|  |
| --- |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析） |
| 1. 学习fork()，exec,pthread库函数的使用，阅读源码，分析fork，exec，pthread\_create函数的机理 2. 代码实现：   进程A创建子进程B  子进程B与父进程A分别对应不同的可执行体:A打印Hello world，B实现sum累加  进程B具有三线程，主线程创建新的线程1实现sum累加（从1到参数x），线程2监控输入参数x  若输入x是非负整数，线程1开始计算；如果线程1正在计算前次程序，线程2的输入非负整数和最近一次的又不同，线程1应重新计算；若相同则不需要重新计算  若输入字母p，则线程1暂停执行。  输入字母e则线程1打印退出信息后整个进程A和B都退出  若输入的是其他字符，则线程1打印一个输入错误提示，任务继续   1. 分析运行中各个执行体的处理器使用，内存使用等基本信息 |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤） |
| 实验环境：  1. vmware workstation 15.0.0   1. ubuntu-18.10-desktop 2. 内核版本：linux-4.18.14   实验步骤：   1. **学习fork()，exec,pthread库函数** 2. **fork()函数**   参考：https://blog.csdn.net/dlutbrucezhang/article/details/8692227  fork()源码：  /\*  \* linux/kernel/fork.c  \* //－－fork()用于创建子进程  \* (C) 1991 Linus Torvalds  \*/  /\*  \* 'fork.c' contains the help-routines for the 'fork' system call  \* (see also system\_call.s), and some misc functions ('verify\_area').  \* Fork is rather simple, once you get the hang of it, but the memory  \* management can be a bitch. See 'mm/mm.c': 'copy\_page\_tables()'  \*/  #include <errno.h>  #include <linux/sched.h>  #include <linux/kernel.h>  #include <asm/segment.h>  #include <asm/system.h>  //－－写页面验证，若页面不可写，则复制页面  extern void write\_verify(unsigned long address);  long last\_pid=0;  //－－进程空间区域写前验证函数  void verify\_area(void \* addr,int size)  {  unsigned long start;  start = (unsigned long) addr;  size += start & 0xfff;  start &= 0xfffff000;  start += get\_base(current->ldt[2]); //－－逻辑地址到线性地址的转换  while (size>0) {  size -= 4096;  write\_verify(start);  start += 4096;  }  }  int copy\_mem(int nr,struct task\_struct \* p) //－－复制内存页表  { //－－由于采用写时复制技术，这里只复制目录和页表项，不分配内存  unsigned long old\_data\_base,new\_data\_base,data\_limit;  unsigned long old\_code\_base,new\_code\_base,code\_limit;  code\_limit=get\_limit(0x0f); //－－取段限长  data\_limit=get\_limit(0x17);  old\_code\_base = get\_base(current->ldt[1]);  old\_data\_base = get\_base(current->ldt[2]);  if (old\_data\_base != old\_code\_base)  panic("We don't support separate I&D");  if (data\_limit < code\_limit)  panic("Bad data\_limit");  new\_data\_base = new\_code\_base = nr \* TASK\_SIZE;  p->start\_code = new\_code\_base;  set\_base(p->ldt[1],new\_code\_base);  set\_base(p->ldt[2],new\_data\_base);  if (copy\_page\_tables(old\_data\_base,new\_data\_base,data\_limit)) { //－－复制页表  free\_page\_tables(new\_data\_base,data\_limit);  return -ENOMEM;  }  return 0;  }  /\*  \* Ok, this is the main fork-routine. It copies the system process  \* information (task[nr]) and sets up the necessary registers. It  \* also copies the data segment in it's entirety.  \*/ //－－fork()子程序，它复制系统进程信息，设置寄存器，复制数据段(代码段)  int copy\_process(int nr,long ebp,long edi,long esi,long gs,long none,  long ebx,long ecx,long edx, long orig\_eax,  long fs,long es,long ds,  long eip,long cs,long eflags,long esp,long ss) //－－复制进程  {  struct task\_struct \*p;  int i;  struct file \*f;  p = (struct task\_struct \*) get\_free\_page(); //－－为新任务数据结构分配内存  if (!p)  return -EAGAIN;  task[nr] = p;  \*p = \*current; /\* NOTE! this doesn't copy the supervisor stack \*/  p->state = TASK\_UNINTERRUPTIBLE;  p->pid = last\_pid;  p->counter = p->priority;  p->signal = 0;  p->alarm = 0;  p->leader = 0; /\* process leadership doesn't inherit \*/  p->utime = p->stime = 0;  p->cutime = p->cstime = 0;  p->start\_time = jiffies;  p->tss.back\_link = 0;  p->tss.esp0 = PAGE\_SIZE + (long) p;  p->tss.ss0 = 0x10;  p->tss.eip = eip;  p->tss.eflags = eflags;  p->tss.eax = 0;  p->tss.ecx = ecx;  p->tss.edx = edx;  p->tss.ebx = ebx;  p->tss.esp = esp;  p->tss.ebp = ebp;  p->tss.esi = esi;  p->tss.edi = edi;  p->tss.es = es & 0xffff;  p->tss.cs = cs & 0xffff;  p->tss.ss = ss & 0xffff;  p->tss.ds = ds & 0xffff;  p->tss.fs = fs & 0xffff;  p->tss.gs = gs & 0xffff;  p->tss.ldt = \_LDT(nr);  p->tss.trace\_bitmap = 0x80000000;  if (last\_task\_used\_math == current)  \_\_asm\_\_("clts ; fnsave %0 ; frstor %0"::"m" (p->tss.i387));  if (copy\_mem(nr,p)) {  task[nr] = NULL;  free\_page((long) p);  return -EAGAIN;  }  for (i=0; i<NR\_OPEN;i++) //－－如果父进程中有文件是打开的，则将对应文件的打开次数增1  if (f=p->filp[i])  f->f\_count++;  if (current->pwd)  current->pwd->i\_count++;  if (current->root)  current->root->i\_count++;  if (current->executable)  current->executable->i\_count++;  if (current->library)  current->library->i\_count++;  set\_tss\_desc(gdt+(nr<<1)+FIRST\_TSS\_ENTRY,&(p->tss)); //－－在GDT表中设置新任务的TSS和LDT  set\_ldt\_desc(gdt+(nr<<1)+FIRST\_LDT\_ENTRY,&(p->ldt));  p->p\_pptr = current;  p->p\_cptr = 0;  p->p\_ysptr = 0;  p->p\_osptr = current->p\_cptr;  if (p->p\_osptr)  p->p\_osptr->p\_ysptr = p;  current->p\_cptr = p;  p->state = TASK\_RUNNING; /\* do this last, just in case \*/  return last\_pid;  }  int find\_empty\_process(void) //－－为新进程取得不重复的进程号last\_pid  {  int i;  repeat:  if ((++last\_pid)<0) last\_pid=1;  for(i=0 ; i<NR\_TASKS ; i++)  if (task[i] && ((task[i]->pid == last\_pid) ||  (task[i]->pgrp == last\_pid)))  goto repeat;  for(i=1 ; i<NR\_TASKS ; i++)  if (!task[i])  return i;  return -EAGAIN;  }  以下给出说明：   fork函数  #include <sys/types.h>  #include <unistd.h> pid\_t fork(void);  fork调用失败则返回-1，调用成功的返回值见下面的解释。我们通过一个例子来理解fork是怎样创建新进程的。  这个程序的运行过程如下图所示。  IMG_256  父进程初始化。  父进程调用fork，这是一个系统调用，因此进入内核。  内核根据父进程复制出一个子进程，父进程和子进程的PCB信息相同，用户态代码和数据也相同。因此，子进程现在的状态看起来和父进程一样，做完了初始化，刚调用了fork进入内核，还没有从内核返回。  现在有两个一模一样的进程看起来都调用了fork进入内核等待从内核返回（实际上fork只调用了一次），此外系统中还有很多别的进程也等待从内核返回。是父进程先返回还是子进程先返回，还是这两个进程都等待，先去调度执行别的进程，这都不一定，取决于内核的调度算法。  如果某个时刻父进程被调度执行了，从内核返回后就从fork函数返回，保存在变量pid中的返回值是子进程的id，是一个大于0的整数，因此执下面的else分支，然后执行for循环，打印"This is the parent\n"三次之后终止。  如果某个时刻子进程被调度执行了，从内核返回后就从fork函数返回，保存在变量pid中的返回值是0，因此执行下面的if (pid == 0)分支，然后执行for循环，打印"This is the child\n"六次之后终止。fork调用把父进程的数据复制一份给子进程，但此后二者互不影响，在这个例子中，fork调用之后父进程和子进程的变量message和n被赋予不同的值，互不影响。  父进程每打印一条消息就睡眠1秒，这时内核调度别的进程执行，在1秒这么长的间隙里（对于计算机来说1秒很长了）子进程很有可能被调度到。同样地，子进程每打印一条消息就睡眠1秒，在这1秒期间父进程也很有可能被调度到。所以程序运行的结果基本上是父子进程交替打印，但这也不是一定的，取决于系统中其它进程的运行情况和内核的调度算法，如果系统中其它进程非常繁忙则有可能观察到不同的结果。  Shell进程是父进程的父进程。父进程运行时Shell进程处于等待状态，当父进程终止时Shell进程认为命令执行结束了，于是打印Shell提示符，而事实上子进程这时还没结束，所以子进程的消息打印到了Shell提示符后面。最后光标停在This is the child的下一行，这时用户仍然可以敲命令，即使命令不是紧跟在提示符后面，Shell也能正确读取。  fork函数的特点概括起来就是“调用一次，返回两次”，在父进程中调用一次，在父进程和子进程中各返回一次。从上图可以看出，一开始是一个控制流程，调用fork之后发生了分叉，变成两个控制流程，这也就是“fork”（分叉）这个名字的由来了。子进程中fork的返回值是0，而父进程中fork的返回值则是子进程的id（从根本上说fork是从内核返回的，内核自有办法让父进程和子进程返回不同的值），这样当fork函数返回后，程序员可以根据返回值的不同让父进程和子进程执行不同的代码。  fork在子进程中返回0，子进程仍可以调用getpid函数得到自己的进程id，也可以调用getppid函数得到父进程的id。在父进程中用getpid可以得到自己的进程id，然而要想得到子进程的id，只有将fork的返回值记录下来，别无它法。  fork的另一个特性是所有由父进程打开的描述符都被复制到子进程中。父、子进程中相同编号的文件描述符在内核中指向同一个file结构体，也就是说，file结构体的引用计数要增加。 使用wait() 函数主程序等子程序执行完成(退出)后再执行  1. **exec函数**   参考：https://www.cnblogs.com/nufangrensheng/p/3510821.html  用fork函数创建子进程后，子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时，该进程执行的程序完全替换为新程序，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用一个全新的程序替换了当前进程的正文、数据、堆和栈段。  fork() 函数用于创建一个新的子进程，该子进程几乎复制了父进程的全部内容，但是，这个新创建的子进程如何执行呢？exec 函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新的进程替换了。另外，这里的可执行文件既可以是二进制文件，也可以是Linux下任何可执行的脚本文件。    在 Linux 中使用exec函数族主要有两种情况：    ●  当进程认为自己不能再为系统和用户做出任何贡献时，就可以调用 exec 函数族中的任意一个函数让自己重生。    ●  如果一个进程想执行另一个程序，那么它就可以调用 fork() 函数新建一个进程，然后调用 exec 函数族中的任意一个函数，这样看起来就像通过执行应用程序而产生了一个新进程（这种情况非常普遍）。  函数族语法    实际上，在Linux中并没有exec()函数，而是由6个以 exec 开头的函数，它们之间的语法有细微差别。下表列出了 exec 函数族的6个成员函数的语法：  IMG_256  这6个函数在函数名和使用语法的规则上都有细微的区别，下面就从可执行文件查找方式、参数传递方式和环境变量这几个方面进行比较。     ●  查找方式：表1中的前4个函数的查找方式都是完整的文件目录路径，而最后两个函数（也就是以 p 结尾的两个函数）可以只给出文件名，系统就会自动按照环境变量“$PATH” 所指定的路径进行查找。     ●   参数传递方式：exec函数族的参数传递有两种：一种是逐个列举的方式，而另一种则是将所有参数整体构造指针数组传递。在这里是以函数名的第5位字母来区分的，字母为 "l"(list)的表示逐个列举参数的方式，其语法为const char \*arg;字母为“v”(vector)的表示将所有参数整体构造指针数组传递，其语法为 char \*const argv[]。这里的参数实际上就是用户在使用这个可执行文件时所需的全部命令选项字符串（包括该可执行程序命令本身）。要注意的是，这些参数必须以NULL结束。     ●   环境变量： exec函数族可以默认系统的环境变量，也可以传入指定的环境变量。这里以 “e”(environment)结尾的两个函数 execle()和 execve()就可以在 envp[]中指定当前进程所使用的环境变量。     表2再对这6个函数中的函数名和对应语法做了一个小结，主要指出了函数名中每一位对应所表明的含义，以此表加以记住这6个函数。  IMG_256     事实上，这6个函数中真正的系统调用只有execve()，其他5个都是库函数，它们最终都会调用execve()这个系统调用。在使用exec函数族时，一定要加上错误判断语句。exec 很容易执行失败，其中最常见的原因有：     ①  找不到文件或路径，此时 errno 被设置为 ENOENT。     ②  数组argv 和envp  忘记用NULL结束，此时，errno被设置为 EFAUL。     ③  没有对应可执行文件的运行权限，此时 errno 被设置为EACCES。  注意，在执行exec前后实际用户ID和实际组ID保持不变，而有效ID是否改变则取决于所执行程序文件的设置用户ID位和设置组ID位是否设置。如果新程序的设置用户ID位已设置，则有效用户ID变成程序文件所有者的ID，否则有效用户ID不变。对组ID的处理方式与此相同。  在很多UNIX实现中，这6个函数中只有execve是内核的系统调用。另外5个只是库函数，它们最终都要调用该系统调用。   1. **pthread库函数**   参考：https://blog.csdn.net/liangxanhai/article/details/7767430  3.1 pthread\_create（）  是UNIX环境创建线程函数  头文件  　　#include<pthread.h>  函数声明  　　int pthread\_create(pthread\_t\*restrict tidp,const pthread\_attr\_t \*restrict\_attr,void\*（\*start\_rtn)(void\*),void \*restrict arg);  返回值  　　若成功则返回0，否则返回出错编号  　　返回成功时，由tidp指向的内存单元被设置为新创建线程的线程ID。attr参数用于制定各种不同的线程属性。新创建的线程从start\_rtn函数的地址开始运行，该函数只有一个万能指针参数arg，如果需要向start\_rtn函数传递的参数不止一个，那么需要把这些参数放到一个结构中，然后把这个结构的地址作为arg的参数传入。  　　linux下用C开发多线程程序，Linux系统下的多线程遵循POSIX线程接口，称为pthread。  　　由 restrict 修饰的指针是最初唯一对指针所指向的对象进行存取的方法，仅当第二个指针基于第一个时，才能对对象进行存取。对对象的存取都限定于基于由 restrict 修饰的指针表达式中。 由 restrict 修饰的指针主要用于函数形参，或指向由 malloc() 分配的内存空间。restrict 数据类型不改变程序的语义。 编译器能通过作出 restrict 修饰的指针是存取对象的唯一方法的假设，更好地优化某些类型的例程。  参数  　　第一个参数为指向线程标识符的指针。  　　第二个参数用来设置线程属性。  　　第三个参数是线程运行函数的起始地址。  　　最后一个参数是运行函数的参数。  另外，在编译时注意加上-lpthread参数，以调用静态链接库。因为pthread并非Linux系统的默认库  3.2线程退出的条件  调用pthread\_exit函数退出。  其他线程调用pthread\_cancel取消该线程，且该线程可被取消。  创建线程的进程退出或者整个函数结束。  当前线程代码执行完毕。  其中的一个线程执行exec类函数执行新的代码，替换当前进程所有地址空间。 3.3线程之间的参数传递 函数pthread\_join用来等待一个线程的结束,线程间同步的操作。头文件 ： #include <pthread.h>  函数定义： int pthread\_join(pthread\_t thread, void \*\*retval);  描述 ：pthread\_join()函数，以阻塞的方式等待thread指定的线程结束。当函数返回时，被等待线程的资源被收回。如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的。  参数 ：thread: 线程[标识符](https://baike.baidu.com/item/%E6%A0%87%E8%AF%86%E7%AC%A6/7105638" \t "https://baike.baidu.com/item/pthread_join/_blank)，即线程ID，标识唯一线程。retval: 用户定义的指针，用来存储被等待线程的返回值。  3.4 线程同步  条件变量    条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待"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）。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。  条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作： 1）一个线程等待"条件变量的条件成立"而挂起； 2）另一个线程使"条件成立"（给出条件成立信号）。 为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。 1.主要涉及到下面的函数： int pthread\_cond\_init(pthread\_cond\_t \*cond, pthread\_condattr\_t \*cond\_attr) ---动态创建条件变量 pthread\_mutex\_lock ---互斥锁上锁 pthread\_mutex\_unlock ----互斥锁解锁  pthread\_cond\_wait() / pthread\_cond\_timedwait -----等待条件变量，挂起线程，区别是后者，会有timeout时间，如 果到了timeout,线程自动解除阻塞，这个时间和 time()系统调用相同意义的。以1970年时间算起。 pthread\_cond\_signal ----激活等待列表中的线程， pthread\_cond\_broadcast() -------激活所有等待线程列表中最先入队的线程 注意：1）上面这几个函数都是原子操作，可以为理解为一条指令，不会被其他程序打断            2）上面这个几个函数，必须配合使用。             3）pthread\_cond\_wait，先会解除当前线程的互斥锁，然后挂线线程，等待条件变量满足条件。一旦条件变                   量满足条件，则会给线程上锁，继续执行pthread\_cond\_wait   1. 代码实现   根据题目要求，可知需要两个执行体A和B。根据模块化的原则先设计A，再设计B。   1. 审题，根据题目要求，写了进程A的代码   **#include<unistd.h>提供fork(),execlp()函数**  **#include <sys/wait.h> 提供wait()函数**  **#include <sys/types.h> 提供类型pid\_t的定义**  **代码比较简单，主要就是创建子进程，子进程用一个新的进程映像替换当前进程映像。**     1. 之后写子进程B的代码   充分利用模块化的设计方法。  先是头文件  #include<pthread.h>提供pthread库函数  而需要注意，gcc编译时加上-lpthread，因为pthread并非Linux默认的库    之后是几个全局变量，减少线程间传递参数的麻烦。    **1.主线程**  **功能是创建2个子线程，线程2监控输入，线程1打印信息和计算累加。**  **当线程1,2都退出后主线程退出。**  **使用pthread\_create创建子线程。**  **使用pthread\_join等待线程结束**    **2.线程2**  **因为线程2要监控输入，所以先写线程2.**  **线程2第一部分：**  **主要进行输入的处理。**  **详情见图片和代码的注释。**  **这里用c++的string处理输入，因为没有限制输入类型**    线程2第二部分：  根据输入确定func值。func作为全局变量调整线程1的工作状态  同时根据recent和cal判断是否要重新计算。target是线程1从1开始的累加目标。    线程2第三部分：  根据第二部分得到的func调整线程1工作状态。  func==2时，利用pthread\_join()先等待线程1打印错误信息，再退出。  pthread\_suspend() pthread\_resume()函数是自己定义的函数，用以实现线程1暂停。暂停的实现具体解释见实验报告后续部分。     1. **线程1**   **第一部分：实现线程1的暂停。**  **利用**  **pthread\_mutex\_lock(),pthread\_mutexunlock()，pthread\_cond\_wait(),pthread\_cond\_broadcast()实现线程的暂停。**  **源码体现的暂停实现过程更清晰。**        **第二部分：线程1进行累加，输出信息等操作**  **sleep是为了体现p操作，两次输入相同，不同操作而设置。否则可能人输入的速度不如计算机计算速度，导致第二次输入还没到，就得出上一次的结果。**    综上，进程B结构如下  void pthread\_suspend(void)  void pthread\_resume(coid)  void thread1(void\* arg)  void thread2(void\*arg)  int main()  三．分析运行中各个执行体的处理器使用，内存使用等基本信息  首先在终端运行进程A。    pgrep 是通过程序的名字来查询进程的工具，一般是用来判断程序是否正在运行。在服务器的配置和管理中，这个工具常被应用，简单明了；  然后打开另一个终端，pgrep -l A 查看进程名和pid。    pstree -p 26638。以树状图显示进程，并显示进程pid。可见父子进程和线程的关系。    ps -T -p 26639 可以查看进程B的线程信息。    使用top查看进程cpu和内存信息。  下图为示例：    top命令的第三行，cpu状态：  依次对应：  us:user 用户空间占用cpu的百分比  sy:system 内核空间占用cpu的百分比  ni:niced 改变过优先级的进程占用cpu的百分比  id:空闲cpu百分比  wa:IO wait IO等待占用cpu的百分比  hi:Hardware IRQ 硬中断 占用cpu的百分比  si:software 软中断 占用cpu的百分比  st:被hypervisor偷去的时间  top命令第四行，内存状态：  total，free ，used ，buff/cache  依次对应：物理内存总量,空闲内存总量,使用中的内存总量,缓冲内存量  进入top后，交互时，输入s，系统提示更改刷新间隔。输入0则不断刷新。    top -p 26638 查看pid为26638的进程（即A）的信息    top -p - H 26639  top命令可以实时显示各个线程情况。要在top输出中开启线程查看，请调用top命令的“-H”选项，该选项会列出所有Linux线程。在top运行时，你也可以通过按“H”键将线程查看模式切换为开或关。    top -p 26640查看进程B的子线程1.    top -p 26641查看进程B的子线程2. |
| 1. 实验过程分析   （实验分工，详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） |
| 遇到的问题及解决：  1.编译时出现c++11标准库未定义错误，因为编译器使用的库版本不同。      解决：使用-l链接 stdc++     1. gcc 编译时报错     解决：  在32位模式时，int 和指针类型变量都占32位 在64位模式下，int占32位，指针变量占64位     1. pthead线程的暂停与恢复   解决：Linux下没有类似windows的suspend和resume函数。故需要自己写   1. 如图     又是-lstdc++解决  实验结果记录：  执行程序A。程序A 父进程fork 出子进程，之后打印Hello, world!然后执行wait()，等待子进程结束。  子进程调用execlp，执行另一个执行体B。  执行体B中线程2开始监控输入。  输入15，是非负整数，之后线程1计算1累加到15.结果为120。120后跟着的15表明计算的是1-15累加。  输入16，是非负整数，之后线程1计算1累加到16.结果为136。136后跟着的16表明计算的是1-16累加。  输入18，之后输入20.线程2监控到由于20与18不同，故线程1不再进行1-18累加，而进行1-20累加。  输出210.  输入25，再输入25.线程2监控到25相同，不需要重新计算，打印信息“输入相同，不需要重新计算”。线程1计算第一个25，输出325.  输入afd。线程2监控，属于其他字符，线程1打印错误信息“input wrong”。  输入fg。线程2监控，属于其他字符，线程1打印错误信息“input wrong”。  输入p，线程1暂停执行。打印信息“pause”表明线程1已暂停。  输入123.线程2检测到是非负整数，先调用恢复函数解除线程1的暂停。之后打印信息“resume”表示线程1已经恢复。之后线程1执行1-123累加，输出7626.  输入e。线程2检测到要退出。之后线程1打印信息“A and B exited”表示主进程，子进程都要退出。之后主进程和子进程都退出。  结束。  IMG_256 |