<https://blog.csdn.net/jmshl/article/details/6773731>

**APR分析-整体篇**

* 由于部门所使用的底层库与Apache Server有着“一定的渊源”，所以总有一种想看看Apache的实现的冲动。最近项目收尾，愿望终可实现。

**一、何为APR?**

Apache Server经过这么多年的发展后，将一些通用的运行时接口封装起来提供给大家，这就是Apache Portable Run-time libraries, APR。

**二、APR的目录组织**

从[www.apache.org](http://www.apache.org/)上下载apr-1.1.1.tar.gz到本地解压后，发现APR的目录结构很清晰。

1) 所有的头文件都放在$(APR)/include目录中;

2) 所有功能接口的实现都放在各自的独立目录下，如threadproc、mmap等;

3) 此外就是相关平台构建工具文件如Makefile.in等。曾经看过ACE的代码，ACE的所有源文件(.cpp)都放在一个目录下，显得很混乱。APR给我的第一印象还不错。

4) 进入各功能接口子目录，以threadproc为例，在其下面的子目录有5个，分别为beos、netware、os2、unix和win32。从APR的名字也可以理解，每个子目录下都存放着各个平台的独特实现源文件。

**三、APR构建**

如果想要使用APR，需要先在特定平台上构建它，这里不考虑多个平台的特性，仅针对Unix平台进行分析。

1) apr.h、apr.h.in、apr.h.hw和apr.h.hnw的关系

在$(APR)/include目录下，由于APR考虑移植性等原因，最基本的apr.h文件是在构建时自动生成的，其中apr.h.in类似一模板作为apr.h生成程序的输入源。其中apr.h.hw和apr.h.hnw分别是Windows和NetWare的特定版本。

2) 编译时注意事项

在Unix上编译时，注意$(APR)/build下\*.sh文件的访问权限，应该先chmod一下，否则Make的时候会提示ERROR。

**四、应用APR**

我们首先make install一下，比如我们在Makefile中指定prefix=$(APR)/dist，则make install后，在$(APR)/dist下会发现4个子目录，分别为bin、lib、include和build，其中我们感兴趣的只有include和lib。下面是一个APR app的例子project。

该工程的目录组织如下：

$(apr\_path)

 - dist

    - lib

    - include

 - examples

    - apr\_app

      - Make.properties

      - Makefile

      - apr\_app.c

我们的Make.properties文件内容如下：

#

# The APR app demo

#

CC             = gcc -Wall

BASEDIR         =$(HOME)/apr-1.1.1/examples/apr\_app

APRDIR          =$(HOME)/apr-1.1.1

APRVER          = 1

APRINCL         =$(APRDIR)/dist/include/apr-$(APRVER)

APRLIB          =$(APRDIR)/dist/lib

DEFS           = -D\_REENTRANT -D\_POSIX\_PTHREAD\_SEMANTICS -D\_DEBUG\_

LIBS           = -L$(APRLIB) -lapr-$(APRVER) /

                 -lpthread -lxnet -lposix4 -ldl -lkstat -lnsl -lkvm -lz -lelf -lm -lsocket -ladm

INCL           = -I$(APRINCL)

CFLAGS          =$(DEFS) $(INCL)

Makefile文件内容如下：

include Make.properties

TARGET  = apr\_app

OBJS    = apr\_app.o

all: $(TARGET)

$(TARGET): $(OBJS)

        $(CC) ${CFLAGS} -o $@$(OBJS) ${LIBS}

clean:

        rm -f core $(TARGET)$(OBJS)

而apr\_app.c文件采用的是$(apr\_path)/test目录下的proc\_child.c文件。编译运行一切OK！

**五、GO ON**

分析APR的过程也是我学习Unix高级系统机制的过程，有时间我会继续APR分析的。

**APR分析-设计篇**

* 作为一个可移植的运行时环境，APR设计当然是很精妙的，但精妙的同时对使用者有一些限制。

APR附带一个简短的设计文档，文字言简意赅，其中很多的设计思想都值得我们所借鉴，主要从三个方面谈。

**1、类型**

1) APR提供并建议用户使用APR自定义的数据类型，好处很多，比如便于代码移植，避免数据间进行不必要的类型转换（如果你不使用APR自定义的数据类型，你在使用某些APR提供的接口时，就需要进行一些参数的类型转换）；自定义数据类型的名字更加具有自描述性，提高代码可读性。APR提供的基本自定义数据类型包括：

typedef unsigned char  apr\_byte\_t;

typedef short    apr\_int16\_t;

typedef unsigned short  apr\_uint16\_t;

typedef int    apr\_int32\_t;

typedef unsigned int  apr\_uint32\_t;

typedef long long   apr\_int64\_t;

typedef unsigned long long  apr\_uint64\_t;

这些都是在apr.h中定义的，而apr.h在UNIX平台是通过configure程序生成的，在不同平台APR自定义类型的实际类型是完全有可能不一致的。

2) 还有一点值得提的是在APR的设计文档中，它称“dso、mmap、process、thread”等为“base types”。很难用中文理解之，估计是指apr\_mmap\_t这些类型吧。权且这么理解吧^\_^

3) 另外的一个特点就是大多APR类型中都包含一个apr\_pool\_t类型的字段，该字段用于分配APR内部使用的内存，任何APR函数需要内存都可以通过它分配。如果你创建一个新的类型，你最好在该类型中加入一个apr\_pool\_t类型的字段，否则所有操作该类型的APR函数都需要一个apr\_pool\_t类型的参数。

**2、函数**

1) 理解APR的函数设计对阅读APR代码很有帮助。看了APR代码你会发现很多类似APR\_DECLARE(apr\_hash\_t \*) apr\_hash\_make(apr\_pool\_t\*pool)带APR\_DECLARE宏的函数声明，到底是什么意思呢？为什么要加一个APR\_DECLARE呢？在apr.h中有这样的解释：“APR的固定个数参数公共函数的声明形式APR\_DECLARE(rettype) apr\_func(args)；而非固定个数参数的公共函数的声明形式为APR\_DECLARE\_NONSTD(rettype) apr\_func(args, ...);”。在Unix上的apr.h中有这两个宏的定义：

#defineAPR\_DECLARE(type)           type

#define APR\_DECLARE\_NONSTD(type)     type

在apr.h文件中解释了这么做就是为了在不同平台上编译时使用“the most appropriate calling convention”，这里的“calling convention”是一术语，翻译过来叫“调用约定”。[注1]

常见的调用约定有：stdcall、cdecl、fastcall、thiscall和naked call，其中cdecl调用约定又称为C调用约定，是C语言缺省的调用约定。

2) 如果你想新增APR函数,APR建议你最好能按如下做，这样会和APR提供的函数保持最好的一致性：

 a) 输出参数为第一个参数；

 b) 如果某个函数需要内部分配内存，则将一个apr\_pool\_t参数放在最后。

**3、错误处理**

大型的系统程序的错误处理是十分重要的，APR作为一通用的库接口集合详细的说明了使用APR时如何进行错误处理。

1) 错误处理的第一步就是“错误码和状态码分类”。APR的函数大部分都返回apr\_status\_t类型的错误码，这是一个int型，在apr\_errno.h中定义，和它在一起定义的还有apr所用的所有错误码和状态码。APR定义了5种错误码类型，它们分别为“0”[注2]、APR\_OS\_START\_ERROR、APR\_OS\_START\_STATUS、APR\_OS\_START\_USEERR和APR\_OS\_START\_SYSERR，它们每个都拥有自己独自的偏移量。

2) 如何定义错误捕捉策略？

由于APR是可移植的，这样就可能遇到这样一个问题：不同平台错误码的不一致。如何处理呢？APR给我们提供了2种策略：

a) 跨多平台返回相同的错误码

这种策略的缺点是转换费时且在转换时有错误码损耗。比如Windows操作系统定义了成百上千错误码，而POSIX才定义了50错误码，如果都转换为规范统一的错误码，势必会有错误码含义丢失，有可能得不到拥有真正含义的错误码。执行流程如：

make syscall that fails

        convert to common errorcode

        return common errorcode

-------------------------------------------------------------------

           decide execution based on common error code

b) 返回平台相关错误码，如果需要将它转换为通用错误码

程序的执行路线往往要根据函数返回错误码来定，这么做的缺点就是把这些工作推给了程序员。执行流程如：

make syscall that fails

        return error code

-------------------------------------------------------------------

           convert to common error code (using ap\_canonical\_error)

           decide execution based on common error code

[注1] 调用约定

我们知道函数调用是通过栈操作来完成的，在栈操作过程中需要函数的调用者和被调用者在下面的两个问题上做出协调，达成协议：

a) 当参数个数多于一个时，按照什么顺序把参数压入堆栈

b) 函数调用后，由谁来把堆栈恢复原来状态

在像C/C++这样的中、高级语言中，使用“调用约定”来说明这两个问题。

[注2] 特殊“0”

每个平台都有0，但是都没有实际的定义，0又的确是一个errnovalue的offset，但是它是“匿名的”，它不像EEXIST那样有着可以“自描述”的名字。

**APR分析-进程篇**

* Apache Server的进程调度一直为人所称道，Apache 2.0推出的APR对进程进行了封装，特别是Apache 2.0的MPM(Multiple Process Management)框架就是以APR封装的进程为基础的，下面就让我们一起来探索一下APR的进程封装吧(以Unix平台为例)。

APR进程封装源代码的位置在$(APR\_HOME)/threadproc目录下，本篇blog着重分析unix子目录下的proc.c文件内容，其相应头文件为$(APR\_HOME)/include/apr\_thread\_proc.h。

**一、APR进程概述**

APR进程封装采用了传统的fork-exec配合方式(spawn)，即父进程在fork出子进程后继续执行其自己的代码，而子进程调用exec函数加载新的程序映像到其地址空间，执行新的程序。我们先来看看使用APR创建一个新的进程的流程，然后再根据流程做细节分析：

apr\_proc\_t   newproc;

apr\_pool\_t   \*p;

apr\_status\_t rv;

const char \*args[2];

apr\_procattr\_t \*attr;

/\* 初始化APR内部使用的内存 \*/

rv = apr\_pool\_initialize();

HANDLE\_RTVAL(apr\_pool\_initialize, rv);[注1]

rv = apr\_pool\_create(&p, NULL);

HANDLE\_RTVAL(apr\_pool\_create, rv);

/\* 创建并初始化新进程的属性 \*/

rv = apr\_procattr\_create(&attr, p);

HANDLE\_RTVAL(apr\_procattr\_create, rv);

rv = apr\_procattr\_io\_set(attr, APR\_FULL\_BLOCK, APR\_FULL\_BLOCK,

                            APR\_NO\_PIPE); /\* 可选 \*/

HANDLE\_RTVAL(apr\_procattr\_io\_set, rv);

rv = apr\_procattr\_dir\_set(attr,"startup\_path"); /\* 可选 \*/

HANDLE\_RTVAL(apr\_procattr\_dir\_set, rv);

rv = apr\_procattr\_cmdtype\_set(attr, APR\_PROGRAM); /\*可选\*/

HANDLE\_RTVAL(apr\_procattr\_cmdtype\_set, rv);

... ... /\* 其他设置进程属性的函数 \*/

/\* 创建新进程 \*/

args[0] = "proc\_child";

args[1] = NULL;

rv = apr\_proc\_create(&newproc, "your\_progname",args, NULL, attr, p);

HANDLE\_RTVAL(apr\_proc\_create, rv);

/\* 等待子进程结束 \*/

rv = apr\_proc\_wait(&newproc, NULL, NULL, APR\_WAIT);

HANDLE\_RTVAL(apr\_proc\_wait, rv);

**二、APR procattr创建**

在我们平时的Unix进程相关编程时，我们大致会接触两类进程操作函数：进程创建函数(如fork和exec等)和进程属性操作函数(getpid、chdir等)，APR将进程的相关属性信息封装到apr\_procattr\_t结构体中，我们来看看这个重要的结构体定义：(这里只列出Unix下可用的属性)

/\* in $(APR\_HOME)/include/arch/unix/apr\_arch\_threadproc.h \*/

struct apr\_procattr\_t {

    /\* PART 1 \*/

    apr\_pool\_t \*pool;

    /\* PART 2 \*/

    apr\_file\_t \*parent\_in;

    apr\_file\_t \*child\_in;

    apr\_file\_t \*parent\_out;

    apr\_file\_t \*child\_out;

    apr\_file\_t \*parent\_err;

    apr\_file\_t \*child\_err;

    /\* PART 3 \*/

    char \*currdir;

    apr\_int32\_t cmdtype;

    apr\_int32\_t detached;

    /\* PART 4 \*/

    struct rlimit \*limit\_cpu;

    struct rlimit \*limit\_mem;

    struct rlimit \*limit\_nproc;

    struct rlimit \*limit\_nofile;

    /\* PART 5 \*/

    apr\_child\_errfn\_t \*errfn;

    apr\_int32\_t errchk;

    /\* PART 6 \*/

    apr\_uid\_t   uid;

    apr\_gid\_t   gid;

};

我这里将apr\_procattr\_t包含的字段大致分为6部分，下面逐一说明：

**[PART 1]**

在上一篇关于APR的blog中说过，大部分的APR类型中都会有一个apr\_pool\_t类型字段，用于APR内部的内存管理，此结构也无例外。该字段用来标识procattr在哪个pool中分配的内存。

**[PART 2]**

进程不是孤立存在的，进程也是有父有子的。父子进程间通过传统的匿名pipe进行通信。在apr\_procattr\_io\_set(attr, APR\_FULL\_BLOCK,APR\_FULL\_BLOCK, APR\_FULL\_BLOCK)调用后，我们可以用下面的图来表示这些字段的状态：[注3]

parent\_in ----------------------------------------------

                                                                      /|/

                    ------------------------------------------

                    filedes[0]    "in\_pipe"       filedes[1]

                    ------------------------------------------

                   /|/

child\_in ------

parent\_out ----

                    /|/

                    -------------------------------------------

                    filedes[0]    "out\_pipe"       filedes[1]

                    -------------------------------------------

                                                                          /|/

child\_out ----------------------------------------------

parent\_err ----

                    /|/

                    -------------------------------------------

                    filedes[0]    "err\_pipe"       filedes[1]

                    -------------------------------------------

                                                                       /|/

child\_err ------------------------------------------------

还有一点值得注意的是apr\_procattr\_io\_set调用apr\_file\_pipe\_create创建pipe的时候，为相应的in/out字段注册了cleanup函数apr\_unix\_file\_cleanup，apr\_unix\_file\_cleanup在相应的in/out字段的pool销毁时被调用，在后面的apr\_proc\_create时还会涉及到这块儿。

**[PART 3]**

进程的一些常规属性。

currdir标识新进程启动时的工作路径(执行路径)，默认时为和父进程相同；

cmdtype标识新的子进程将执行什么类型的命令；共5种类型，默认为APR\_PROGRAM，定义见[注2]

detached标识新进程是否为分离后台进程，默认为前台进程。

**[PART 4]**

这4个字段标识平台对进程资源的限制，一般我们接触不到。struct rlimit的定义在/usr/include/sys/resource.h中。

**[PART 5]**

errfn为一函数指针，原型为typedef void (apr\_child\_errfn\_t)(apr\_pool\_t \*proc,apr\_status\_t err, const char \*description); 这个函数指针如果被赋值，那么当子进程遇到错误退出前将调用该函数。

errchk一个标志值，用于告知apr\_proc\_create是否对子进程属性进行检查，如检查curdir的access属性等。

**[PART 6]**

用户ID和组ID，用于检索允许该用户所使用的权限。

**三、APR proc创建**

APR proc的描述结构为apr\_proc\_t：

typedef struct apr\_proc\_t {

    /\*\* The process ID \*/

    pid\_t pid;

    /\*\* Parent's side of pipe to child's stdin \*/

    apr\_file\_t \*in;

    /\*\* Parent's side of pipe to child's stdout \*/

    apr\_file\_t \*out;

    /\*\* Parent's side of pipe to child's stdouterr\*/

    apr\_file\_t \*err;

} apr\_proc\_t;

结构中有很清晰明了的注释，这里就不再说了。

创建一个新的进程的接口为apr\_proc\_create，其参数也都很简单。前面说过apr\_proc\_create先fork出一个子进程，众所周知fork后子进程是父进程的复制品[注4]，然后子进程再通过exec函数加载新的程序映像，并开始执行新的程序。这里分析一下apr\_proc\_create的执行流程，其伪码如下：

apr\_proc\_create

{

   if (attr->errchk)

      对attr做有效性检查，让错误尽量发生在parentprocess中，而不是留给child process; ----(1)

   fork子进程;

   { /\* 在子进程中 \*/

      清理一些不必要的从父进程继承下来的描述符等，为

      exec提供一个“干净的”环境；------(2)

      关闭attr->parent\_in、parent\_out和parent\_err，

      并分别重定向attr->child\_in、child\_out和child\_err为

      STDIN\_FILENO、STDOUT\_FILENO和STDERR\_FILENO;-----(3)

      判断attr->cmdtype，选择执行exec函数; ------(4)

   }

   /\* 在父进程中 \*/

   关闭attr->child\_in、child\_out和child\_err;

}

下面针对上述伪码进行具体分析：

(1) 有效性检查

attr->errchk属性可以通过apr\_procattr\_error\_check\_set函数在apr\_proc\_create之前设置。一旦设置，apr\_proc\_create就会在fork子进程前对procattr的有效性进行检查，比如attr->curdir的访问属性(利用access检查)、progname文件的访问权限检查等。这些的目的就是一个：“让错误发生在fork前，不要等到在子进程中出错”。

(2) 清理“不必要的”继承物

由于子进程复制了父进程的地址空间，随之而来的还包含一些“不必要”的“垃圾”。为了给exec提供一个“干净的”环境，在exec之前首先要做一下必要的清理，APR使用apr\_pool\_cleanup\_for\_exec来完成这项任务。apr\_pool\_cleanup\_for\_exec究竟做了些什么呢？这涉及到了apr\_pool的设计，这里仅仅作简单说明。apr\_pool\_cleanup\_for\_exec通过pool内部的global\_pool搜索其子结点，并逐一递归cleanup，这里的cleanup并不释放任何内存，也不flushI/O Buffer，仅是调用结点注册的相关cleanup函数，这里我们可以回顾一下apr\_procattr\_io\_set调用，在创建相关pipe时就为相应的in/out/err描述符注册了cleanup函数。同样就是因为这点，子进程在调用apr\_pool\_cleanup\_for\_exec之前，首先要kill掉（这里理解就是去掉相关文件描述符上的cleanup注册函数）这些注册函数。防止相关pipe的描述符被意外关闭。

(3) 建立起与父进程“对话通道”

父进程在创建procattr时就建立了若干个pipe，fork后子进程继承了这些。为了关掉一些不必要的描述符和更好的和父进程通讯，子进程作了一些重定向的工作，这里用2副图来表示重定向前后的差别：（图中显示的是子进程关闭parent\_in/out/err三个描述符后的文件描述表）

重定向前：

子进程文件描述表

-----------------------|

[0] STDIN\_FILENO |

-----------------------|

[1] STDOUT\_FILENO|

-----------------------|

[2] STDERR\_FILENO|

-----------------------|

[3] child\_in.fd | ----> in\_pipe的filedes[0]

-----------------|

[4] child\_out.fd| ----> out\_pipe的filedes[1]

-----------------|

[5] child\_err.fd| ----> err\_pipe的filedes[1]

-----------------|

重定向后：

------------------|

[0] child\_in.fd  | ----> in\_pipe的filedes[0]

------------------|

[1] child\_out.fd | ----> out\_pipe的filedes[1]

------------------|

[2] child\_err.fd | ----> err\_pipe的filedes[1]

------------------|

为了能更好的体现出“对话通道”的概念，这里再画出父进程再关闭ttr->child\_in、child\_out和child\_err后的文件描述表：

父进程文件描述表

-----------------------|

[0] STDIN\_FILENO  |

-----------------------|

[1] STDOUT\_FILENO |

------------------------|

[2] STDERR\_FILENO |

-------------------|

[3] parent\_in.fd  | ----> in\_pipe的filedes[1]

-------------------|

[4] parent\_out.fd | ----> out\_pipe的filedes[0]

-------------------|

[5] parent\_err.fd | ----> err\_pipe的filedes[0]

-------------------|

(4) 启动新的程序

根据APR proc的设计，子进程在被fork出来后，将根据procattr的cmdtype等属性信息决定调用哪种exec函数。当子进程调用一种exec函数时，子进程将完全由新程序代换，而新程序则从其main函数开始执行(与fork不同，fork返回后子进程从fork点开始往下执行)。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用另一个新程序替换了当前进程的正文、数据、堆和栈段。这里不详述这几种函数的差别，在参考资料中有相关描述[注5]。

**四、总结**

简单分析了一下APR的进程封装，APR的源代码注释很详尽，很多细节可以直接察看源码。

[注1]

#define HANDLE\_RTVAL(func, rv) do { /

 if (rv != APR\_SUCCESS) { /

  printf("%s executes error!/n", #func); /

  return rv; /

 } /

} while(0)

[注2]

typedef enum {

   APR\_SHELLCMD,           /\*use the shell to invoke the program \*/

   APR\_PROGRAM,           /\* invoke the program directly, no copied env \*/

   APR\_PROGRAM\_ENV,        /\* invoke theprogram, replicating our environment \*/

   APR\_PROGRAM\_PATH,       /\* find program on PATH,use our environment \*/

   APR\_SHELLCMD\_ENV        /\* use the shell toinvoke the program, replicating our environment \*/

} apr\_cmdtype\_e;

[注3]

xx\_in/xx\_out都是相对于child process来说的，xx\_in表示通过该描述符child process从in\_pipe读出parent process写入in\_pipe的数据；xx\_out表示通过该描述符child process将数据写入out\_pipe供parent process使用；xx\_err则是child process将错误信息写入err\_pipe供parent process使用。

[注4]

fork后子进程和父进程的同和异

同：

子进程从父进程那继承了

-- 父进程已打开的文件描述符；

-- 实际用户ID、实际组ID、有效用户ID、有效组ID；

-- 添加组ID；

-- 进程组ID；

-- 对话期ID；

-- 控制终端；

-- 设置用户ID标志和设置组ID标志；

-- 当前工作目录；

-- 根目录；

-- 文件方式创建屏蔽字；

-- 信号屏蔽和排列；

-- 对任一打开文件描述符的在执行时关闭标志；

-- 环境；

-- 连接的共享存储段；

-- 资源限制。

异：

-- fork的返回值；

-- 进程ID；

-- 不同的父进程ID；

-- 子进程的tms\_utime, tms\_stime, tms\_cutime以及tme\_ustime设置为0；

-- 父进程设置的锁，子进程不继承；

-- 子进程的未决告警被清除；

-- 子进程的未决信号集设置为空集。

[注5]

这里引用《Unix环境高级编程》中关于如何区分和记忆exec函数族的方法：“这六个exec函数的参数很难记忆。函数名中的字符会给我们一些帮助。字母p表示该函数取filename作为参数，并且用PATH环境变量寻找可执行文件。字母l表示该函数取一个参数列表，它与字母v互斥。v表示该函数取一个argv[]。最后，字母e表示该函数取envp[] 数组，而不使用当前环境。”

**参考资料：**

1、《Unix环境高级编程》

2、《Unix系统编程》

**APR分析-内存篇**

* 内存管理一直是让C程序员头痛的问题，作为一个通用接口集，APR当然也提供其自己的内存管理接口--APR Pool。APR Pool作为整个APR的一个基础功能接口，直接影响着APR的设计风格。在这篇Blog中，我们就要和APR Pool来一次“亲密接触”。(还是以Unix平台实现为例)

APR Pool源代码的位置在$(APR\_HOME)/memory目录下，本篇blog着重分析unix子目录下的apr\_pools.c文件内容，其相应头文件为$(APR\_HOME)/include/apr\_pools.h；在apr\_pools.c中还实现了负责APR内部内存分配的APRallocator的相关操作接口(APR allocator相关头文件为$(APR\_HOME)/include/apr\_allocator.h)。

**一、APR Pool概述**

我们平时常用的内存管理方式都是基于“request-style”的，即分配所请求大小的内存，使用之，销毁之。而APR Pool的设计初衷是为Complex Application提供良好的内存管理接口，其使用方式与“request-style”有所不同。在$(APR\_HOME)/docs/pool-design.htm文档中，设计者道出了“使用好”APR Pool的几个Rules，同时也从侧面反映出APRPool的设计。

1、任何Object都不应该有自己的Pool，它应该在其构造函数的调用者的Pool中分配。因为一般调用者知道该Object的生命周期，并通过Pool管理之。也就是说Object无须自己调用"Close" or "Free"，这些操作在Object所在Pool被摧毁时会被隐式调用的。

2、函数无须为了他们的行为而去Create/Destroy Pool，它们应该使用它们调用者传给它们的Pool。

3、为了防止内存无限制的增长，APR Pool建议当遇到unbounded iteration时使用sub\_pool，标准格式如下：

subpool = apr\_poll\_create(pool, NULL);

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

  apr\_pool\_clear(subpool);

  ... ...

  do\_operation(..., subpool);

}

apr\_pool\_destroy(subpool);

**二、深入APR Pool**

到目前为止我们已经知道了该如何“很好的”使用APR Pool，接下来我们来深入APRPool的内部，看究竟有什么“奥秘”。

**1、分析apr\_pool\_initialize**

任何使用APR的应用程序一般都会调用apr\_app\_initalize来初始化APR的内部使用的数据结构，察看一下app\_app\_initialize的代码，你会发现apr\_pool\_initialize在被apr\_app\_initialize调用的apr\_initialize中被调用，该函数用来初始化使用Pool所需的内部结构(用户无须直接调用apr\_pool\_initialize，在apr\_app\_initialize时它被自动调用，而apr\_app\_initialize又是APR program调用的第一个function，其在apr\_general.h中声明，在misc/unix/start.c中实现)。

apr\_pool\_initialize的伪码如下（这里先不考虑多线程的情况）：

static apr\_byte\_t apr\_pools\_initialized = 0;

static apr\_pool\_t \*global\_pool = NULL;

static apr\_allocator\_t \*global\_allocator = NULL;

apr\_pool\_initialize

{

 如果(!apr\_pools\_initialized)

 {

  创建global\_allocator; ------(1)

 }

 创建global\_pool; -------(2)

 给global\_pool起名为"apr\_global\_pool";

}

**(1) Pool和Allocator**

每个Pool都有一个allocator相伴，这个allocator可能是Pool自己的，也可能是其ParentPool的。allocator的结构如下：

/\* in apr\_pools.c \*/

struct apr\_allocator\_t {

   apr\_uint32\_t        max\_index;

   apr\_uint32\_t        max\_free\_index;

   apr\_uint32\_t        current\_free\_index;

    ... ...[注1]

   apr\_pool\_t         \*owner;

    apr\_memnode\_t     \*free[MAX\_INDEX];

};

在(1)调用后，global\_allocator的所有xx\_index字段都为0，owner-->NULL，free指针数组中的指针也都-->NULL。这里的index是大小的级别，这里最大级别为20(即MAX\_INDEX = 20)，free指针数组中free[0]所指的node大小为MIN\_ALLOC大小，即8192，即2的13次幂。按此类推free[19]所指的node大小应为2的32次幂，即4G byte。allocator\_alloc中是通过index =(size >> BOUNDARY\_INDEX) - 1来得到这一index的。allocator维护了一个index不同的memnode池，每一index级别上又有一个memnode list，以后用户调用apr\_palloc分配size大小内存时，allocaotr\_alloc函数就会在free memnode池中选和要寻找的size的index级别相同的memnode，而不是重新malloc一个size大小的memnode。另外要说明一点的是APR Pool中所有ADT中的xx\_index字段都是大小级别的概念。

**(2) 创建global\_pool**

在APR Pool初始化的时候，唯一创建一个Pool-- global\_pool。apr\_pool\_t的非Debug版本如下：

/\* in apr\_pools.c \*/

struct apr\_pool\_t {

   apr\_pool\_t           \*parent;

   apr\_pool\_t           \*child;

   apr\_pool\_t          \*sibling;

   apr\_pool\_t           \*\*ref;

   cleanup\_t           \*cleanups;

   cleanup\_t           \*free\_cleanups;

    apr\_allocator\_t     \*allocator;

    struct process\_chain \*subprocesses;

   apr\_abortfunc\_t       abort\_fn;

   apr\_hash\_t          \*user\_data;

    constchar           \*tag;

   apr\_memnode\_t        \*active;

   apr\_memnode\_t        \*self; /\* The nodecontaining the pool itself \*/

   char                \*self\_first\_avail;

    ... ...

}

而apr\_memnode\_t的结构如下：

/\* in apr\_allocator.h \*/

struct apr\_memnode\_t {

    apr\_memnode\_t\*next;           /\*\*< next memnode \*/

    apr\_memnode\_t\*\*ref;           /\*\*< reference to self \*/

    apr\_uint32\_t  index;           /\*\*< size\*/

    apr\_uint32\_t  free\_index;      /\*\*< how much free \*/

   char         \*first\_avail;     /\*\*< pointer to first free memory \*/

   char         \*endp;           /\*\*< pointer to end of free memory \*/

};

apr\_pool\_create\_ex首先通过allocator寻找合适的node用于创建Pool，但由于global\_allocator尚未分配过任何node，所以global\_allocator创建一个新的node，该node大小为MIN\_ALLOC(即8192)，该node的当前状态如下：

node -->|---------------|0

             |                     |

             |                     |

             |                     |

             |---------------|APR\_MEMNODE\_T\_SIZE <-------- node->first\_avail

             |                     |

             |                     |

             |                     |

             ----------------- size(一般为8192) <-------- node->endp

其他属性值如下：

node->next = NULL;

node->index = (APR\_UINT32\_TRUNC\_CAST)index; /\* 这里为1 \*/

创建完node后，我们将在该node上的avail space划分出我们的global\_pool来。划分后状态如下(pool与node关系)：

node -->|---------------|0 <---pool->self = pool\_active

            |                      |

            |                      |

            |---------------|APR\_MEMNODE\_T\_SIZE <-------- global\_pool

           |                       |

            |                      |

           |---------------|APR\_MEMNODE\_T\_SIZE+SIZEOF\_POOL\_T<--------node->first\_avail = pool->self\_first\_avail

          |                        |

          |                       |

          ----------------- size(一般为8192) <-------- node->endp

pool其他一些属性值(pool与pool之间关系)如下：

pool->allocator = global\_allocator;

pool->child = NULL;

pool->sibling = NULL;

pool->ref = NULL;

也许现在你仍然不能看清楚APRPool的结构，无需着急，我们继续往下分析。

**2、APR Sub\_Pool创建(pool与pool之间关系)**

上面我们已经初始化了global\_pool，但是global\_pool是不能直接拿来就用的，我们需要创建其sub\_pool，也就是用户自己的pool。一般创建user的sub\_pool我们都使用apr\_pool\_create宏，它只需要2个参数，并默认sub\_pool继承parent\_pool的allocator和abort\_fn。在apr\_pool\_create内部调用的还是apr\_pool\_create\_ex函数。我们来看一下创建sub\_pool后pool之间的关系：

例:

static apr\_pool\_t \*sub\_pool = NULL;

apr\_pool\_create(&sub\_pool, NULL);

这里sub\_pool的创建过程与global\_pool相似，也是先创建其承载体node，然后设置相关属性，使其成为global\_pool的child\_pool。创建完后global\_pool和该sub\_pool的关系如下图：

global\_pool <-----/    ----->  sub\_pool

-----------             / /          ------------

sibling --->NULL    /-------   parent

-----------            /            ------------

child ------------ /                sibling ----->NULL

-----------                           ------------

                                            child  ------>NULL

                                           ------------

APR Pool是按照二叉树结构组织的，并采用“child-sibling”的链式存储方式，global\_pool作为整个树的Root Node。如果APR Pool中存在多个Pool，其节点结构关系如下：

              /-child-->

             / --------Pool\_level1-a

            / / parent   /|/    |

           /|/\_             |     | sibling

global\_pool             |    |

           /                 |    /|/

            /-child-> Pool\_level1-b

           /|/                 |

            -parent------

**3、从pool中分配内存**

上面我们已经拥有了一个sub\_pool，我们现在就可以从sub\_pool中分配内存了。APR提供了函数apr\_palloc来做这件事情。

例如：apr\_alloc(sub\_pool,wanted\_mem\_size);

apr\_palloc在真正分配内存前会把wanted\_mem\_size做一下处理。它使用APR\_ALIGN\_DEFAULT宏处理wanted\_mem\_size得到一个圆整到8的new\_size，然后再在pool中分配new\_size大小的内存，也就是说pool中存在的用户内存块的大小都是8的倍数。举个例子来说，如果wanted\_mem\_size= 30，apr\_alloc实际会在pool中划分出32个字节的空间。

apr\_palloc的工作流程简单描述是这样的：

a) 如果在pool->active node的avail space足够满足要申请的内存大小size时，则直接返回active->first\_avail，并调整active->first\_avail= active->first\_avail + size；

b) 如果a)不满足，则察看active->next这个node满足与否；如果满足则将返回所要内存，并将该node设为active node，将以前的active node放在新active node的next位置上;

c) 如果b)也不满足，则新创建一个memnode，这个node可能为新创建的，也可能是从allocator的free memnode池中取出的，取决于当时整个Pool的状态。

从上面我们也可以看出node分为2类，一种是作为pool的承载体，但pool结构的空间不足以完全占满一个node，所以也可以用来分配用户内存；另一种就是完全用于分配用户内存的了。每个pool有一个node list，当然这个list中包括它自己所在的node了。

**4、apr\_pool\_clear和apr\_pool\_destroy**

创建和分配结束后，我们需要clear或者destroy掉Pool。

clear和destroy的区别在于clear并不真正free内存，只是清理便于以后alloc时重用，而destroy则是真正的free掉内存了。

**三、总结**

本文并未说明APR Pool有哪些优点或缺点(除了概述中的一些Rules)，仅是把其来龙去脉弄清。

[注1]

在本文中出现的"......"的符号表示与多线程相关的字段和代码的省略。

**APR分析-信号篇**

* U know 信号是Unix的重要系统机制。信号机制使用起来很简单，但是理解起来有并不是那么Easy。APR Signal的封装也并不繁琐，代码量很少，所以分析APR Signal的过程其实就是学习Signal机制的过程。

**一、信号介绍**

1、Signal“历史久远”，在最初的Unix系统上就能看到它“伟岸”的身影。它的引入用来进行User Mode进程间的交互，系统内核也可以利用它通知User Mode进程发生了哪些系统事件。从最开始引入到现在，信号只是做了很小的一些改动（不可靠信号模型到可靠信号模型）。

2、信号服务于两个目的：

 1) 通知某进程某特定事件发生了；

 2) 强制其通知进程执行相应的信号处理程序。

**二、基础概念**

1、信号的一个特性就是可以在任何时候发给某一进程，而无需知道该进程的状态。如果该进程当前并未处于执行态，则该信号被内核Save起来，直到该进程恢复执行才传递给它；如果一个信号被进程设置为阻塞，则该信号的传递被延迟，直到其阻塞被取消它才被传递给进程。

2、系统内核严格区分信号传送的两个阶段：

 1) Signal Generation : 系统内核更新目标进程描述结构来表示一个信号已经被发送出去。

 2) Signal Delivery : 内核强制目标进程对信号做出反应，或执行相关信号处理函数，或改变进程执行状态。

信号的诞生和传输我们可以这样理解：把信号作为“消费品”，其Generation状态就是“消费品诞生”，其Delivery状态就是理解为“被消费了”。这样势必存在这样的一个情况：“消费品诞生了，但是还没有被消费掉”，在信号模型中，这样的状态被称为“pending”(悬而未决)。

任何时候一个进程只能有一个这样的某类型的pending信号，同一进程的其他同类型的pending信号将不排队，将被简单的discard(丢弃)掉。

3、如何消费一个signal

 1) 忽略该信号；[注1]

 2) 响应该信号，执行一特定的信号处理函数；

 3) 响应该信号，执行系统默认的处理函数。包括：Terminate、Dump、Ignore、Stop、Continue等。

这里有特殊：SIGKILL和SIGSTOP两个信号不能忽略、不能捕捉、不能阻塞，而只是执行系统默认处理函数。

**三、APR Signal封装**

APR Signal源代码的位置在$(APR\_HOME)//threadproc目录下，本篇blog着重分析unix子目录下的signals.c文件内容，其相应头文件为$(APR\_HOME)/include/apr\_signal.h。

**1、apr\_signal函数**

Unix信号机制提供的最简单最常见的接口是signal函数，用来设置某特定信号的处理函数。但是由于早期版本和后期版本处理信号方式的不同，导致现在直接使用signal函数在不同的平台上可能得到不同的结果。

早期版本处理方式：进程每次处理信号后，随即将信号的处理动作重置为默认值。

后期版本处理方式：进程每次处理信号后，信号的处理动作不被重置为默认值。

我们举例测试一下：分别在Solaris9 、Cygwin和RedHat Linux 9上。

例子：

E.G 1:

void siguser1\_handler(int sig);

int main(void)

{

        if (signal(SIGUSR1,siguser1\_handler) == SIG\_ERR) {

               perror("siguser1\_handler error");

               exit(1);

        }

        while (1) {

               pause();

        }

}

void siguser1\_handler(int sig)

{

        printf("insiguser1\_handler, %d/n", sig);

}

input:

kill -USR1 9122

kill -USR1 9122

output:(Solaris 9)

in siguser1\_handler, 16

用户信号1 (程序终止)

output:(Cygwin and RH9)

in siguser1\_handler, 30

in siguser1\_handler, 30

...

..

E.G 1结果表示在Solaris 9上，信号的处理仍然按照早期版本的方式，而Cygwin和RH9则都按照后期版本的方式。

那么有什么替代signal函数的办法么？在最新的X/Open和UNIXspecifications中都推荐使用一个新的信号接口sigaction，该接口采用后期版本的信号处理方式。在《Unix高级环境编程》中就有使用sigaction实现signal的方法，而APR恰恰也是使用了该方法实现了apr\_signal。其代码如下：

APR\_DECLARE(apr\_sigfunc\_t \*) apr\_signal(int signo, apr\_sigfunc\_t \*func)

{

    struct sigaction act, oact;

    act.sa\_handler = func;

   sigemptyset(&act.sa\_mask); ------------------(1)

    act.sa\_flags = 0;

#ifdefSA\_INTERRUPT            /\* SunOS \*/

    act.sa\_flags |= SA\_INTERRUPT;

#endif

    ... ...

    if (sigaction(signo, &act, &oact) <0)

        return SIG\_ERR;

    return oact.sa\_handler;

}

(1) 这里有一个Signal Set(信号集)的概念，通过相关函数操作信号集以改变内核传递信号给进程时的行为。Unix用sigset\_t结构来表示信号集。信号集总是和sigprocmask或sigaction一起使用。关于信号集和sigprocmask函数将在下面详述。

**2、apr\_signal\_block和apr\_signal\_unblock**

这两个函数分别负责阻塞和取消阻塞内核传递某信号给目标进程。其主要利用的就是sigprocmask函数来实现的。每个进程都有其对应的信号屏蔽字，它让目标进程能够通知内核“哪些传给我的信号该阻塞，哪些畅通无阻”。在《Unix高级环境编程》中作者有这么一段说明“如果在调用sigprocmask后有任何未决的、不再阻塞的信号，则在sigprocmask返回前，至少将其中之一递送给该进程。”能理解这句我想信号屏蔽字这块儿也就没什么问题了。在Unix高级环境编程》中作者举了一个很不错的例子，讲解的也很详细。这里想举例说明的是：如果多次调用SET\_BLOCK的sigprocmask设置屏蔽字，结果是什么呢？

E.G 3

int main(void)

{

        sigset\_t newmask,oldmask, pendmask;

        /\* 设置进程信号屏蔽字, 阻塞SIGQUIT \*/

       sigemptyset(&newmask);

        sigaddset(&newmask,SIGQUIT);

        if(sigprocmask(SIG\_BLOCK, &newmask, &oldmask) < 0) {

               perror("SIG\_BLOCK error");

        }

        printf("1st towait 30 seconds/n");

        sleep(30);

        /\* 第一次察看当前的处于pend状态的信号 \*/

        if(sigpending(&pendmask) < 0) {

               perror("sigpending error");

        }

        if(sigismember(&pendmask, SIGQUIT)) {

               printf("SIGQUIT pending/n");

        } else {

               printf("SIGQUIT unpending/n");

        }

        if(sigismember(&pendmask, SIGUSR1)) {

        if(sigismember(&pendmask, SIGUSR1)) {

               printf("SIGUSR1 pending/n");

        } else {

               printf("SIGUSR1 unpending/n");

        }

        /\* 重新设置屏蔽字, 阻塞SIGUSR1 \*/

       sigemptyset(&newmask);

        sigaddset(&newmask,SIGUSR1);

        if(sigprocmask(SIG\_BLOCK, &newmask, &oldmask) < 0) {

               perror("SIG\_BLOCK error");

        }

        printf("2nd towait 30 seconds/n");

        sleep(30);

        /\* 再次察看当前的处于pend状态的信号 \*/

        if(sigpending(&pendmask) < 0) {

               perror("sigpending error");

        }

        if(sigismember(&pendmask, SIGQUIT)) {

               printf("SIGQUIT pending/n");

        } else {

               printf("SIGQUIT unpending/n");

        }

        if(sigismember(&pendmask, SIGUSR1)) {

               printf("SIGUSR1 pending/n");

        } else {

               printf("SIGUSR1 unpending/n");

        }

        exit(0);

}

//output:

1st to wait 30 seconds

^/

SIGQUIT pending

SIGUSR1 unpending

2nd to wait 30 seconds -- 这之后发送kill -USR128821

SIGQUIT pending

SIGUSR1 pending

第一次输出SIGUSR1unpending是因为并未发送USR1信号，所以自然为unpending状态；我想说的是第二次重新sigprocmask时我们仅加入了SIGUSR1,并未显示加入SIGQUIT，之后察看pending信号中SIGQUIT仍然为pending状态，这说明两次SET\_BLOCK的sigprocmask调用是"或"的关系，第二次SET\_BLOCK的sigprocmask调用不会将第一次SET\_BLOCK的sigprocmask调用设置的阻塞信号变为非阻塞的。

**四、总结**

信号简单而强大，如果想深入了解signal的实现，参考资料中的第二本书会给你满意的答案。

**五、参考资料：**

1、《Unix高级环境编程》

2、《深入理解Linux内核》

[注1]

忽略信号和阻塞信号

前者相当于一个消费行为，该信号的状态为“已消费”，而后者只是将信号做缓存，等待阻塞打开，再交给进程消费，其状态为“未消费”，也相当于处于pending状态。

**APR分析-文件IO篇**

* 文件I/O在Unix下占据着非常重要的地位，曾有一句经典语句绝对可以说明file在Unix下的重要性，That is "In UNIX, everything is a file"，APR就是本着这个思想对Unix文件I/O进行了再一次的抽象封装，以提供更为强大和友善的文件I/O接口。

APR File I/O源代码的位置在$(APR\_HOME)/file\_io目录下，本篇blog着重分析unix子目录下的相关.c文件内容，其相应头文件为$(APR\_HOME)/include/apr\_file\_io.h和apr\_file\_info.h。

**一、APR File I/O介绍**

APR用了"不小的篇幅"来"描述"文件I/O，在$(APR\_HOME)/file\_io/unix目录下，你会看到多个.c文件，每个.c都是一类文件I/O操作。比如：

 open.c -- 封装了文件的打开、关闭、改名和删除等操作；

 readwrite.c -- 顾名思义，它里面包含了文件的读写操作；

 pipe.c -- 包含了pipe相关操作。

还有许多这里不多说，由于文件I/O操作复杂，我们下面将仅挑出最常用的文件I/O操作进行分析。

**二、基本APR I/O**

APR定义了apr\_file\_t类型来表示广义的文件。先来看一下这个核心数据结构的“模样”：

/\* in apr\_arch\_file\_io.h \*/

struct apr\_file\_t {

    apr\_pool\_t \*pool;

    int filedes;

    char \*fname;

    apr\_int32\_t flags;

    int eof\_hit;

    int is\_pipe;

    apr\_interval\_time\_t timeout;

    int buffered;

    enum {BLK\_UNKNOWN, BLK\_OFF, BLK\_ON } blocking;

    int ungetchar;    /\* Last charprovided by an unget op. (-1 = no char)\*/

#ifndef WAITIO\_USES\_POLL

    /\* if there is a timeout set, then this pollsetis used \*/

    apr\_pollset\_t \*pollset;

#endif

    /\* Stuff for buffered mode \*/

    char \*buffer;

    intbufpos;              /\* Read/Write position in buffer \*/

    unsigned long dataRead;   /\* amountof valid data read into buffer \*/

    intdirection;            /\*buffer being used for 0 = read, 1 = write \*/

    unsigned long filePtr;    /\*position in file of handle \*/

#if APR\_HAS\_THREADS

    struct apr\_thread\_mutex\_t \*thlock;

#endif

};

在这个数据结构中有些字段的含义一目了然，如filedes、fname、is\_pipe等，而有些呢即使看了注释也不能够马上了解其真正的含义，这就需要在阅读源码时来体会。

**1、apr\_file\_open**

ANSI C标准库和Unix系统库函数都提供对“打开文件”这个操作语义的支持。他们提供的接口很相似，参数一般都为“文件名+打开标志位+权限标志位”，apr\_file\_open也不能忽略习惯的巨大力量，也提供了类似的接口如下：

APR\_DECLARE(apr\_status\_t) apr\_file\_open(apr\_file\_t \*\*new,

                                       const char \*fname,

                                       apr\_int32\_t flag,

                                       apr\_fileperms\_t perm,

                                       apr\_pool\_t \*pool);

其中fname、flag和perm三个参数你应该很眼熟吧:)。每个封装都有自定义的一些标志宏，这里也不例外，flag和perm参数都需要用户传入APR自定义的一些宏组合，不过由于这些宏的可读性都很好，不会成为你使用过程的绊脚石。由于apr\_file\_open操作是其他操作的基础所以这里作简单分析，还是采用老办法伪码法：

apr\_file\_open

{

 “打开标志位”转换；-----(1)

 “权限标志位”转换；-----(2)

 调用Unix原生API打开文件；

 设置apr\_file\_t变量相关属性值；------(3)

}

(1) 由于上面说了，APR定义了自己的“文件打开标志位”，所以在apr\_file\_open的开始需要将这些专有的“文件打开标志位”转换为Unix平台通用的“文件打开标志位”；

(2) 同(1)理，专有的“权限标志位”需要转换为Unix平台通用的“权限标志位”；

(3) APR file I/O封装支持非阻塞I/O带超时等待以及缓冲I/O，默认情况下为阻塞的，是否缓冲可通过“文件打开标志位”设置。一旦设置为缓冲I/O，则apr\_file\_open会在pool中开辟大小为APR\_FILE\_BUFSIZE(4096)的缓冲区供使用。

**2、apr\_file\_read/apr\_file\_write**

该两个接口的看点是其缓冲区管理（前提：在apr\_file\_open该文件时指定了是Buffer I/O及非阻塞I/O带超时等待）。还有一点就是通过这两个接口的实现我们可以了解到上面提到的apr\_file\_t中某些“晦涩”字段的真正含义。

(1) 带缓冲I/O

这里的缓冲是APR自己来管理的，带缓冲的好处很简单，即减少直接操作文件的次数，提高I/O性能。要知道无论lseek还是read/write都是很耗时的，尽可能的减少直接I/O操作次数，会带来性能上明显的改善。这里将用图示说明缓冲区与文件的对应关系，以帮助理解APR缓冲I/O：

             thefile->filePtr

                  |

0               /|/                     文件末尾

-----------------------------------------------

///////////////////                   <---- thefile->filedes (文件)

-----------------------------------------------

    /             /

   /              /

  /                /

0|/\_             \_/|          APR\_FILE\_BUFSIZE

-----------------------------------------------

////////////////////////                              (缓冲区)

//////////

-----------------------------------------------

/|/     /|/           /|/

 |       |             |

 |       |            thefile->dataRead

 |  thefile->bufpos

thefile->buffer

说明："//////"-- 表示从文件读到缓冲区的数据；

      "//////" --表示从用户已从缓冲区读出的数据。

thefile->bufpos : 缓冲区中的读写位置

thefile->dataRead: 标识缓冲区从文件读取的数据的大小

thefile->fileptr: 标识文件本身被读到什么位置

读写切换：如果先读后写，则每次写的时候都要重新定位文件指针到上次读的结尾处；如果先写后读，则每次读前都要flush缓冲区。

(2)非阻塞I/O带超时等待

这里分析下面一段apr\_file\_read的代码：

 do {

           rv = read(thefile->filedes, buf, \*nbytes);

        } while (rv == -1&& errno == EINTR); --------------(a)

#ifdef USE\_WAIT\_FOR\_IO

        if (rv == -1 &&

           (errno == EAGAIN || errno == EWOULDBLOCK) &&

           thefile->timeout != 0) {

           apr\_status\_t arv = apr\_wait\_for\_io\_or\_timeout(thefile, NULL, 1); ------(b)

           if (arv != APR\_SUCCESS) {

               \*nbytes = bytes\_read;

               return arv;

           }

           else {

               do {

                   rv = read(thefile->filedes, buf, \*nbytes);

               } while (rv == -1 && errno == EINTR);

           }

        }

#endif

 (a) 第一个do-while块：之所以使用一个do-while块是为了当read操作被信号中断后重启read操作；

 (b) 一旦文件描述符设为非阻塞，(a)则瞬间返回，一旦(a)并未读出数据，则rv = -1并且errno被设置为errno = EAGAIN，这时开始带超时的等待该文件描述符I/O就绪。这里的apr\_wait\_for\_io\_or\_timeout使用了I/O的多路复用技术Poll，在后面的APR分析中会详细理解之。apr\_file\_t中的timeout字段就是用来做超时等待的。

**3、apr\_file\_close**

该接口主要完成的工作为刷新缓冲区、关闭文件描述符、删除文件(如果设置了APR\_DELONCLOSE标志位)和清理Pool中内存的工作，这里不详述了。

**三、总结**

复杂的文件I/O，让我们通过三言两语就说完了。大家慢慢体会，看看世界著名开源项目的源代码，收获是颇丰的，不妨尝试一下。

**APR分析-高级IO篇**

* 近两天稍轻闲了些，便抓紧时间学习、学习再学习。在“APR分析-文件IO篇”，我们只分析了最基本的I/O操作，如文件的open、close、write和read。当然File I/O操作不止这些，在这一篇中我们来看看APR提供的一些高级I/O设施，包括记录锁、I/O多路复用和内存映射文件(内存映射文件将和共享内存一起分析)。

**一、记录锁或(区域锁)[注1]**

我见过的对记录锁讲解最详细的书就是《Unix高级环境编程》，特别是关于进程、文件描述符和记录锁三者之间关系的讲解更是让人受益匪浅，有此书的朋友一定不要放过哟。这里将其中的三原则摘录到这：

关于记录锁的自动继承和释放有三条规则：

(1) 锁与进程、文件两方面有关。这有两重含意：第一重很明显，当一个进程终止时，它所建立的锁全部释放；第二重意思就不很明显，任何时候关闭一个描述符时，则该进程通过这一描述符可以存访的文件上的任何一把锁都被释放（这些锁都是该进程设置的）。

(2) 由fork产生的子程序不继承父进程所设置的锁。这意味着，若一个进程得到一把锁，然后调用fork，那么对于父进程获得的锁而言，子进程被视为另一个进程，对于从父进程处继承过来的任一描述符，子进程要调用fcntl以获得它自己的锁。这与锁的作用是相一致的。锁的作用是阻止多个进程同时写同一个文件（或同一文件区域）。如果子进程继承父进程的锁，则父、子进程就可以同时写同一个文件。

(3) 在执行exec后，新程序可以继承原执行程序的锁。

话归正题谈APR的记录锁，平心而论APR的提供的加索和解锁接口并没有什么独到的地方，APR之所以将之封装起来，无非是为了提供一个统一的跨平台接口，并且不破坏APR整体代码风格的一致性。APR记录锁源码位置在$(APR\_HOME)/file\_io/unix目录下flock.c，头文件仍然是apr\_file\_io.h。apr\_file\_lock和apr\_file\_unlock仅提供对整个文件的加锁和解锁，而并不支持对文件中任意范围数据的加锁和解锁。至于该锁是建议锁(advisory lock)还是强制锁(mandatory lock)，需要看具体的平台的实现了。两个函数均利用fcntl实现记录锁功能(前提是所在平台支持fcntl，由于fcntl是POSIX标准，绝大多数平台都支持)。代码中有一处值得鉴赏：

while ((rc = fcntl(thefile->filedes, fc, &l)) < 0&& errno == EINTR)

           continue;

这里这么做的原因就是考虑到fcntl的调用可能被某信号中断，一旦中断我们去要重启fcntl函数。

**二、I/O多路复用[注2]**

在经典的《Unix网络编程第1卷》Chapter 6中作者详细介绍了五种I/O模型，分别为：

 - blocking I/O

 - nonblocking I/O

 - I/O multiplexing (select and poll)

 - signal driven I/O (SIGIO)

 - asynchronous I/O (the POSIX aio\_functions)

作者同时对这5种I/O模型作了很详细的对比分析，很值得一看。这里所说的I/O多路复用就是第三种模型，它既解决了Blocking I/O数据处理不及时，又解决了Non-Blocking I/O采用轮旬的CPU浪费问题，同时它与异步I/O不同的是它得到了各大平台的广泛支持。

APR I/O多路复用源码主要在$(APR\_HOME)/poll/unix目录下的poll.c和select.c中，头文件为apr\_poll.h。APR提供统一的apr\_poll接口，但是apr\_pollset\_t结构定义和apr\_poll的实现则根据宏POLLSET\_USES\_SELECT、POLL\_USES\_POLL和POLLSET\_USES\_POLL的定义与否而不同。这里拿poll的实现(That is 使用poll来实现apr\_poll及apr\_pollset\_xx相关，与之对应的是使用select来实现apr\_poll及apr\_pollset\_xx相关)来分析：在poll的实现下，apr\_pollset\_t的定义如下：

/\* in poll.c \*/

struct apr\_pollset\_t

{

    apr\_pool\_t \*pool;

    apr\_uint32\_t nelts;

    apr\_uint32\_t nalloc;

    struct pollfd \*pollset;

    apr\_pollfd\_t \*query\_set;

    apr\_pollfd\_t \*result\_set;

};

统一的apr\_pollfd\_t定义如下：

/\* in apr\_poll.h \*/

struct apr\_pollfd\_t {

    apr\_pool\_t\*p;             /\* associated pool \*/

    apr\_datatype\_e desc\_type;   /\*descriptor type \*/

    apr\_int16\_treqevents;      /\* requested events \*/

    apr\_int16\_trtnevents;      /\* returned events \*/

    apr\_descriptordesc;        /\* @see apr\_descriptor \*/

    void\*client\_data;          /\* allowsapp to associate context \*/

};

把数据结构定义贴出来便于后面分析时参照理解。

假设我们像这样apr\_pollset\_create(&mypollset,10, p, 0)调用，那么在apr\_pollset\_create后，我们可以用图示来表示mypollset变量的状态：

mypollset

-------

nalloc  ----> 10 /\* 该mypollset的“容量”，在create的时候由参数指定 \*/

-------

nelts   ----> 0  /\* 刚初始化，mypollset中并没有任何element，之后每add一次，nelts就+1 \*/

-------

                       ---------------------------------------------

pollset --------->  pollset[0] | pollset[1] |...|pollset[nalloc-1]

                       ---------------------------------------------

-------

                       -----------------------------------------------------

query\_set --------->  query\_set[0] | query\_set[1] |...|query\_set[nalloc-1]

                      -----------------------------------------------------

-------

                      ---------------------------------------------------------

result\_set --------->  result\_set[0] | result\_set[1] |...|result\_set[nalloc-1]

                      ---------------------------------------------------------

-------

pollset、query\_set和result\_set这几个集合的关系通过下图说明：

apr\_pollfd\_t \*descriptor ---> [pollset\_add]--------> query\_set ------ [pollset\_poll] -----> result\_set (输出)

                                                    |                                                                                           /|/

                                                      ------------------->pollset ------ [pollset\_poll] --------------------

apr\_pollset\_xx系列是改版后APR I/O复用新增的接口集，它以apr\_pollset\_t作为其管理的基本单位，其中apr\_pollset\_poll用于监视pollset中的所有descriptor(s)。而apr\_poll则是旧版的APR I/O复用接口，它同样可以实现apr\_pollset\_poll的功能，只是它的基本管理单位是apr\_pollfd\_t，其相关函数还包括apr\_poll\_setup、apr\_poll\_socket\_add等在apr-1.1.1版中已看不到的几个接口。新版本中建议使用apr\_pollset\_poll，起码APR的测试用例(testpoll.c)是这么做的。

select实现的思路与poll实现的思路是一致的，只是apr\_pollset\_t的结构不同，原因不言自明。

**三、总结**

由于APR对高级I/O的封装很“薄”，所以基本上没有太多很精致的东西。

**四、参考资料**

1、《Unix高级环境编程》

2、《Unix网络编程卷1、2》

[注1]

对于Unix，“记录”这个定语也是误用，因为Unix内核根本没有使用文件记录这种概念。一个更适合的术语可能是“区域锁”，因为它锁定的只是文件的一个区域（也可能是整个文件）-- 摘自《Unix高级环境编程》。

[注2]

在《Unix网络编程卷1》译者译为"多路复用"，在《Unix高级环境编程》中译者译为"多路转接"，我更倾向于前者。I/O多路复用其英文为"I/OMultiplexing"。

**APR分析-共享内存篇**

* 共享内存是一种重要的IPC方式。在项目中多次用到共享内存，只是用而并未深入研究。这次趁研究APR代码的机会复习了共享内存的相关资料。

APR共享内存封装的源代码的位置在$(APR\_HOME)/shmem目录下，本篇blog着重分析unix子目录下的shm.c文件内容，其相应头文件为$(APR\_HOME)/include/apr\_shm.h。

**一、共享内存简单小结**

共享内存是最快的IPC方式，因为一旦这样的共享内存段映射到各个进程的地址空间，这些进程间通过共享内存的数据传递就不需要内核的帮忙了。Stevens的解释是“各进程不是通过执行任何进入内核的系统调用来传递数据，显然内核的责任仅仅是建立各进程地址空间与共享内存的映射，当然像处理页面故障这一类的底层活还是要做的”。相比之下，管道和消息队列交换数据时都需要内核来中转数据，速度就相对较慢。

Unix“历史悠久”，所以在历史上不同版本的Unix提供了不同的支持共享内存的方式，我想这也是Stevens在《Unix网络编程第2卷》中花费三章来讲解共享内存的原因吧。你也不妨先看看shm.c中的代码，代码用条件宏分割不同Share Memory的实现。

**二、APR共享内存封装**

APR提供多种创建共享内存的方式，其中最主要的就是apr\_shm\_create接口，其伪码如下：

apr\_shm\_create

{

 if (要创建匿名shm) {

#if APR\_USE\_SHMEM\_MMAP\_ZERO || APR\_USE\_SHMEM\_MMAP\_ANON

#if APR\_USE\_SHMEM\_MMAP\_ZERO

  xxxx ---------- (1)

#elif APR\_USE\_SHMEM\_MMAP\_ANON

  xxxx ---------- (2)

#endif

#endif /\* APR\_USE\_SHMEM\_MMAP\_ZERO || APR\_USE\_SHMEM\_MMAP\_ANON \*/

#if APR\_USE\_SHMEM\_SHMGET\_ANON

  xxxx ---------- (3)

#endif

 } else { /\* 创建有名shm \*/

#if APR\_USE\_SHMEM\_MMAP\_TMP || APR\_USE\_SHMEM\_MMAP\_SHM

#if APR\_USE\_SHMEM\_MMAP\_TMP

  xxxx ---------- (4)

#endif

#if APR\_USE\_SHMEM\_MMAP\_SHM

  xxxx ---------- (5)

#endif

#endif /\* APR\_USE\_SHMEM\_MMAP\_TMP || APR\_USE\_SHMEM\_MMAP\_SHM \*/

#if APR\_USE\_SHMEM\_SHMGET

  xxxx ---------- (6)

#endif

 }

}

apr\_shm\_create函数代码很长，之所以这样是因为其支持多种创建Share Memory的方式，在上面的伪代码中共用条件宏分隔了6种方式，这6种方式将在下面分析。可以看出shmem主要分为"匿名的"和"有名的"，其中"有名的"都是通过filename来标识(或通过ftok转换filename而得到的shmid来标识)。

其中不同版本Unix创建匿名shmem的做法如下：

(1) SVR4通过映射"/dev/zero"设备文件来获得匿名共享内存，其代码一般为：

fd = open("/dev/zero", ..);

ptr = mmap(..., MAP\_SHARED, fd, ...);

(2) 4.4 BSD提供更加简单的方式来支持匿名共享内存(注意标志参数MAP\_XX)

ptr = mmap(..., MAP\_SHARED | MAP\_ANON, -1, ...);

(3) System V匿名共享内存区的做法如下：

shmid = shmget(IPC\_PRIVATE, ...);

ptr = shmat(shmid, ...);

匿名共享内存一般都用于有亲缘关系的进程间的数据通讯。由父进程创建共享内存，子进程自动继承下来。由于是匿名，没有亲缘关系的进程是不能动态连接到该共享内存区的。

不同版本Unix创建有名shmem的做法如下：

(4) 由于是有名的shmem，所以与匿名不同的地方在于用filename替代"/dev/zero"做映射。

fd = open(filename, ...);

apr\_file\_trunc(...);

ptr = mmap(..., MAP\_SHARED, fd, ...);

(5) Posix共享内存的做法

fd = shm\_open(filename, ...);

apr\_file\_trunc(...);

ptr = mmap(..., MAP\_SHARED, fd, ...);

值得注意的一点就是通过shm\_open映射的共享内存可以供无亲缘关系的进程共享。apr\_file\_trunc用于重新设定共享内存对象长度。

(6) System V有名共享内存区的做法如下：

shmkey = ftok(filename, 1);

shmid = shmget(shmkey, ...); //相当于open orshm\_open

ptr = shmat(shmid, ...); //相当于mmap

有名共享内存一般都与一个文件相关，该文件映射到共享内存段，而不同的进程(包括无亲缘关系的进程)则都映射到该文件以达到目的。在APR中通过apr\_shm\_attach可以动态将调用进程连接到已存在的共享内存区上，前提是你必须知道该共享内存区的标识，在APR中一律用filename做标识。

**三、总结**

内核架起了多个进程间共享数据的纽带--共享内存。通过上面的叙述你会发现共享内存的创建其实并不困难，真正困难的是共享内存的管理[注1]，在正规的软件公司像内存/共享内存管理这样的重要底层功能都是封装成库形式的，当然内存管理的内容不是这篇blog重点涉及的内容。

**四、参考资料：**

1、《Unix网络编程第2卷》

2、《Unix环境高级编程》

[注1] SIGSEGV和SIGBUS

涉及共享内存的管理就不能不提到访问共享内存对象。谈到访问共享内存对象就要留神“SIGSEGV和SIGBUS”这两个信号。

系统分配内存页来承载内存映射区，由于内存页大小是固定的，所以存在多余的页空间空闲，比如待映射文件大小为5000 bytes，内存映射区大小也为5000 bytes。而一个内存页大小4096，系统势必要分配两页来承载，这时空闲的有效空间为从5000-8191，如果进程访问这段地址空间也不会发生错误。但是要超出8191，就会收到SIGSEGV信号，导致程序停止。关于SIGBUS信号的来历，这里也举例说明：若待映射文件大小为5000 bytes，我们在mmap时指定内存映射区size = 15000 > 5000，这时内核真正的共享区承载体大小只有8192（能包容映射文件大小即可），此时在[0，8191]内访问均没问题，但在[8192,14999]之间会得到SIGBUS信号；超出15000访问时会触发SIGSEGV信号。

**APR分析-环篇**

* APR中少见对数据结构的封装，好像唯一例外的就是其对循环链表，即环(RING)的封装。

在大学的时候学的不是计算机专业，但大三的时候我所学的专业曾开过一门好像叫“计算机软件开发基础”的课，使用的是清华的一本教材，课程的内容包括数据结构。说实话听过几节课，那个老师讲的还不错，只是由于课程目标所限，没讲那么深罢了。当然我接触数据结构要早于这门课的开课时间。早在大一下学期就开始到计算机专业旁听“数据结构”，再说一次实话，虽号称名校名专业，但是那个老师的讲课水平却不敢恭维。

言归正传! 简单说说环(RING)：环是一个首尾相连的双向链表，也就是我们所说的循环链表。对应清华的那本经典的《数据结构》一书中线性表一章的内容，按照书中分类其属于线性表中的链式存储的一种。环是很常见也很实用的数据结构，相信在这个世界上环的实现不止成千上万，但是APR RING(按照APR RING源代码中的注释所说，APR RING的实现源自4.4BSD)却是其中较独特的一个，其最大的特点是其所有对RING的操作都由一组宏(大约30个左右)来实现。在这里不能逐个分析，仅说说一些让人印象深刻的方面吧。

**1、如何使用APR RING？**

我们先来点感性认识! 下面是一个典型的使用APRRING的样例：

假设环节点的结构如下：

struct  elem\_t {    /\* APR RING链接的元素类型定义 \*/

    APR\_RING\_ENTRY(elem\_t) link; /\* 链接域 \*/

   int                                     foo; /\*数据域\*/

};

APR\_RING\_HEAD(elem\_head\_t, elem\_t);

int main() {

    struct elem\_head\_t  head;

    structelem\_t       \*el;

    APR\_RING\_INIT(&head, elem\_t, link);

    /\* 使用其他操作宏插入、删除等操作，例如 \*/

    el = malloc(sizeof(elem\_t);

    el->foo = 20051103;

    APR\_RING\_ELEM\_INIT(el, link);

    APR\_RING\_INSERT\_TAIL(&h, el, elem\_t, link);

}

**2、APR RING的难点--“哨兵”**

环是通过头节点来管理的，头节点是这样一种节点，其next指针指向RING的第一个节点，其prev指针指向RING的最后一个节点，即尾节点。但是通过察看源码发现APR RING通过APR\_RING\_HEAD宏定义的头节点形式如下：

#define APR\_RING\_HEAD(head, elem)     /

    struct head{       /

            struct elem \*next;      /

            struct elem \*prev;      /

    }

如果按照上面的例子进行宏展开，其形式如下：

struct elem\_head\_t {

     struct elem\_t \*next;

     struct elem\_t \*prev;

};

而一个普通的元素elem\_t展开形式如下：

struct elem\_t {

     struct{       /

        struct elem\_t\*next;     /

        struct elem\_t\*prev;     /

     } link;

     int foo;

};

通过对比可以看得出头节点仅仅相当于一个elem\_t的link域。这样做的话必然带来对普通节点和头节点在处理上的不一致，为了避免这种情况的发生，APR RING引入了“哨兵(sentinel)”节点的概念。我们先看看哨兵节点在整个链表中的位置。

sentinel->next = 链表的第一个节点；

sentinel->prev = 链表的最后一个节点；

但是察看APR RING的源码你会发现sentinel节点只是个虚拟存在的节点，这个虚拟节点既有数据域(虚拟出来的，不能引用)又有链接域，好似与普通节点并无差别。在APR RING的源文件中使用了下面这幅图来说明sentinel的位置，同时也指出了sentinel和head的关系 -- head即为sentinel虚拟节点的link域。

 普通节点

+->+-------+<--

   |struct |

   |elem   |

   +-------+

   |prev   |

   |   next|

   +-------+

   | etc.  |

   .       .

   .       .

sentinel节点

+->+--------+<--

   |sentinel|

   |elem    |

   +--------+

   |ring    |

   |   head |

   +--------+

再看看下面APR\_RING\_INIT的源代码：

#define APR\_RING\_INIT(hp, elem, link) do{    /

           APR\_RING\_FIRST((hp)) = APR\_RING\_SENTINEL((hp), elem, link); /

          APR\_RING\_LAST((hp))  = APR\_RING\_SENTINEL((hp), elem, link); /

    } while (0)

你会发现：初始化RING实际上是将head的next和prev指针都指向了sentinel虚拟节点了。从sentinel的角度来说相当于其自己的link域的next和prev都指向了自己。所以判断APRRING是否为空只需要判断RING的首个节点是否为sentinel虚拟节点即可。APR\_RING\_EMPTY宏就是这么做的：

#define APR\_RING\_EMPTY(hp, elem,link)     /

    (APR\_RING\_FIRST((hp)) ==APR\_RING\_SENTINEL((hp), elem, link))

那么如何计算sentinel虚拟节点的地址呢？

我们这样思考：从普通节点说起，如果我们知道一个普通节点的首地址(elem\_addr)，那么我们计算其link域的地址(link\_addr)的公式就应该为link\_addr = elem\_addr + offsetof(elem\_t, link)；前面我们一直在说sentinel虚拟节点看起来和普通节点没什么区别，所以它仍然符合该计算公式。前面我们又说过head\_addr是sentinel节点的link域，这样的话我们将head\_addr输入到公式中得到head\_addr = sentinel\_addr + offsetof(elem\_t, link)，做一下变换即可得到sentinel\_addr= head\_addr - offsetof(elem\_t, link)。看看APR RING源代码就是这样实现的：

#define APR\_RING\_SENTINEL(hp, elem, link)    /

    (struct elem \*)((char \*)(hp) -APR\_OFFSETOF(struct elem, link))

至此APR RING使用一个虚拟sentinel节点分隔RING的首尾节点，已达到对节点操作一致的目的。

**3、使用时注意事项**

这里在使用APR RING时有几点限制：

a) 在定义RING的元素结构时，需要把APR\_RING\_ENTRY放在结构的第一个字段的位置。

b) 链接一种类型的元素就要使用APR\_RING\_HEAD宏定义该种类型RING的头节点类型。学过C++或者了解泛型的人可能都会体味到这里的设计有那么一点范型的味道。比如：

模板：APR\_RING\_HEAD(T\_HEAD,T) ---- 链接----> T类型元素

实例化：APR\_RING\_HEAD(elem\_head\_t,elem\_t) --- 链接---->elem\_t类型元素

**4、APR RING不足之处**

1) 缺少遍历接口

浏览APR RING源码后发现缺少一个遍历宏接口，这里提供一种正向遍历实现：

#define APR\_RING\_TRAVERSE(ep, hp, elem,link)    /

           for ((ep)  = APR\_RING\_FIRST((hp));     /

           (ep) != APR\_RING\_SENTINEL((hp), elem, link);   /

          (ep)  = APR\_RING\_NEXT((ep), link))

大家还可以模仿写出反向遍历的接口APR\_RING\_REVERSE\_TRAVERSE。

**APR分析-进程同步篇**

* 最新的统计数据显示Apache服务器在全世界仍然占据着Web服务器龙头老大的位置，而且市场占有率遥遥领先，所以学习Apache相关知识是完全正确的方向，这里我们继续分析APR进程同步相关内容。

进程同步的源代码的位置在$(APR\_HOME)/locks目录下，本篇blog着重分析unix子目录下的proc\_mutex.c、global\_mutex文件内容，其相应头文件为$(APR\_HOME)/include/apr\_proc\_mutex.h、apr\_global\_mutex.h。其用于不同进程之间的同步以及多进程多线程中的同步问题。

APR提供三种同步措施，分别为：

apr\_thread\_mutex\_t - 支持单个进程内的多线程同步；

apr\_proc\_mutex\_t - 支持多个进程间的同步；

apr\_global\_mutex\_t  - 支持不同进程内的不同线程间同步。

在本篇中着重分析apr\_proc\_mutex\_t。

**1、同步机制**

APR提供多种进程同步的机制供选择使用。在apr\_proc\_mutex.h中列举了究竟有哪些同步机制：

typedef enum {

   APR\_LOCK\_FCNTL,         /\* 记录上锁 \*/

   APR\_LOCK\_FLOCK,         /\* 文件上锁 \*/

   APR\_LOCK\_SYSVSEM,       /\* 系统V信号量 \*/

    APR\_LOCK\_PROC\_PTHREAD,  /\* 利用pthread线程锁特性 \*/

   APR\_LOCK\_POSIXSEM,      /\* POSIX信号量 \*/

   APR\_LOCK\_DEFAULT        /\* 默认进程间锁 \*/

} apr\_lockmech\_e;

这几种锁机制，随便拿出哪一种都很复杂。APR的代码注释中强调了一点就是“只有APR\_LOCK\_DEFAULT”是可移植的。这样一来用户若要使用APR进程同步机制接口，就必须显式指定一种同步机制。

**2、实现点滴**

APR提供每种同步机制的实现，每种机制体现为一组函数接口，这些接口被封装在一个结构体类型中：

/\* in apr\_arch\_proc\_mutex.h \*/

struct apr\_proc\_mutex\_unix\_lock\_methods\_t {

    unsigned int flags;

    apr\_status\_t (\*create)(apr\_proc\_mutex\_t \*,const char \*);

    apr\_status\_t (\*acquire)(apr\_proc\_mutex\_t \*);

    apr\_status\_t (\*tryacquire)(apr\_proc\_mutex\_t \*);

    apr\_status\_t (\*release)(apr\_proc\_mutex\_t \*);

    apr\_status\_t (\*cleanup)(void \*);

    apr\_status\_t (\*child\_init)(apr\_proc\_mutex\_t \*\*,apr\_pool\_t \*, const char \*);

    const char \*name;

};

之后在apr\_proc\_mutex\_t类型中，apr\_proc\_mutex\_unix\_lock\_methods\_t的出现也就在情理之中了:)

/\* in apr\_arch\_proc\_mutex.h \*/

struct apr\_proc\_mutex\_t {

    apr\_pool\_t \*pool;

    const apr\_proc\_mutex\_unix\_lock\_methods\_t \*meth;

    const apr\_proc\_mutex\_unix\_lock\_methods\_t\*inter\_meth;

    int curr\_locked;

    char \*fname;

    ... ...

#if APR\_HAS\_PROC\_PTHREAD\_SERIALIZE

    pthread\_mutex\_t \*pthread\_interproc;

#endif

};

这样APR提供的用户接口其实就是对mech各个“成员函数”功能的“薄封装”，而真正干活的其实是apr\_proc\_mutex\_t中的meth字段的“成员函数”，它们的工作包括mutex的创建、获取(加锁)和清除(解锁)等。以“获取锁”为例APR的实现如下：

APR\_DECLARE(apr\_status\_t) apr\_proc\_mutex\_lock(apr\_proc\_mutex\_t\*mutex)

{

    return mutex->meth->acquire(mutex);

}

**3、同步机制**

按照枚举类型apr\_lockmech\_e的声明，我们知道APR为我们提供了5种同步机制，下面分别简单说说：

(1) 记录锁

记录锁是一种建议性锁，它不能防止一个进程写已由另一个进程上了读锁的文件，它主要利用fcntl系统调用来完成锁功能的，记得在以前的一篇关于APR 文件I/O的Blog中谈过记录锁，这里不再详细叙述了。

(2) 文件锁

文件锁是记录锁的一个特例，其功能由函数接口flock支持。值得说明的是它仅仅提供“写入锁”(独占锁)，而不提供“读入锁”(共享锁)。

(3) System V信号量

System V信号量是一种内核维护的信号量，所以我们只需调用semget获取一个System V信号量的描述符即可。值得注意的是与POSIX的单个“计数信号量”不同的是System V信号量是一个“计数信号量集”。所以我们在注意的是在初始化时设定好信号量集的属性以及在调用semop时正确选择信号量集中的信号量。在APR的System V信号量集中只是申请了一个信号量。

(4) 利用线程互斥锁机制

APR使用pthread提供的互斥锁机制。原本pthread互斥锁是用来互斥一个进程内的各个线程的，但APR在共享内存中创建了pthread\_mutex\_t，这样使得不同进程的主线程实现互斥，从而达到进程间互斥的目的。截取部分代码如下：

new\_mutex->pthread\_interproc = (pthread\_mutex\_t \*)mmap(

                                      (caddr\_t) 0,

                                      sizeof(pthread\_mutex\_t),

                                      PROT\_READ | PROT\_WRITE, MAP\_SHARED,

                                      fd, 0);

(5) POSIX信号量

APR使用了POSIX有名信号量机制，从下面的代码中我们可以看出这一点：

/\* in proc\_mutex.c \*/

apr\_snprintf(semname, sizeof(semname),"/ApR.%lxZ%lx", sec, usec); /\* APR自定义了一种POSIX信号量命名规则，在源代码中有说明\*/

psem = sem\_open(semname, O\_CREAT, 0644, 1);

**4、如何使用**

我们知道父进程的锁其子进程并不继承。APR进程同步机制的一个典型使用方法就是：“Createthe mutex in the Parent， Attach to it in the Child”。APR提供接口apr\_proc\_mutex\_child\_init在子进程中re-open themutex。

**5、小结**

APR提供多种锁机制，所以使用的时候要根据具体应用情况细心选择。

**APR分析-线程篇**

* 并行一直是程序设计领域的难点，而线程是并行的一种重要的手段，而且线程的一些特性也能在进程并行时发挥很好的作用(在“线程同步篇”中详细阐述)。

APR线程的源代码的位置在$(APR\_HOME)/threadproc目录下，本篇blog着重分析unix子目录下的thread.c文件内容，其相应头文件为$(APR\_HOME)/include/apr\_threadproc.h。

**一、线程基础**

《深入理解计算机系统》（以下称CS.APP）一书中对线程基础概念的讲解让我眼前豁然开朗，这里不妨引述一下：

(1) 在传统观点中，进程是由存储于用户虚拟内存中的代码、数据和栈，以及由内核维护的“进程上下文”组成的，其中“进程上下文”又可以看成“程序上下文”和“内核上下文”组成，可参见下面图示：

进程--

      |- 进程上下文

            |- 程序上下文

                  |- 数据寄存器

                  |- 条件码

                  |- 栈指针

                  |- 程序计数器

           |- 内核上下文

                  |- 进程ID

                  |- VM结构

                  |- Open files

                  |- 已设置的信号处理函数

                  |- brk pointer

    |- 代码、数据和栈(在虚存中)

           |- 栈区<-- SP

           |- 共享库区

           |- 运行时堆区 <-- brk

           |- 可读/写数据区

           |- 只读代码/数据区 <-- PC

(2) 另种观点中，进程是由线程、代码和数据以及内核上下文组成的，下图更能直观的展示出两种观点的异同：

进程 --+

   |- 线程

          |- 栈区<-- SP

          |- 线程上下文

          |- 线程ID

          |- 数据寄存器

          |- 条件码

          |- 栈指针

          |- 程序计数器

  |- 内核上下文

          |- 进程ID

          |- VM结构

           |-Open files

          |- 已设置的信号处理函数

           |-brk pointer

  |- 代码、数据(在虚存中)

          |- 共享库区

          |- 运行时堆区 <-- brk

          |- 可读/写数据区

          |- 只读代码/数据区 <-- PC

对比两种观点我们可以得出以下几点结论：

(a) 从观点(2)可以看出进程内的多个线程共享进程的内核上下文和代码、数据(当然不包括栈区)；

(b) 线程上下文比进程上下文小，且切换代价小；

(c) 线程不像进程那样有着“父-子”体系，同一个进程内的线程都是“对等的”，主线程与其他线程不同之处就在于其是进程创建的第一个线程。

**二、APR线程管理接口**

如今应用最广泛的线程包就是PosixThread了。APR对线程的封装也是基于Posix thread的。

APR线程管理接口针对apr\_thread\_t这个基本的数据结构进行操作，apr\_thread\_t的定义很简单：

/\* apr\_arch\_threadproc.h \*/

struct apr\_thread\_t {

    apr\_pool\_t \*pool;

    pthread\_t \*td;

    void \*data;

    apr\_thread\_start\_t func;

    apr\_status\_t exitval;

};

这个结构中包含了线程ID、线程函数以及该函数的参数数据。不过APR的线程函数定义与Pthread的有不同，“Pthread线程函数”是这样的：

typedef void \*(start\_routine)(void\*);

而“APR线程函数”如下：

typedef void \*(APR\_THREAD\_FUNC \*apr\_thread\_start\_t)(apr\_thread\_t\*,void\*);

**1、apr\_thread\_create**

apr\_thread\_create内部定义了一个dummy\_worker的“Pthread线程函数”，并将apr\_thread\_t结构作为参数传入，然后在dummy\_worker中启动“APR的线程函数”。在该函数的参数列表中有一项类型为apr\_threadattr\_t：

struct apr\_threadattr\_t {

    apr\_pool\_t \*pool;

    pthread\_attr\_t attr;

};

这个类型封装了线程的属性，不同的线程属性会导致线程的行为有所不同。Pthread提供多种线程属性设置接口，可是APR并未全部提供，必要时我觉得可以自己来调用Pthread接口。APR提供的属性设置接口包括设置线程的可分离性、线程栈大小和栈Guard区域属性。

**2、apr\_thread\_exit**

进程退出我们可以直接调用exit函数，而线程退出也有几种方式：

(1) 隐式退出 - 可以理解为线程main routine代码结束返回；

(2) 显式退出 - 调用线程包提供的显式退出接口，在apr中就是apr\_thread\_exit；

(3) 另类显式退出 - 调用exit函数，不仅自己退出，其所在线程也跟着退出了；

(4) 被“黑”退出 - 被别的“对等”线程调用pthread\_cancel而被迫退出。

apr\_thread\_exit属于种类(2)，该种类退出应该算是线程的优雅退出了。apr\_thread\_exit做了3个工作，分别为设置线程返回值、释放pool中资源和调用pthread\_exit退出。

**3、apr\_thread\_join和apr\_thread\_detach**

进程有waitpid，线程有join。线程在调用apr\_thread\_exit后，只是其执行停止了，其占有的“资源”并不一定释放，这里的“资源”我想就是“另种观点”中的“线程上下文”，线程有两种方式来释放该“资源”，这主要由线程的“可分离”属性决定的。如果线程是“可分离的”，当线程退出后就会自动释放其“资源”，如果线程为“非可分离的”，则必须由“对等线程”调用join接口来释放其资源。apr\_thread\_detach用来将其调用线程转化为“可分离”线程，而apr\_thread\_join用来等待某个线程结束并释放其资源。

**三、小结**

基本的线程管理接口相对较简单，关键是对线程概念的理解。接下来的“线程同步”则是件比较有趣的话题。

**APR分析-网络IO篇**

* “这个世界如果没有了网络就好比没有了石油、没有了电一样，是多么的可怕呀。”相信世界上已经有很多很多的人能够同意这种观点了，通过这个观点也可以看出网络在现代人们心中的地位。而运行在网络节点上的网络应用程序则是在幕后默默地为人们提供着服务。Apache Server就是其中一个典型的代表。而APR网络I/O库则像磐石一样支撑着Apache Server的运行。

APR网络I/O的源代码的位置在$(APR\_HOME)/network\_io目录下，本篇blog着重分析unix子目录下的各.c文件内容，其相应头文件为$(APR\_HOME)/include/apr\_network\_io.h。

以程序员的视角来看待网络，这样我们可以忽略一些网络的基础概念。下面将循序渐进地接触网络，并说明APR是如何支持这些网络概念的。

**一、IP地址 -- 主机通信**

我们熟知的并且每天工作于其上的因特网是一个世界范围的主机的集合，这个主机集合被映射为一个32位(目前)或者64位(将来)IP地址；而IP地址又被映射为一组因特网域名；一个网络中的主机上的进程能通过一个连接(connection)和任何其他网络中的主机上的进程通信。

**1、IP地址存储**

在如今的IPV4协议中我们一般使用一个unsignedint来存储IP地址，在UNIX平台下，使用如下结构来存储一个IP地址的值：

/\* Internet address structure \*/

struct in\_addr {

 unsigned int s\_addr; /\* network byte order (big-endian) \*/

};

这里值得一提的是APR关于IP地址存储的做法，看如下代码：

#if (!APR\_HAVE\_IN\_ADDR)

/\*\*

 \* We need to make sure we always have an in\_addr type, soAPR will just

 \* define it ourselves, if the platform doesn't provide it.

 \*/

struct in\_addr {

    apr\_uint32\_t  s\_addr;

};

#endif

APR保证了其所在平台上in\_addr的存在。还有一点儿需要注意的是在in\_addr中，s\_addr是以网络字节序存储的。如果你的IP地址不符合条件，可通过调用一些辅助接口来做转换，这些接口包括：

htonl : host to network long ;

htons : host to network short ;

ntohl : network to host long ;

ntohs : network to host short.

**2、IP地址表示**

我们平时看到的IP地址都是类似“xxx.xxx.xxx.xxx”这样的点分十进制的。上面说过IP地址使用的是一个unsignedint整形数来表示。这样就存在着一个IP地址表示和IP地址存储之间的一个转换过程。APR提供这一转换支持，我们用一个例子来说明：

#include <apr.h>

#include <apr\_general.h>

#include "apr\_network\_io.h"

#include "apr\_arch\_networkio.h"

int main(int argc, const char \* const \* argv, const char \* const\*env)

{

       apr\_app\_initialize(&argc, &argv, &env);

        char   presentation[100];

       int     networkfmt;

        memset(presentation, 0,sizeof(presentation));

        apr\_inet\_pton(AF\_INET,"255.255.255.255", &networkfmt);

       printf("0x%x/n", networkfmt);

        apr\_inet\_ntop(AF\_INET,&networkfmt, presentation, sizeof(presentation));

       printf("presentation is %s/n", presentation);

        apr\_terminate();

        return 0;

}

APR提供apr\_inet\_pton将我们熟悉的点分十进制形式转换成一个整型数存储的IP地址；而apr\_inet\_ntop则将一个存整型数存储的IP地址转换为我们可读的点分十进制形式。这两个接口的功能类似于系统调用inet\_pton和inet\_ntop，至于使用哪个就看你的喜好了^\_^。

**二、SOCKET -- 进程通信**

前面提到过通过一个连接(connection)可以连接两个internet不同或相同主机上的不同进程，这个连接是点对点的。而从Unix内核角度来看，SOCKET则是连接的一个端点。每个SOCKET都有一个地址，其地址由主机IP地址和通讯端口号组成。一个连接有两个端点，这样一个连接就可以由一个SOCKET对唯一表示了。这个SOCKET对是这个样子的(cliaddr:cliport, servaddr:servport)。

那么在应用程序中我们如何获取和使用这一互联网上的进程通讯利器呢？每个平台都为应用程序提供了一套SOCKET编程接口，APR又在不同平台提供的接口之上进行了封装，使代码可以在不同平台上编译运行，而且易用性也有所提高。

**1、SOCKET描述符**

SOCKET属于系统资源，我们必须通过系统调用来申请该资源。SOCKET资源的申请类似于FILE，在使用文件时我们通过调用open函数获取文件描述符，类似我们也可通过调用下面的接口来获取SOCKET描述符：

int socket(int domain, int type, int protocol);

从Unix程序的角度来看，SOCKET就是一个有相应描述符的打开的文件。在APR中我们可以通过调用apr\_socket\_create来创建一个APR自定义的SOCKET对象，该SOCKET结构如下：

/\* apr\_arch\_networkio.h \*/

struct apr\_socket\_t {

    apr\_pool\_t \*cntxt;

    int socketdes;

    int type;

    int protocol;

    apr\_sockaddr\_t \*local\_addr;

    apr\_sockaddr\_t \*remote\_addr;

    apr\_interval\_time\_t timeout;

#ifndef HAVE\_POLL

    int connected;

#endif

    int local\_port\_unknown;

    int local\_interface\_unknown;

    int remote\_addr\_unknown;

    apr\_int32\_t options;

    apr\_int32\_t inherit;

    sock\_userdata\_t \*userdata;

#ifndef WAITIO\_USES\_POLL

    /\* if there is a timeout set, then this pollsetis used \*/

    apr\_pollset\_t \*pollset;

#endif

};

该结构中的socketdes字段其实是真正存储由socket函数返回的SOCKET描述符的，其他字段都是为APR自己所使用的，这些字段在Bind、Connect等过程中使用。另外需要提及的就是要分清SOCKET描述符和SOCKET地址(IP地址，端口号)，前者是系统资源，而后者用来描述一个连接的一个端点的地址。SOCKET描述符可以代表任意的SOCKET地址，也可以绑定到某个固定的SOCKET地址上(在后面有说明)。我们如果不显式将SOCKET描述符绑定到某SOCKET地址上，系统内核就会自动为该SOCKET描述符分配一个SOCKET地址。

**2、SOCKET属性**

还是与文件对比，在文件系统调用中有一个fcntl接口可以用来获取或设置已分配的文件描述符的属性，如是否Block、是否Buffer等。SOCKET也提供类似的接口调用setsockopt和getsockopt。在APR中等价于该功能的接口是apr\_socket\_opt\_set和apr\_socket\_opt\_get。APR在apr\_network\_io.h中提供如下SOCKET的参数属性：

#define APR\_SO\_LINGER       1    /\*\*< Linger \*/

#define APR\_SO\_KEEPALIVE    2    /\*\*< Keepalive \*/

#defineAPR\_SO\_DEBUG        4    /\*\*< Debug \*/

#define APR\_SO\_NONBLOCK     8    /\*\*< Non-blocking IO \*/

#define APR\_SO\_REUSEADDR     16  /\*\*< Reuse addresses \*/

#define APR\_SO\_SNDBUF       64   /\*\*< Send buffer \*/

#define APR\_SO\_RCVBUF       128  /\*\*< Receive buffer \*/

#define APR\_SO\_DISCONNECTED  256  /\*\*< Disconnected\*/

... ...

另外从上面这些属性值(都是2的n次方)可以看出SOCKET也是使用一个属性控制字段中的“位”来控制SOCKET属性的。

再有APR提供一个宏apr\_is\_option\_set来判断一个SOCKET是否拥有某个属性。

**3、Connect、Bind、Listen、Accept -- 建立连接**

这里不详述C/S模型了，只是说说APR支持C/S模型的一些接口。

(1) apr\_socket\_connect

客户端连接服务器端的唯一调用就是connect，connect试图建立一个客户端进程与服务器端进程的连接。apr\_socket\_connect的参数分别为客户端已经打开的一个SOCKET以及指定的服务器端的SOCKET地址(IP ADDR : PORT)。apr\_socket\_connect内部实现的流程大致如以下代码：

apr\_socket\_connect

{

 do {

         rc =connect(sock->socketdes,

                    (const struct sockaddr \*)&sa->sa.sin,

                    sa->salen);

     } while (rc == -1 && errno ==EINTR);   -------- (a)

 if ((rc == -1) && (errno == EINPROGRESS || errno ==EALREADY)

                  && (sock->timeout > 0)) {

        rc =apr\_wait\_for\_io\_or\_timeout(NULL, sock, 0); --------- (b) 注[1]

        if (rc != APR\_SUCCESS){

           return rc;

        }

 if (rc == -1 && errno != EISCONN) {

         returnerrno;   --------- (c)

     }

 初始化sock->remote\_addr;

 ... ...

}

对上述代码进行若干说明：

(a) 执行系统调用connect连接服务器端，注意这里做了防止信号中断的处理，这个技巧在以前的文章中提到过，这里不详述；

(b) 如果系统操作正在进行中，调用apr\_wait\_for\_io\_or\_timeout进行超时等待；

(c) 错误返回，前提errno不是表示已连接上。

一旦apr\_socket\_connect成功返回，我们就已经成功建立了一个SOCKET对，即一个连接。

(2) apr\_socket\_bind

Bind、Listen和Accept这三个过程是服务器端用于接收“连接”的必经之路。其中Bind就是告诉操作系统内核显式地为该SOCKET描述符分配一个SOCKET地址，这个SOCKET地址就不能被其他SOCKET描述符占用了。在服务器编程中Bind几乎成为了“必选”之调用，因为一般服务器程序都有自己的“名气很大”的SOCKET地址，如TELNET服务端口号23等。apr\_socket\_bind也并未做太多的工作，只是简单的调用了bind系统接口，并设置了apr\_socket\_t结构的几个local\_addr字段。

(3) apr\_socket\_listen

按照《Unix网络编程 Vol1》的说法，SOCKET描述符在初始分配时都处于“主动连接”状态，Listen过程将该SOCKET描述符从“主动连接”转换为“被动状态”，并告诉内核接受该SOCKET描述符的连接请求。apr\_socket\_listen的背后直接就是listen接口调用。

(4) apr\_socket\_accept

Accept过程在“被动状态”SOCKET描述符上接受一个客户端的连接，这时系统内核会自动分配一个新的SOCKET描述符，内核为该描述符自动分配一个SOCKET地址，来代表这条连接的服务器端。注意在SOCKET编程接口中除了socket函数能分配新的SOCKET描述符之外，accept也是另外的一个也是唯一的一个能分配新的SOCKET描述符的系统调用了。apr\_socket\_accept首先在pool中分配一个新的apr\_socket\_t结构变量，然后调用accept，并设置新变量的各个字段。

**4、Send/Recv -- 数据传输**

网络通信最重要的还是数据传输，在SOCKET编程接口中最常见的两个接口就是recv和send。在APR中分别有apr\_socket\_recv和apr\_socket\_send与前面二者对应。下面逐一分析。

(1) apr\_socket\_recv

首先来看看apr\_socket\_recv的实现过程：

apr\_socket\_recv

{

 if (上次调用apr\_socket\_recv没有读完所要求的字节数) { ----------(a)

  设置sock->options;

  goto do\_select;

 }

 do {

         rv =read(sock->socketdes, buf, (\*len)); ------ (b)

     } while (rv == -1 && errno ==EINTR);

 if ((rv == -1) && (errno == EAGAIN || errno ==EWOULDBLOCK)

                   && (sock->timeout > 0)) {

do\_select:

         arv =apr\_wait\_for\_io\_or\_timeout(NULL, sock, 1);

         if (arv !=APR\_SUCCESS) {

             \*len = 0;

             return arv;

         }

         else {

             do {

                 rv = read(sock->socketdes, buf, (\*len));

             } while (rv == -1 && errno == EINTR);

         }

     }  ------------ (c)

 设置(\*len)和sock->options;  -------------(d)

 ... ...

}

针对上面代码进行简单说明：

(a) 一次apr\_socket\_recv调用完全有可能没有读完所要求的字节数，这里做个判断以决定是否继续读完剩下的数据；

(b) 调用read读取SOCKET缓冲区数据，注意这里做了防止信号中断的处理，这个技巧在以前的文章中提到过，这里不详述；

(c) 如果SOCKET操作正在忙，我们调用apr\_wait\_for\_io\_or\_timeout等待，直到SOCKET可用。这里我觉得好像有个问题，想象一下如果上一次SOCKET的状态为APR\_INCOMPLETE\_READ，那么重新调用apr\_socket\_read后在SOCKET属性中去掉APR\_INCOMPLETE\_READ，然后进入apr\_wait\_for\_io\_or\_timeout过程，一旦apr\_wait\_for\_io\_or\_timeout失败，那么就直接返回了。而实际上SOCKET仍然应该处于APR\_INCOMPLETE\_READ状态，而下次再调用apr\_socket\_read就直接进入一轮完整数据的读取过程了，不知道这种情形是否能否发生。

(d) 将(\*len)设置为实际从SOCKET Buffer中读取的字节数，并根据这一实际数据与要求数据作比较来设置sock->options。

(2) apr\_socket\_send

apr\_socket\_send负责发送数据到SOCKET Buffer，其实现的方式与apr\_socket\_recv大同小异，这里就不分析了。

**三、小结**

APR Network I/O中还有对Multicast的支持，由于平时不常接触，这里不分析了。

注[1]：

/\* in errno.h \*/

#define EISCONN        133     /\* Socket is already connected \*/

#define EALREADY       149     /\* operation already in progress \*/

#define EINPROGRESS    150     /\* operation now in progress \*/

**APR分析-线程同步篇**

* 在线程同步方面，Posix标准定义了3种同步模型，分别为互斥量、条件变量和读写锁。APR也“浅”封装了这3种模型，只是在“读写锁”一块儿还没有全部完成。

线程同步的源代码的位置在$(APR\_HOME)/locks目录下，本篇blog着重分析unix子目录下的thread\_mutex.c、thread\_rwlock.c和thread\_cond.c文件的内容，其相应头文件为(APR\_HOME)/include/apr\_thread\_mutex.h、apr\_thread\_rwlock.h和apr\_thread\_cond.h。

由于APR的封装过于“浅显”，实际上也并没有多少值得分析的“靓点”。所以本篇实际上是在讨论线程同步的3种运行模型。

**一、互斥量**

互斥量是线程同步中最基本的同步方式。互斥量用于保护代码中的临界区，以保证在任一时刻只有一个线程或进程访问临界区。

**1、互斥量的初始化**

在POSIX Thread中提供两种互斥量的初始化方式，如下：

(1) 静态初始化

互斥量首先是一个变量，Pthread提供预定义的值来支持互斥量的静态初始化。举例如下：

static pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

既然是静态初始化，那么必然要求上面的mutex变量需要静态分配。在APR中并不支持apr\_thread\_mutex\_t的使用预定值的静态初始化(但可以变通的利用下面的方式进行静态分配的mutex的初始化)。

(2) 动态初始化

除了上面的情况，如果mutex变量在堆上或在共享内存中分配的话，我们就需要调用一个初始化函数来动态初始化该变量了。在Pthread中的对应接口为pthread\_mutex\_init。APR封装了这一接口，我们可以使用下面方式在APR中初始化一个apr\_thread\_mutex\_t变量。

       apr\_thread\_mutex\_t \*mutex = NULL;

       apr\_pool\_t  \*pool = NULL;

       apr\_status\_t  stat;

        stat =apr\_pool\_create(&pool, NULL);

        if (stat !=APR\_SUCCESS) {

               printf("error in pool %d/n", stat);

        } else {

               printf("ok in pool/n");

        }

        stat =apr\_thread\_mutex\_create(&mutex, APR\_THREAD\_MUTEX\_DEFAULT, pool);

        if (stat !=APR\_SUCCESS) {

               printf("error %d in mutex/n", stat);

        } else {

               printf("ok in mutex/n");

        }

**2、互斥锁的软弱性所在**

互斥锁之软弱性在于其是一种协作性锁，其运作时对各线程有一定的要求，即“所有要访问临界区的线程必须首先获取这个互斥锁，离开临界区后释放该锁”，一旦某一线程不遵循该要求，那么这个互斥锁就形同虚设了。如下面的例子：

举例：我们有两个线程，一个线程A遵循要求，每次访问临界区均先获取锁，然后将临界区的变量x按偶数值递增，另一个线程B不遵循要求直接修改x值，这样即使在线程A获取锁的情况下仍能修改临界区的变量x。

static apr\_thread\_mutex\_t      \*mutex  = NULL;

staticint                               x       = 0;

staticapr\_thread\_t            \*t1     = NULL;

staticapr\_thread\_t            \*t2     = NULL;

static void \* APR\_THREAD\_FUNC thread\_func1(apr\_thread\_t \*thd, void\*data)

{

       apr\_time\_t      now;

        apr\_time\_exp\_t xt;

        while (1) {

               apr\_thread\_mutex\_lock(mutex);

               now = apr\_time\_now();

               apr\_time\_exp\_lt(&xt, now);

               printf("[threadA]: own the lock, time[%02d:%02d:%02d]/n", xt.tm\_hour,xt.tm\_min,

                        xt.tm\_sec);

               printf("[threadA]: x = %d/n", x);

               if (x % 2 || x == 0) {

                       x += 2;

               } else {

                       printf("[threadA]: Warning: x变量值被破坏，现重新修正之/n");

                       x += 1;

               }

               apr\_thread\_mutex\_unlock(mutex);

               now = apr\_time\_now();

               apr\_time\_exp\_lt(&xt, now);

               printf("[threadA]: release the lock, time[%02d:%02d:%02d]/n",xt.tm\_hour, xt.tm\_min,

                        xt.tm\_sec);

               sleep(2);

        }

        return NULL;

}

static void \* APR\_THREAD\_FUNC thread\_func2(apr\_thread\_t \*thd, void\*data)

{

       apr\_time\_t      now;

        apr\_time\_exp\_t xt;

        while (1) {

               x ++;

               now = apr\_time\_now();

               apr\_time\_exp\_lt(&xt, now);

               printf("[threadB]: modify the var, time[%02d:%02d:%02d]/n",xt.tm\_hour, xt.tm\_min,  xt.tm\_sec);

               sleep(2);

        }

        return NULL;

}

int main(int argc, const char \* const \* argv, const char \* const\*env)

{

       apr\_app\_initialize(&argc, &argv, &env);

        apr\_status\_t stat;

        //...

        /\*

         \* 创建线程

         \*/

        stat =apr\_thread\_create(&t1, NULL, thread\_func1, NULL, pool);

        stat =apr\_thread\_create(&t2, NULL, thread\_func2, NULL, pool);

         //...

        apr\_terminate();

        return 0;

}

//output

... ...

[threadA]: own the lock, time[10:10:15]

[threadB]: modify the var, time[10:10:15]

[threadA]: x = 10

[threadA]: Warning: x变量值被破坏，现重新修正之

[threadA]: release the lock, time[10:10:15]

当然这个例子不一定很精确的表明threadB在threadA拥有互斥量的时候修改了x值。

**二、条件变量**

互斥量一般用于被设计被短时间持有的锁，一旦我们不能确定等待输入的时间时，我们可以使用条件变量来完成同步。我们曾经说过I/O复用，在我们调用poll或者select的时候实际上就是在内核与用户进程之间达成了一个协议，即当某个I/O描述符事件发生的时候内核通知用户进程并且将处于挂起状态的用户进程唤醒。而这里我们所说的条件变量让对等的线程间达成协议，即“某一线程发现某一条件满足时必须发信号给阻塞在该条件上的线程，将后者唤醒”。这样我们就有了两种角色的线程，分别为

(1) 给条件变量发送信号的线程

其流程大致为：

{

        获取条件变量关联锁；

        修改条件为真；

        调用apr\_thread\_cond\_signal通知阻塞线程条件满足了；------(a)

        释放变量关联锁；

}

(2) 在条件变量上等待的线程

其流程大致为：

{

        获取条件变量关联锁；

        while (条件为假) {--------------------- (c)

               调用apr\_thread\_cond\_wait阻塞在条件变量上等待；------(b)

        }

        修改条件；

        释放变量关联锁；

}

上面两个流程中，理解三点最关键：

a) apr\_thread\_cond\_signal中调用的pthread\_cond\_signal保证至少有一个阻塞在条件变量上的线程恢复；在《Unix网络编程 Vol2》中也谈过这里存在着一个race。即在发送cond信号的同时，该发送线程仍然持有条件变量关联锁，那么那个恢复线程的apr\_thread\_cond\_wait返回时仍然拿不到这把锁就会再次挂起。这里的这个race要看各个平台实现是如何处理的了。

b) apr\_thread\_cond\_wait中调用的pthread\_cond\_wait原子的将调用线程挂起，并释放其持有的条件变量关联锁；

c) 这里之所以使用while反复测试条件，是防止“伪唤醒”的存在，即条件并未满足就被唤醒。所以无论怎样，唤醒后我都需要重新测试一下条件，保证该条件的的确确满足了。

条件变量在解决“生产者-消费者”问题中有很好的应用，在我以前的一篇blog中也说过这个问题。

**三、读写锁**

前面说过，互斥量把想进入临界区而又试图获取互斥量的所有线程都阻塞住了。读写锁则改进了互斥量的这种霸道行为，它区分读临界区数据和修改临界区数据两种情况。这样如果有线程持有读锁的话，这时再有线程想读临界区的数据也是可以再获取读锁的。读锁和写锁的分配规则在《Unix网络编程 Vol2》中有详细说明，这里不详述。

**四、小结**

三种同步方式如何选择？场合不同选择也不同。互斥量在于完全同步的临界区访问；条件变量在解决“生产者-消费者”模型问题上有独到之处；读写锁则在区分对临界区读写的时候使用。