

项目3 *LiteOS* 同步实验 —— 生产者-消费者问题

22920212204392 黄勛

1 实验目的

2 实验环境

3 实验思路

4 实验内容

4.1 编写程序hxproj3.c

4.2 编译并烧录运行

4.2.1 makefile

4.2.2 烧录运行

5 实验结果

6 实验分析

7 实验总结

8 参考文献

9 附录

9.1 pthread线程库各函数作用

9.2 线程属性pthread_attr_t

1 实验目的

在 `LiteOS` 中利用 Pthread 库，实现生产者-消费者问题。

2 实验环境

宿主机操作系统: `Windows 10`

虚拟机操作系统: `Ubuntu 18.04.6`，在完成Project2的虚拟机中完成

开发板: `IMAX6ULL MINI`

终端工具: `MobaXterm`

3 实验思路

根据前面的project的经验和实现，显然我们已经可以直接利用 Pthread 库进行编程，所以我们只需要编写实现生产者-消费者问题的代码并烧录运行即可。

4 实验内容

4.1 编写程序hxproj3.c

首先编写实现上述功能的程序，大致思路已写在注释中，具体函数解释详见后文。

```
/* LiteOS proj3 */
#include <stdio.h>
#include <pthread.h>

#define MAX_NUM 10

typedef struct {
    int buffer[MAX_NUM]; // 缓冲区
    size_t produce_idx; // 生产者索引
    size_t consume_idx; // 消费者索引
    size_t count; // 缓冲区中的产品数量
} BUFFER;

BUFFER buf; // 缓冲区
pthread_t thread[2]; // 两个线程
pthread_mutex_t mutex; // 锁
pthread_cond_t prod, cons; // 用来实现条件变量

void *producer(void * nul) {
    // 生产者
    int i;
    for (i = 1; i < MAX_NUM; ++i) {
```

```

pthread_mutex_lock(&mutex); // 加锁

while(buf.count != 0) { // 等待条件变量
    pthread_cond_wait(&prod, &mutex);
}

buf.buffer[buf.produce_idx] = i; // 生产产品
buf.produce_idx = (buf.produce_idx + 1) % MAX_NUM;
++buf.count;

printf("producer produces %d \n", i );

pthread_cond_signal(&cons); // 通知 consumer
pthread_mutex_unlock(&mutex); // 解锁
}

pthread_exit(NULL);
}

void *consumer(void * nul) {
    // 消费者
    int i;
    for (i = 1; i < MAX_NUM; ++i) {
        pthread_mutex_lock(&mutex);

        while(buf.count == 0) { // 如果缓冲区空了, 就等待
            pthread_cond_wait(&cons, &mutex); // 等待条件变量
        }

        int item = buf.buffer[buf.consume_idx]; // 消费产品
        buf.consume_idx = (buf.consume_idx + 1) % MAX_NUM;
        --buf.count;

        printf("consumer consumes %d \n", item);

        pthread_cond_signal(&prod); //通知 producer
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main(int argc, char const *argv[]) {
    //锁和条件变量
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&prod, NULL);
    pthread_cond_init(&cons, NULL);

    //producer

```

```

printf("producer start\n");fflush(stdout);
pthread_create(&thread[0], NULL, producer, NULL);

//consumer
printf("consumer start\n");fflush(stdout);
pthread_create(&thread[1], NULL, consumer, NULL);

//等待线程结束
pthread_join(thread[0], NULL);
pthread_join(thread[1], NULL);

//清理资源
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&cons);
pthread_cond_destroy(&prod);
pthread_exit(NULL);

return 0;
}

```

4.2 编译并烧录运行

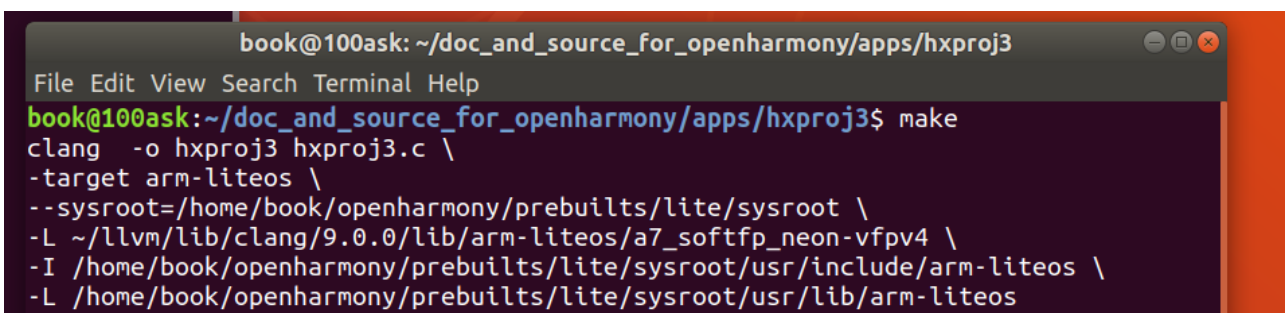
4.2.1 makefile

```

LITEOS_SOURCE_TOP_DIR = /home/book/openharmony
all:
    clang -o hxproj3 hxproj3.c \
    -target arm-liteos \
    --sysroot=$(LITEOS_SOURCE_TOP_DIR)/prebuilts/lite/sysroot \
    -L ~/llvm/lib/clang/9.0.0/lib/arm-liteos/a7_softfp_neon-vfpv4 \
    -I $(LITEOS_SOURCE_TOP_DIR)/prebuilts/lite/sysroot/usr/include/arm-liteos \
    -L $(LITEOS_SOURCE_TOP_DIR)/prebuilts/lite/sysroot/usr/lib/arm-liteos
clean:
    rm -f *.o hxproj3

```

执行make



```

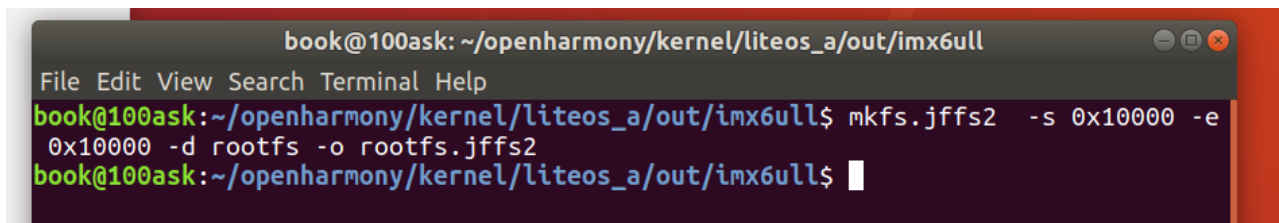
book@100ask: ~/doc_and_source_for_openharmony/apps/hxproj3
File Edit View Search Terminal Help
book@100ask:~/doc_and_source_for_openharmony/apps/hxproj3$ make
clang -o hxproj3 hxproj3.c \
-target arm-liteos \
--sysroot=/home/book/openharmony/prebuilts/lite/sysroot \
-L ~/llvm/lib/clang/9.0.0/lib/arm-liteos/a7_softfp_neon-vfpv4 \
-I /home/book/openharmony/prebuilts/lite/sysroot/usr/include/arm-liteos \
-L /home/book/openharmony/prebuilts/lite/sysroot/usr/lib/arm-liteos

```

4.2.2 烧录运行

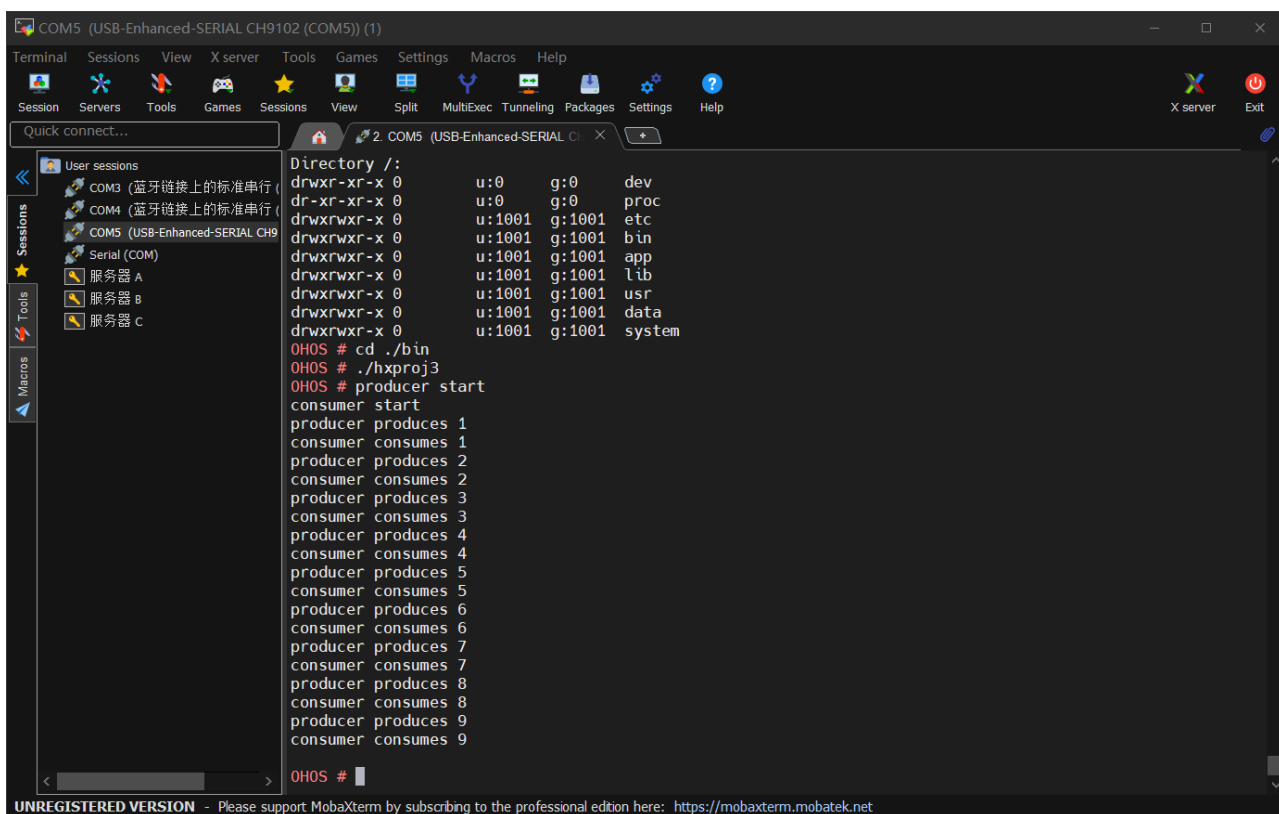
把hxproj3程序放入rootfs后

```
mkfs.jffs2 -s 0x10000 -e 0x10000 -d rootfs -o rootfs.jffs2
```



led.imx	2020/11/2 17:21	IMX 文件	7 KB
liteos.bin	2023/11/15 17:15	BIN 文件	888 KB
rootfs.img	2020/11/2 17:21	光盘映像文件	2,259 KB
rootfs.jffs2	2023/11/23 16:46	JFFS2 文件	907 KB
rt-smart-imx	2020/11/2 17:21	IMX 文件	1.899 KB

5 实验结果



从运行的结果看，程序成功实现并体现出了生产者生产内容，消费者消费内容的过程，实现了生产者-消费者功能。

6 实验分析

在此处大致分析一下编写的程序：

1. **初始化：** 在程序开始时，初始化了一个缓冲区（BUFFER结构体）、两个线程、一个互斥锁和两个条件变量。

2. **生产者函数**: `producer` 函数表示生产者的行为。它会不断地生成数据并将数据放入缓冲区。在这个过程中, 它使用互斥锁确保在对缓冲区进行读写时不会被其他线程中断。同时, 它使用条件变量 `prod` 来判断是否可以生产, 如果缓冲区不为空, 就等待。生产完成后, 通知消费者线程, 然后释放互斥锁。
3. **消费者函数**: `consumer` 函数表示消费者的行为。它会不断地从缓冲区中取出数据并进行消费。在这个过程中, 它使用互斥锁确保在对缓冲区进行读写时不会被其他线程中断。同时, 它使用条件变量 `cons` 来判断是否可以消费, 如果缓冲区为空, 就等待。消费完成后, 通知生产者线程, 然后释放互斥锁。
4. **主函数**: 在 `main` 函数中, 创建了两个线程, 一个是生产者线程, 一个是消费者线程。然后等待这两个线程运行结束。最后, 清理资源, 包括销毁互斥锁和条件变量。

通过这种方式, 程序确保了生产者和消费者之间的同步和互斥, 避免了潜在的竞争条件和数据不一致性问题。这是一种常见的多线程编程模型, 用于解决多个线程对共享资源进行读写时可能引发的问题。

7 实验总结

这次project我学会了如何通过互斥锁和条件变量来实现线程之间的同步, 从而解决生产者-消费者问题。这是多线程编程中常见的一种模型, 用于确保多个线程之间对共享资源的安全访问。同时, 通过条件变量的巧妙运用, 实现了高效的线程同步。

8 参考文献

csdn: <https://blog.csdn.net/pingxiaozhao/article/details/122224969>

9 附录

9.1 pthread线程库各函数作用

1. `pthread_create(pthread_t *restrict thread, const pthread_attr_t restrict attr, void (start_routine)(void), void *restrict arg);`

- **作用**: 创建一个新线程, 执行 `start_routine` 函数, 并传递 `(void*)args` 作为参数。
- **参数**:
 - `thread`: 所创建的线程号。
 - `attr`: 所创建的线程属性, 这个将在 8.2 详细说明。
 - `start_routine`: 即将运行的线程函数。
 - `ar`: 传递给线程函数的参数。

2. `pthread_mutex_init(&mutex, NULL);`

- **作用**: 用于初始化互斥量。
- **参数**:

- `&mutex`: 互斥量的地址, 表示将要被初始化的互斥量。
- `NULL`: 用于指定互斥量的属性, `NULL` 表示使用默认属性。

3. `pthread_mutex_destroy(&mutex)`:

- 作用: 用于销毁互斥量。
- 参数:
 - `&mutex`: 要销毁的互斥量的地址。

4. `pthread_mutex_lock(&mutex)`:

- 作用: 锁定互斥量, 确保只有一个线程能够进入临界区。
- 参数:
 - `&mutex`: 要锁定的互斥量的地址。

5. `pthread_mutex_unlock(&mutex)`:

- 作用: 解锁互斥量, 允许其他线程锁定它。
- 参数:
 - `&mutex`: 要解锁的互斥量的地址。

6. `pthread_cond_init(&cond, NULL)`:

- 作用: 初始化条件变量。
- 参数:
 - `&cond`: 条件变量的地址, 表示将要被初始化的条件变量。
 - `NULL`: 用于指定条件变量的属性, `NULL` 表示使用默认属性。

7. `pthread_cond_destroy(&cond)`:

- 作用: 销毁条件变量。
- 参数:
 - `&cond`: 要销毁的条件变量的地址。

8. `pthread_cond_signal(&cond)`:

- 作用: 发送信号给等待在条件变量 `cond` 上的一个线程。
- 参数:
 - `&cond`: 要发送信号的条件变量的地址。

9. pthread_cond_wait(&cond, &mutex):

- 作用： 使调用线程等待条件变量 `cond` 被通知。
- 参数：
 - `&cond`: 要等待的条件变量的地址。
 - `&mutex`: 相关联的互斥量的地址。在等待期间，会释放这个互斥量，等待结束后重新获取。

10. pthread_join(thread[i], NULL):

- 作用： 等待线程 `thread[i]` 的结束。
- 参数：
 - `thread[i]`: 要等待的线程的标识符。
 - `NULL`: 用于存储目标线程的返回值。如果不关心返回值，可以传递NULL。

这些函数用于实现线程同步、互斥和条件等待，确保多线程程序的正确执行。在使用时，需要小心处理锁的获取和释放，以及条件变量的正确使用，以避免潜在的并发问题。

9.2 线程属性pthread_attr_t

Posix线程中的线程属性pthread_attr_t主要包括scope属性、detach属性、堆栈地址、堆栈大小、优先级。在pthread_create中，把第二个参数设置为NULL的话，将采用默认的属性配置。

pthread_attr_t的主要属性的意义如下：

1. `__detachstate`，表示新线程是否与进程中其他线程脱离同步，如果设置为 `PTHREAD_CREATE_DETACHED` 则新线程不能用pthread_join()来同步，且在退出时自行释放所占用的资源。缺省为 `PTHREAD_CREATE_JOINABLE` 状态。这个属性也可以在线程创建并运行以后用 pthread_detach() 来设置，而一旦设置为 `PTHREAD_CREATE_DETACH` 状态（不论是创建时设置还是运行时设置）则不能再恢复到 `PTHREAD_CREATE_JOINABLE` 状态。

注：在本次实验中，我们设置的是默认的 `PTHREAD_CREATE_JOINABLE` 状态，故在代码中省略了相关内容，实际在main()函数中应当有：

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
pthread_create(&thread[0], &attr, producer, NULL);
pthread_create(&thread[1], &attr, consumer, NULL);
.....
pthread_attr_destroy(&attr);
```

2. `__schedpolicy`，表示新线程的调度策略，主要包括 `SCHED_OTHER`（正常、非实时）、`SCHED_RR`（实时、轮转法）和 `SCHED_FIFO`（实时、先入先出）三种，缺省为

`SCHED_OTHER`，后两种调度策略仅对超级用户有效。运行时可以用过 `pthread_setschedparam()` 来改变。

3. `__schedparam`，一个 struct `sched_param` 结构，目前仅有一个 `sched_priority` 整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即 `SCHED_RR` 或 `SCHED_FIFO`）时才有效，并可以在运行时通过 `pthread_setschedparam()` 函数来改变，缺省为 0。
4. `__inheritsched`，有两种值可供选择：`PTHREAD_EXPLICIT_SCHED` 和 `PTHREAD_INHERIT_SCHED`，前者表示新线程使用显式指定调度策略和调度参数（即 `attr` 中的值），而后者表示继承调用者线程的值。缺省为 `PTHREAD_EXPLICIT_SCHED`。
5. `__scope`，表示线程间竞争CPU的范围，也就是说线程优先级的有效范围。POSIX的标准中定义了两个值：`PTHREAD_SCOPE_SYSTEM` 和 `PTHREAD_SCOPE_PROCESS`，前者表示与系统中所有线程一起竞争CPU时间，后者表示仅与同进程中的线程竞争CPU。目前LinuxThreads仅实现了 `PTHREAD_SCOPE_SYSTEM` 一值。

为了设置这些属性，POSIX定义了一系列属性设置函数，包括 `pthread_attr_init()`、`pthread_attr_destroy()` 和与各个属性相关的 `pthread_attr_get XXX/ pthread_attr_set XXX` 函数。

在设置线程属性 `pthread_attr_t` 之前，通常先调用 `pthread_attr_init` 来初始化，之后来调用相应的属性设置函数。