int \_\_pthread\_create\_2\_1(pthread\_t \*thread, const pthread\_attr\_t \*attr,

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

// thread 为pthread\_t，实则是pthread指针。

// typedef struct pthread \*pthread\_t; ->unsigned long int

// attr 创建线程时的附加属性

// start\_routine 执行的线程函数

// arg参数首地址指针

{

pthread\_descr self = thread\_self();

// 向管理线程的请求

struct pthread\_request request;

// 在初始化时 全局 int \_\_pthread\_manager\_request = -1;

if (\_\_pthread\_manager\_request < 0) {

// 首先Linuxthreads需要有\_\_pthread\_manager\_thread，这个线程是个很特殊的线程

// 初始化Linuxthreads系统线程

if (\_\_pthread\_initialize\_manager() < 0) return EAGAIN;

}

request.req\_thread = self;

// 请求类型 CREATE ，并将新线程（轻量进程）的参数都打包到request结构中。

request.req\_kind = REQ\_CREATE;

request.req\_args.create.attr = attr;

request.req\_args.create.fn = start\_routine;

request.req\_args.create.arg = arg;

//sigprocmask用于改变进程的当前阻塞信号集

sigprocmask(SIG\_SETMASK, (const sigset\_t \*) NULL,

&request.req\_args.create.mask);

// 向\_\_pthread\_manager\_request写入request请求。 request请求将被\_pthread\_manager\_thread读取并处理。

\_\_libc\_write(\_\_pthread\_manager\_request, (char \*) &request, sizeof(request));

// 挂起一下

suspend(self);

if (THREAD\_GETMEM(self, p\_retcode) == 0)

\*thread = (pthread\_t) THREAD\_GETMEM(self, p\_retval);

return THREAD\_GETMEM(self, p\_retcode);

}

#if defined HAVE\_ELF && defined PIC && defined DO\_VERSIONING

default\_symbol\_version (\_\_pthread\_create\_2\_1, pthread\_create, GLIBC\_2.1);

int \_\_pthread\_create\_2\_0(pthread\_t \*thread, const pthread\_attr\_t \*attr,

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

{

/\* The ATTR attribute is not really of type `pthread\_attr\_t \*'. It has

the old size and access to the new members might crash the program.

We convert the struct now. \*/

pthread\_attr\_t new\_attr;

if (attr != NULL)

{

size\_t ps = \_\_getpagesize ();

memcpy (&new\_attr, attr,

(size\_t) &(((pthread\_attr\_t\*)NULL)->\_\_guardsize));

new\_attr.\_\_guardsize = ps;

new\_attr.\_\_stackaddr\_set = 0;

new\_attr.\_\_stackaddr = NULL;

new\_attr.\_\_stacksize = STACK\_SIZE - ps;

attr = &new\_attr;

}

return \_\_pthread\_create\_2\_1 (thread, attr, start\_routine, arg);

}

symbol\_version (\_\_pthread\_create\_2\_0, pthread\_create, GLIBC\_2.0); // 由此可知pthread\_create入口函数为\_\_pthread\_create\_2\_1

#else

strong\_alias (\_\_pthread\_create\_2\_1, pthread\_create)

#endif

由上面的函数可知，pthread\_create对线程的创建实际上是向pthread\_manager\_thread发送个request请求。

下面我们就看pthread\_manager\_thread是怎样初始化的：

首先要进行基本系统的初始化工作：

即填充\_\_pthread\_initial\_thread（算是模板吧？）其他属性：

/\* Descriptor of the initial thread \*/

struct \_pthread\_descr\_struct \_\_pthread\_initial\_thread = {

&\_\_pthread\_initial\_thread, /\* pthread\_descr p\_nextlive \*/ // 将进程中所有用户线程串在了一起

&\_\_pthread\_initial\_thread, /\* pthread\_descr p\_prevlive \*/

NULL, /\* pthread\_descr p\_nextwaiting \*/

NULL, /\* pthread\_descr p\_nextlock \*/

PTHREAD\_THREADS\_MAX, /\* pthread\_t p\_tid \*/

0, /\* int p\_pid \*/

0, /\* int p\_priority \*/

&\_\_pthread\_handles[0].h\_lock, /\* struct \_pthread\_fastlock \* p\_lock \*/

0, /\* int p\_signal \*/

NULL, /\* sigjmp\_buf \* p\_signal\_buf \*/

NULL, /\* sigjmp\_buf \* p\_cancel\_buf \*/

0, /\* char p\_terminated \*/

0, /\* char p\_detached \*/

0, /\* char p\_exited \*/

NULL, /\* void \* p\_retval \*/

0, /\* int p\_retval \*/

NULL, /\* pthread\_descr p\_joining \*/

NULL, /\* struct \_pthread\_cleanup\_buffer \* p\_cleanup \*/

0, /\* char p\_cancelstate \*/

0, /\* char p\_canceltype \*/

0, /\* char p\_canceled \*/

NULL, /\* int \*p\_errnop \*/

0, /\* int p\_errno \*/

NULL, /\* int \*p\_h\_errnop \*/

0, /\* int p\_h\_errno \*/

NULL, /\* char \* p\_in\_sighandler \*/

0, /\* char p\_sigwaiting \*/

PTHREAD\_START\_ARGS\_INITIALIZER, /\* struct pthread\_start\_args p\_start\_args \*/

{NULL}, /\* void \*\* p\_specific[PTHREAD\_KEY\_1STLEVEL\_SIZE] \*/

{NULL}, /\* void \* p\_libc\_specific[\_LIBC\_TSD\_KEY\_N] \*/

0, /\* int p\_userstack \*/

NULL, /\* void \* p\_guardaddr \*/

0, /\* size\_t p\_guardsize \*/

&\_\_pthread\_initial\_thread, /\* pthread\_descr p\_self \*/

0, /\* Always index 0 \*/

0, /\* int p\_report\_events \*/

{{{0, }}, 0, NULL}, /\* td\_eventbuf\_t p\_eventbuf \*/

ATOMIC\_INITIALIZER, /\* struct pthread\_atomic p\_resume\_count \*/

0, /\* char p\_woken\_by\_cancel \*/

NULL /\* struct pthread\_extricate\_if \*p\_extricate \*/

};

static void pthread\_initialize(void)

{

struct sigaction sa;

sigset\_t mask;

struct rlimit limit;

int max\_stack;

//\_\_pthread\_initial\_thread\_bos: Limit between the stack of the initial thread (above) and the

//stacks of other threads (below). Aligned on a STACK\_SIZE boundary.

/\* 如果初始化工作已完成，则退出 \*/

if (\_\_pthread\_initial\_thread\_bos != NULL) return;

#ifdef TEST\_FOR\_COMPARE\_AND\_SWAP

/\* Test if compare-and-swap is available \*/

\_\_pthread\_has\_cas = compare\_and\_swap\_is\_available();

#endif

/\* For the initial stack, reserve at least STACK\_SIZE bytes of stack

below the current stack address, and align that on a

STACK\_SIZE boundary. \*/

\_\_pthread\_initial\_thread\_bos =

(char \*)(((long)CURRENT\_STACK\_FRAME. - 2 \* STACK\_SIZE) & ~(STACK\_SIZE - 1));

/\* 对initial\_thread.pid赋值：主进程pid \*/

\_\_pthread\_initial\_thread.p\_pid = \_\_getpid();

/\* If we have special thread\_self processing, initialize that for the

main thread now. \*/

#ifdef INIT\_THREAD\_SELF

INIT\_THREAD\_SELF(&\_\_pthread\_initial\_thread, 0);

#endif

/\* 共享主进程errno/h\_errno.\*/

\_\_pthread\_initial\_thread.p\_errnop = &\_errno;

\_\_pthread\_initial\_thread.p\_h\_errnop = &\_h\_errno;

/\* Play with the stack size limit to make sure that no stack ever grows

beyond STACK\_SIZE minus two pages (one page for the thread descriptor

immediately beyond, and one page to act as a guard page). \*/

getrlimit(RLIMIT\_STACK, &limit); // 获取STACK limit

max\_stack = STACK\_SIZE - 2 \* \_\_getpagesize();

if (limit.rlim\_cur > max\_stack) {

limit.rlim\_cur = max\_stack;

setrlimit(RLIMIT\_STACK, &limit);

}

#ifdef \_\_SIGRTMIN

/\* Initialize real-time signals. \*/

init\_rtsigs ();

#endif

/\* Setup signal handlers for the initial thread.

Since signal handlers are shared between threads, these settings

will be inherited by all other threads. \*/

// 设置initial thread信号处理函数 其他线程都将继承

#ifndef \_\_i386\_\_

sa.sa\_handler = pthread\_handle\_sigrestart;

#else

sa.sa\_handler = (\_\_sighandler\_t) pthread\_handle\_sigrestart;

#endif

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = 0;

\_\_sigaction(\_\_pthread\_sig\_restart, &sa, NULL);

#ifndef \_\_i386\_\_

sa.sa\_handler = pthread\_handle\_sigcancel;

#else

sa.sa\_handler = (\_\_sighandler\_t) pthread\_handle\_sigcancel;

#endif

sa.sa\_flags = 0;

\_\_sigaction(\_\_pthread\_sig\_cancel, &sa, NULL);

if (\_\_pthread\_sig\_debug > 0) {

sa.sa\_handler = pthread\_handle\_sigdebug;

sigemptyset(&sa.sa\_mask);

sa.sa\_flags = 0;

\_\_sigaction(\_\_pthread\_sig\_debug, &sa, NULL);

}

/\* Initially, block \_\_pthread\_sig\_restart. Will be unblocked on demand. \*/

sigemptyset(&mask);

sigaddset(&mask, \_\_pthread\_sig\_restart);

sigprocmask(SIG\_BLOCK, &mask, NULL);

/\* Register an exit function to kill all other threads. \*/

/\* Do it early so that user-registered atexit functions are called

before pthread\_exit\_process. \*/

\_\_on\_exit(pthread\_exit\_process, NULL);

}

// 下面是\_\_pthread\_manager\_thread 的初始化

/\* Descriptor of the manager thread; none of this is used but the error

variables, the p\_pid and p\_priority fields,

and the address for identification. \*/

struct \_pthread\_descr\_struct \_\_pthread\_manager\_thread = {

NULL, /\* pthread\_descr p\_nextlive \*/ // 这两个值为空！

NULL, /\* pthread\_descr p\_prevlive \*/

NULL, /\* pthread\_descr p\_nextwaiting \*/

NULL, /\* pthread\_descr p\_nextlock \*/

0, /\* int p\_tid \*/

0, /\* int p\_pid \*/

0, /\* int p\_priority \*/

&\_\_pthread\_handles[1].h\_lock, /\* struct \_pthread\_fastlock \* p\_lock \*/

0, /\* int p\_signal \*/

NULL, /\* sigjmp\_buf \* p\_signal\_buf \*/

NULL, /\* sigjmp\_buf \* p\_cancel\_buf \*/

0, /\* char p\_terminated \*/

0, /\* char p\_detached \*/

0, /\* char p\_exited \*/

NULL, /\* void \* p\_retval \*/

0, /\* int p\_retval \*/

NULL, /\* pthread\_descr p\_joining \*/

NULL, /\* struct \_pthread\_cleanup\_buffer \* p\_cleanup \*/

0, /\* char p\_cancelstate \*/

0, /\* char p\_canceltype \*/

0, /\* char p\_canceled \*/

&\_\_pthread\_manager\_thread.p\_errno, /\* int \*p\_errnop \*/

0, /\* int p\_errno \*/

NULL, /\* int \*p\_h\_errnop \*/

0, /\* int p\_h\_errno \*/

NULL, /\* char \* p\_in\_sighandler \*/

0, /\* char p\_sigwaiting \*/

PTHREAD\_START\_ARGS\_INITIALIZER, /\* struct pthread\_start\_args p\_start\_args \*/

{NULL}, /\* void \*\* p\_specific[PTHREAD\_KEY\_1STLEVEL\_SIZE] \*/

{NULL}, /\* void \* p\_libc\_specific[\_LIBC\_TSD\_KEY\_N] \*/

0, /\* int p\_userstack \*/

NULL, /\* void \* p\_guardaddr \*/

0, /\* size\_t p\_guardsize \*/

&\_\_pthread\_manager\_thread, /\* pthread\_descr p\_self \*/

1, /\* Always index 1 \*/

0, /\* int p\_report\_events \*/

{{{0, }}, 0, NULL}, /\* td\_eventbuf\_t p\_eventbuf \*/

ATOMIC\_INITIALIZER, /\* struct pthread\_atomic p\_resume\_count \*/

0, /\* char p\_woken\_by\_cancel \*/

NULL /\* struct pthread\_extricate\_if \*p\_extricate \*/

};

int \_\_pthread\_initialize\_manager(void)

{

// 管理线程与其他线程之间的通信是通过管道完成的。

/\*在一个进程空间内，管理线程与其他线程之间通过一对"管理管道（manager\_pipe[2]）"来通讯，

该管道在创建管理线程之前创建，在成功启动了管理线程之后，管理管道的读端和写端分别赋给两个全局变量

\_\_pthread\_manager\_reader和\_\_pthread\_manager\_request，之后，每个用户线程都通过

\_\_pthread\_manager\_request向管理线程发请求，但管理线程本身并没有直接使用\_\_pthread\_manager\_reader，

管道的读端（manager\_pipe[0]）是作为\_\_clone()的参数之一传给管理线程的，

管理线程的工作主要就是监听管道读端，并对从中取出的请求作出反应。\*/

int manager\_pipe[2];

int pid;

struct pthread\_request request;

/\* 如果基本的初始化未进行，那么进行初始化\*/

if (\_\_pthread\_initial\_thread\_bos == NULL) pthread\_initialize();

/\* 设置stack \*/

\_\_pthread\_manager\_thread\_bos = malloc(THREAD\_MANAGER\_STACK\_SIZE);

if (\_\_pthread\_manager\_thread\_bos == NULL) return -1;

// top值

\_\_pthread\_manager\_thread\_tos =

\_\_pthread\_manager\_thread\_bos + THREAD\_MANAGER\_STACK\_SIZE;

/\* 设置通信管道 \*/

if (pipe(manager\_pipe) == -1) {

free(\_\_pthread\_manager\_thread\_bos);

return -1;

}

/\* 在创建线程时 pid一定不会为0 \*/

pid = 0;

if (\_\_pthread\_initial\_thread.p\_report\_events)

{

/\* It's a bit more complicated. We have to report the creation of

the manager thread. \*/

int idx = \_\_td\_eventword (TD\_CREATE);

uint32\_t mask = \_\_td\_eventmask (TD\_CREATE);

if ((mask & (\_\_pthread\_threads\_events.event\_bits[idx]

| \_\_pthread\_initial\_thread.p\_eventbuf.eventmask.event\_bits[idx]))

!= 0)

{

// sys\_fork：创建 pthread\_manager线程 ：共享VM FS FILES等

pid = \_\_clone(\_\_pthread\_manager\_event,

(void \*\*) \_\_pthread\_manager\_thread\_tos,

CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND,

(void \*)(long)manager\_pipe[0]); // 读管道

if (pid != -1)

{

/\* Now fill in the information about the new thread in

the newly created thread's data structure. We cannot let

the new thread do this since we don't know whether it was

already scheduled when we send the event. \*/

// 对新线程添加属性

\_\_pthread\_manager\_thread.p\_eventbuf.eventdata =

&\_\_pthread\_manager\_thread;

\_\_pthread\_manager\_thread.p\_eventbuf.eventnum = TD\_CREATE;

\_\_pthread\_last\_event = &\_\_pthread\_manager\_thread;

// \_pthread\_manager\_thread的是2\*PTHREAD\_THREADS\_MAX+1

\_\_pthread\_manager\_thread.p\_tid = 2\* PTHREAD\_THREADS\_MAX + 1;

\_\_pthread\_manager\_thread.p\_pid = pid; // clone进程pid

/\* Now call the function which signals the event. \*/

\_\_linuxthreads\_create\_event ();

/\* Now restart the thread. \*/

\_\_pthread\_unlock(\_\_pthread\_manager\_thread.p\_lock);

}

}

}

// 如果\_\_pthread\_initial\_thread.p\_report\_events标记了，自然不再创建新线程了

if (pid == 0)

pid = \_\_clone(\_\_pthread\_manager, (void \*\*) \_\_pthread\_manager\_thread\_tos,

CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND,

(void \*)(long)manager\_pipe[0]);

if (pid == -1) {

free(\_\_pthread\_manager\_thread\_bos);

\_\_libc\_close(manager\_pipe[0]);

\_\_libc\_close(manager\_pipe[1]);

return -1;

}

// \_\_pthread\_manager\_request fd = manager\_pipe[1]; 交给其他线程写请求

\_\_pthread\_manager\_request = manager\_pipe[1]; /\* 全局变量writing end \*/

\_\_pthread\_manager\_reader = manager\_pipe[0]; /\* 全局变量reading end \*/

\_\_pthread\_manager\_thread.p\_tid = 2\* PTHREAD\_THREADS\_MAX + 1;

\_\_pthread\_manager\_thread.p\_pid = pid;

/\* Make gdb aware of new thread manager \*/

if (\_\_pthread\_threads\_debug && \_\_pthread\_sig\_debug > 0)

{

raise(\_\_pthread\_sig\_debug);

/\* We suspend ourself and gdb will wake us up when it is

ready to handle us. \*/

\_\_pthread\_wait\_for\_restart\_signal(thread\_self());

}

/\* Synchronize debugging of the thread manager \*/

request.req\_kind = REQ\_DEBUG;

\_\_libc\_write(\_\_pthread\_manager\_request, (char \*) &request, sizeof(request));

return 0;

}

至此，\_\_pthread\_manager\_thread创建成功。

下面看看它是怎样管理的：\_\_pthread\_manager函数

/\* The server thread managing requests for thread creation and termination \*/

int \_\_pthread\_manager(void \*arg)

{

int reqfd = (int) (long int) arg; // 管道的读口fd

struct pollfd ufd;

sigset\_t mask;

int n;

struct pthread\_request request;

/\* If we have special thread\_self processing, initialize it. \*/

#ifdef INIT\_THREAD\_SELF

INIT\_THREAD\_SELF(&\_\_pthread\_manager\_thread, 1);

#endif

/\* Set the error variable. \*/

\_\_pthread\_manager\_thread.p\_errnop = &\_\_pthread\_manager\_thread.p\_errno;

\_\_pthread\_manager\_thread.p\_h\_errnop = &\_\_pthread\_manager\_thread.p\_h\_errno;

/\* Block all signals except \_\_pthread\_sig\_cancel and SIGTRAP \*/

sigfillset(&mask);

sigdelset(&mask, \_\_pthread\_sig\_cancel); /\* for thread termination \*/

sigdelset(&mask, SIGTRAP); /\* for debugging purposes \*/

sigprocmask(SIG\_SETMASK, &mask, NULL);

/\* 提高优先级 等于主线程的优先级 \*/

\_\_pthread\_manager\_adjust\_prio(\_\_pthread\_main\_thread->p\_priority);

/\* Synchronize debugging of the thread manager \*/

// 从管道中读取请求

n = \_\_libc\_read(reqfd, (char \*)&request, sizeof(request));

ASSERT(n == sizeof(request) && request.req\_kind == REQ\_DEBUG);

ufd.fd = reqfd;

ufd.events = POLLIN;

/\* Enter server loop \*/

while(1) {

n = \_\_poll(&ufd, 1, 2000); // 2秒超时

/\* main thread的终止 将导致所有thread终止 \*/

if (getppid() == 1) {

pthread\_kill\_all\_threads(SIGKILL, 0);

\_exit(0);

}

/\* Check for dead children \*/

if (terminated\_children) {

terminated\_children = 0;

pthread\_reap\_children();

}

/\* 读取请求 \*/

if (n == 1 && (ufd.revents & POLLIN)) {

n = \_\_libc\_read(reqfd, (char \*)&request, sizeof(request));

ASSERT(n == sizeof(request));

switch(request.req\_kind) {

case REQ\_CREATE: //创建

request.req\_thread->p\_retcode =

pthread\_handle\_create((pthread\_t \*) &request.req\_thread->p\_retval,

request.req\_args.create.attr,

request.req\_args.create.fn,

request.req\_args.create.arg,

&request.req\_args.create.mask,

request.req\_thread->p\_pid,

request.req\_thread->p\_report\_events,

&request.req\_thread->p\_eventbuf.eventmask);

restart(request.req\_thread);

break;

case REQ\_FREE:

pthread\_handle\_free(request.req\_args.free.thread\_id);

break;

case REQ\_PROCESS\_EXIT:

pthread\_handle\_exit(request.req\_thread,

request.req\_args.exit.code);

break;

case REQ\_MAIN\_THREAD\_EXIT:

main\_thread\_exiting = 1;

if (\_\_pthread\_main\_thread->p\_nextlive == \_\_pthread\_main\_thread) {

restart(\_\_pthread\_main\_thread);

return 0;

}

break;

case REQ\_POST:

\_\_new\_sem\_post(request.req\_args.post);

break;

case REQ\_DEBUG:

/\* Make gdb aware of new thread and gdb will restart the

new thread when it is ready to handle the new thread. \*/

if (\_\_pthread\_threads\_debug && \_\_pthread\_sig\_debug > 0)

raise(\_\_pthread\_sig\_debug);

break;

}

}

}

}

下面我们看看新的线程是怎样创建的：

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

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

sigset\_t \* mask, int father\_pid,

int report\_events,

td\_thr\_events\_t \*event\_maskp)

{

size\_t sseg;

int pid;

pthread\_descr new\_thread;

char \* new\_thread\_bottom;

pthread\_t new\_thread\_id;

char \*guardaddr = NULL;

size\_t guardsize = 0;

int pagesize = \_\_getpagesize();

/\* First check whether we have to change the policy and if yes, whether

we can do this. Normally this should be done by examining the

return value of the \_\_sched\_setscheduler call in pthread\_start\_thread

but this is hard to implement. FIXME \*/

if (attr != NULL && attr->\_\_schedpolicy != SCHED\_OTHER && geteuid () != 0)

return EPERM;

/\* Find a free segment for the thread, and allocate a stack if needed \*/

for (sseg = 2; ; sseg++)

{

if (sseg >= PTHREAD\_THREADS\_MAX)

return EAGAIN;

if (\_\_pthread\_handles[sseg].h\_descr != NULL)

continue;

if (pthread\_allocate\_stack(attr, thread\_segment(sseg), pagesize,

&new\_thread, &new\_thread\_bottom,

&guardaddr, &guardsize) == 0)

break;

}

\_\_pthread\_handles\_num++;

/\* Allocate new thread identifier \*/

pthread\_threads\_counter += PTHREAD\_THREADS\_MAX;

new\_thread\_id = sseg + pthread\_threads\_counter;

/\* Initialize the thread descriptor. Elements which have to be

initialized to zero already have this value. \*/

// 对新的线程进行属性赋值

new\_thread->p\_tid = new\_thread\_id; //tid=n\*PTHREAD\_THREADS\_MAX+n+1

new\_thread->p\_lock = &(\_\_pthread\_handles[sseg].h\_lock);

new\_thread->p\_cancelstate = PTHREAD\_CANCEL\_ENABLE;

new\_thread->p\_canceltype = PTHREAD\_CANCEL\_DEFERRED;

new\_thread->p\_errnop = &new\_thread->p\_errno;

new\_thread->p\_h\_errnop = &new\_thread->p\_h\_errno;

new\_thread->p\_guardaddr = guardaddr;

new\_thread->p\_guardsize = guardsize;

new\_thread->p\_self = new\_thread;

new\_thread->p\_nr = sseg;

/\* Initialize the thread handle \*/

\_\_pthread\_init\_lock(&\_\_pthread\_handles[sseg].h\_lock);

\_\_pthread\_handles[sseg].h\_descr = new\_thread;

\_\_pthread\_handles[sseg].h\_bottom = new\_thread\_bottom;

/\* Determine scheduling parameters for the thread \*/

new\_thread->p\_start\_args.schedpolicy = -1;

if (attr != NULL) {

new\_thread->p\_detached = attr->\_\_detachstate;

new\_thread->p\_userstack = attr->\_\_stackaddr\_set;

switch(attr->\_\_inheritsched) {

case PTHREAD\_EXPLICIT\_SCHED:

new\_thread->p\_start\_args.schedpolicy = attr->\_\_schedpolicy;

memcpy (&new\_thread->p\_start\_args.schedparam, &attr->\_\_schedparam,

sizeof (struct sched\_param));

break;

case PTHREAD\_INHERIT\_SCHED:

new\_thread->p\_start\_args.schedpolicy = \_\_sched\_getscheduler(father\_pid);

\_\_sched\_getparam(father\_pid, &new\_thread->p\_start\_args.schedparam);

break;

}

new\_thread->p\_priority =

new\_thread->p\_start\_args.schedparam.sched\_priority;

}

/\* Finish setting up arguments to pthread\_start\_thread \*/

// 新线程 函数 参数赋值

new\_thread->p\_start\_args.start\_routine = start\_routine;

new\_thread->p\_start\_args.arg = arg;

new\_thread->p\_start\_args.mask = \*mask;

/\* Raise priority of thread manager if needed \*/

\_\_pthread\_manager\_adjust\_prio(new\_thread->p\_priority);

/\* Do the cloning. We have to use two different functions depending

on whether we are debugging or not. \*/

pid = 0; /\* Note that the thread never can have PID zero. \*/

if (report\_events)

{

/\* See whether the TD\_CREATE event bit is set in any of the

masks. \*/

int idx = \_\_td\_eventword (TD\_CREATE);

uint32\_t mask = \_\_td\_eventmask (TD\_CREATE);

if ((mask & (\_\_pthread\_threads\_events.event\_bits[idx]

| event\_maskp->event\_bits[idx])) != 0)

{

/\* Lock the mutex the child will use now so that it will stop. \*/

\_\_pthread\_lock(new\_thread->p\_lock, NULL);

/\* We have to report this event. \*/

pid = \_\_clone(pthread\_start\_thread\_event, (void \*\*) new\_thread,

CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND |

\_\_pthread\_sig\_cancel, new\_thread);

if (pid != -1)

{

/\* Now fill in the information about the new thread in

the newly created thread's data structure. We cannot let

the new thread do this since we don't know whether it was

already scheduled when we send the event. \*/

new\_thread->p\_eventbuf.eventdata = new\_thread;

new\_thread->p\_eventbuf.eventnum = TD\_CREATE;

\_\_pthread\_last\_event = new\_thread;

/\* We have to set the PID here since the callback function

in the debug library will need it and we cannot guarantee

the child got scheduled before the debugger. \*/

new\_thread->p\_pid = pid;

/\* Now call the function which signals the event. \*/

\_\_linuxthreads\_create\_event ();

/\* Now restart the thread. \*/

\_\_pthread\_unlock(new\_thread->p\_lock);

}

}

}

if (pid == 0)

pid = \_\_clone(pthread\_start\_thread, (void \*\*) new\_thread,

CLONE\_VM | CLONE\_FS | CLONE\_FILES | CLONE\_SIGHAND |

\_\_pthread\_sig\_cancel, new\_thread);

/\* Check if cloning succeeded \*/

if (pid == -1) {

/\* Free the stack if we allocated it \*/

if (attr == NULL || !attr->\_\_stackaddr\_set)

{

if (new\_thread->p\_guardsize != 0)

munmap(new\_thread->p\_guardaddr, new\_thread->p\_guardsize);

munmap((caddr\_t)((char \*)(new\_thread+1) - INITIAL\_STACK\_SIZE),

INITIAL\_STACK\_SIZE);

}

\_\_pthread\_handles[sseg].h\_descr = NULL;

\_\_pthread\_handles[sseg].h\_bottom = NULL;

\_\_pthread\_handles\_num--;

return errno;

}

/\* 将线程添加到\_\_pthread\_main\_thread 双向链表中 由管理线程管理\*/

new\_thread->p\_prevlive = \_\_pthread\_main\_thread;

new\_thread->p\_nextlive = \_\_pthread\_main\_thread->p\_nextlive;

\_\_pthread\_main\_thread->p\_nextlive->p\_prevlive = new\_thread;

\_\_pthread\_main\_thread->p\_nextlive = new\_thread;

/\* Set pid field of the new thread, in case we get there before the

child starts. \*/

new\_thread->p\_pid = pid; // 记录了clone 返回的pid

/\* We're all set \*/

\*thread = new\_thread\_id;

return 0;

}