_size = 4096;
const int mapped_count = mapped_size / sizeof(int);
int main( int argc, char * const argv[] )
{
  //  打开文件作为内存映射的对象，确保文件尺寸足够存储1024个整数
  int fd = open( argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR );
  lseek( fd, mapped_size - 1, SEEK_SET );
  write( fd, "", 1 );
  lseek( fd, 0, SEEK_SET );
  int * base = ( int * )mmap( 0, mapped_size, PROT_READ | PROT_WRITE,
     MAP_SHARED | MAP_ANONYMOUS, fd, 0 );
  close( fd );                        //  创建映射内存后，关闭文件的文件描述符
```

读写映射内存

```
pid_t pid = fork();
if( pid == (pid_t)0 )          //  子进程写入数据
{
  //  写入数据0～1023
  for( int i = 0, * p = base; i < mapped_count; *p++ = i++ )
    ;
  munmap( base, mapped_size );
}
else if( pid > (pid_t)0 )      // 父进程读取数据
{
  sleep( 10 );                 //  等待10秒
  for( int i = 0, *p = base; i < mapped_count; i++, p++ )
    std::cout << std::setw(5) << *p << " ";
  std::cout << std::endl;
  munmap( base, mapped_size );
}
return 0;
}
```

消息队列

消息队列：在两个进程间传递二进制块数据

- 数据块具有类别信息，接收方可根据消息类别有选择地接收
- 头文件：“sys/type.h”、“sys/ipc.h”和“sys/msg.h”

msgget()函数：创建或获取消息队列

- 原型：int msgget(ket_t key, int msgflg);
- 参数：key为键值，标识全局唯一的消息队列；msgflg为创建标志，与semget()的标志相同
- 返回值：成功时返回正整数作为消息队列的标识符，失败时返回-1，并设errno值
- 如果用于创建消息队列，相关内核数据结构struct msqid_ds将被创建并初始化

消息队列

msgsnd()函数：将消息添加到消息队列中

- 原型：int msgsnd(int msqid, const void * msg_ptr, size_t msg_sz, int msgflg);
- 参数： msqid为msgget()返回的消息队列标识符；msg_ptr指向准备发送的消息； msg_sz为消息数据长度；msgflg控制消息发送行为，一般仅支持IPC_NOWAIT标志，即以非阻塞的方式发送消息
- 返回值：成功时返回0，失败时返回-1，并设errno值

消息缓冲区结构

- msg_ptr指向的数据结构如右
- mtype为消息类别，必须为正整数
- mtext为消息数据，msg_sz为其实际长度

```
struct msgbuf
{
  long int mtype;
  char mtext[512];
};
```

消息队列

msgrcv()函数：从消息队列中获取消息

- 原型：int msgrcv(int msqid, void * msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
- 参数： msqid为msgget()返回的消息队列标识符；msg_ptr用于存储接收的消息；msg_sz为消息数据长度；msgtype为消息类别；msgflg控制消息发送行为，可以为IPC_NOWAIT、MSG_EXCEPT（msgtype大于0时，读取第一个非msgtype类别的消息）和MSG_NOERROR的位或
- 返回值：成功时返回0，失败时返回-1，并设errno值
- 消息类别msgtype说明：为0则读取队列第一条消息，大于0则读取队列中第一条类别为msgtype的消息，小于0则读取队列中第一个类别比msgtype绝对值小的消息

消息队列

msgctl()函数：控制消息队列的某些属性

- 原型：int msgctl(int msqid, int cmd, struct msqid_ds * buf);
- 参数： msqid为msgget()返回的消息队列标识符；cmd指定要执行的命令，支持的命令有IPC_STAT、IPC_SET、IPC_RMID、IPC_INFO、MSG_INFO、MSG_STAT；buf的意义与cmd参数有关
- 返回值：成功时返回值取决于cmd参数，失败时返回-1，并设errno值

进程池

动机：为什么需要引入进程池？

- 进程需要频繁创建子进程，以执行特定任务
- 动态创建子进程的过程效率较低，客户响应速度较慢
- 动态创建的子进程一般只为单一客户提供服务，当客户较多时，系统中会存在大量子进程，进程切换的开销过高
- 动态创建的子进程为当前进程的完整映像，当前进程必须谨慎地管理系统资源，以防止子进程不适当地复制这些资源

什么是进程池？

- 主进程预先创建一组子进程，并统一管理
- 子进程运行同样代码，具有同样属性，个数多与CPU数目一致，很少超过CPU数目的两倍

进程池

进程池的工作原理是什么？

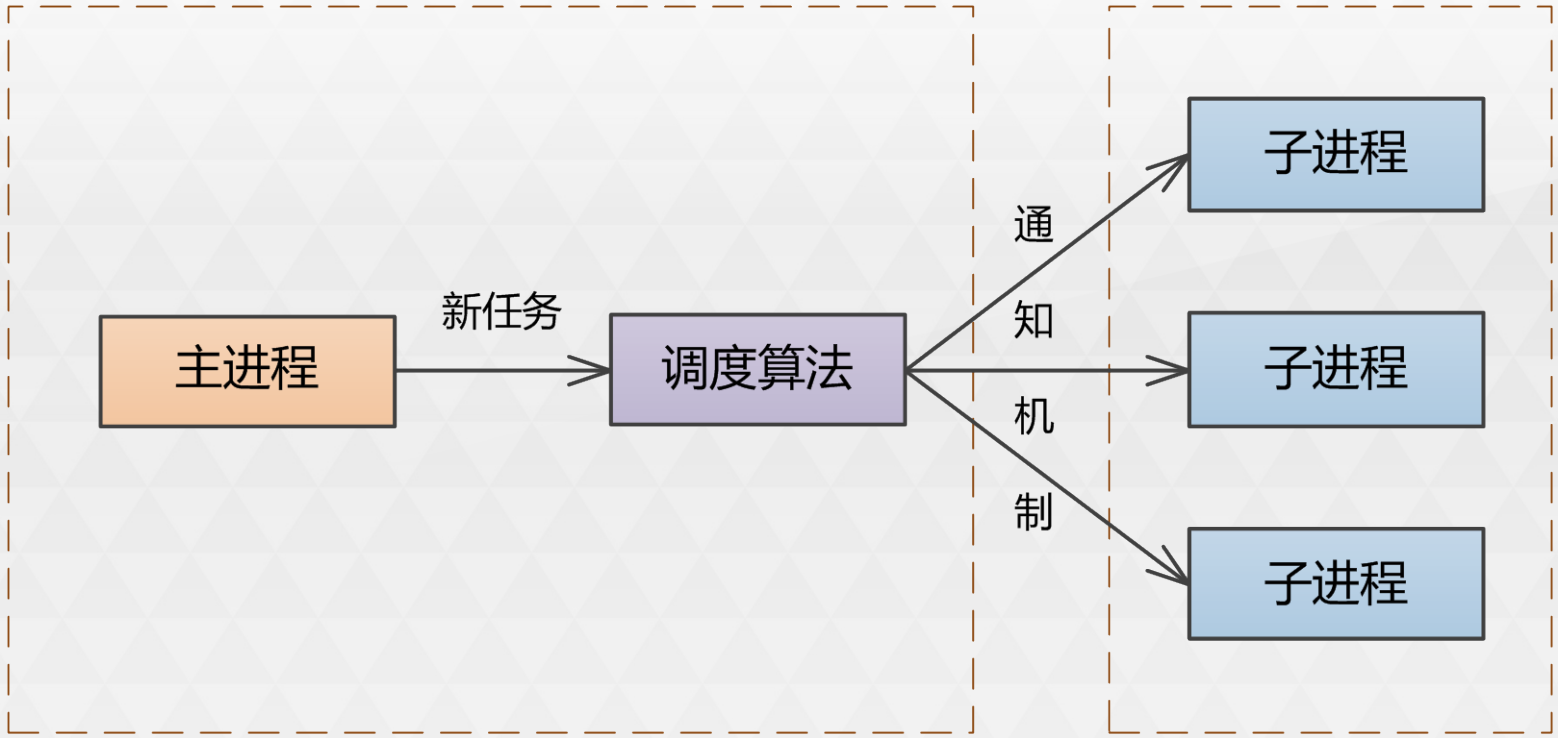
- 主进程充当服务器，子进程充当服务员，按照服务器的需要提供服务
- 在任务到达时，主进程选择一个子进程进行服务
- 相对于动态创建子进程，选择的代价显然更小——这些子进程未来还可以被复用
- 存在多种选择子进程的策略，如随机选择或轮值制度，如共享单一任务队列，还可以使用更加智能化的负载平衡技术
- 父子进程之间应该具有传递信息的通道，如管道、共享内存、消息队列等，也可能需要同步机制

编程实践

13.1 编写程序，调用fork()创建子进程，使用二元进程信号量进行同步。提示：在创建子进程前，使用IPC_PRIVATE创建新的二元进程信号量，其创建的进程信号量并不是该进程私有的，子进程可以通过复制的semid访问该二元进程信号量。父进程在等待子进程结束后释放该二元进程信号量。

13.2 编程实现进程池类ProcessPool。提示：（1）进程池应实现为类模板，从而可以针对不同的任务类别构造不同的进程池；（2）每个任务类别的进程池应实现为单子类；（3）可以统一事件源，即统一管理同类的任务序列，典型的策略是实现父子进程通信的信号管道；（4）调度算法随意。

编程实践



游双，Linux高性能服务器编程，机械工业出版社，2013年5月