**实验二. 多线程应用程序设计**

1. 实验目的

 了解 Linux 下多线程程序设计的基本原理

 学习 pthread 库函数的使用

 学习多线程间通讯的方法

2. 实验内容

 读懂 pthread.c 的源代码，熟悉几个重要的PTHREAD 库函数的使用，掌握互斥锁和条件变量在多线程 间通讯的使用方法。

 进入/imx6/SRC/exp/basic/02\_pthread 目录，运行 make 产生 pthread 程序，在 ARM 设备端使用 NFS 方式 连接宿主机端试验目录，运行实验测试。

3. 实验环境

 硬件：IMX6 教学平台，PC 机酷睿 i3 以上, 硬盘 120G 以上, 内存 2G 以上

 软件：Vmware Workstation +Yocto 项目

4. 实验原理

4.1 多线程概述

**线程** 是进程的一条执行路径。每个线程共享其所附属的进程的所有的资源，包括打开的文件、页表 （因此也就共享整个用户态地址空间）、信号标识及动态分配的内存等等。

线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一 物理内存空间，当进程退出时该进程所产生的线程都会被强制退出并清除。

线程技术早在 60 年代就被提出，但真正应用多线程到操作系统中去，是在 80 年代中期。传统的Unix 也支持线程的概念，但是在一个进程（process）中只允许有一个线程，这样多线程就意味着多进程。现在， 多线程技术已经被许多操作系统所支持，包括 Windows/NT ，当然，也包括 Linux。

为什么有了进程的概念后，还要再引入线程呢？使用多线程到底有哪些好处？什么系统应该选用多线 程？

使用多线程的理由之一是和进程相比，它是一种非常"节俭"的多任务操作方式。我们知道，在 Linux 系 统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和 数据段，这是一种"昂贵"的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地 址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程 间彼此切换所需的时间也远远小于进程间切换所需要的时间。据统计，总的说来，一个进程的开销大约是 一个线程开销的30 倍左右，当然，在具体的系统上，这个数据可能会有较大的区别。

使用多线程的理由之二是线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进 行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然， 由于同一进程 下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。当 然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为 static 的 数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

除了以上所说的优点外，不和进程比较，多线程程序作为一种多任务、并发的工作方式，还有以下的 优点：

1) 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待 这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming） 置于一个新的线程，可以避免这种尴尬的情况。

2) 使多CPU 系统更加有效。操作系统会保证当线程数不大于 CPU 数目时，不同的线程运行于不同的 CPU 上。

3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分， 这样的程序会利于理解和修改。

Linux 系统下的多线程遵循 POSIX 线程接口，称为 pthread 。编写 Linux 下的多线程程序，需要使用头 文件 pthread.h ，连接时需要使用库 libpthread.a 。LIBC 中的 pthread 库提供了大量的 API 函数，为用户编写 应用程序提供支持。

4.2 软件架构及流程

本实验为著名的生产者－消费者问题模型的实现，主程序中分别启动生产者线程和消费者线程。生产 者线程不断顺序地将 0 到 1000 的数字写入共享的循环缓冲区，同时消费者线程不断地从共享的循环缓冲区 读取数据。流程图如图 2.2.1 所示：

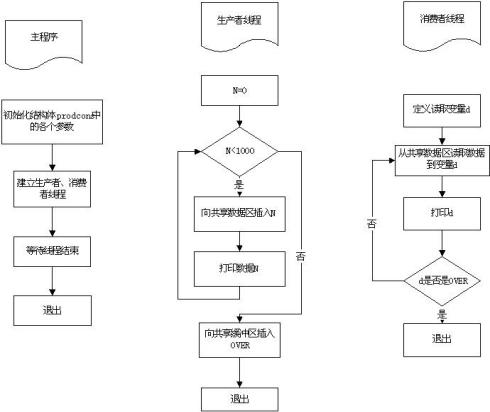


图 2.2.1 生产者-消费者实验源代码结构流程图

4.3 关键代码分析

**1、p** **thread.c** **源文件**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* The classic producer-consumer example.

\* Illustrates mutexes and conditions .

\* by Zou jian guo <ah\_zou@tom .com>

\* 2013-05-15

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio .h>

#include <stdlib.h>

#include <time.h>

#include "pthread .h"

#define BUFFER SIZE 16

/\* 设置一个整数的圆形缓冲区 \*/ struct prodcons {

int buffer [BUFFER\_SIZE];

pthread\_mutex\_t lock;

int readpos, writepos;

pthread\_cond\_t notempty;

pthread\_cond\_t notfull;

};

/\* 缓冲区数组 \*/ /\* 互斥锁 \*/

/\* 读写的位置\*/

/\* 缓冲区非空信号 \*/ /\*缓冲区非满信号 \*/

/\*--------------------------------------------------------\*/

/\*初始化缓冲区\*/

void init (struct prodcons \* b)

{

pthread\_mutex\_init (&b->lock, NULL);

pthread\_cond\_init (&b->notempty, NULL); pthread\_cond\_init (&b->notfull, NULL); b->readpos = 0;

b->writepos = 0; }

/\*--------------------------------------------------------\*/

/\* 向缓冲区中写入一个整数\*/

void put (struct prodcons \* b, int data)

{

pthread\_mutex\_lock (&b->lock);

/\*等待缓冲区非满\*/

while ( (b->writepos + 1) % BUFFER\_SIZE == b->readpos) {

printf ("wait for not full\n");

pthread\_cond\_wait (&b->notfull, &b->lock);

}

/\*写数据并且指针前移\*/

b->buffer [b->writepos] = data; b->writepos++;

if (b->writepos >= BUFFER\_SIZE) b->writepos = 0;

/\*设置缓冲区非空信号\*/

pthread\_cond\_signal (&b->notempty);

pthread\_mutex\_unlock (&b->lock);

}

/\*--------------------------------------------------------\*/

/\*从缓冲区中读出一个整数 \*/

int get (struct prodcons \* b)

{

int data; pthread\_mutex\_lock (&b->lock);



/\* 等待缓冲区非空\*/

while (b->writepos == b->readpos) {

printf ("wait for not empty\n");

pthread\_cond\_wait (&b->notempty, &b->lock); }

/\* 读数据并且指针前移 \*/

data = b->buffer [b->readpos];

b->readpos++;

if (b->readpos >= BUFFER\_SIZE) b->readpos = 0;

/\* 设置缓冲区非满信号\*/

pthread\_cond\_signal (&b->notfull);



pthread\_mutex\_unlock (&b->lock); return data;

}

/\*--------------------------------------------------------\*/

#define OVER (-1)

struct prodcons buffer;

/\*--------------------------------------------------------\*/

void \* producer (void \* data) {

int n;

for (n = 0; n < 1000; n++) {

printf (" put-->%d\n", n);

put (&buffer, n);

}

put (&buffer, OVER);

printf ("producer stopped !\n"); return NULL;

}

/\*--------------------------------------------------------\*/

void \* consumer (void \* data) {

int d;

while (1) {

d = get (&buffer);

if (d == OVER ) break;

printf (" %d-->get\n", d); }

printf ("consumer stopped !\n"); return NULL;

}

/\*--------------------------------------------------------\*/

int main (void) {

pthread\_t th\_a, th\_b; void \* retval;

init (&buffer);

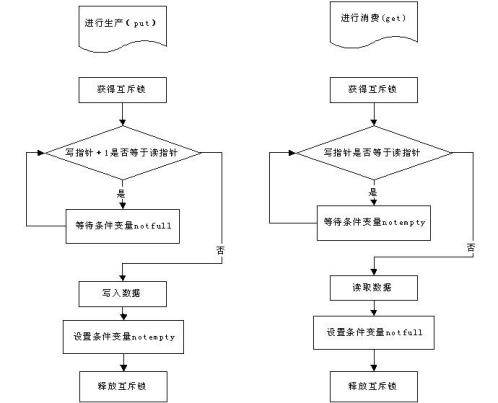
pthread\_create (&th\_a, NULL, producer, 0); pthread\_create (&th\_b, NULL, consumer, 0);

/\* 等待生产者和消费者结束 \*/

pthread\_join (th\_a, &retval); pthread\_join (th\_b, &retval); return 0;

}

下面我们来看一下，生产者写入缓冲区和消费者从缓冲区读数的具体流程，生产者首先要获得互斥锁， 并且判断写指针+1 后是否等于读指针，如果相等则进入等待状态，等候条件变量 notfull；如果不等则向缓 冲区中写一个整数，并且设置条件变量为 notempty ，最后释放互斥锁。消费者线程与生产者线程类似，这 里就不再过多介绍了。流程图如下图 2.2.2 所示：



s

图 2.2.2 生产消费流程图 生产者写入共享的循环缓冲区函数 PUT

void put (struct prodcons \* b, int data) {

pthread\_mutex\_lock (&b->lock); //获取互斥锁

while ( (b->writepos + 1) % BUFFER\_SIZE == b->readpos) {

//如果读写位置相同

pthread\_cond\_wait (&b->notfull, &b->lock); //等待状态变量 b->notfull，不满则跳出阻塞

}

b->buffer [b->writepos] = data; //写入数据

b->writepos++;

if (b->writepos >= BUFFER\_SIZE) b->writepos = 0;

pthread\_cond\_signal (&b->notempty); //设置状态变量 pthread\_mutex\_unlock (&b->lock); //释放互斥锁

}

消费者读取共享的循环缓冲区函数 GET



int get (struct prodcons \* b)

{

int data;

pthread\_mutex\_lock (&b->lock); //获取互斥锁

while (b->writepos == b->readpos) { //如果读写位置相同

pthread\_cond\_wait (&b->notempty, &b->lock);

//等待状态变量 b->notempty，不空则跳出阻塞。否则无数据可读。

}

data = b->buffer [b->readpos]; //读取数据

b->readpos++;

if (b->readpos >= BUFFER\_SIZE) b->readpos = 0;

pthread\_cond\_signal (&b->notfull); //设置状态变量 pthread\_mutex\_unlock (&b->lock); //释放互斥锁

return data;

}

**1、** **主要的多线程API**

在 本 程 序 的 代 码 中 大 量 的 使 用 了 线 程 函 数 ， 如 pthread\_cond\_signal 、 pthread\_mutex\_init 、 pthread\_mutex\_lock 等等，这些函数的作用是什么，在哪里定义的，我们将在下面的内容中为大家做一个简 单的介绍，并且为其中比较重要的函数做一些详细的说明。

线程创建函数：

int pthread\_create (pthread\_t \* thread\_id, const pthread\_attr\_t \* attr,

void \* (\* start\_routine) (void \*),void \* restrict arg)

获得父进程 ID：

pthread\_t pthread\_self (void)

测试两个线程号是否相同：

int pthread\_equal (pthread\_t thread1, pthread\_t thread2)

线程退出：

void pthread\_exit (void \* retval)

等待指定的线程结束：

int pthread\_join (pthread\_t th, void \*\* thread\_return)

互斥量初始化：

pthread\_mutex\_init (pthread\_mutex\_t \*, const pthread\_mutexattr\_t \*) 销毁互斥量：

int pthread\_mutex\_destroy (pthread\_mutex\_t \* mutex)

再试一次获得对互斥量的锁定（非阻塞）：

int pthread\_mutex\_trylock (pthread\_mutex\_t \* mutex)

锁定互斥量（阻塞）：

int pthread\_mutex\_lock (pthread\_mutex\_t \* mutex)

解锁互斥量：

int pthread\_mutex\_unlock (pthread\_mutex\_t \* mutex)

条件变量初始化：

int pthread\_cond\_init (pthread\_cond\_t \* restrict cond,

const pthread\_condattr\_t \* restrict cond\_attr)

销毁条件变量 COND：

int pthread\_cond\_destroy (pthread\_cond\_t \* cond)

唤醒线程等待条件变量：

int pthread\_cond\_signal (pthread\_cond\_t \* cond)

等待条件变量（阻塞）：

int pthread\_cond\_wait (pthread\_cond\_t \* restrict cond, pthread\_mutex\_t

\* restrict mutex)

在指定的时间到达前等待条件变量：

int pthread\_cond\_timedwait (pthread\_cond\_t \* restrict cond,

pthread\_mutex\_t \* restrict mutex, const struct timespec \* restrict

abstime)

PTHREAD 库中还有大量的 API 函数，用户可以参考其他相关书籍。下面我们对几个比较重要的函数 做一下详细的说明：

pthread\_create 线程创建函数

int pthread\_create (pthread\_t \* thread\_id, const pthread\_attr\_t \* attr,

void \* (\* start\_routine) (void \*),void \* restrict arg)

线程创建函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。这里，我们的函数 thread 不需要参数，所以最 后一个参数设为空指针。第二个参数我们也设为空指针，这样将生成默认属性的线程。当创建线程成功时， 函数返回 0 ，若不为 0 则说明创建线程失败，常见的错误返回代码为 EAGAIN 和 EINVAL 。前者表示系统 限制创建新的线程，例如线程数目过多了；后者表示第二个参数代表的线程属性值非法。创建线程成功后， 新创建的线程则运行参数三和参数四确定的函数，原来的线程则继续运行下一行代码。

pthread\_join 函数 用来等待一个线程的结束。函数原型为：

int pthread\_join (pthread\_t th, void \*\* thread\_return)

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程 的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数 返回时，被等待线程的资源被收回。

pthread\_exit 函数

一个线程的结束有两种途径，一种是象我们上面的例子一样，函数结束了，调用它的线程也就结束了； 另一种方式是通过函数 pthread\_exit 来实现。它的函数原型为：

void pthread\_exit (void \* retval)

唯一的参数是函数的返回代码，只要 pthread\_join 中的第二个参数 thread\_return 不是 NULL ，这个值将 被传递给 thread\_return 。最后要说明的是，一个线程不能被多个线程等待，否则第一个接收到信号的线程成 功返回，其余调用 pthread\_join 的线程则返回错误代码 ESRCH。

下面我们来介绍有关条件变量的内容。使用互斥锁来可实现线程间数据的共享和通信，互斥锁一个明 显的缺点是它只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的 方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满 足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知

第 79 页

相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件 是否满足。一般说来，条件变量被用来进行线线程间的同步。

pthread\_cond\_init 函数

条件变量的结构为 pthread\_cond\_t ，函数 pthread\_cond\_init（） 被用来初始化一个条件变量。它的原型 为：

int pthread\_cond\_init (pthread\_cond\_t \* cond, const pthread\_condattr\_t \*

cond attr)

其中 cond 是一个指向结构 pthread\_cond\_t 的指针，cond\_attr 是一个指向结构 pthread\_condattr\_t 的指针。 结构 pthread\_condattr\_t 是条件变量的属性结构，和互斥锁一样我们可以用它来设置条件变量是进程内可用 还是进程间可用，默认值是 PTHREAD\_ PROCESS\_PRIVATE，即此条件变量被同一进程内的各个线程使用。 注意初始化条件变量只有未被使用时才能重新初始化或被释放。释放一个条件变量的函数为 pthread\_cond\_ destroy（pthread\_cond\_t cond）。

pthread\_cond\_wait 函数 使线程阻塞在一个条件变量上。它的函数原型为：

extern int pthread\_cond\_wait (pthread\_cond\_t \* restrict cond,

pthread\_mutex\_t \* restrict mutex)

线程解开 mutex 指向的锁并被条件变量 cond 阻塞 。线程可以被函数 pthread\_cond\_signal 和函数 pthread\_cond\_broadcast 唤醒，但是要注意的是，条件变量只是起阻塞和唤醒线程的作用，具体的判断条件 还需用户给出，例如一个变量是否为 0 等等，这一点我们从后面的例子中可以看到。线程被唤醒后，它将 重新检查判断条件是否满足，如果还不满足，一般说来线程应该仍阻塞在这里，被等待被下一次唤醒。这 个过程一般用 while 语句实现。

pthread\_cond\_timedwait 函数

另一个用来阻塞线程的函数是 pthread\_cond\_timedwait（） ，它的原型为：

extern int pthread\_cond\_timedwait P ( (pthread\_cond\_t \* cond,

pthread\_mutex\_t \* mutex, const struct timespec \* abstime))

它比函数 pthread\_cond\_wait（） 多了一个时间参数，经历 abstime 段时间后，即使条件变量不满足，阻 塞也被解除。

pthread\_cond\_signal 函数 它的函数原型为：

extern int pthread\_cond\_signal (pthread\_cond\_t \* cond)

它用来释放被阻塞在条件变量 cond 上的一个线程。多个线程阻塞在此条件变量上时，哪一个线程被唤 醒是由线程的调度策略所决定的。要注意的是，必须用保护条件变量的互斥锁来保护这个函数，否则条件 满足信号又可能在测试条件和调用 pthread\_cond\_wait 函数之间被发出，从而造成无限制的等待。

5. 实验步骤

5. 1 实验目录

/imx6/SRC/exp/basic/02\_pthread

5.2 编译源程序

1 、进入实验目录：

root@uptech-virtual-machine:/#**cd /imx6/SRC/exp/basic/02 pthread/**

root@uptech-virtual-machine:/imx6/SRC/exp/basic/02\_pthread# **ls**

Makefile pthread pthread.c pthread.o

2 、清除中间代码，重新编译

root@uptech-virtual-machine:/imx6/SRC/exp/basic/02\_pthread#**source**

**/opt/fsl-imx-wayland/4 .9 .88-2.0.0/environment-setup-cortexa9hf-neon-poky-linu**

**x-gnueabi**

root@uptech-virtual-machine:/imx6/SRC/exp/basic/02\_pthread#**make clean**

rm -f ../bin/pthread ./pthread \*.elf \*.gdb \*.o

root@uptech-virtual-machine:/imx6/SRC/exp/basic/02\_pthread#**make**

root@uptech-virtual-machine:/imx6/SRC/exp/basic/02\_pthread#**ls**

Makefile pthread pthread.c pthread.o

root@uptech-virtual-machine:/imx6/SRC/exp/basic/02\_pthread#

当前目录下生成可执行程序 pthread。

5.3 NFS 挂载实验目录测试

启动 IMX6 嵌入式教学科研平台，连好网线、串口线。通过串口终端挂载宿主机实验目录。

root@IMX6DLsabresd :~# **mount -t nfs 192.168.12 .157 :/imx6 /mnt/**

进入串口终端的 NFS 共享实验目录。

root@IMX6DLsabresd :~# **cd /mnt/SRC/exp/basic/02 pthread/**

root@IMX6DLsabresd:/mnt/SRC/exp/basic/02\_pthread# **ls**

Makefile pthread pthread .c pthread .o

执行程序。

root@IMX6DLsabresd:/mnt/SRC/exp/basic/02\_pthread# **./pthread**

6. 实验效果

root@IMX6DLsabresd:/mnt/SRC/exp/basic/02\_pthread# **./pthread**

put-->0

put-->1

put-->2

put-->3

put-->4

put-->5

put-->6

put-->7

put-->8

put-->9

put-->10

put-->11

put-->12

put-->13

put-->14

put-->15

wait for not full

0-->get

1-->get

2-->get

3-->get

4-->get

5-->get

6-->get

7-->get

8-->get

9-->get

10-->get

11-->get

12-->get

13-->get

14-->get

wait for not empty

15-->get

wait for not empty

put-->16

16-->get

wait for not empty

put-->17

17-->get

wait for not empty

put-->18

18-->get

wait for not empty

put-->19

19-->get

wait for not empty

put-->20

20-->get

wait for not empty

put-->21

21-->get

wait for not empty

put-->22

22-->get

wait for not empty

put-->23

23-->get

wait for not empty

put-->24

24-->get

wait for not empty