第2章 进程与线程

实验目的

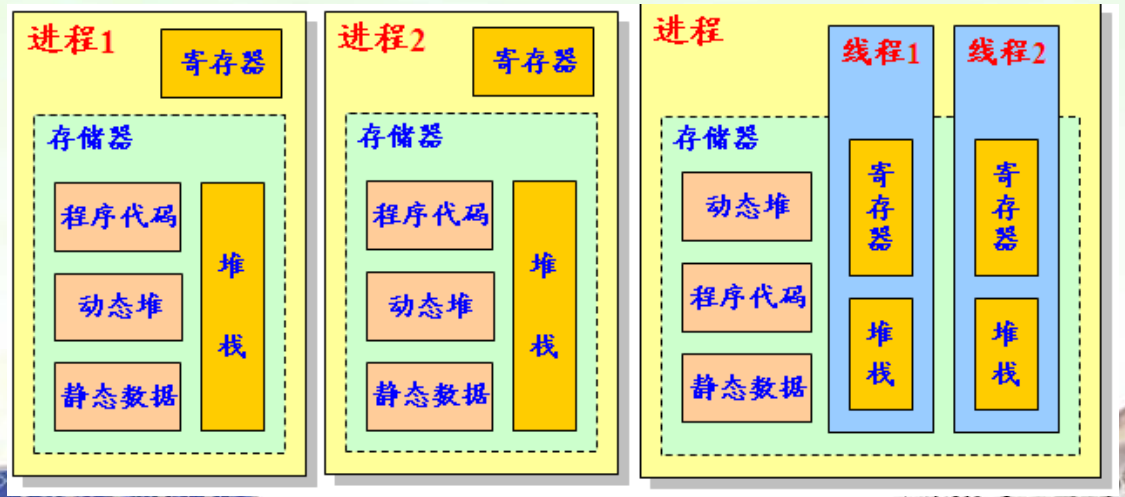
- 加深理解进程和程序、进程和线程的联系与区别
- 深入理解进程及线程的重要数据结构及实现机制
- 熟悉进程及线程的创建、执行、阻塞、唤醒、终止等控制方法
- 学会使用进程及线程开发应用程序

主要内容

- 背景知识
  - 进程与线程基本概念
  - 多进程编程
  - 多线程编程
- 实验内容
  - 创建进程
  - 线程共享进程中的数据
  - 多线程实现单词统计工具

进程及线程基本定义

- 进程（process）
  - 处于执行期的程序及其所包含资源的总称
    - ✓程序：可执行程序代码
    - ✓资源：打开文件、挂起信号、地址空间、数据段等
- 线程（thread）
  - 进程中活动的对象
    - ✓有独立的程序计数器、进程栈及一组进程寄存器



进程与线程的区别

- 从形态角度
  - 一个进程可包含一个或多个线程
- 从调度角度
  - 进程是资源分配的基本单位
  - 线程是处理器调度的独立单位
- 从虚拟化角度
  - 进程提供两种虚拟机制
    - ✓ 虚拟处理器：进程独享处理器的假象
    - ✓ 虚拟内存：进程拥有系统内所有内存资源的假象
  - 线程之间可共享虚拟内存，但各自拥有独立虚拟处理器

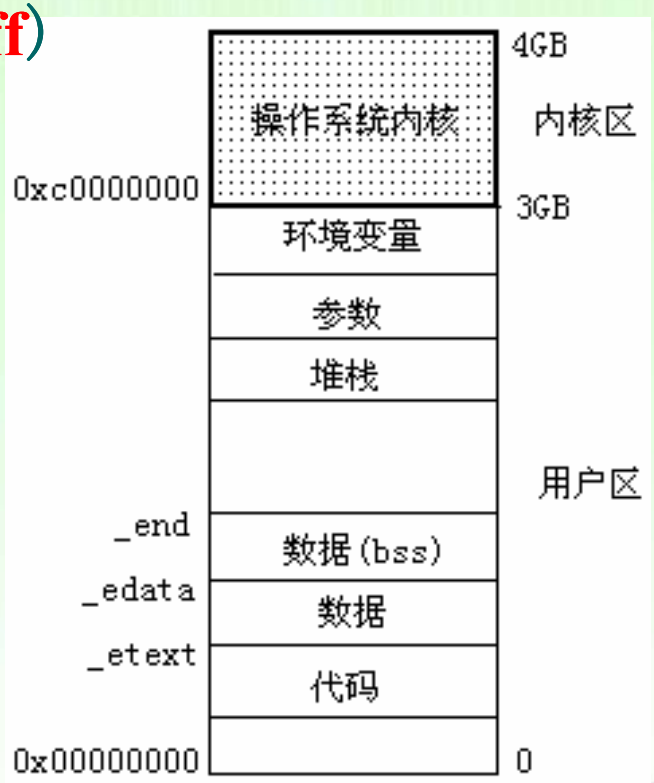
对Linux系统而言，线程只是一种特殊的进程！

进程构成要素

- 正文段
  - 存放进程运行的代码，描述进程需完成的功能
- 进程数据段
  - 存放正文段在执行期间所需的数据和工作区
    - ✓ 包括全局变量、动态分配的空间（调用malloc函数）
  - 用户栈也在该数据段开辟，存放函数调用时的栈帧、局部变量等
- 系统堆栈
  - 每个进程捆绑一个，进程在内核态下工作时使用
  - 保存中断现场、执行函数调用时的参数和返回地址等
  - 其中最重要的数据结构是进程控制块

进程虚拟地址结构

- 以Linux系统为例（共4G空间）
  - 用户空间(0x 0000 0000 ~ 0x bfff ffff)
    - 可执行映象
    - 进程运行时堆栈
    - 进程控制信息，如进程控制块
  - 内核空间(0x c000 0000以上)
    - 内核被映射进程内核空间
    - 只允许进程在核心态下访问



Linux进程描述符

- 进程描述符
  - 数据结构：task_struct
  - 定义位置：include/linux/sched.h
- 进程描述符向量结构
  - 数据结构：task[NR_TASKS]
  - 定义位置：include/linux/sched.h
  - 定义格式

```
struct task_struct *task[NR_TASKS] = {&init_task}
#define NR_TASKS 512
```

  - 说明
    - ✓全局变量NR_TASKS记录系统可容纳进程数，默认大小是512

Linux进程描述符的信息组成

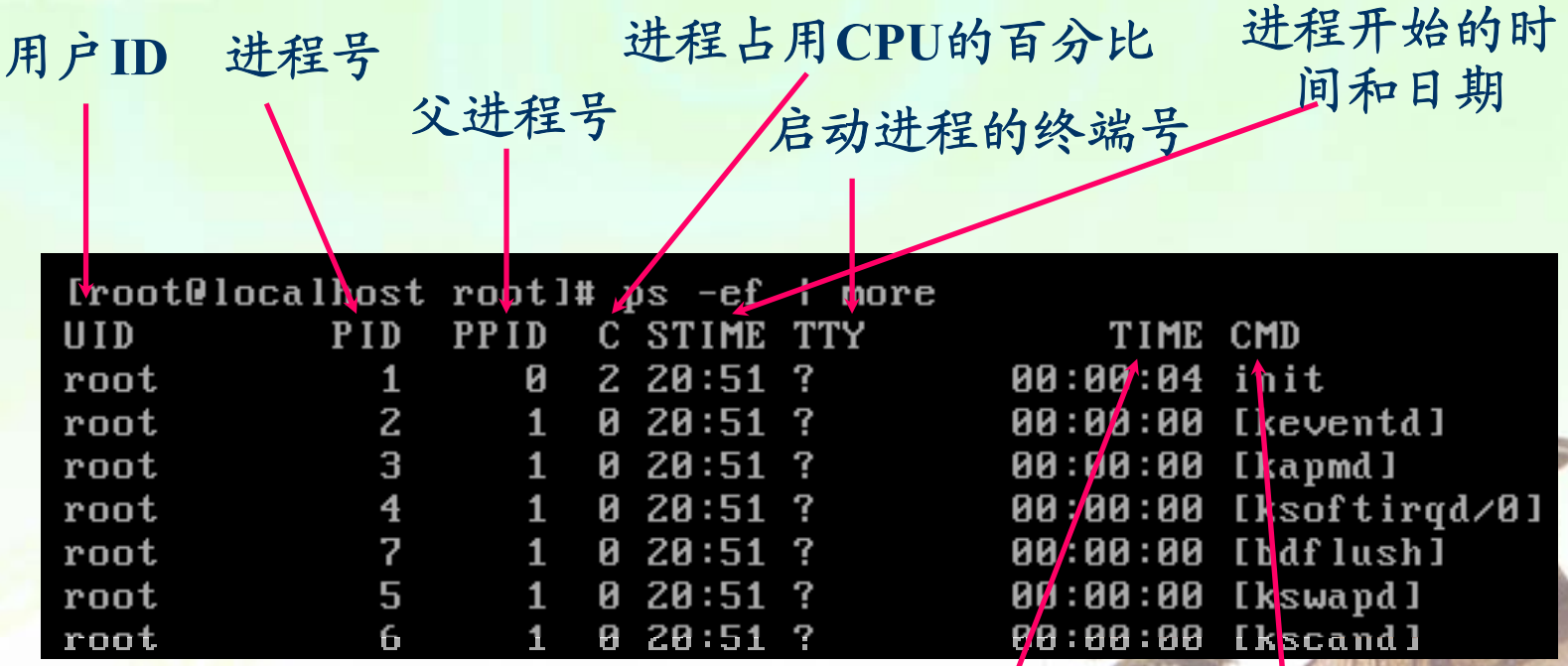
- 进程状态信息(state, flags, ptrace)
- 调度信息(static_prio, normal_proi, run_list, array, policy)
- 内存管理(mm, active_mm)
- 进程状态位信息(binfmt, exit_state, exit_code, exit_signal)
- 身份信息(pid, tgid, uid, suid, fsuid, gid, egid, sgid, fsgid)
- 家族信息(real_parent, parent, children, sibling)
- 进程耗间信息(realtime, utime, stime, starttime)
- 时钟信息(it_prof_expires, it_virt_expires, it_sched_expires)
- 文件系统信息(link_count, fs, files)
- IPC信息(sysvsem, signal, sighand, blocked, sigmask, pending)

进程标识

- 成员名：pid_t pid
- 功能
  - 内核通过pid标识每个进程
  - pid与进程描述符之间有严格的一一对应关系
- 数据类型说明
  - pid_t实际上是一个int类型
  - 取值范围：0～32767
  - 最大值修改：/proc/sys/kernel/pid_max
  - 生成新pid：get_pid()
  - 获取进程pid
    - ✓ ps命令
    - ✓ 访问/proc/pid
    - ✓ getpid()←sys_getpid()

通过ps命令获取进程信息

- 功能
  - 查看系统中正在运行的进程
- 语法
  - ps [-ef][-n name][-t ttys][-p pids][-u users][-groups]



进程组标识

- 成员名：pid_t tgid
- 功能
  - 标识进程是否属于同组，组ID是第一个组内线程（父进程）的ID
  - 线程组中的所有线程共享相同的PID
- 组ID赋值相关操作

```
p->tgid = p->pid;
if (clone_flags & CLONE_THREAD)
    p->tgid = current->tgid;
```

  - 单线程进程：tgid和pid相等
  - 多线程进程：组内所有线程tgid都相等，且等于父进程pid

用户相关的进程标识信息

- 功能：控制用户对系统资源的访问权限
- 分类
  - 用户标识**uid**及组标识**gid**
    - ✓ 通常是进程创建者的uid和gid
  - 有效用户标识**euid**及有效组标识**egid**
    - ✓ 有时系统会赋予一般用户暂时拥有root的uid和gid(作为用户进程的euid和egid)，以便于进行运作
  - 备份用户标识**suid**及备份组标识**sgid**
    - ✓ 在使用系统调用改变uid和gid时，利用其保留真正uid和gid
  - 文件系统标识**fsuid**及文件系统组标识**fsgid**
    - ✓ 检查对文件系统访问权限时使用，通常与euid及egid相等
    - ✓ 在NSF文件系统中，NSF服务器需要作为一个特殊的进程访问文件，此时只修改客户进程的fsuid和fsgid，而不改变euid及egid，可避免受恶意攻击

获取用户相关的进程标识信息的代码示例

❖getpidtest.c

- 获取用户相关的进程标识信息的方法：getXXX()

```
#include <sys/types.h>
int main(int argc,char **argv) {
  pid_t my_pid,parent_pid;
  uid_t my_uid,my_euid;
  gid_t my_gid,my_egid;
  my_pid=getpid( );
  parent_pid=getppid( );
  my_uid=getuid( );
  my_euid=geteuid( );
  my_gid=getgid( );
  my_egid=getegid( );

  printf("Process ID:%ld\n",my_pid);
  printf("Parent ID:%ld\n",parent_pid);
  printf("User ID:%ld\n",my_uid);
  printf("Effective User ID:%ld\n",my_euid);
  printf("Group ID:%ld\n",my_gid);
  printf("Effective Group ID:%ld\n",my_egid);
}
```

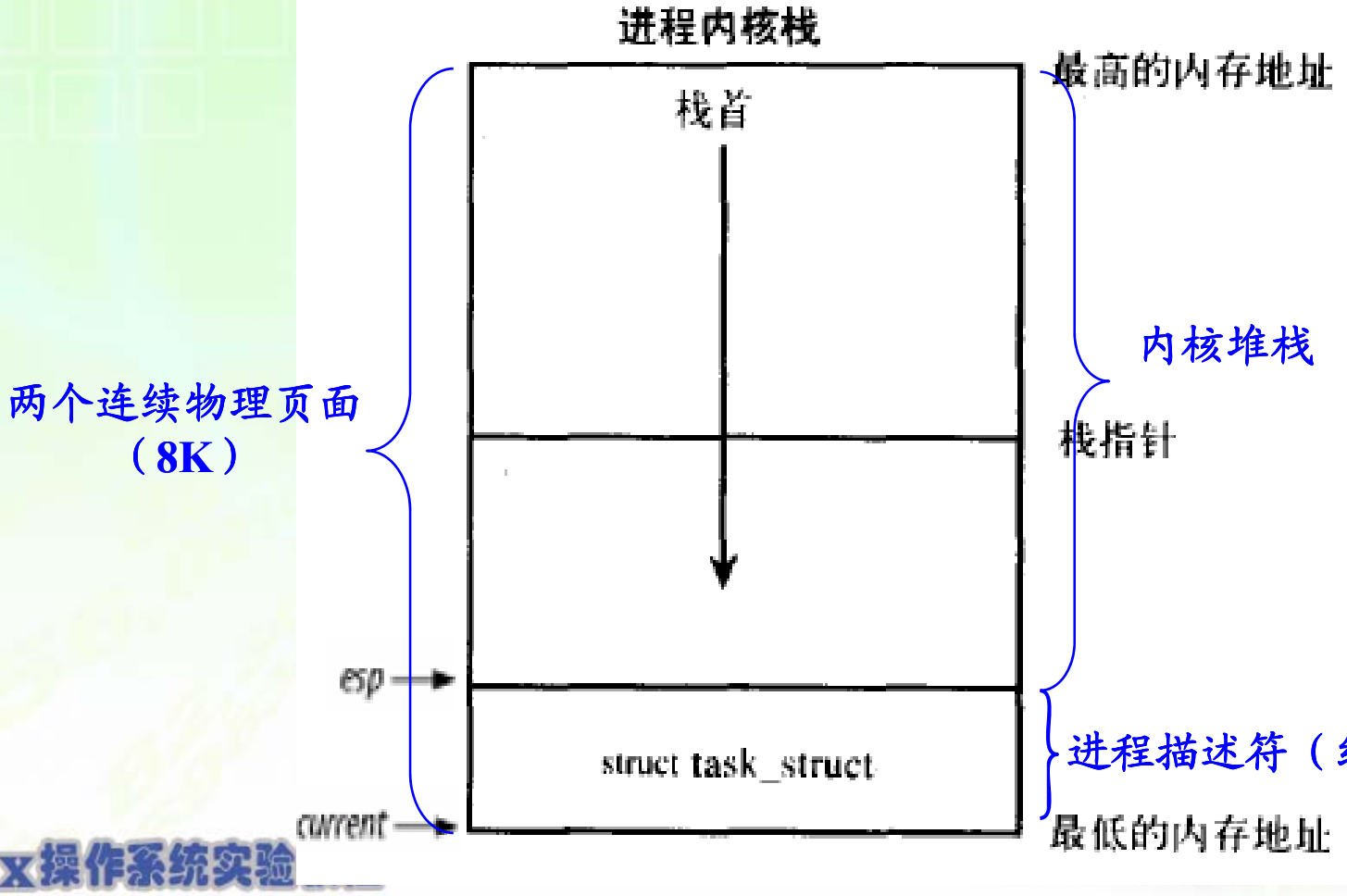
Linux系统进程系统堆栈结构

- 每个进程而都要单独分配一个系统堆栈
- 结构组成
  - 内核态的进程堆栈
  - 进程描述符信息(task_struct)
- 结构特点
  - 8192（2^{13}）字节，两个页框
  - 占据连续两个页框，且第一个页框起始地址为2^{13}的倍数

Linux 2.4进程系统堆栈结构

❖Linux系统进程个数限制

- 所有进程的PCB及系统堆栈占用空间≤1/2的物理内存总和



Linux 2.4进程系统堆栈相关数据结构

❖结构定义(/include/linux/sched.h)

```
union task_union {
        struct task_struct task;
        unsigned long stack[INIT_TASK_SIZE/ sizeof(long)];
};
```

#define INIT_TASK_SIZE 2048*sizeof(long)

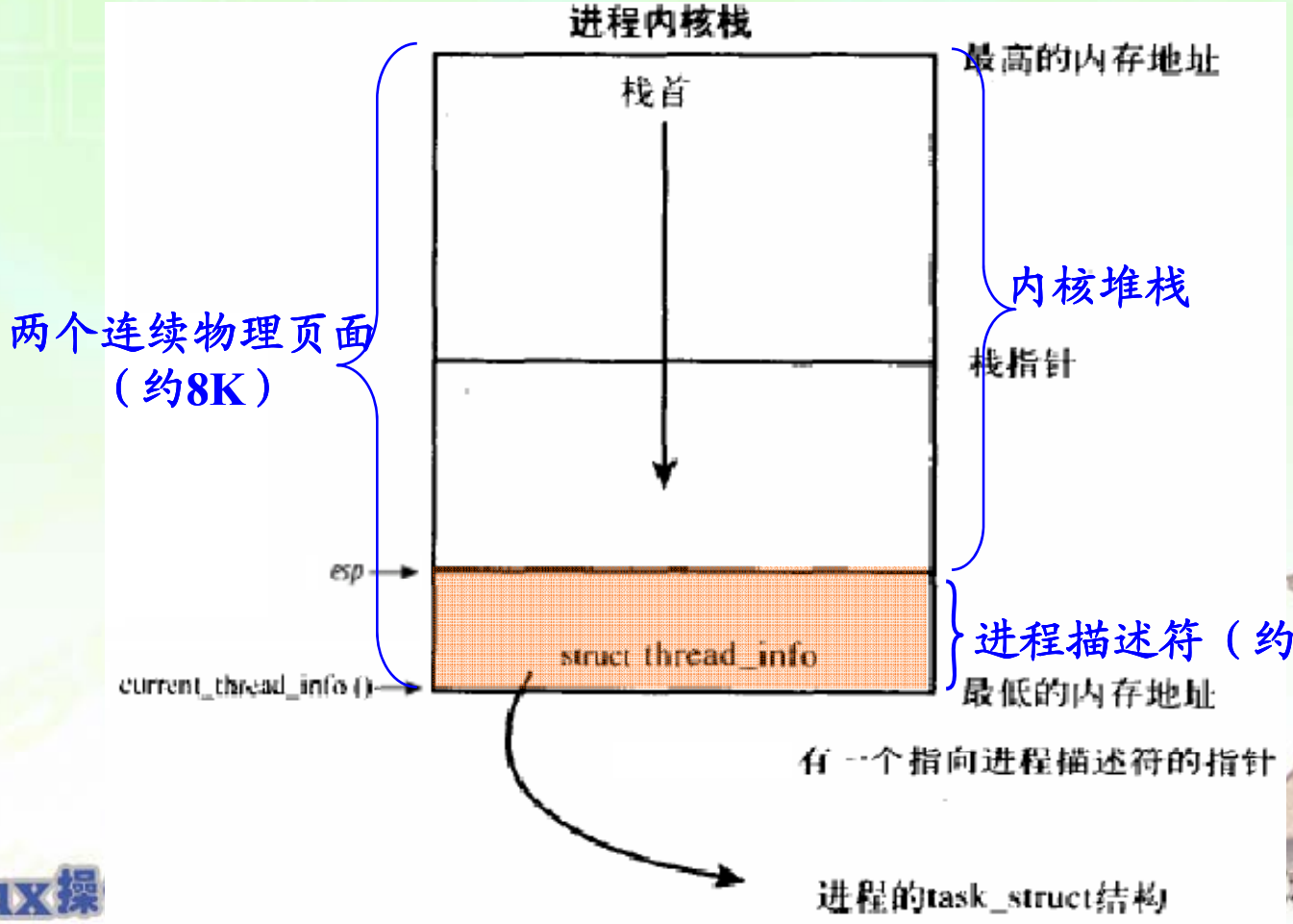
2048

❖进程描述符管理

- 分配：alloc_task_struct()
- 回收：free_task_struct()
- 访问：get_task_struct()

Linux 2.6进程系统堆栈结构

- 进程描述符由slab分配器动态生成
- 栈底用新结构struct thread_info，指向进程描述符



Linux 2.6进程系统堆栈相关数据结构

- **thread_info结构定义[include/asm-x86/thread_info_32.h]**
  - **thread_info（52个字节）**

```
struct thread_info {
 struct task_struct *task; /* main task structure */
  struct exec_domain *exec_domain; /* execution domain */
  unsigned long flags; /* low level flags */
  unsigned long status; /* thread-synchronous flags */
  ... ..
}
```

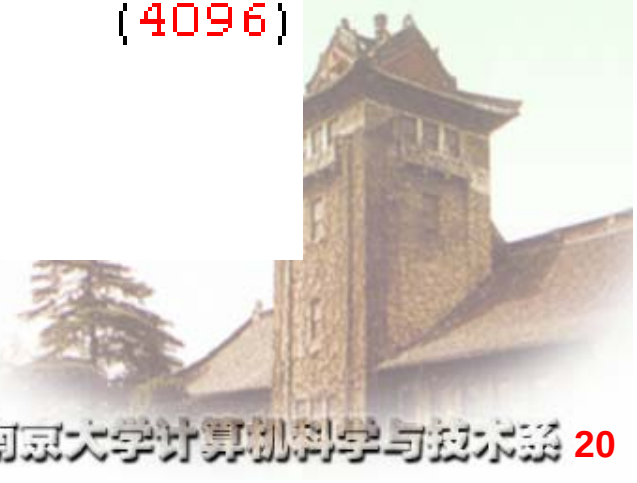
Linux 2.6进程系统堆栈相关数据结构定义

❖ **thread_uion定义[/include/linux/sched.h]**

```
union thread_union {
  struct thread_info thread_info;
  unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

❖ **THREAD_SIZE定义[include/asm-x86/thread_info_32.h]**

```
#ifdef CONFIG_4KSTACKS
#define THREAD_SIZE              (4096)
#else
#define THREAD_SIZE    (8192)
#endif
```

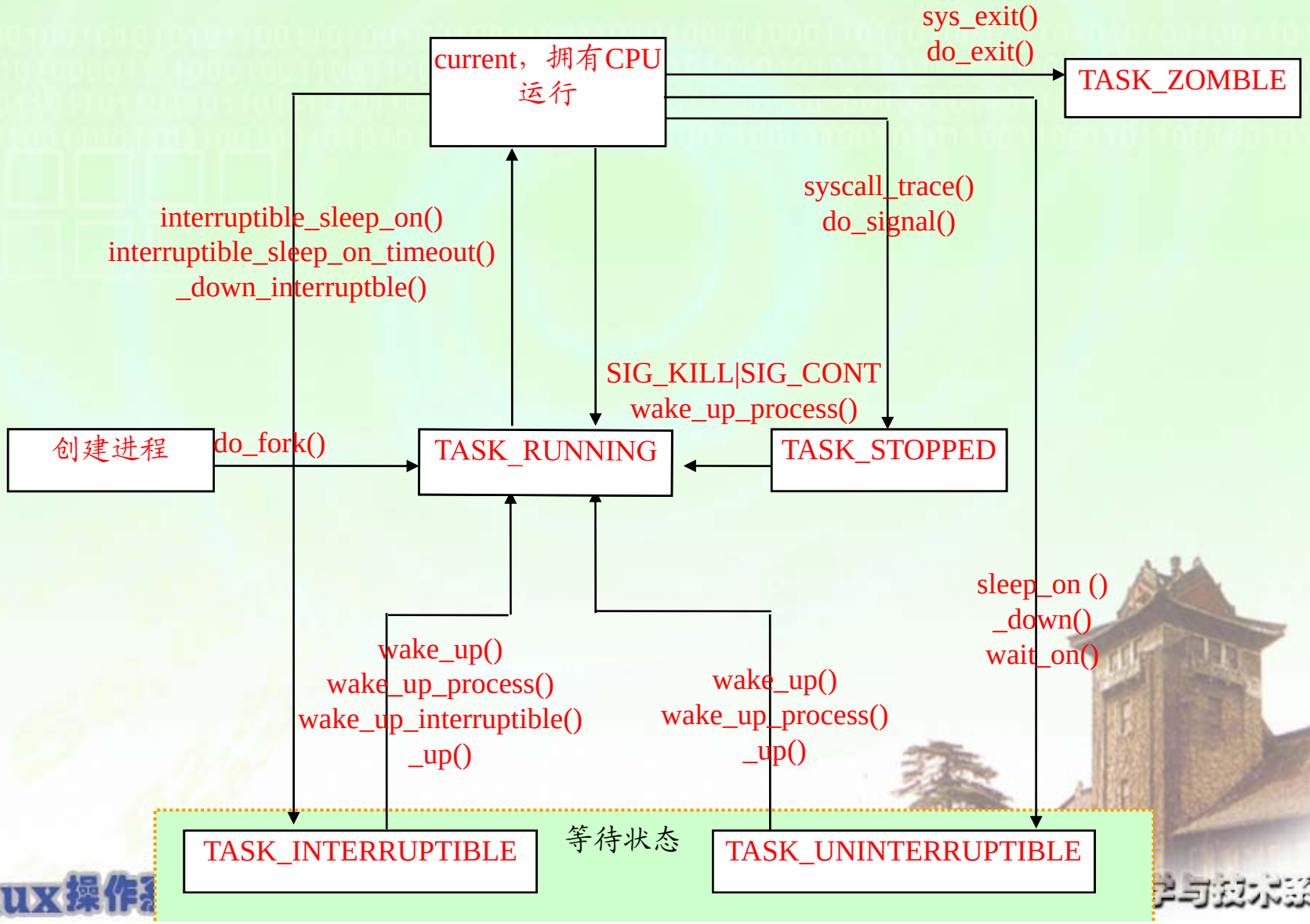


Linux 2.6 CPU运行进程进程描述符的获取

- 由current来获取当前进程的进程描述符
  - 根据THREAD_SIZE大小，分别屏蔽掉内核栈的12-bit LSB(4K)或13-bit LSB(8K)，从而获得内核栈的起始位置

```
#define current get_current()
static inline struct task_struct * get_current(void){
  return current_thread_info()->task;
}
static inline struct thread_info *current_thread_info(void){
  struct thread_info *ti;
  __asm__("andl %%esp,%0; ":"=r" (ti) : "" (~(THREAD_SIZE - 1)));
  return ti;
}
```

Linux系统中的进程状态转换图



Linux进程关系及系统启动过程

主要内容

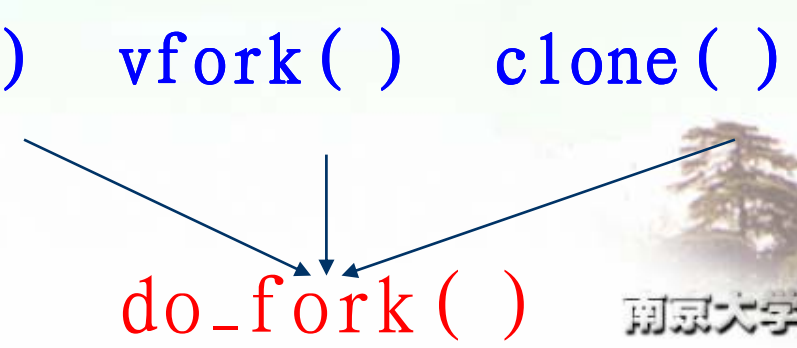
- 背景知识
  - 进程与线程基本概念
  - 多进程编程
  - 多线程编程
- 实验内容
  - 创建进程
  - 线程共享进程中的数据
  - 多线程实现单词统计工具

Linux进程创建特点

- 引入三种机制优化进程创建效率
  - 写时复制技术（Copy-On-Writing）
  - 轻量级进程
    - ✓ 允许父子进程共享页表、打开文件列表、信号处理等数据结构
    - ✓ 但每个进程应该有自己的程序计数器、寄存器集合、核心栈和用户栈
  - vfork()
    - ✓ 新进程可共享父进程的内存地址空间

Linux进程创建方法

- 在终端输入命令，由shell进程创建一个新进程
- 进程创建函数
  - pid_t fork(void);
  - pid_t vfork(void);
  - int clone(int (*fn)(void * arg), void *stack, int flags, void * arg);
    - ✓创建轻量级线程
- 三函数都调用同一内核函数**do_fork( ) [/kernel/fork.c]**



do_fork()内核函数原型

❖函数调用形式

- do_fork(unsigned long clone_flag, unsigned long stack_start, struct pt_regs *regs, unsigned long stack_size, int _user *parent_tidptr, int _user *child_tidptr);

❖参数说明

- clone_flag: 子进程创建相关标志
- stact_start: 将用户态堆栈指针赋给子进程的esp
- regs: 指向通用寄存器值的指针
- stack_size: 未使用（总设为0）
- parent_tidptr: 父进程的用户态变量地址，若需父进程与新轻量级进程有相同PID，则需设置CLONE_PARENT_SETTID
- child_tidptr: 新轻量级进程的用户态变量地址，若需让新进程具有同类进程的PID，需设置CLONE_CHILD_SETTID

子进程创建CLONE参数标志说明

参数标志	含　义
CLONE_FILES	父子进程共享打开的文件
CLONE_FS	父子进程共享文件系统信息
CLONE_IDLETASK	将PID设置为0（只供idle进程使用）
CLONE_NEWNS	为子进程创建新的命名空间
CLONE_PARENT	指定子进程与父进程拥有同一个父进程
CLONE_PTRACE	继续调试子进程
CLONE_SETTID	将TID回写至用户空间
CLONE_SETTLS	为子进程创建新的TLS
CLONE_SIGHAND	父子进程共享信号处理函数
CLONE_SYSVSEM	父子进程共享System V SEM_UNDO语义
CLONE_THREAD	父子进程放入相同的线程组
CLONE_VFORK	调用vfork()，所以父进程准备睡眠等待子进程将其唤醒
CLONE_UNTRACED	防止跟踪进程在子进程上强制执行CLONE_PTRACE
CLONE_STOP	以TASK_STOPPED状态开始进程
CLONE_SETTLS	为子进程创建新的TLS(thread-local storage)
CLONE_CHILD_CLEARTID	清除子进程的TID
CLONE_CHILD_SETTID	设置子进程的TID
CLONE_PARENT_SETTID	设置父进程的TID
CLONE_VM	父子进程共享地址空间

fork()函数

```
asmlinkage int sys_fork(struct pt_regs regs)
{
        return do_fork(SIGCHLD, regs.esp, &regs, 0);
}
```

- 说明
  - 子进程完全复制父进程的资源
  - 子进程的执行独立于父进程
  - 进程间数据共享需通过专门的通信机制来实现
- 返回值
  - 父进程执行fork()返回子进程的PID值
  - 子进程执行fork()返回0
  - 调用失败返回-1

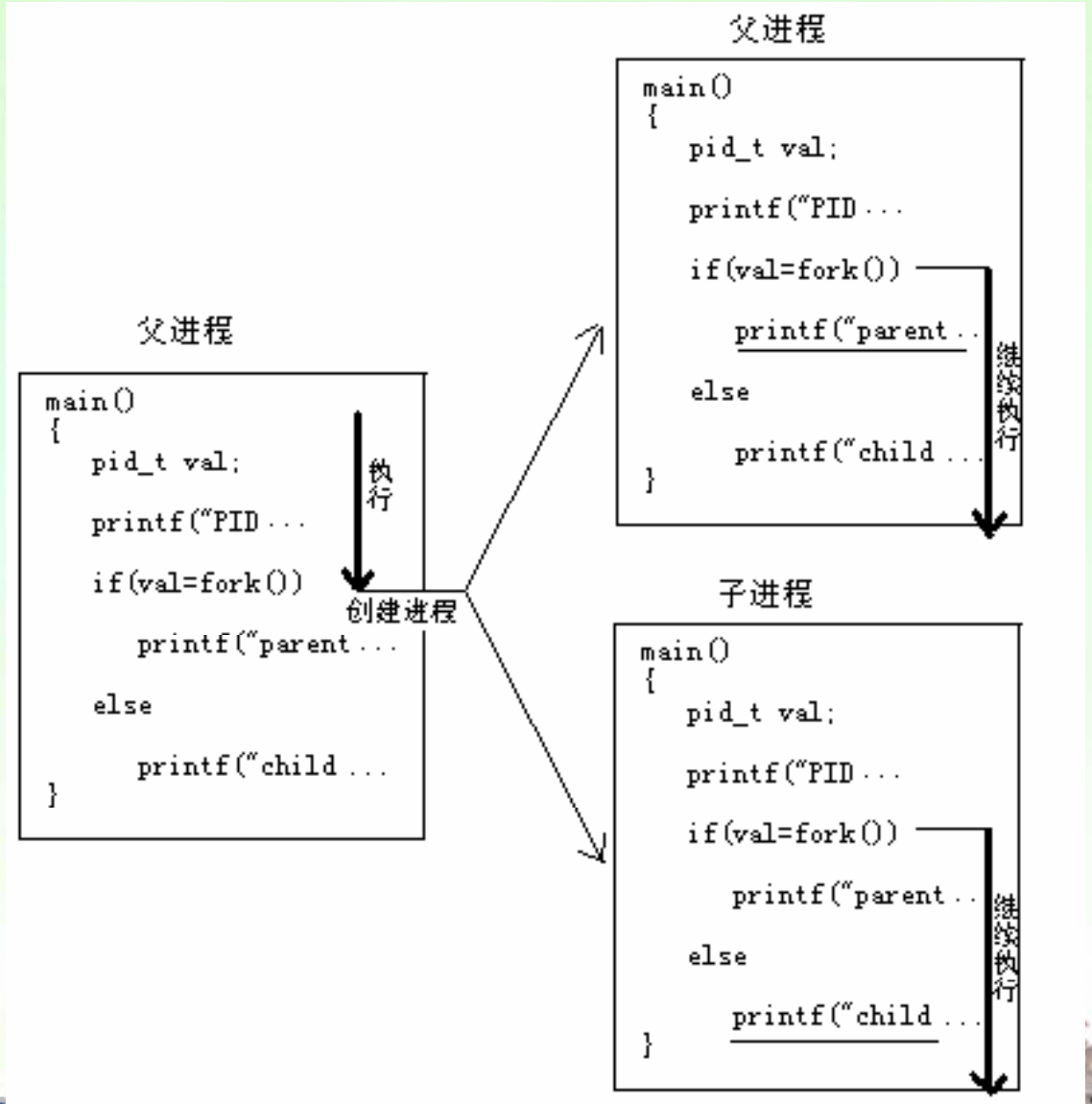
fork()调用代码结构

❖ forktest.c

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

void main(){
int i,p_id;
if((p_id=fork())==0){
for(i=1; i<3; i++)
  printf("This is child  process\n");
}
else if(p_id==-1){
  printf("fork new process error\n");
  exit(-1);
}
else{
  for(i=1; i<3; i++)
    printf("This is parent process\n");
  }
}
```

父子进程执行线索


父进程
main()
{
pid_t val;
printf("PID ...
if(val=fork())
printf("parent ...
else
printf("child ...
}
执行
创建进程
父进程
main()
{
pid_t val;
printf("PID ...
if(val=fork())
printf("parent ...
else
printf("child ...
}
继续执行
子进程
main()
{
pid_t val;
printf("PID ...
if(val=fork())
printf("parent ...
else
printf("child ...
}
继续执行

vfork()调用

```
asmlinkage int sys_vfork(struct pt_regs regs)
{
    return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.esp, &regs, 0);
}
```

- 说明
  - vfork()创建的子进程与父进程共享地址空间
    - ✓ 子进程作为父进程的一个单独线程在其地址空间运行
    - ✓ 子进程从父进程继承控制终端、信号标志位、可访问的主存区、环境变量和其他资源分配
    - ✓ 子进程对虚拟空间任何数据的修改都可为父进程所见
    - ✓ 父进程将被阻塞，直到子进程调用execve()或exit()
- 与fork()的关系
  - 功能相同，但vfork()但不拷贝父进程的页表项
  - 子进程只执行exec()时，vfork()为首选

vfork()系统调用示例

vforktest.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
  int data = 0 ;
  pid_t pid ;
  int choose = 0 ;
  while((choose = getchar( ))!='q') {
    switch(choose) {
    case '1':
      pid = fork( );
      if(pid < 0 ) {
        printf("Error !\n");
      }
      if(pid == 0 ) {
        data++;
        exit(0);
      }
      wait(pid);
      if(pid > 0 ) {
        printf("data is %d\n",data);
      }
      break;
    case '2' :
      pid = vfork( );
      if(pid < 0 ){
        perror("Error !\n");
      }
      if(pid == 0 ) {
        data++;
        exit(0);
      }
      wait(pid);
      if(pid > 0 ) {
        printf("data is %d\n",data);
      }
      break;
    default :
      break;
    }
  }
}
```

```
1
data is 0
2
data is 1
```

clone()系统调用

- 函数原型
  - int clone(int (*fn)(void *), void *child_stack, int clone_flag, void *arg);
- 参数说明
  - fn: 待执行的程序
  - child_stack: 进程所使用的堆栈
  - clone_flag: 由用户指定，可以是多个标志的组合
  - arg: 执行fn所需的参数
- 功能
  - 创建轻量级进程(LWP)的系统调用
  - 通过clone_flag控制

clone()函数对应的系统调用

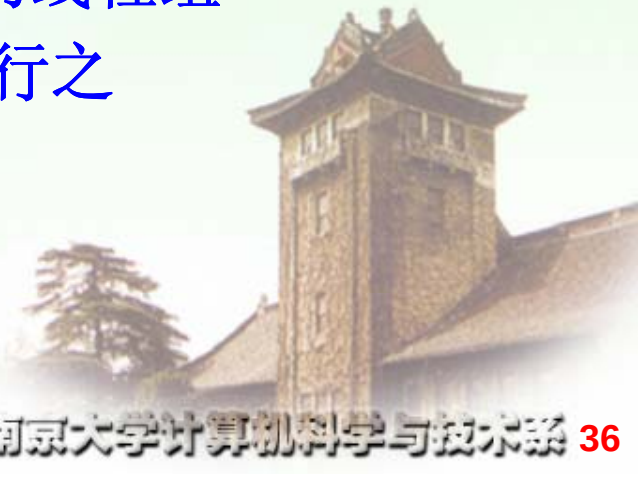
```
asmlinkage int sys_clone(struct pt_regs regs)
{
    unsigned long clone_flags;
    unsigned long newsp;
    int __user *parent_tidptr, *child_tidptr;

    clone_flags = regs.bx;
    newsp = regs.cx;
    parent_tidptr = (int __user *)regs.dx;
    child_tidptr = (int __user *)regs.di;
    if (!newsp)
        newsp = regs.sp;
    return do_fork(clone_flags, newsp, &regs, 0,
        parent_tidptr, child_tidptr);
}
```

- fn及arg参数的获取
  - do_fork()返回时,可通过进程切换机制（jmp+ret）使CPU从stack中获取返回地址，并得到fn地址

clone()函数的常用CLONE标志

- CLONE_VM：父子进程共享内存描述符及所有页表
- CLONE_FS：父子进程共享文件系统信息
- CLONE_FILES：父子进程共享文件描述符表
- CLONE_SIGHAND：父子进程共享信号描述符
- CLONE_PTRACE：若父进程被跟踪，子进程也被跟踪
- CLONE_PARENT：父进程的real_parent登记为子进程的parent和real_parent
- CLONE_THREAD：子进程加入父进程的线程组
- CLONE_STOPPED：创建新进程但不运行之



clone()函数调用示例

```
void* func(int arg){
   ......
}
int main(){
   int clone_flag, arg;
   ......
   clone_flag = CLONE_VM | CLONE_SIGHAND | CLONE_FS | CLONE_FILES;
   stack = (char *)malloc(STACK_FRAME);
   stack += STACK_FRAME;
   retval = clone((void *)func, stack, clone_flag, arg);
   ......
}
```

exec()函数

- 大多情况下子进程从fork返回后都调用exec()函数来执行新的程序
  - 进程调用exec()函数时，该进程完全由新程序替代，新程序从main开始执行
  - exec()并不创建新进程，前后进程ID不变，但用另外一个程序替代当前进程的正文、数据、堆栈等
  - 函数原型
    - ✓ 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 execve(const char *filename, char *const argv [ ], char *const envp[ ]);
    - ✓ int execvp(const char *file, char *const argv[ ]);

exec()函数调用示例

❖ execlptest.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <wait.h>
#include <errno.h>
#include <error.h>
char command[256];
void main(){
  int rtn;
int errorno;
while(1){
printf(">");
  fgets(command,256,stdin);
   command[strlen(command)-1]=0;
   if(fork==0){
     execlp(command,command);

    perror(command);
    exit(errorno);
   }
   else{
     wait(&rtn);
     printf("child process return %d\n",rtn);
   }
  }
}
```

进程等待

- 函数原型
  - pid_t wait(int *status);
  - pid_t waitpid(pid_t pid, int *status, int options);
- 说明
  - 均通过wait4()系统调用实现
  - 进程终止时，会向父进程发送SIGCHLD信号
- 调用wait( )和waitpid( )的进程的可能状态
  - 阻塞
    - ✓如果子进程还在运行
  - 正常返回
    - ✓返回子进程的终止状态（其中一个子进程终止）
  - 出错返回
    - ✓没有子进程

进程等待示例

❖ **waittest1.c**

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
main(){
   pid_t pc,pr;
   pc=fork();
   if(pc<0)  // 如果出错 */
     printf("error ocurred!\n");
   else if(pc==0){ // 如果是子进程 */
     printf("This is child process with pid of %d\n",getpid());
     sleep(10);  // 睡眠10秒钟 */
   }
   else{ // 如果是父进程 */
     pr=wait(NULL);  // 在这里等待 */
     printf("I catched a child process with pid of %d\n",pr);
   }
   exit(0);
}
```

进程等待示例

❖ **waittest2.c**

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
main(){
   int status;
   pid_t pc,pr;
   pc=fork();
   if(pc<0)  // 如果出错 */
     printf("error ocurred!\n");
   else if(pc==0){ // 子进程 */
     printf("This is child process with pid of %d.\n",getpid());
     exit(3);  // 子进程返回3 */
   }
   else{ // 父进程 */
     pr=wait(&status);
     if(WIFEXITED(status)){  // 如果WIFEXITED返回非零值 */
       printf("the child process %d exit normally.\n",pr);
       printf("the return code is %d.\n",WEXITSTATUS(status));
     }else // 如果WIFEXITED返回零 */
       printf("the child process %d exit abnormally.\n",pr);
   }
}
```

进程等待示例

❖ **waitpidtest.c**

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
main(){
  pid_t pc, pr;
  pc=fork();
  if(pc<0)     //如果fork出错
    printf("Error occured on forking.\n");
  else if(pc==0){    //如果是子进程
    sleep(10);  /* 睡眠10秒 */
    exit(0);
  }
  //如果是父进程 */
  do{
    pr=waitpid(pc, NULL, WNOHANG);  //WNOHANG使waitpid不会等待 */
    if(pr==0){  //如果没有收集到子进程 */
  printf("No child exited\n");
    sleep(1);
    }
  }while(pr==0);  //没有收集到子进程，就回去继续尝试 */
  if(pr==pc)
    printf("successfully get child %d\n", pr);
  else
    printf("some error occured\n");
}
```

进程终止

- 释放进程占有的大部分资源
- 终止方式
  - 进程终止的一般方式是exit()或_exit()系统调用
    - ✓ 该系统调用可由编程者明确地在代码中插入
    - ✓ 控制流到达主过程最后一条语句时，自动执行exit()
  - 信号机制

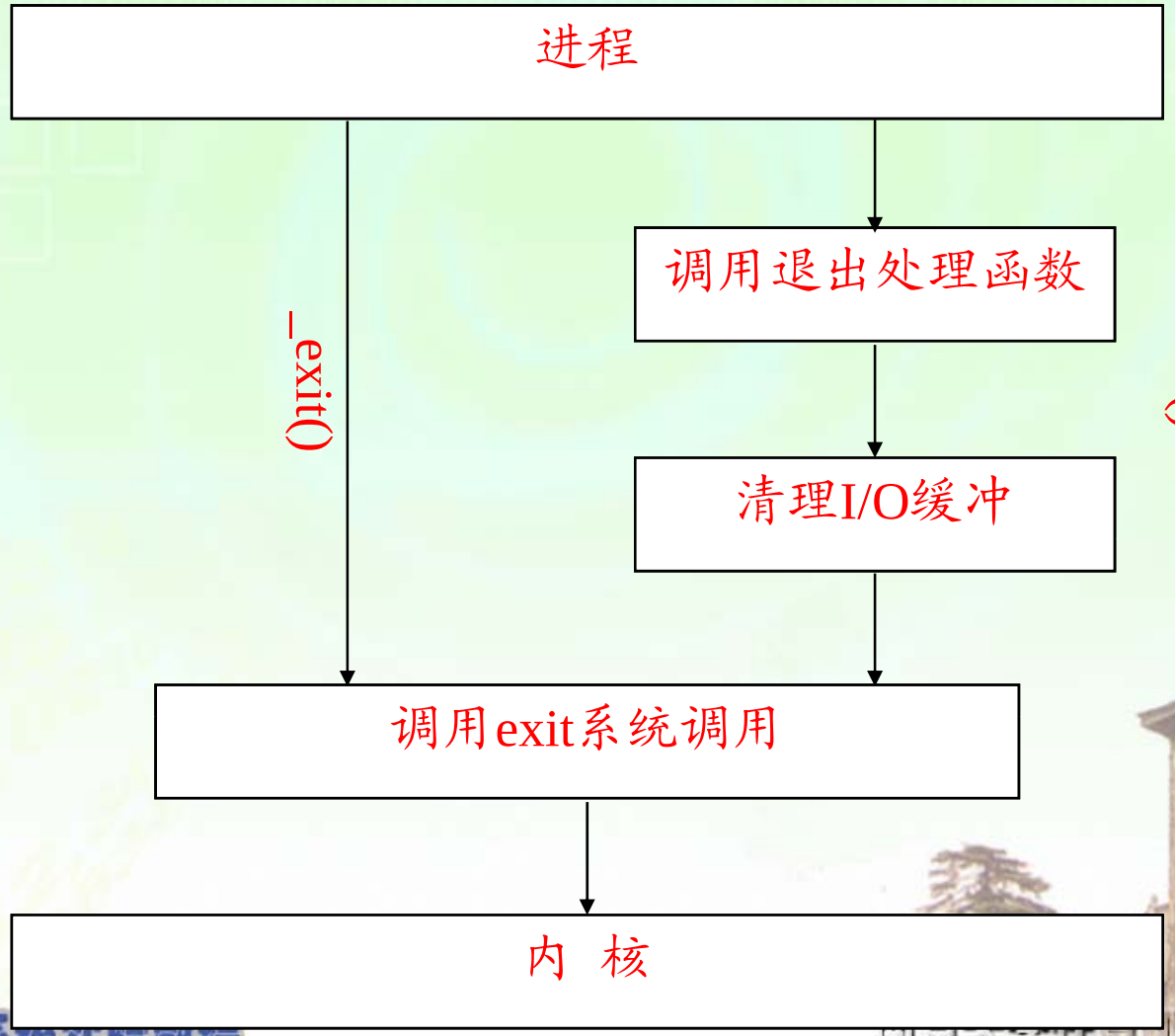
进程终止函数

- 函数原型
  - exit(int status)
  - _exit(int status)
- status说明
  - 正常结束时返回0，否则表示出错信息
- 两函数共性
  - 都通过do_exit() 系统调用

```
asmlinkage long sys_exit(int error_code)
{
        do_exit((error_code&0xff)<<8);
}
```

- 两函数区别
  - _exit()直接使进程停止工作，做清除和销毁工作
  - exit()在调用do_exit()系统调用前检查文件打开情况，清理I/O缓冲

exit()与_exit()区别



进程终止示例

❖ exittest1.c

```
#include<stdlib.h>
main( ) {
   printf("output begin\n");
   printf("content in buffer");
   exit(0);
}
```

❖ exittest2.c

```
#include<stdlib.h>
main( ) {
   printf("output begin\n");
   printf("content in buffer");
   _exit(0);
}
```

主要内容

- 背景知识
  - 进程与线程基本概念
  - 多进程编程
  - 多线程编程
- 实验内容
  - 创建进程
  - 线程共享进程中的数据
  - 多线程实现单词统计工具

Linux的线程描述机制

❖其他多线程操作系统

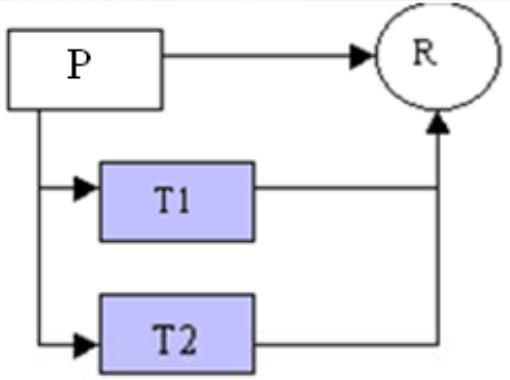
- 通过两种数据结构来表示线程和进程：进程表项和线程表项
- 每个进程表项可以指向若干个线程表项，调度器在进程的时间片内再调度线程

❖Linux系统

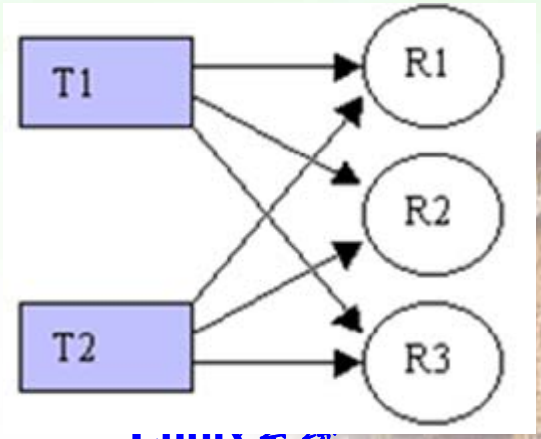
- 从内核角度，没有线程概念，所有线程当进程来实现
- 线程被视为与其他进程共享某些资源的进程，都有自己的进程描述符task_struct
- 内核没有针对线程的调度算法或数据结构

Linux的线程描述机制

- 举例（包含两个线程的进程）
  - 对于非Linux系统
    - ✓有一个包含两个不同线程指针的进程描述符
    - ✓每个线程指针描述其独占资源
  - 对于Linux系统
    - ✓创建两个进程并分配两个普通task_struct结构
    - ✓建立两个进程时只要指定它们共享的某些资源



非Linux系统



Linux系统

Linux的线程分类

- 内核线程
  - 在内核空间内执行线程的创建、调度和管理
  - 内核线程的创建和管理慢于用户线程的创建和管理
  - 特点
    - ✓支持多处理器，支持用户进程中的多线程、内核线程切换的速度快
    - ✓对用户的线程切换来说，系统开销大
- 用户线程
  - 存在于用户空间中，通过线程库来实现
  - 线程创建和调度都在用户空间进行，以进程为单位调度
  - 特点
    - ✓同一进程内的线程切换不需要转换到内核
    - ✓系统调度阻塞问题，不能充分利用多处理器

Linux内核线程

- 独立运行在内核空间的标准进程，支持内核在后台执行一些操作
  - 刷新磁盘高速缓存
  - 交换出不用的页框
  - 维护网络链接等待
- 与普通进程的区别
  - 只运行在内核态，内核线程没有独立的地址空间（mm指针被设置为NULL）
  - 每个内核线程执行一个单独的内核函数
  - 只使用大于PAGE_OFFSET的线性地址空间

Linux内核线程的创建

- 只能由其他内核线程来创建
- 函数原型
  - int kernel_thread(int (*fn) (void *), void *arg, unsigned long flags)
  - 说明
    - ✓ 函数返回时，父线程退出，并返回一个指向子线程task_struct的指针
    - ✓ 子线程执行fn指向的函数，arg是运行时所需的参数
    - ✓ flag定义内核线程的常用参数标志，如CLONE_FS, CLONE_FILES, CLONE_SIGHAND
    - ✓ 一般情况下，内核线程所调用函数一直执行（该函数通常由一个循环构成），直到系统关闭

Linux内核线程的创建过程

```
int kernel_thread(int (*fn)(void *), void * arg,
  unsigned long flags){
  struct pt_regs regs;
  memset(&regs, 0, sizeof(regs));
  regs.bx = (unsigned long) fn;
  regs.dx = (unsigned long) arg;
  regs.ds = __USER_DS;
  regs.es = __USER_DS;
  regs.fs = __KERNEL_PERCPU;
  regs.orig_ax = -1;
  regs.ip = (unsigned long) kernel_thread_helper;
  regs.cs = __KERNEL_CS | get_kernel_rpl();
  regs.flags = X86_EFLAGS_IF | X86_EFLAGS_SF
    | X86_EFLAGS_PF | 0x2;
  /* Ok, create the new process.. */
  return do_fork(flags | CLONE_VM | CLONE_UNTRACED,
    0, &regs, 0, NULL, NULL);
}
```

Linux用户线程的创建

- 函数调用形式
  - int clone(int (*fn)(void * arg), void *stack, int flags, void * arg);
  - int pthread_create(pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void), void *restrict arg);
    - ✓ tidp:新线程的线程描述表指针
    - ✓ attr:为新线程定义不同属性(如栈尺寸)默认为NULL
    - ✓ 第三个和第四个参数指定执行的函数start_rtn和传递给函数的参数arg
- 两者区别
  - clone()创建内核支持的用户线程，对内核可见且由内核调度
  - pthread_create()由基于POSIX标准的线程库创建的用户线程
    - ✓ 但在Linux里， pthread_create()最终调用clone()实现

pthread介绍

- 基于POSIX标准的线程编程接口
  - 包括一个pthread.h头文件和一个线程库
  - 编译方法
    - gcc –g **.c -o *** –lpthread
- 功能
  - 线程管理
    - 支持线程创建/删除、分离/联合，设置/查询线程属性
  - 互斥
    - 处理同步，称为“mutex”
      - 创建/销毁、加锁/解锁互斥量，设置/修改互斥量属性
  - 条件变量
    - 支持基于共享互斥量的线程间通信
      - 建立/销毁、等待/触发特定条件变量，设置/查询条件变量属性

pthread变量类型的命名规则

前缀形式	功能组
pthread_	线程自身及其他子例程
pthread_attr_	线程属性对象
pthread_mutex_	互斥变量
pthread_mutexattr_	互斥属性对象
pthread_cond_	条件变量
pthread_condattr_	条件属性对象
pthread_key_	特定线程数据键

线程创建

- 函数原型
  - int pthread_create(pthread_t *thread, const pthread_attr_t *attr，void *(*start_rtn) (void*)，void * arg);
- 参数说明
  - thread: 待创建线程的id指针
  - attr: 创建线程时的线程属性
  - v(*start_rtin)(void*): 返回值是void*类型的指针函数
  - arg: start_routine的参数
- 返回值
  - 成功返回0
  - 失败返回错误编号
    - ✓ EAGAIN：表示系统限制创建新的线程，如线程数目过多
    - ✓ EINVAL：代表线程属性值非法

线程创建示例

❖ **threadcreatetest.c**

```
#include <pthread.h>
#include <stdio.h>
void *create(void *arg) {
    printf("new thread created ..... ");
}

int main(int argc,char *argv[]){
      pthread_t tidp;
      int error;
      error=pthread_create(&tidp, NULL, create, NULL);
        if(error!=0)
         {
           printf("pthread_create is not created ... ");
           return -1;
         }
        printf("prthread_create is created... ");
        return 0;
}
```

pthread_create()的参数传递问题

❖基本问题

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr，void *(*start_routine) (void*)，void * arg);
  - ✓仅允许传递一个参数给线程待执行的函数
  - ✓如何传递多个参数

❖解决途径

- 构造一个包含所有参数的结构，将结构指针作为参数传递给pthread_create()
- 所有参数必须利用(void *)来传递

pthread_create()参数传递问题—正确示例

- 向新建线程同时传入线程号/消息/线程号总和

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8
char *messages[NUM_THREADS];
struct thread_data{
   int   thread_id;
   int   sum;
   char *message;
};
struct thread_data thread_data_array[NUM_THREADS];
```

pthread_create() 参数传递问题—正确示例

- 向新建线程同时传入线程号/消息/线程号总和

```
void *PrintHello(void *threadarg){
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;
   sleep(1);
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}
```

pthread_create()参数传递问题—正确示例

- 向新建线程同时传入线程号/消息/线程号总和

```
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int *taskids[NUM_THREADS];
  int rc, t, sum;
  sum=0;
  messages[0] = "English: Hello World!";
  messages[1] = "French: Bonjour, le monde!";
  messages[2] = "Spanish: Hola al mundo";
  messages[3] = "Klingon: Nuq neH!";
  messages[4] = "German: Guten Tag, Welt!";
  messages[5] = "Russian: Zdravstvytye, mir!";
  messages[6] = "Japan: Sekai e konnichiwa!";
  messages[7] = "Latin: Orbis, te saluto!";
```

pthread_create()参数传递问题—正确示例

- 向新建线程同时传入线程号/消息/线程号总和

```
  for(t=0;t<NUM_THREADS;t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
      (void *)&thread_data_array[t]);
    if (rc) {
      printf("ERROR; return code f is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

pthread_create()参数传递问题—正确示例

- 向新建线程同时传入线程号/消息/线程号总和

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!  Sum=0
Thread 1: French: Bonjour, le monde!  Sum=1
Thread 2: Spanish: Hola al mundo  Sum=3
Thread 3: Klingon: Nuq neH!  Sum=6
Thread 4: German: Guten Tag, Welt!  Sum=10
hread 5: Russian: Zdravstvytye, mir!  Sum=15
Thread 6: Japan: Sekai e konnichiwa!  Sum=21
Thread 7: Latin: Orbis, te saluto!  Sum=28
```

pthread_create()参数传递问题—错误示例

- 向新建线程传入线程号信息

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     8

void *PrintHello(void *threadid){
   int *id_ptr, taskid;
   sleep(1);
   id_ptr = (int *) threadid;
   taskid = *id_ptr;
   printf("Hello from thread %d\n", taskid);
   pthread_exit(NULL);
}
```

pthread_create()参数传递问题—错误示例

❖错误示例

```
int main(int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int rc, t;
  for(t=0;t<NUM_THREADS;t++) {
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL,
      PrintHello, (void *) &t);
    if (rc) {
      printf("ERROR; return code is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

pthread_create()参数传递问题—错误示例

❖运行结果

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
Hello from thread 8
```

线程ID的访问

- 获取线程自身的id
  - 函数原型
    - pthread_t pthread_self(void);
  - 返回值
    - 调用线程的线程id
- 比较线程ID
  - 函数原型
    - int pthread_equal(pthread_t tid1, pthread_t tid2);
  - 参数
    - tid1：线程1的id
    - tid2：线程2的id
  - 返回值
    - 相等返回非0值，否则返回0

线程ID的访问示例

❖ threadselftest.c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *create(void *arg){
  printf("new thread...");
  printf("thread_tid == %u",(unsigned int )pthread_self());
  printf("thread_pid  is  %d",getpid());
  return (void *)0;
}

int main(int arg,char *argv[]){
  pthread_t tid;
  int error;
  printf("Main thread is starting...");
  error = pthread_create(&tid,NULL,create,NULL);
  if(error!=0){
    printf("thread is not created...");
    return -1;
  }
  printf("main pid is %d ",getpid());
  sleep(1);
  return 0;
}
```

线程终止

- 正常终止
  - 方法1：线程自己调用**pthread_exit()**
    - ✓ **void pthread_exit(void *rval_ptr);**
    - ✓ **rval_ptr 线程退出返回的指针，进程中其他线程可调用 pthread_join（）访问到该指针**
  - 方法2：在线程函数执行**return**
- 非正常终止
  - 其它线程的干预
  - 自身运行出错

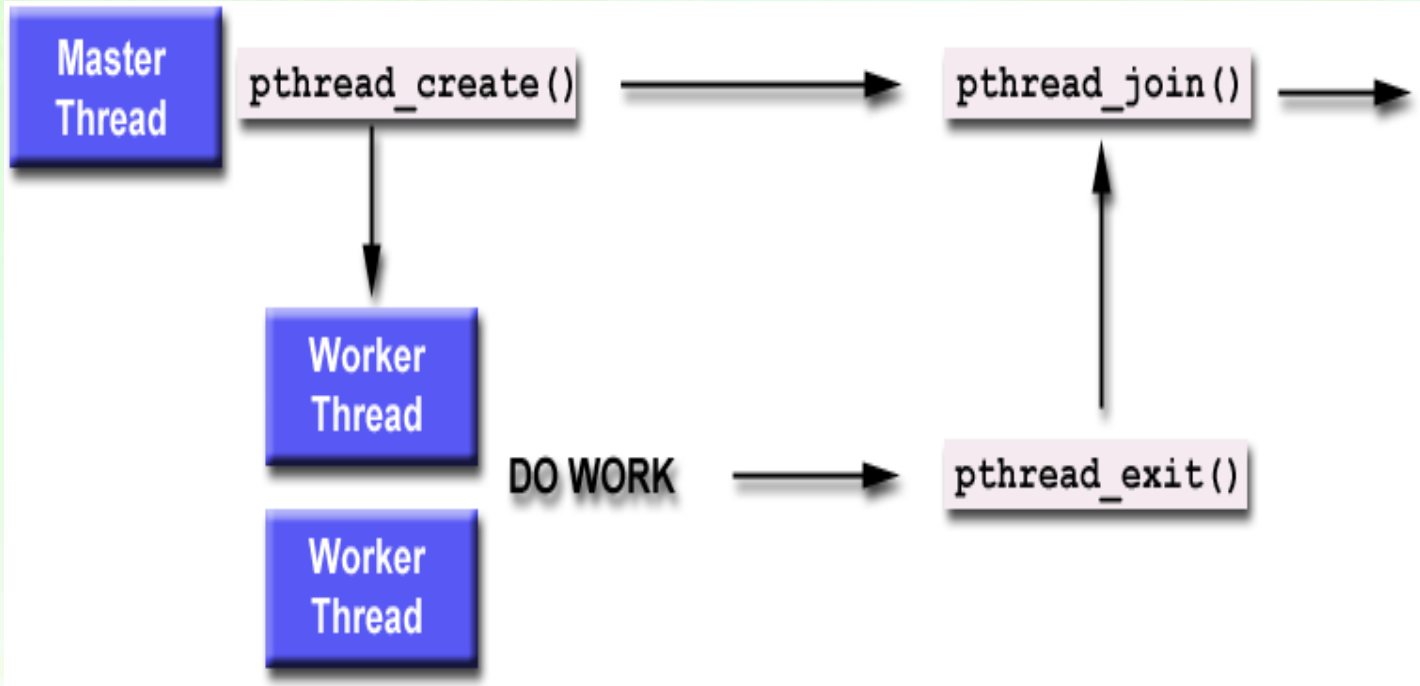
线程执行轨迹

- 同步方式（非分离状态）
  - 等待新创建线程结束
  - 只有当pthread_join（）函数返回时，创建的线程才算终止，才能释放自己占用的系统资源
- 异步方式（分离状态）
  - 未被其他线程等待，自己运行结束即可终止线程并释放系统资源

线程同步终止

- 函数原型
  - int pthread_join(pthread_t thread, void ** rval_ptr);
- 功能
  - 调用者将挂起并等待新进程终止
  - 当新线程调用pthread_exit()退出或者return时，进程中的其他线程可通过pthread_join()获得进程的退出状态
- 使用约束
  - 一个新线程仅仅允许一个线程使用该函数等待它终止
  - 被等待的线程应该处于可join状态，即非DETACHED状态
- 返回值
  - 成功结束返回值为0,否则为错误编码
- 说明
  - 类似于waitpid()

线程同步终止


Master Thread
pthread_create()
pthread_join()
Worker Thread
DO WORK
pthread_exit()
Worker Thread

线程同步终止示例

❖ threadexittest.c

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void *create(void *arg) {
printf("new thread is created ... ");
return (void *)2;
}
int main(int argc,char *argv[]) {
pthread_t tid;
int error;
void *temp;
error=pthread_create(&tid, NULL, create, NULL);
    if(error!=0)    {
 printf("thread is not created ... ");
 return -1;
}
error=pthread_join(tid,&temp);
        if(error!=0){
  printf("thread is not exit ... ");
  return -2;
}
printf("thread is exit code %d  ",(int )temp);     sleep(1);
printf("thread is created... ");
return 0;
}
```

线程分离终止

- 函数原型
  - int pthread_detach(pthread_t thread)
- 功能
  - 执行该函数后线程处于DETACHED状态
  - 处于该状态的线程结束后自动释放内存资源，不能被pthread_join()同步
- 说明
  - 当线程被分离时，不能用pthread_join()等待其终止状态
  - 为避免内存泄漏，线程终止要么处于分离状态，要么处于同步状态

线程分离终止示例

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

pthread_t tid[2];
void * func_1(void *arg){
  //为函数func_2创建线程
  pthread_create(&tid[1],NULL,func_2,NULL);
  //tid[0]将自己挂起，等待线程tid[1]的结束
  pthread_join(tid[1],NULL);
}
void * func_2(void *arg){
  //函数内容
}
int main(){
  //为函数func_1创建线程并将其分离
  pthread_create(&tid[0],NULL,func_1,NULL);
  pthread_detach(tid[0],NULL);
  //主进程不退出，但这两个线程的资源都可以释放掉
  while(1){
    sleep(10);
  }
  return 0;
}
```

Linux线程的互斥机制

- **mutex功能**
  - 阻止资源竞争
  - 实现线程同步和保护共享数据的主要方式
- **pthreads中mutex的基本特点**
  - 在任一时刻，只有一个线程可获得mutex
  - 其他线程需要等待，直到拥有mutex的线程放弃

mutex的一般使用步骤

- 创建和初始化mutex
- 使用mutex
  - 各线程尝试获取mutex
    - ✓但仅有一个线程能够获取mutex并拥有它
  - 拥有mutex的线程执行需访问临界资源的特定处理例程
  - 拥有线程释放mutex
  - 其他线程阐述获取mutex
  - 重复上述步骤
- 销毁 mutex

互斥锁创建

- 声明mutex变量：pthread_mutex_t类型
- 在使用前必须已经初始化（两种方式）
  - 静态方式
    - ✓ pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER
  - 动态方式
    - ✓ int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
  - 返回值
    - ✓ 成功返回0
    - ✓ 失败返回错误编号
  - 说明
    - ✓ mutex初始时是unlocked（未加锁的）的

互斥锁的销毁

- 函数原型
  - int pthread_mutex_destroy(pthread_mutex_t *mutex)
    - ✓销毁一个互斥锁
    - ✓释放它锁占用的资源，且要求锁当前处于开放状态

互斥锁操作

- 加锁
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
    - ✓ 若mutex已被其他线程加锁，该调用会阻塞线程直到mutex被解锁
- 尝试加锁
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
    - ✓ 若mutex已经被加锁，该调用会立即返回一个“busy”错误码
    - ✓ 利用此调用可以防止在优先级倒置所出现的死锁
- 解锁
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
    - ✓ 当拥有mutex的线程使用完保护资源后，应该调用该解锁mutex。在下面的情况中，将返回一个错误
      - mutex已解锁
      - mutex被其他线程加锁

mutex变量使用示例—修改前

```
#include <stdio.h>
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void *thread_function(void *arg);
int run_now=1;  /*用run_now代表共享资源*/
void *thread_function(void *arg){
  int print_count2=0;
  while(print_count2++<5){
    if(run_now==2){ /*子线程：如果run_now为1就把它修改为1*/
      printf("function thread is run\n");
      run_now=1;
    }
    else {
      printf("function thread is sleep\n");
      sleep(1);
    }
  }
  pthread_exit(NULL);
}
```

mutex变量使用示例—修改前

```
int main(){
  int print_count1=0; /*用于控制循环*/
  prhread_t a_thread;
  /*创建一个进程*/
  if(pthread_create(&a_thread,NULL, thread_function,NULL)!=0){
    perror("Thread createion failed");
    exit(1);
  }
  while(print_count1++<5){
    if(run_now==1){ /*主线程：如果run_now为1就把它修改为2*/
      printf("main thread is run\n");
      run_now=2;
    }
    else{
      printf("main thread is sleep\n");
      sleep(1);
    }
  }
  pthread_join(a_thread,NULL); /*等待子线程结束*/
  exit(0);
}
```

mutex变量使用示例—修改前

- 运行结果
  - main线程和function线程是交替运行，都可对run_now进行操作

```
function thread is sleep
main thread is run
main thread is sleep
main thread is sleep
function thread is run
function thread is sleep
main thread is run
main thread is sleep
function thread is run
function thread is sleep
```

mutex变量使用示例—修改后

```
#include <stdio.h>
#include <pthread.h>
#include <stdio.h>
void *thread_function(void *arg);
int run_now=1;   /*用run_now代表共享资源*/
pthread_mutex_t work_mutex;   /*定义互斥量*/

void *thread_function(void *arg){
   int print_count2=0;
   sleep(1);
   if(pthread_mutex_lock(&work_mutex)!=0){
      perror("Lock failed");
      exit(1);
   }
   else
      printf("function lock\n");
```

mutex变量使用示例—修改后

```
    while(print_count2++<5){
        if(run_now==2){   /*分进程：如果run_now为1就把它修改为1*/
            printf("function thread is run\n");
            run_now=1;
        }
        else{
            printf("function thread is sleep\n");
            sleep(1);
        }
    }
    if(pthread_mutex_unlock(&work_mutex)!=0){   /*对互斥量解锁*/
        perror("unlock failed");
        exit(1);
    }
    else
        printf("function unlock\n");
    pthread_exit(NULL);
}
```

mutex变量使用示例—修改后

```
int main(){
  int res;
  int print_count1=0;
  prhread_t a_thread;
  /*初始化互斥量*/
  if(pthread_mutex_init(&work_mutex,NULL)!=0){
    perror("Mutex init faied");
    exit(1);
  }
   /*创建新线程*/
  if(pthread_create(&a_thread,NULL,thread_function,NULL)!=0){
    perror("Thread createion failed");
    exit(1);
  }
  if(pthread_mutex_lock(&work_mutex)!=0){ /*对互斥量加锁*/
    preeor("Lock failed");
    exit(1);
  }
  else
    printf("main lock\n");
```

mutex变量使用示例—修改后

```
while(print_count1++<5){
    if(run_now==1){  /*主线程：如果run_now为1就把它修改为2*/
        printf("main thread is run\n");
        run_now=2;
    }
    else{
        printf("main thread is sleep\n");
        sleep(1);
    }
}
if(pthread_mutex_unlock(&work_mutex)!=0){ /*对互斥量解锁*/
    preeor("unlock failed");
    exit(1);
}
else
    printf("main unlock\n");
pthread_mutex_destroy(&work_mutex); /*收回互斥量资源*/
pthread_join(a_thread,NULL); /*等待子线程结束*/
exit(0);
}
```

mutex变量使用示例—修改后

❖运行结果

```
main lock
main thread is run
main thread is sleep
main thread is sleep
main thread is sleep
main thread is sleep
main unlock
function lock
function thread is run
function thread is sleep
function thread is sleep
function thread is sleep
function thread is sleep
function unlock
```

mutex变量示例

❖ threadsyntest.c

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

void *thread_function(void *arg);
//互斥量，保护工作区及额外变量time_to_exit //
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE]; //工作区
int time_to_exit = 0;
int main( ) {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_mutex_init(&work_mutex, NULL); //初始化工作区
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_create(&a_thread, NULL, thread_function, NULL); //创建新线程
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
```

mutex变量示例

❖ threadsyntest.c(续)

```
pthread_mutex_lock(&work_mutex);
printf("Input some text. Enter 'end' to finish\n");
while(!time_to_exit) {
    fgets(work_area, WORK_SIZE, stdin);
    pthread_mutex_unlock(&work_mutex);
    while(1) {
        pthread_mutex_lock(&work_mutex);
        if (work_area[0] != '\0') {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
        }
        else {
            break;
        }
    }
}
pthread_mutex_unlock(&work_mutex);
printf("\nWaiting for thread to finish...\n");
res = pthread_join(a_thread, &thread_result);
if (res != 0) {
    perror("Thread join failed");
    exit(EXIT_FAILURE);
}
printf("Thread joined\n");
pthread_mutex_destroy(&work_mutex);
exit(EXIT_SUCCESS);
```

mutex变量示例

❖threadsyntest.c(续)

```
void *thread_function(void *arg) {
    sleep(1);
    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

条件变量

- 互斥锁的缺点
  - 通过控制存取数据来实现线程同步
  - 线程需不断轮询条件是否满足（忙等），消耗很多资源
- 条件变量
  - 利用线程间共享的全局变量实现同步
  - 条件变量使线程睡眠等待特定条件出现（无需轮询）
  - 使用方法
    - ✓通常条件变量和互斥锁同时使用
    - ✓一个线程因等待“条件变量的条件成立”而挂起
    - ✓另一个线程使"条件成立"（给出条件成立信号）

条件变量典型使用步骤

- 申明和初始化需要同步的全局数据/变量（如count）
- 申明和初始化一个条件变量对象
- 申明和初始化对应的mutex
- 创建若干进程并运行之

条件变量典型使用步骤

进程1	进程2
1、执行工作直到必须等待特定条件（如计数器必须满足特定值）； 2、对关联mutex上锁并检查全局变量的当前值; 3、调用ptread_cond_wait()进入阻塞等待，等待进程2发送的信号（说明：ptread_cond_wait()会自动对关联mutex解锁，以便可以被线程B使用）； 4、接收到信号后，将唤醒并自动对关联mutex上锁; 5、显式对关联mutex解锁; 6、继续执行。	1、执行自身工作; 2、对关联mutex上锁并修改进程1所等待的全局变量的值; 3、检查进程1所等待的全局变量的值，若满足期待条件，调用pthread_cond_signal()向进程1发送信号; 4、显式对关联mutex解锁; 5、继续执行。
主线程 (join/continue)	

条件变量检测

- 条件检测在互斥锁的保护下进行
  - 如果条件为假，一个线程自动阻塞
  - 若另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件

条件变量的初始化

- 声明条件变量：pthread_cond_t 类型
- 使用前必须初始化，有两种方法初始化方法
  - 静态方式
    - ✓ pthread_cond_t condition=PTHREAD_COND_INITIALIZER
  - 动态方式
    - ✓ int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
      - 被创建的条件变量ID通过 参数返回给调用线程
      - 该方法允许设置条件变量属性

条件变量的销毁

- 函数原型
  - int pthread_cond_destroy(pthread_cond_t *cond);
- 功能说明
  - 销毁指定条件变量，同时释放为其分配的资源

条件变量的等待

- 函数原型
  - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  - int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t mytex, const struct timespec *abstime);
- 说明
  - 阻塞调用线程，直到满足特定的条件
    - ✓ 当该线程运行时，会被加锁，阻塞时会自动解锁
    - ✓ 当收到信号唤醒线程时，会被线程自动上锁当线程完成更新共享数据后，开发者有责任解锁
  - 这里的互斥锁必须是普通锁或者适应锁
  - 调用前必须由本线程加锁，激活前要保持锁是锁定状态

条件变量的激活

- 函数原型
  - int pthread_cond_signal(pthread_cond_t *cond);
  - int pthread_cond_broadcast(pthread_cond_t *cond);
- 说明
  - 用于通知（唤醒）等待在条件变量上的另一线程
  - 在被加锁后被调用，在完成pthread_cond_wait()运行后必须解锁
  - 二者区别
    - ✓ pthread_cond_signal()激活一个等待该条件的线程
    - ✓ pthread_cond_broadcast()激活所有等待的线程
    - ✓ 如果多于一个线程处于阻塞状态，应该用pthread_cond_broadcast()代替pthread_cond_signal()

条件变量等待与激活使用说明

- 如果在调用pthread_cond_wait()前先调用pthread_cond_signal()，将出现逻辑错误
- 当使用上述函数时，必须正确的加锁和解锁
  - 在调用pthread_cond_wait()之前没有成功加锁mutex会导致线程不会阻塞
  - 在调用 pthread_cond_signal()后没有成功解锁mutex，会导致pthread_cond_wait()一直运行 (保持线程阻塞)

条件变量示例

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int count = 0;
int thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

条件变量示例

```
void *inc_count(void *idp){
int j,i;
double result=0.0;
int *my_id = idp;
for (i=0; i<TCOUNT; i++) {
  pthread_mutex_lock(&count_mutex);
  count++;
  if (count == COUNT_LIMIT) {
    pthread_cond_signal(&count_threshold_cv);
    printf("inc_count(): thread %d, count = %d  Threshold reached.\n",
           *my_id, count);
  }
  printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
    *my_id, count);
  pthread_mutex_unlock(&count_mutex);
  for (j=0; j<1000; j++)
    result = result + (double)random();
  }
  pthread_exit(NULL);
}
```

条件变量示例

```
void *watch_count(void *idp){
  int *my_id = idp;
  printf("Starting watch_count(): thread %d\n", *my_id);
  pthread_mutex_lock(&count_mutex);
  if (count<COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal
      received.\n", *my_id);
  }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

条件变量示例

```
int main (int argc, char *argv[]){
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count, (void *)&thread_ids[0]);
  pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
  pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);
  for (i=0; i<NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```

条件变量示例

❖输出结

```
inc_count(): thread 0, count = 1, unlocking mutex
Starting watch_count(): thread 2
inc_count(): thread 1, count = 2, unlocking mutex
inc_count(): thread 0, count = 3, unlocking mutex
inc_count(): thread 1, count = 4, unlocking mutex
inc_count(): thread 0, count = 5, unlocking mutex
inc_count(): thread 0, count = 6, unlocking mutex
inc_count(): thread 1, count = 7, unlocking mutex
inc_count(): thread 0, count = 8, unlocking mutex
inc_count(): thread 1, count = 9, unlocking mutex
inc_count(): thread 0, count = 10, unlocking mutex
inc_count(): thread 1, count = 11, unlocking mutex
inc_count(): thread 0, count = 12  Threshold reached.
inc_count(): thread 0, count = 12, unlocking mutex
watch_count(): thread 2 Condition signal received.
inc_count(): thread 1, count = 13, unlocking mutex
inc_count(): thread 0, count = 14, unlocking mutex
inc_count(): thread 1, count = 15, unlocking mutex
inc_count(): thread 0, count = 16, unlocking mutex
inc_count(): thread 1, count = 17, unlocking mutex
inc_count(): thread 0, count = 18, unlocking mutex
inc_count(): thread 1, count = 19, unlocking mutex
inc_count(): thread 1, count = 20, unlocking mutex
Main(): Waited on 3  threads. Done.
```

主要内容

- 背景知识
  - 进程与线程基本概念
  - 多进程编程
  - 多线程编程
- 实验内容
  - 创建进程
  - 线程共享进程中的数据
  - 多线程实现单词统计工具

进程创建实验

❖实验说明

- 学会通过基本的Linux进程控制函数，由父进程创建子进程，并实现协同工作
- 创建两个进程，让子进程读取一个文件，父进程等待子进程读完文件后继续执行

❖解决方案

- 协调两个进程，使之安排好先后次序并依次执行，可用wait( )或waitpid( )来实现这一点
- 当只需要等待任一子进程运行结束时，可在父进程中调用wait( )
- 若需要等待某一特定子进程的运行结果时，需调用waitpid( )，它是非阻塞型函数

主要内容

- 背景知识
  - 进程与线程基本概念
  - 多进程编程
  - 多线程编程
- 实验内容
  - 创建进程
  - 线程共享进程中的数据
  - 多线程实现单词统计工具

线程共享进程中的数据实验

- 实验说明
  - 了解线程与进程之间的数据共享关系
  - 创建一个线程，在线程中更改进程中的数据
- 解决方案
  - 在进程中定义共享数据，在线程中直接引用并输出该数据
  - 在进程中定义共享数据，创建一个线程，在线程中直接引用并输出该数据

主要内容

- 背景知识
  - 进程与线程基本概念
  - 多进程编程
  - 多线程编程
- 实验内容
  - 创建进程
  - 线程共享进程中的数据
  - 多线程实现单词统计工具

多线程实现单词统计工具实验

❖实验说明

- 多线程实现单词统计工具

❖解决方案

- 区分单词原则：凡是一个非字母或数字的字符跟在字母或数字的后面，那么这个字母或数字就是单词的结尾
- 允许线程使用互斥锁来修改临界资源，确保线程间的同步与协作
- 如果两个线程需要安全地共享一个公共计数器，需要把公共计数器加锁

第2章 进程与线程