APUE - 14

阅读目的：了解进程间通信中的信号量、共享内存的几实现方式及用法。
阅读时间：4 小时
阅读概况：第 15 章

第 15 章　进程间通信

5. 信号量

信号量的作用和消息队列不太一样，消息队列的作用 是进程之间传递消息。而信号量的作用是为了同步多个进程的操作。

一般来说，信号量是和某种预先定义的资源相关联的。信号量元素的值表示与之关联的资源的个数。内核会负责维护信号量的值，并确保其值不小于0。

信号量上支持的操作有：

- 将信号量的值设置成一个绝对值。
- 在信号量当前值的基础上加上一个数量。
- 在信号量当前值的基础上减去一个数量。
- 等待信号量的值等于 0。

将信号量和某种资源关联起来，就起到了同步使用某种资源的功能：

操作	功能
将信号量的值设为某绝对值	初始化资源的个数为某绝对值
在信号量当前值的基础上加上一个数量 N	释放 N 个资源
在信号量当前值的基础上减去一个数量 M	申请 M 个资源，可能因资源不足而陷入阻塞
等待信号量的值变为 0	等待可用资源个数变为 0，可能会陷入阻塞

使用最广泛的信号量是二值信号量（binary semaphore）。对于这种信号量而言，它只有两种合法值：0 和 1，对应一个可用的资源。

若当前有资源可用，则与之对应的二值信号量的值为 1；若资源已被占用，则与之对应的二值信号量的值为 0。当进程申请资源时，如果当前信号量的值为0，那么进程会陷入阻塞，直到有其他进程释放资源，将信号量的值加 1 才能被唤醒。

> 二值信号量和互斥量所起的作用非常类似。那信号量和互斥量有何不同之处呢?
>
> 互斥量（mutex）是用来保护临界区的，所谓临界区，是指同一时间只能容许一个进程进入。而信号量（semaphore）是用来管理资源的，资源的个数不一定是 1，可能同时存在多个一模一样的资源，因此容许多个进程同时使用资源。
>
> 信号量是互斥量的一个扩展，由于资源数目增多，增强了并行度。但是这仅仅是一个方面。更重要的区别是，互斥量和信号量解决的问题是不同的。
>
> **互斥量的关键在于互斥、排它，同一时间只允许一个线程访问临界区。**这种严格的互斥，决定了解铃还须系铃人，即加锁进程必然也是解锁进程。
>
> **而信号量的关键在于资源的多少和有无。申请资源的进程不一定要释放资源，**信号量同样可以用于生产者--消费者的场景。在这种场景下， 生产者进程只负责增加信号量的值，而消费者进程只负责减少信号量的值。彼此之间通过信号量的值来同步。

创建或打开信号量

创建或打开信号量的函数为semget，其接口定义如下:

```c
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

在创建信号量时，需要考虑的问题是系统限制。系统的限制可以分成三个层面：

- 系统容许的信号量集的上限：SEMMNI
- 单个信号量集中信号量的上限：SEMMSL
- 系统容许的信号量的上限：SEMMNS

**上限数量根据系统实现决定*

控制信号量

控制信号量的函数为 semctl 函数，其定义如下:

```c
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd,/* union semun arg*/);
```

某些特定的操作需要第四个参数，第四个参数是联合体：

```c
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
    struct seminfo *__buf; /*Linux特有的*/
};
```

根据第三个参数 cmd 值的不同，semctl支持以下命令：

IPC_STAT	Fetch the semid_ds structure for this set, storing it in the structure pointed to by *arg.buf*.
IPC_SET	Set the sem_perm.uid, sem_perm.gid, and sem_perm.mode fields from the structure pointed to by *arg.buf* in the semid_ds structure associated with this set. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or by a process with superuser privileges.
IPC_RMID	Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of EIDRM on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals sem_perm.cuid or sem_perm.uid or by a process with superuser privileges.
GETVAL	Return the value of semval for the member *semnum*.
SETVAL	Set the value of semval for the member *semnum*. The value is specified by *arg.val*.
GETPID	Return the value of sempid for the member *semnum*.
GETNCNT	Return the value of semncnt for the member *semnum*.
GETZCNT	Return the value of semzcnt for the member *semnum*.
GETALL	Fetch all the semaphore values in the set. These values are stored in the array pointed to by *arg.array*.
SETALL	Set all the semaphore values in the set to the values pointed to by *arg.array*.

操作信号量

semop函数负责修改集合中一个或多个信号量的值，其定义如下:

```c
#include <sys/sem.h>
```

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

函数的第一个参数是通过 semget 获取到的信号量的标识符 ID。第二个参数是 sembuf 类型的指针。sembuf 结构体定义在 sys/sem.h 头文件中。一般来说，该结构体至少包含以下三个成员变量：

```
struct sembuf {
    unsigned short int sem_num ;
    short sem_op ;
    short sem_flg;
}
```

成员变量 sem_num 解决的是操作哪个信号量的问题。因为信号量集中可能存在多个信号量，需要用这个参数来告知 semop 函数要操作的是哪个信号量，0 表示第一个信号量，1 表示第二个信号量，依此类推，最大为nsems - 1，即不得超过集合中信号量的个数。如果sem_num的值小于 0，或者大于等于集合中信号量的个数，semop 调用则会返回失败，并置 errno 为 EFBIG。

6. 共享内存

共享内存是所有 IPC 手段中最快的一种。它之所以快是因为共享内存一旦映射到进程的地址空间，进程之间数据的传递就不须要涉及内核了。

> 建立共享内存之后，内核完全不参与进程间的通信，这种说法严格来讲并不是正确的。因为当进程使用共享内存时，可能会发生缺页，引发缺页中断，这种情况下内核还是会参与进来的。

进程从此就像操作普通进程的地址空间一样操作这块共享内存，一个进程可以将信息写入这片内存区域，而另一个进程也可以看到共享内存里面的信息，从而达到通信的目的。

允许多个进程同时操作共享内存，就不得不防范竞争条件的出现，比如有两个进程同时执行更新操作，或者一个进程在执行读取操作时，另外一个进程正在执行更新操作。因此，共享内存这种进程间通信的手段通常不会单独出现，总是和信号量、文件锁等同步的手段配合使用。

创建或打开共享内存

shmget 函数负责创建或打开共享内存段，其接口定义如下:

```
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

其中第二个参数 size 必须是正整数，表示要创建的共享内存的大小。内核以页面大小的整数倍来分配共享内存，因此，实 size 会被向上取整为页面大小的整数倍。

第三个参数支持 IPC_CREAT 和 IPC_EXCL 标志位。如果没有设置 IPC_CREAT 标志位，那么第二个参数 size 对共享内存段并无实际意义，但是必须小于或等于共享内存的大小，否则会有 EINVAL 错误。

和消息队列及信号量一样，对于创建共享内存，系统也存在一些限制：

- SHMMNI：系统所能够创建的共享内存的最大个数。
- SHMMIN：一个共享内存段的最小字节数。
- SHMMAX：一个共享内存段的最大字节数。
- SHMALL：系统中共享内存的分页总数。
- SHMSEG：一个进程允许 attach 的共享内存段的最大个数

控制共享内存

shmctl 函数用来控制共享内存，函数接口定义如下:

```
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

cmd 参数指定下列 5 种命令中的一种，使其在 shmid 指定的段上执行。

`IPC_STAT`	Fetch the `shmid_ds` structure for this segment, storing it in the structure pointed to by *buf*.
`IPC_SET`	Set the following three fields from the structure pointed to by *buf* in the `shmid_ds` structure associated with this shared memory segment: `shm_perm.uid`, `shm_perm.gid`, and `shm_perm.mode`. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.
`IPC_RMID`	Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the `shm_nattch` field in the `shmid_ds` structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that `shmat` can no longer attach the segment. This command can be executed only by a process whose effective user ID equals `shm_perm.cuid` or `shm_perm.uid` or by a process with superuser privileges.

Two additional commands are provided by Linux and Solaris, but are not part of the Single UNIX Specification.

`SHM_LOCK`	Lock the shared memory segment in memory. This command can be executed only by the superuser.
`SHM_UNLOCK`	Unlock the shared memory segment. This command can be executed only by the superuser.

使用共享内存

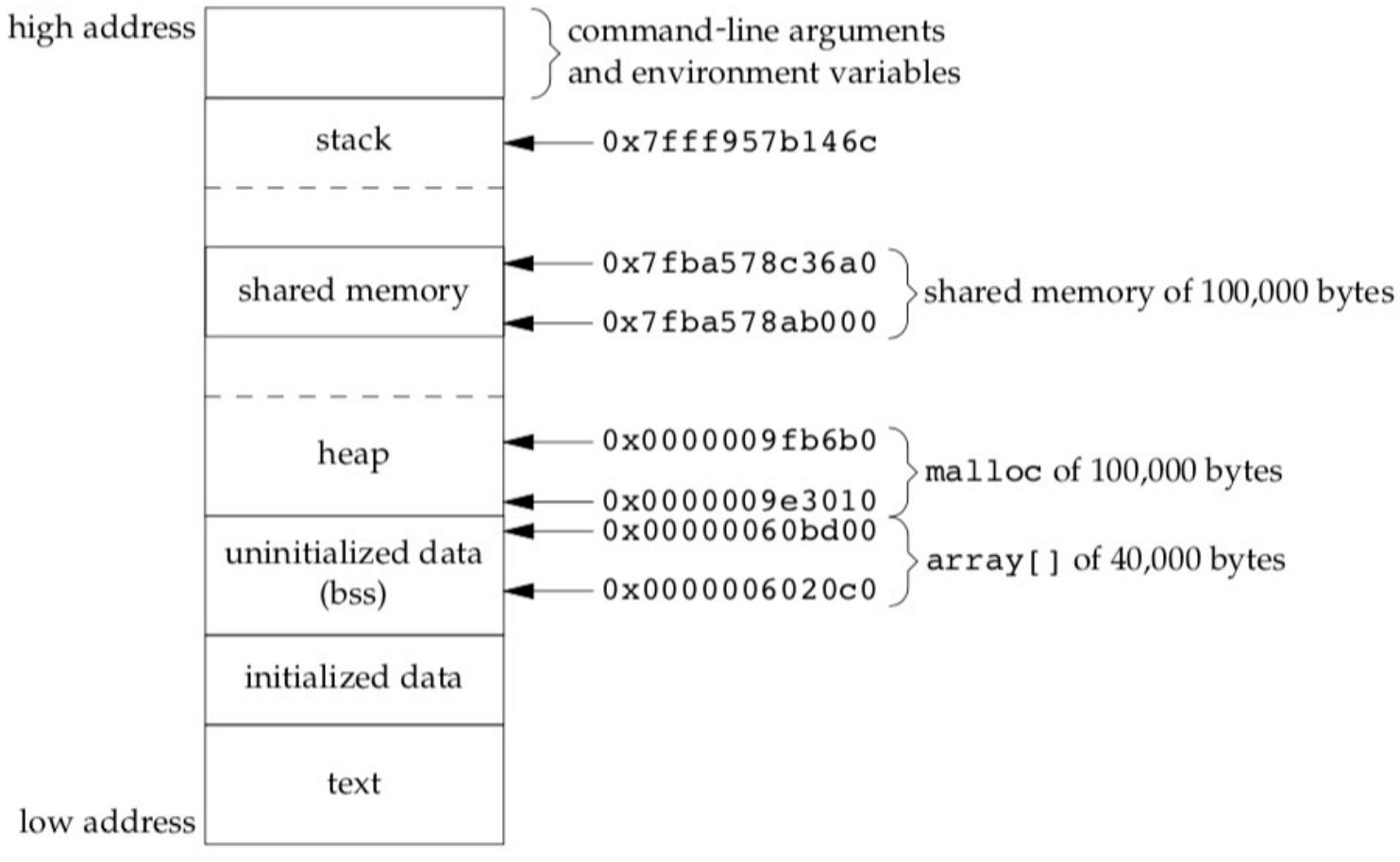
shmget 函数，只是创建或找到一块共享内存区域，但是这块内存和进程尚没有任何关系。要想使用该共享内存，必须先把共享内存引入进程的地址空间，这就是 attach 操作。 attach 操作的接口定义如下：

```
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

> 使用共享内存和使用 malloc 分配的空间的区别。
>
> 共享内存段用于多个进程间的通信，因此，写入共享内存的内容要事先约定好，读取进程才可以正常地解析写入进程写入的内容。
>
> malloc 分配的内存区域完全归调用进程所有，其他进程不可见，但共享内存则不然，其他进程也可能会同时操作该共享内存，因此使用者需要考虑进程间同步的问题。

注意：**共享存储段紧靠在栈之下。**



浅薄的总结：进程间通信的标准较多（System V IPC、POSIX IPC、XSI IPC），且 XSI IPC 不使用文件系统命名空间，所以很多对文件操作的函数不能直接使用。管道和 FIFO 虽然简单且基础，却足以应对大多数应用程序。实际工作和学习很大的区别在于**权衡**：开发时间、犯错成本、稳定性等等。

Keep It Simple，Stupid！