# 实验 4-1：Linux进程与线程

# 内　　容

1. Linux下C程序设计
2. 进程/线程控制
3. 同步与互斥

# Linux下C语言编程

- **一、源程序编辑器**
  - Vi，Emacs，gedit 等
- **二、gcc编译器**

  一个全功能的 ANSI C 兼容编译器，它是所有UNIX系统可用的C编译器

  - 预处理：对源代码文件中的文件包含(include)、预编译语句(如宏定义define等)进行分析。
  - 编　　译：就是把C/C++代码“翻译”成汇编代码。
  - 汇　　编：将第二步输出的汇编代码翻译成符合一定格式的机器代码，生成以.s为后缀的目标文件。
  - 链　接：将上步生成的目标文件和系统库的目标文件和库文件链接起来，最终生成了可以在特定平台运行的可执行文件。

# Linux下C语言编程

- 简单实例

编辑好源文件hello.c，编译并链接生成二进制执行文件的过程：

方法1：

① gcc -c hello.c

编译生成目标文件，输出：hello.o

② gcc hello.o -o hello

链接生成可执行文件，输出：hello二进制可执行文件

执行：./hello

方法2：

gcc hello.c -o hello

一步操作，完成方法1的两步内容,输出名为hello的二进制可执行文件

方法3：

gcc hello.c

缺省输出到一个名为a.out的可执行文件

# Linux下C语言编程

- gcc的各种选项

1　-S 编译后输出汇编语言文件

hi.c

```
#include <stdio.h>
int main()
{
   printf("good evening,everyone!\n");
}
```

hi.s

```
        .file   "hi.c"
        .section        .rodata
.LC0:
        .string "good evening,everyone!"
        .text
.globl main
        .type   main, @function
main:
        leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $4, %esp
        movl    $.LC0, (%esp)
        call    puts
        addl    $4, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.2 20080704 (Red Hat 4.1.2-48)"
        .section        .note.GNU-stack,"",@progbits
```

2 -c 编译或汇编源文件，产生目标文件；一般缺省采用.0后缀替换源文件的后缀。

3　-E 预处理后即停止，不进行编译. 预处理后的代码送往标准输出

4　-o 指定一个输出文件

# Linux下C语言编程

- gcc的各种选项

  5 -l*library* 链接名为*library*的库文件

```c
#include <stdio.h>
#include <math.h>
int main()
{
   double x;
   double nm=16;
   x=sqrt(nm);
   printf("The SQRT of %lf is %lf \n", nm, x);
}
```

```
[root@localhost sang]# gcc -o hi hi.c
/tmp/ccS5nNZd.o: In function `main':
hi.c:(.text+0x35): undefined reference to `sqrt'
collect2: ld returned 1 exit status
```

```
gcc -o hi hi.c -lm
```

```
linux-gate.so.1 =>  (0x0020e000)
libm.so.6 => /lib/libm.so.6 (0x005de000)
libc.so.6 => /lib/libc.so.6 (0x00496000)
/lib/ld-linux.so.2 (0x00478000)
```

| | |
|---|---|
| -lm | 链接数学库 |
| -lpthread | 链接线程库 |

libm.so

libpthread.so

# 进程创建

- LINUX启动创建init进程（PID=1），所有进程的祖先
- fork系统调用可用于创建进程
- 父进程通过fork创建（复制）子进程
- 子进程也可以再创建子进程
- 子进程可以通过exec加载其它不同的程序

# 进程创建函数1： pid_t fork(void);

**功 能：**创建一个副进程的副本

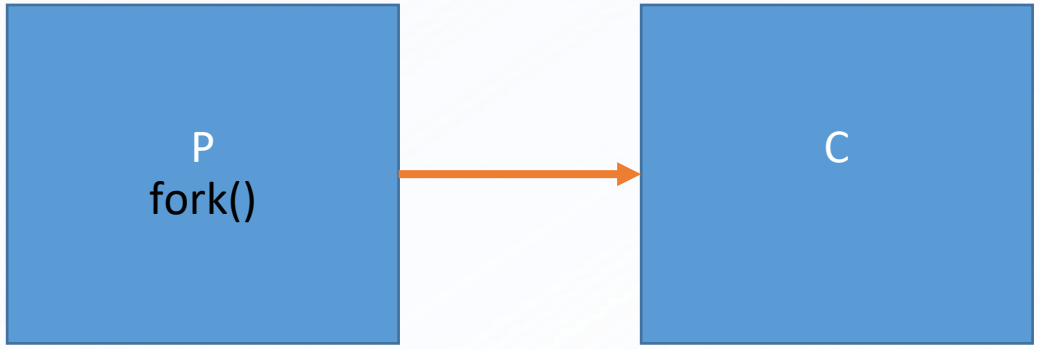
**特 点：**

调用一次，返回两次，三种不同的返回值：

1、父进程中返回新创建的子进程的PID

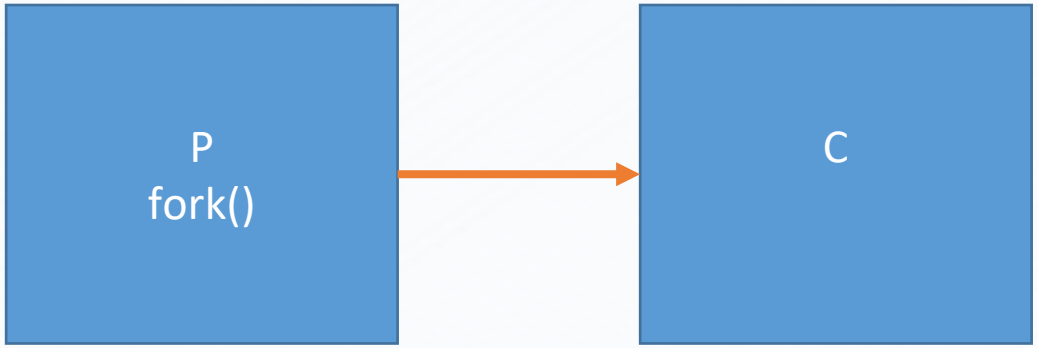
2、子进程中返回0；

3、如果出现错误返回一个负值。



# 进程创建函数1： pid_t fork(void);

- fork() 创建一个进程的副本，调用一次，返回两次



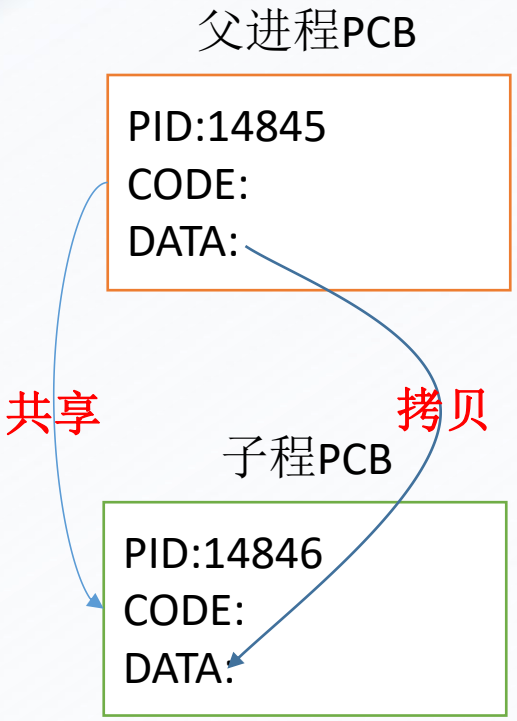

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
  pid_t pid;
  pid=fork();
  printf("Hello, the PID of this process is  %d. \n",getpid());
}
```

```
Hello, the PID of this process is  24839.
Hello, the PID of this process is  24838.
```

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
  int i;
  pid_t pid;
  pid=fork();
  for (i=0;i<5;i++)
  {
    printf("After run fork, the current PID is %d \n",getpid());
    sleep(1);
  }
}
```

```
After run fork, the current PID is 25653
After run fork, the current PID is 25652
After run fork, the current PID is 25652
After run fork, the current PID is 25653
After run fork, the current PID is 25652
After run fork, the current PID is 25653
After run fork, the current PID is 25652
After run fork, the current PID is 25653
After run fork, the current PID is 25652
After run fork, the current PID is 25653
```

# 进程创建函数1： pid_t fork(void);



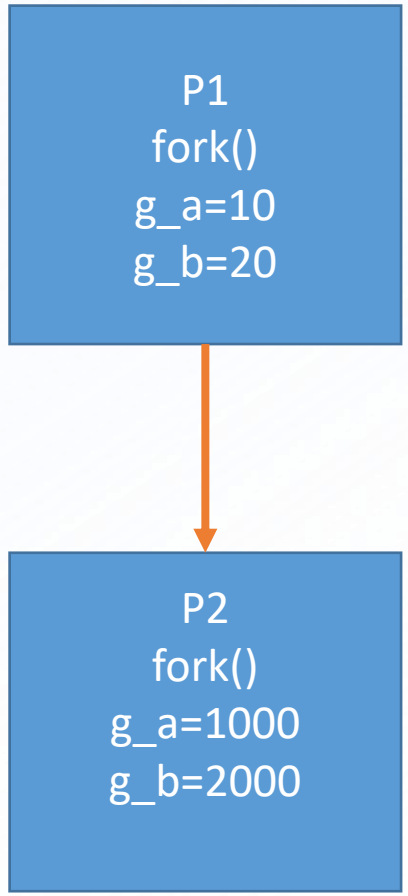

```
printf("-------begin--------\n");
int pid;
pid = fork();
if (pid > 0 )
{

   printf("Output from the Parent proc, returned pid:%d,  MyPid:%d, MyParent:%d \n", pid, getpid(), getppid());
}
else if (pid == 0)
{

  printf("Output from the Child proc, returned pid:%d, MyPid:%d, MyParent:%d \n", pid, getpid(), getppid());
}
else
{

  printf("Error \n");

}

printf("------end---------\n");

return 0;
```

```
-------begin--------
Output from the Parent proc, returned pid:14846,  MyPid:14845, MyParent:6213
------end---------
Output from the Child proc, returned pid:0, MyPid:14846, MyParent:14845
------end---------
```

# 进程创建函数1： pid_t fork(void);

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

int g_a=10;
int g_b=20;

int main()
{

  pid_t pid;
  pid=fork();

  if (pid<0)
     printf("error in fork!\n");
  else if (pid==0)
     {
        g_a=1000;
        g_b=2000;
        printf("Child process, my pid is %d, and g_a= %d, g_b=%d \n",getpid(),g_a,g_b);
     }
  else
    {

        printf("Parent process, my pid is %d, and g_a=%d, and g_b=%d \n",getpid(),g_a,g_b);
    }

}
```

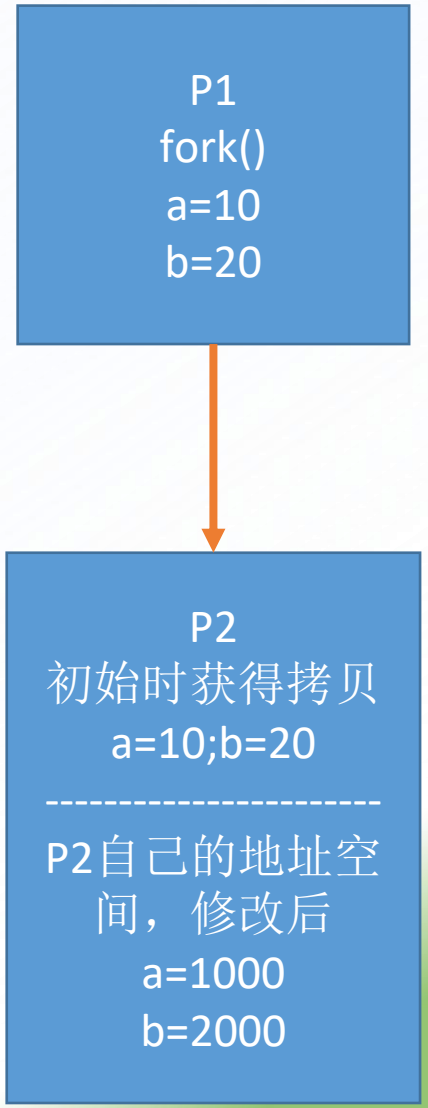
```
[root@localhost sang]# ./as
Child process, my pid is 25234, and g_a= 1000, g_b=2000
Parent process, my pid is 25233, and g_a=10, and g_b=20
```




数据拷贝，但地址不同

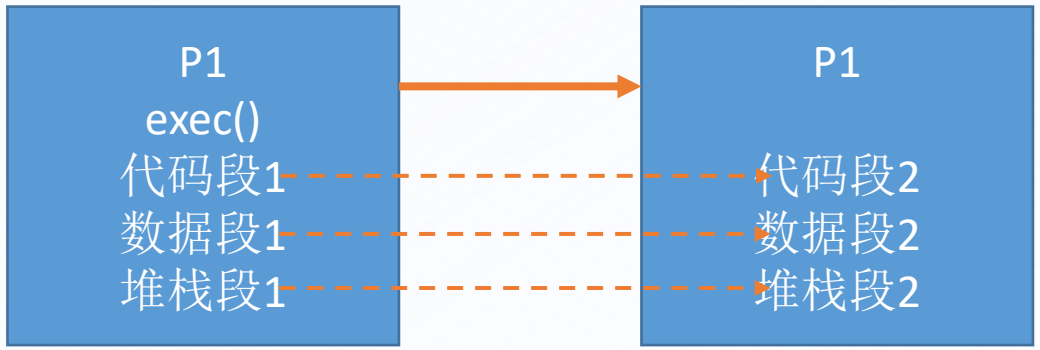
# 进程创建函数1： pid_t fork(void);

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

int main()
{
        int pid;
        int a=10;
        int b=20;
        pid = fork();
        if (pid > 0 )
        {

           printf("in Parent proc,return pid:%d,  MyPid:%d, MyParent:%d \n \n", pid, getpid(), getppid());
           sleep(10);
           printf("in parent proc, a=%d, b=%d \n", a, b);

        }
        else if (pid == 0)
        {
           printf("in Child proc, get a copy of vars from father proc,  ori_a=%d, ori_b=%d \n", a, b);
           a=1000;
           b=2000;
           printf("in Child proc, set new values for private vars,  new_a=%d, new_b=%d \n", a, b);
        }
        else
        {

           printf("Error \n");

        }

        printf(" \n");
        return 0;
```



# 进程创建方法2：exec函数族

- fork创建子进程，子进程几乎拷贝父进程的全部内容；
- exec函数族提供在一个进程中启动另一个完全不同程序的方法。
  - 它可以根据指定的路径和文件名找到可执行文件，并取代原进程的代码段、数据段和堆栈段；
  - 原进程除了进程号外，其它内容全被新的内容替换；
  - 启动的可执行文件可以是二进制文件和可执行脚本




exec并不创建新进程；而是进行进程替代，改变原调用进程的内容，完成不同的工作

# exec函数族-工作原理

1. **验证参数**：检查提供的参数（程序路径、参数列表、环境变量等）的有效性，如路径是否存在、是否有执行权限等。

2. **加载新程序**：从磁盘读取指定程序的可执行文件（通常为ELF格式），将其加载到当前进程的地址空间中。这包括解析文件头、映射代码段、数据段、堆、栈等内存区域，并初始化全局变量、静态变量等。

3. **清理旧资源**：释放当前进程原有的代码、数据等内存区域，关闭不再需要的文件描述符，清除信号处理函数等。由于新程序将完全接管进程，所以原进程的大部分资源都会被清理或重置。

4. **设置执行环境**：根据提供的参数和环境变量列表，设置新程序的执行环境。这包括设置进程参数（`argv`）、环境变量（`envp`）、工作目录、用户ID和组ID等。

5. **跳转执行**：完成上述准备工作后，处理器指令指针（IP）被设置为新程序的入口点（通常是`.text`段的起始地址），然后执行`JMP`指令跳转到新程序的起始代码处执行。至此，当前进程的上下文完全被新程序替换，原程序的所有痕迹消失，仿佛新程序是从头开始执行一样。

# exec函数族

--核心功能相同，参数传递差异；

--执行成功后，原程序的内存空间被新程序所占据，且不再返回到原程序；

--若执行失败，exec函数将返回-1，并设置全局变量errno来指示失败原因。

- 头文件<unistd.h>

```
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 execve(const char *path, char *const argv[], char *const envp[]);
```

| 后缀 | 操作能力 |
| --- | --- |
| l | 希望接收以逗号分隔的参数列表，列表以 NULL 指针作为结束标志 |
| v | 希望接收到一个以 NULL 结尾的字符串数组的指针 |
| p | 是一个以 NULL 结尾的字符串数组指针,函数可以利用 DOS 的 PATH 变量查找子程序文件 |
| e | 函数传递指定参数 envp，允许改变子进程的环境，无后缀 e 时，子进程使用当前程序的环境 |

# exec类函数比较

| 函数 | 参数传递方式 | 自动路径查找 | 环境变量控制 |
| --- | --- | --- | --- |
| execve() | 参数数组+环境数组 | 否 | 是 |
| execl() | 可变参数列表 | 否 | 否 |
| execlp() | 可变参数列表 | 是 | 否 |
| execle() | 可变参数列表+环境数组 | 否 | 是 |
| execv() | 参数数组 | 否 | 否 |
| execvp() | 参数数组 | 是 | 否 |

选择哪个exec函数取决于具体需求：

- ✓ 如果需要完全控制新程序的执行环境（包括环境变量），使用execve()或execle()。
- ✓ 如果希望简化参数传递，避免创建参数数组，可以选择execl()、execlp()或execle()。
- ✓ 如果希望操作系统自动在PATH环境变量中查找程序，使用execlp()或execvp()。
- ✓ 如果仅需替换程序而不关心环境变量，且愿意创建参数数组，使用execv()或execvp()。

# execl函数

- 函数原型
  - int execl(const char *pathname,
    const char *arg0, .../*(char*)0*/); (unistd.h)
- 返回值与参数
  - 出错返回-1，成功不返回值
  - pathname：要执行程序的绝对路径名
  - 可变参数：要执行程序的命令行参数，以“(char *)0”结束

# execl函数示例

```
#include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
int main(void)
{
    printf("entering main process---\n");
    execl("/bin/ls","ls","-l",NULL);
    printf("exiting main process ----\n");
     return 0;
}
```

```
[zxy@test unixenv_c]$ cc execl.c
[zxy@test unixenv_c]$ ./a.out
entering main process---
total 104
-rwxrwxr-x. 1 zxy zxy       5976 Jul 12 22:54 a.out
-rw-r--r--. 1 zxy zxy        527 Jul 12 15:48 atexit02.c
-rw-r--r--. 1 zxy zxy        426 Jul 12 15:59 atexit03.c
-rw-r--r--. 1 zxy zxy        287 Jul 12 15:44 atexit.c
-rw-r--r--. 1 zxy zxy        472 Jul 10 12:39 creathole.c
```

# execl函数

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

int main()
{
  pid_t pid;
  printf("Current PID is: %d\n", getpid());
  execl("/bin/ps","ps","-aux",NULL);

}
```

```
[root@localhost sang]# ./execFns
The current PID is: 11655
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   2072   656 ?        Ss   May01   0:01 init [5]
root         2  0.0  0.0      0     0 ?        S<   May01   0:00 [migration/0]
root         3  0.0  0.0      0     0 ?        SN   May01   0:00 [ksoftirqd/0]
root         4  0.0  0.0      0     0 ?        S<   May01   0:00 [watchdog/0]
root      7756  0.0  0.1  13304  1272 ?        S    May01   0:01 scim-bridge
root      7757  0.0  0.0   2484   664 ?        S    May01   0:00 gnome-pty-helper
root      7758  0.0  0.1   2596  1332 pts/1    Ss+  May01   0:00 bash
root      8480  0.0  0.1   2596  1472 pts/2    Ss+  May01   0:00 bash
root      8518  0.0  3.1 119912 32968 ?        S    May01   0:45 gedit file:///home/sang/p
root     10273  0.0  0.1   2596  1400 pts/5    Ss+  09:12   0:00 bash
root     11446  0.0  0.0   2316   488 ?        Ss   09:44   0:00 /sbin/dhclient -1 -q -lf
root     11655  0.0  0.0   2184   828 pts/4    R+   09:51   0:00 ps aux
root     11918  0.0  0.1   2596  1352 pts/3    Ss+  May02   0:00 bash
root     31880  0.0  0.1   2596  1340 pts/4    Ss   May02   0:00 bash
```

# execv函数

- execv函数
  - int execv(const char *pathname,
    char *const argv[]);
- 返回值与参数
  - 出错返回-1，成功不返回值
  - pathname：要执行程序的绝对路径名
  - argv：数组指针维护的程序参数列表，该数组的最后一个成员必须为NULL

# execv函数示例

```
#include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
int main(void)
{
    printf("entering main process---\n");
    int ret;
    char *argv[] = {"ls","-l",NULL};
    ret = execv("/bin/ls",argv);
    if(ret == -1) perror("execl error");
    printf("exiting main process ----\n");
    return 0;
}
```

```
[zxy@test unixenv_c]$ cc execl.c
[zxy@test unixenv_c]$ ./a.out
entering main process---
total 104
-rwxrwxr-x. 1 zxy zxy       6231 Jul 12 23:09 a.out
-rw-r--r--. 1 zxy zxy        527 Jul 12 15:48 atexit02.c
-rw-r--r--. 1 zxy zxy        426 Jul 12 15:59 atexit03.c
-rw-r--r--. 1 zxy zxy        287 Jul 12 15:44 atexit.c
-rw-r--r--. 1 zxy zxy        472 Jul 10 12:39 creathole.c
```

# execle函数

- execle函数
  - int execle(const char *pathname,
    const char *arg0,.../* (char*)0, char *const envp[] */);
- 返回值与参数
  - 出错返回-1，成功不返回值
  - pathname：要执行程序的绝对路径名
  - 可变参数：要执行程序的命令行参数，以“(char *)0”结束
  - envp指向环境字符串指针数组的指针

# execle函数示例

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

int main()
{
    char *envp[]={“PATH=/tmp”,“USER=shan”,NULL};
    if(fork()==0)
    {
            if(execle(“/bin/ls”,“ls”, “-l”,NULL,envp)<0)
            perror("execle error!");
    }
    return 0;
}
```

# 其他exec类函数

- execve函数
  - int execve(const char *pathname,
    char *const argv[], char *const envp[]);
- execlp函数
  - int execlp(const char *filename,
    const char *arg0, .../* (char*)0*/);
  - Filename参数仅需指定要执行的文件名（其路径信息从环境变量PATH中获取）
  - 例如：PATH=/bin:/usr/bin:/usr/local/bin/
- execvp函数
  - int execvp(const char *filename,
    char *const argv[]);

# Linux线程~进程

- 线程自己基本上不拥有系统资源，只拥有少量在运行中必不可少的资源
  - 程序计数器：标识当前线程执行的位置；
  - 一组寄存器：当前线程执行的上下文内容；
  - 栈：用于实现函数调用、局部变量。因此，局部变量是私有的；
  - 线程信号掩码：因此可以设置每线程阻塞的信号，见本书下一章内容；
  - 局部线程变量：在栈中申请的数据；
  - 线程私有数据
- 进程在使用时占用了大量的内存空间，特别是进行进程间通信时一定要借助操作系统提供的通信机制，这使得进程有自身的弱点；线程占用资源少，使用灵活，很多应用程序中都大量使用线程，而较少的使用多进程
- 线程不能脱离进程而存在，线程的层次关系，执行顺序并不明显，对于初学者大量使用线程会增加程序的复杂度

# Linux线程~进程

- 同进程一样，每个线程也有一个线程ID
- 进程ID在整个系统中是唯一的，但线程ID不同，线程ID只在它所属的进程环境中有效
- 线程ID的类型是pthread_t，在Linux中的定义：
  - 在/usr/include/bits/pthreadtypes.h中
  - typedef unsigned long int pthread_t;

| 应用功能 | 线程 |
| --- | --- |
| 创建 | pthread_create |
| 退出 | pthread_exit |
| 等待 | pthread_join |
| 取消/终止 | pthread_cancel |
| 读取ID | pthread_self() |

# 线程创建-原型

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
```

## 参数：

***（1）pthread_t *thread***：

类型为pthread_t的指针，用于存放新创建线程的标识符，用于后续对线程的管理和同步操作。

***（2）*const pthread_attr_t attr***：

pthread_attr_t结构体的指针，指定线程属性：堆栈大小、调度策略、优先级等，NULL默认。

***（3）void *(*start_routine)(void *)：***

类型为函数指针，指向新线程将要执行的主要工作函数。

***（4）*void arg：***

指向void的指针，用于传递给start_routine函数的参数。

## 返回值：

成功：如果线程创建成功，pthread_create()返回0。

失败：返回非零值，即一个错误编号：

EAGAIN，系统资源暂时不足以创建新的线程；

EINVAL：传递的attr参数包含无效的属性值；ENOMEM：内存不足；（参见errno.h）

# 线程使用注意事项

**线程同步**：新创建的线程与主线程以及其他已存在的线程通常是并发执行的。因此，如果多个线程访问共享资源，需要使用互斥锁（`pthread_mutex_t`）、条件变量（`pthread_cond_t`）等同步机制来确保数据的一致性和避免竞态条件。

**线程生命周期管理**：创建线程后，通常需要通过 `pthread_join()` 来等待线程完成其任务并回收其资源，或者使用 `pthread_detach()` 将其设置为分离状态，以便系统自动清理。不适当的线程管理可能导致资源泄漏或行为不确定。

**线程安全**：确保 `start_routine` 函数以及它所调用的所有函数都是线程安全的，或者在必要时采取适当的同步措施。

**线程局部存储**：如果线程有私有的数据，可以使用 `pthread_key_t` 和相关函数来创建线程局部存储（Thread-Local Storage, TLS），使得每个线程都有自己的数据副本。

# 线程创建例子（访问全局变量）

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

int g_i=10;

void *func(void *arg)
{
   printf("The global g_i is %d \n", g_i);
   return NULL;
}


int main()
{
   pthread_t pth_id;
   int ret;

   ret=pthread_create(&pth_id,NULL,func,NULL);
   if (ret<0)
   {
          printf("Create error!\n");
          return -1;
   }

   printf("Main: the global g_i is %d \n", g_i);
   sleep(1);

   return 0;

}
```

```
Main: the global g_i is 10
The global g_i is 10
```

# 利用互斥锁实现互斥

- 互斥锁数据类型 *pthread_mutex_t*
- 基本操作

（1）初始化互斥锁*int pthread_mutex_init*

- *int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);*
  - ✓ *mutex*：互斥锁地址；
  - ✓ *Attr* ：互斥量属性，默认为*NULL*；
  - ✓ 成功 ：0
  - ✓ 失败 ：非0

# 利用互斥锁实现互斥

（2）上　　锁 *int pthread_mutex_lock(pthread_mutex_t *mutex);*

- **功能：**对互斥锁上锁，若互斥锁已经上锁，则调用者一直阻塞，直到互斥锁解锁后再上锁。
  - ✓成功 ： 0
  - ✓失败 ： 非0

（3）解　锁　 *int pthread_mutex_unlock(pthread_mutex_t * mutex);*

（4）销毁锁　*int pthread_mutex_destroy(pthread_mutex_t *mutex);*

**功能：**销毁指定的一个互斥锁。使用完毕后，必须对互斥锁进行销毁，以释放资源。

# 不考虑互斥

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void printer(char *str)
{
   while(*str!='\0')
    {
        putchar(*str);
        fflush(stdout);
        str++;
        sleep(1);
    }
    printf("\n");
}

void *thread_fun_1(void *arg)
{
    char *str = "Test string from thread1.";
    printer(str);
}

void *thread_fun_2(void *arg)
{
    char *str = "1+2+3+4+5=15 from thread2.";
    printer(str);
}


int main()
{
pthread_t tid1,tid2;
pthread_create(&tid1, NULL, thread_fun_1, NULL);
pthread_create(&tid2, NULL, thread_fun_2, NULL);

pthread_join(tid1, NULL);
pthread_join(tid2,NULL);

}
```

```
T1e+s2t+ 3s+t4r+i5n=g1 5f rformo mt htrheraeda1d.2
```

# 引入互斥锁

```c
pthread_mutex_t mutex; //互斥锁

void printer(char *str)
{
    pthread_mutex_lock(&mutex); //上锁
    while(*str!='\0')
    {
        putchar(*str);
        fflush(stdout);
        str++;
        sleep(1);
    }
    printf("\n");
    pthread_mutex_unlock(&mutex); //解锁
}

int main(void)
{
    pthread_t tid1, tid2;

    pthread_mutex_init(&mutex, NULL); //初始化互斥锁

    // 创建 2 个线程
    pthread_create(&tid1, NULL, thread_fun_1, NULL);
    pthread_create(&tid2, NULL, thread_fun_2, NULL);

    // 等待线程结束，回收其资源
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_mutex_destroy(&mutex); //销毁互斥锁

    return 0;
}
```

```
[root@localhost sang]# ./res
Test string from thread1.
1+2+3+4+5=15 from thread2.
```

# 进程-线程比较

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>

// 假设这是需要并发执行的任务
void task(int id) {
    printf("Task %d started.\n", id);
    sleep(2); // 模拟耗时操作
    printf("Task %d finished.\n", id);
}

int main() {
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time); // 记录程序开始时间

    for (int i = 1; i <= 4; ++i) {
        pid_t child_pid = fork();
        if (child_pid == 0) {
            task(i);
            return 0;
        } else {
            // 父进程等待子进程结束
            wait(NULL);
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &end_time); // 记录程序结束时间
    long long elapsed_ns = (end_time.tv_sec - start_time.tv_sec) * 1000000000LL +
                           (end_time.tv_nsec - start_time.tv_nsec);

    printf("All tasks completed.\n");
    printf("Total time elapsed: %.2f seconds\n", (double)elapsed_ns / 1000000000.0);

    return 0;
}
```

```
Task 1 started.
Task 1 finished.
Task 2 started.
Task 2 finished.
Task 3 started.
Task 3 finished.
Task 4 started.
Task 4 finished.
All tasks completed.
Total time elapsed: 8.00 seconds
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <time.h>

// 假设这是需要并发执行的任务
void *task(void *arg) {
    int id = *(int *)arg;
    printf("Task %d started.\n", id);
    sleep(2); // 模拟耗时操作
    printf("Task %d finished.\n", id);
    pthread_exit(NULL);
}

int main() {
    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time); // 记录程序开始时间

    pthread_t threads[4];
    int task_ids[] = {1, 2, 3, 4};

    for (int i = 0; i < 4; ++i) {
        pthread_create(&threads[i], NULL, task, &task_ids[i]);
    }

    for (int i = 0; i < 4; ++i) {
        pthread_join(threads[i], NULL);
    }

    clock_gettime(CLOCK_MONOTONIC, &end_time); // 记录程序结束时间
    long long elapsed_ns = (end_time.tv_sec - start_time.tv_sec) * 1000000000LL +
                           (end_time.tv_nsec - start_time.tv_nsec);

    printf("All tasks completed.\n");
    printf("Total time elapsed: %.2f seconds\n", (double)elapsed_ns / 1000000000.0);

    return 0;
}
```

```
Task 2 started.
Task 1 started.
Task 3 started.
Task 4 started.
Task 2 finished.
Task 4 finished.
Task 1 finished.
Task 3 finished.
All tasks completed.
Total time elapsed: 2.00 seconds
```