操作系统 2

jask

09/10/2024

补充 1

Linux 对于 Process 和 Thread 在内核中并没有明显区别，都用的是 task_struct 这样的结构体，但是，对于用户来说，区别很大。

早期 Linux 中的 Thread 由 LinuxThread 来提供，这个库不完全兼容 Posix Thread 标准，对于每一个 Thread 都会分配一个 pid，后来变为 NPTL project 并且与 NGPT(Next Generation Posix Thread) 竞争。前者胜出，直至今天还在使用。

进程间通信（IPC）

IPC 对象的持续性

随进程持续的 IPC 对象一直存在到打开该对象的最后一个进程关闭该对象为止。例如管道和 FIFO。

随内核持续的 IPC 对象一直存在到内核重新自举或显式删除该对象为止。例如 Posix 的消息队列，信号量和共享内存区至少是随内核持续的，但是是 Implementation Detail。

随文件系统的 IPC 对象一直存在到显式删除为止。

名字空间

当多于一个无亲缘关系的进程使用某种类型的 IPC 对象进行交换信息时，该 IPC 对象必须具有名字/标识符。

对于一个给定的 IPC 类型，其名字的集合叫做名字空间。

Posix IPC

包含 Posix 消息队列，Posix 信号量，Posix 共享内存区。都是用路径名标识。

对应的创建与打开 IPC 通道的函数分别为：mq_open, sem_open, shm_open。

管道（pipe）

第一个广泛使用的 IPC 形式，可在程序中使用/shell 中使用。

问题是必须由具有共同祖先的进程间才能使用（虽有名管道 named pipe 诞生而解决）。

所谓管道，就是内核里面的一串缓存，任何一个管道只有读取和写入两个口

对于父子进程，父进程关闭读取的口，留下写入的口，子进程则相反，如果需要双向通信需要两个管道

对于匿名管道，它的通信范围是存在父子关系的进程，对于命名管道，它可以在不相关的进程之间也能通信

通信数据采用先进先出原则

管道的通信方式效率低，管道不适合进程之间频繁的交换数据

管道的生命周期随着进程创建而建立，随着进程结束而销毁

内核为管道与 FIFO 维护一个访问计数器，统计访问同一个管道/FIFO 的打开着的描述符的个数。因此如果打开多个 FIFO 时，如果先前的 FIFO 已经打开就要 unlink 删除。

常用函数：

```c
#include<unistd.h>
int pipe(int fd[2]);
```

返回两个 fd，第一个打开来读，第二各打开来写。

宏 S_ISFIFO 可用于确定一个描述符是文件还是管道还是 FIFO。唯一参数是 stat 结构体的 st_mode。

FIFO

FIFO 又被称为有名管道。

由 mkfifo 函数创建。

```
#include<sys/stat.h>
#include<sys/types.h>
int mkfifo(const char* pathname,mode_t mode);
```

FIFO 是半双工的，不能又读又写。

消息队列

System V 消息队列和 Posix 消息队列

Posix 消息队列

```
#include<mqueue.h>
mqg_t mq_open(const char* name,int oflag,...);
int mq_close(mqd_t mqdes);
int mq_unlink(const char* name);
int mq_getattr(mqd_t mqdes,struct mq_sttr *attr);
int mq_setattr(mqd_t mqdes,const struct mq_sttr *attr,struct mq_attr *attr);
int mq_send(mqd_t mqdes,const char* ptr,size_t len,unsigned int prio);
ssize_t mq_receive(mqd_t mqdes,char *ptr,size_t len,unsigned int *priop);
int mq_notify(mqd_t mqdes,const struct sigevent *notification);
```

mq_open 返回值是消息队列描述符，其值用于其余七个消息队列函数的第一个参数

如果消息队列是使用内存映射文件实现的，就具有跟随文件系统的持续性，但这是 Implementation Detail。

Posix 消息队列允许异步事件通知，有两种方式：1. 产生一个信号。
2. 创建一个线程来执行一个指定的函数。

System V 消息队列

```
#include<sys/msg.h>
int msgget(key_t key,int oflag);
int msgsnd(int msqid,const void* *ptr,size_t length,int flag);
ssize_t msgrcv(int msqid,void *ptr,size_t length, long type,int flag);
int msgctl(int msqid,int cmd,struct msqid_ds *buff);
```

共享内存

解决了消息队列的问题。通过 mmap，将一块虚拟地址映射到相同的物理内存中。

共享内存是可用 IPC 形式中最快的。

mmap，munmap，msync 函数

mmap 函数把一个文件或者一个 Posix 共享内存区对象映射到调用进程的地址空间，作用有：

```
void *mmap(void* addr,size_t len,int prot, int flags,int fd,off_t offset);
```

1. 使用普通文件以提供内存映射 I/O;

2. 使用特殊文件以提供匿名内存映射;

3. 使用 shm_open 提供无亲缘关系进程间的 Posix 共享内存区。

为从某个进程的地址空间删除一个映射关系，使用 munmap。

```
int munmap(void *addr,size_t len);
```

addr 是由 mmap 返回的地址，len 是映射区的大小。再次访问这些地址将导致向调用进程产生一个 SIGSEGV 信号。如果被映射区是使用 MAP_PRIVATE 映射的，那么调用进程对他所做的所有变动都会被丢弃。

如果我们希望硬盘上的文件内容与内存映射区的内容一致，需要调用 msync 来执行。

```
int msync(void* addr,size_t len,int flags);
```

flags 是 MS_ASYNC,MS_SYNC,MS_INVALIDATE 的组合。

并不是所有文件都可以 mmap, socket, 终端调用 mmap 会返回一个错误。

Posix 共享内存

1. 指定一个名字参数调用 shm_open, 创建一个新的共享内存去对象或者打开一个已经存在的共享内存区对象。

2. 调用 mmap 把这个共享内存区映射到调用进程的地址空间。

```
#include<sys/mman.h>
int shm_open(const char* name,int oflag,mode_t mode);
int shm_unlink(const char* name);
```

shm_open 返回一个描述符的原因是：mmap 用于把一个内存区对象映射到调用进程地址空间的是该对象的一个已打开的描述符。

shm_open 的返回值可用于 mmap 的第五个参数。

处理 mmap 时，普通文件或共享内存区对象的大小都可以通过调用 ftruncate 修改。

```
#include<unistd.h>
int ftruncate(int fd,off_t length);
```

信号

Posix 信号量

信号是进程间通信机制中唯一的异步通信机制，信号同样可用于线程。

信号的处理方式：执行默认操作，捕捉信号，忽略信号。

进程间共享信号量

信号量本身必须驻留在由所有希望共享他的进程所共享的内存区中，而且 sem_init 第二个参数必须为 1。

Socket

远程过程调用

Unix 上的 rpc 机制使用的是 door 机制。

```
#include<door.h>
int door_call(int fd,door_arg_t *argp);
typedef void Door_server_proc(void *cookie,char *dataptr,size_t datasize,door_desc_t *descptr,size_t ndesc);
int door_create(Door_server_proc *proc,void *cookie,u_int addr);
int door_return(char *dataptr,size_t datasize,door_desc_t *descptr,size_t *ndesc);
int door_cred(door_cred_t *cred);
typedef void Door_create_proc(door_info_t *);
Door_create_proc *door_server_create(Door_create_proc *proc);
```

服务器调用 door_create 创建一个门，并给他关联一个服务器过程，然后调用 fattach 给该门附加一个文件系统中的路径名。

客户对该路径名调用 open, 然后调用 door_call 以调用服务器进程中的服务器过程，该服务器过程通过调用 door_return 返回。

传递描述符

通过一个 door 在客户端和服务端传递描述符的方法：

把 door_arg_t 结构的 desc_ptr 成员设置为指向 door_desc_t 的数组，door_num 成员设置为这些 door_desc_t 结构的数目。

从服务器向客户端传回描述符的手段是，把 door_return 的第三个参数设置成指向 door_desc_t 结构的数组，第四个参数设置成待传递描述符的数目。

同步机制

某些神秘的文件系统的特性

Posix 保证，如果以 O_CREATE 和 O_EXCL 标志调用 open 打开文件，如果文件已经存在就会返回错误，因此可以当做锁用。

问题在于：

1. 持有锁的进程没有释放锁就终止，那么文件就不会删除。

2. 如果另外某个进程已打开锁文件，那么当前进程就会在循环中不断调用 Open，这是轮询，浪费 cpu 资源。

3.open，unlink 效率通常要比 fcntl 低很多。

记录上锁（80 年代添加到 Unix 内核，88 年由 Posix 标准化）

使用 fcntl 进行记录上锁。

Posix 记录上锁称为劝告性上锁，其含义是内核维护这已由各个进程上锁的所有文件的正确信息，但他不能防止一个进程写已由另一个进程读锁定的某个文件。

类似的，他也不能防止一个进程读另一各进程写锁定的某个文件。

信号量

三种操作：创建，等待，挂出。

有名信号量：

```
sem_t *sem_open(const char* name,int oflag,...);
int sem_close(sem_t *sem);//对于 sem_open 打开的要用 sem_close 关闭
int sem_unlink(const char* name);
```

每个信号量有一个引用计数器记录当前的打开次数，sem_unlink 类似于 File IO 的 unlink 函数，当引用计数还是大于 0 时，name 能够删除，然而其信号量的析构要等待最后一个 sem_close 发生为止。

互斥锁与条件变量

互斥锁用来保护临界区（实际上是其中的数据），保证任何时刻只有一个线程在执行其中的代码。

互斥锁是协作性锁，如果共享数据是一个链表，那么操纵该链表的所有线程都必须在实际操作前获取该互斥锁。

读写锁

读写锁的分配规则：
1. 只要没有线程持有某个给定的读写锁用于写，那么任意数目的线程可以持有该读写锁用于读。

2. 仅当没有线程持有某个给定的读写锁用于读或者写时，才能分配该读写锁用于写。

这种给定资源的共享访问也称为共享-独占（shared-exclusive）上锁，读锁叫做 shared lock，写锁叫做 exclusive lock。