Lab4 挑战性任务答辩

崔怿恺 20373866 姜博老师班

2022 年 7 月 4 日

目录

目录

分析线程与进程的异同:

- 进程概念之下，但仍具有进程的诸多特征，仍可被视为进程
  $\rightarrow$ 在进程控制块添加字段

分析线程与进程的异同：

- 进程概念之下，但仍具有进程的诸多特征，仍可被视为进程 → 在进程控制块添加字段
- 不同的：envid，寄存器现场（包括栈位置 → 运行时恢复自己的现场

分析线程与进程的异同：

- 进程概念之下，但仍具有进程的诸多特征，仍可被视为进程 → 在进程控制块添加字段
- 不同的：envid，寄存器现场（包括栈位置 → 运行时恢复自己的现场
- 相同的：地址空间 → 运行时切换父进程的地址空间

线程控制块：env结构体添加字段

字段名	字段含义
`u_int thread_is_thread`	线程标记位，0或父进程id
`u_int thread_join_envid`	正在等待此线程结束的进程id
`void **thread_join_ptr`	指向用于接收返回值的指针
`u_int thread_is_canceled`	是否接收到来自其他线程的撤销信号

线程控制块：其他函数的适应性改造

子线程被创建时

- 使用 `thread_is_thread` 字段维护父进程
- 维护自己的寄存器值

子线程被执行时

需要对`env_run()`函数动手脚：

- 通过 `thread_is_thread` 字段（循环地）找到父进程，切换至父进程的地址空间
- 还原自己保存的寄存器现场

Show me the code

```
void env_run(struct Env *e) {
    if (curenv) {
        struct Trapframe *old;
        old = (struct Trapframe *)(TIMESTACK - sizeof(struct Trapframe));
        bcopy((void *)old, (void *)(&curenv->env_tf), sizeof(struct Trapframe));
        curenv->env_tf.pc = curenv->env_tf.cp0_epc;
    }

    curenv = e;

    if (e->thread_is_threa  > 0) {
        strdct Env *father;
        do { envid2env(e->thread_is_threa , &father, 0); }
        while (father->thrdad_is_threa  > 0);
        lcontext(fatherd->env_pgdir);
        env_pop_tf(&(e->env_tf), GET_ENV_ASID(father->env_id));
    }
    else {
        lcontext(e->env_pgdir);
        env_pop_tf(&(e->env_tf), GET_ENV_ASID(e->env_id));
    }
}
```

创建

```
int pthread_create(pthread_t *thread, const void *attr, void
*(*start_routine)(void *), void *arg);
```

线程的创建与fork操作大致类似但细节上有差异，具体体现在子进程（子线程）的处理、关于页表的处理。总体而言线程因为不需要拷贝页表，创建步骤比fork轻量化，基本符合对于线程的描述。

线程创建

- 必要的合法性检查
- 分配一个进程控制块[1]
- 父进程：标记为线程，修改栈顶寄存器[2]，设置为 RUNNABLE
- 新线程：执行 `start_routine(arg)` 后退出

[1]此时继承了父进程的寄存器现场
[2]参考后文“全用户地址空间共享”的实现方式

阻塞至结束（join）

```
int pthread_join(pthread_t thread, void **__thread_return);
```

与Java相比，POSIX标准下的阻塞至结束还附带了消息传递的功能，通过retval来在线程结束时传递想要传递的消息。阻塞需要与退出相互配合。其主要的步骤如下：

阻塞至目标线程结束（系统调用完成）

- 必要的合法性检查
- 如果目标已经结束，带走（并抹除）他的遗言[3]
- 如果目标还未结束，设置其 env 结构体相应字段，阻塞此线程等候被唤醒

[3]线程结束后将自己的返回指针放在控制块的 `thread_join_ptr` 字段，如果在此之后被join，立即带走此遗言

线程退出

```
void pthread_exit(void *retval);
```

线程退出（系统调用完成）

- 必要的合法性检查
- 如果有正在等待自己的线程，为其join_ptr赋值后唤醒它
- 如果没有，留下遗言
- 最后销毁此进程，因为没有页表，所以也不会对其他父母兄弟进程造成影响，最终沉默而壮烈地结束他的一生

线程撤销

```
int pthread_cancel(pthread_t th);
```

线程的撤销可以不以自己作为目标。但是处于安全性、现实性、实现难度考虑，POSIX标准下的撤销不以自己作为目标时只会为目标线程给出信号，目标线程使用testcancel来创建撤销点，主动地安排检查是否被撤销的时间。换句话说，线程哪怕永远都不调用testcancel也完全没问题，这样任何非自身的撤销都不会有效。

线程撤销（系统调用完成）

- 必要的合法性检查
- 如果目标是自身，则可以调用pthread_exit(PTHREAD_CANCELED[4]);
- 如果目标是别的线程，为它的线程控制块相应字段留下记号

[4]POSIX标准指出，该变量字面值为-1

线程检验是否被撤销

`void pthread_testcancel();`

检验是否被撤销

- 必要的合法性检查
- 查看自身控制块相应位[5]
- 若被撤销，调用`pthread_exit(PTHREAD_CANCELED);`，否则不做任何事

[5]这里仅读取是否被撤销，不需要原子性保护。

Linux下的行为

代码演示：Ubuntu 20.04 x64 on Windows Subsystem for Linux

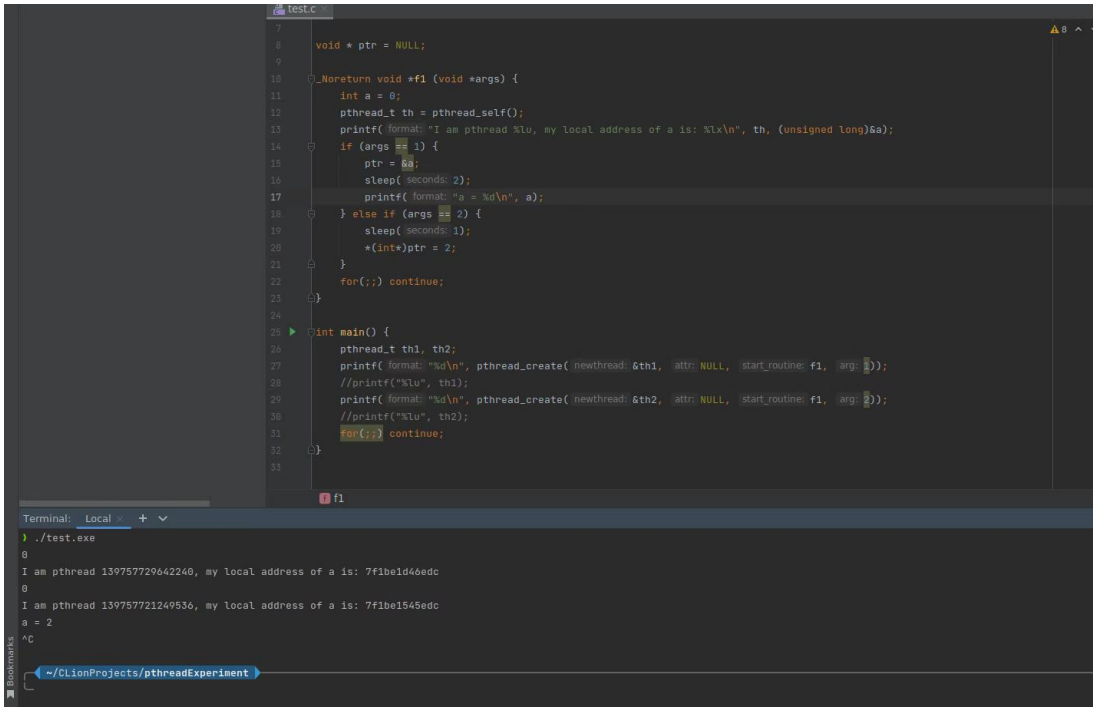

图: linux下各线程有不同栈，栈之间可以互相读写

Linux下的行为

能够得到的结论

两线程的栈顶地址不同 区别于进程栈，系统为每个进程分配了一段栈空间，每个线程对应了一个栈顶地址，并自己的栈顶开始生长。

两线程可以互相访问彼此的内部变量 两线程在同一个地址空间。可以通过万能的指针彼此读写。

线程栈空间的大小为 0x800000 Byte = 8M 这个栈大小基本够用（了吧[6]

[6]查阅文献知，如无特殊设置，线程栈大小默认为8M，可以用ulimit -a指令查询

MOS下的改造

由于POSIX标准并没有对内存空间给出任何的要求与限制，所以为了方便实现，以下对必要的数据进行自行约定，值得指出这样的约定可能会产生一些缺陷，但也是没有办法的办法了。

线程栈放在哪里？ 0x7f000000

线程栈大小设置为多少？ 0x01000000B = 16M，（凑个整，另外整个MOS都只有64MB内存，16M肯定够用了）

这样做当然会存在问题，例如用户态程序在 0x70000000 甚至以上都有文本和数据，那么就会和栈空间撞车。这个问题一方面要触发几乎不可能，64M内存使得排不到那么高的地址空间，用户也没有理由非要指定使用那段地址；另一方面Linux可以避免这样的问题的原因是使用了段页式内存管理，避免了在栈空间对应的va安排程序段。

MOS下的改造

如何追踪进程的线程数？

在pthread.c–pthread_create()中维护静态变量c，用于计算当前创建的线程应该对应哪一个栈顶。

```
syscall_mark_as_thread(newenvid, UTSTACKTOP - howmany
_thread * UTSTACKSIZE - BY2PG);
```

思考

一方面，POSIX允许内部变量跨线程的互相访问，另一方面，完成这样的操作的方式并不优雅。
此外，安全性方面内部变量随时都有可能被更改这非常不安全；数据交换方面，共享全局变量显然更加可行。不禁让人想问这样要求的意义何在。

目录

sem_t的定义

- 有效位（是否经过初始化，正在工作）
- 剩余量
- 正在等待此信号量的线程队列

sem_t应该出现在哪里？

在哪里？由谁管理？

- 当在Linux实验环境下为sem_init传递NULL的指针时会报错 → sem_t变量应当在用户空间创建，对应用户空间的一段地址。
- 信号量相关函数均需要一个指向sem_t的指针作为参数 → 信号量由操作系统和库函数管理。

库函数和操作系统利用系统调用的原子性，通过维护无名信号量对应的数据结构中的相应字段，实现信号量的相关操作。

初始化和销毁

```
int sem_init(sem_t *__sem, int __pshared, unsigned int __value);
int sem_destroy(sem_t *__sem);
```

信号量的初始化

- 初始化链表
- 设置余量
- 将有效位置1

信号量的销毁

- 将有效为置0[7]

[7]同时需要对还在等待此信号量的线程特殊处理（返回等待失败的结果）

信号量的等待

```
int sem_wait(sem_t *__sem);
```

PV操作可以参考Lab3的一次Extra的实现方式，在这里就只讲讲思路了

信号量的等待（系统调用完成）

- 信号量需要有效
- 当还有余量时，减一退出，完成P操作
- 没有余量时，阻塞自身，加入等待队列，等待随时被唤醒
- 唤醒后完成P操作

信号量的尝试等待

```
int sem_trywait(sem_t *__sem);
```

尝试等待在有余量的时候行为同普通的P操作，但没有余量的时候不会阻塞自身。

信号量的等待（系统调用完成）

- 信号量需要有效
- 当还有余量时，减一退出，完成P操作
- 没有余量时，返回非0值

信号量的增加

```
int sem_post(sem_t *__sem);
```

也可以参考Lab3的一次Extra

信号量的V操作（系统调用完成）

- 信号量需要有效
- 信号量等待队列不为空时，选择最先进来的一个唤醒
- 信号量等待队列为空时，增加其余量

Linux下的sem_t

Linux下的标准库中对sem_t结构体做出如下的定义。

semaphore.h

```
typedef union {
    char __size[__SIZEOF_SEM_T];
    long int __align;
} sem_t;
```

只保证了大小，但内存布局对用户不透明。Linux希望即使变量在用户空间，对sem_t的维护也仅由sem_xx函数来完成。
这样的用法还有很多，如pthread_attr_t

Linux下的sem_t

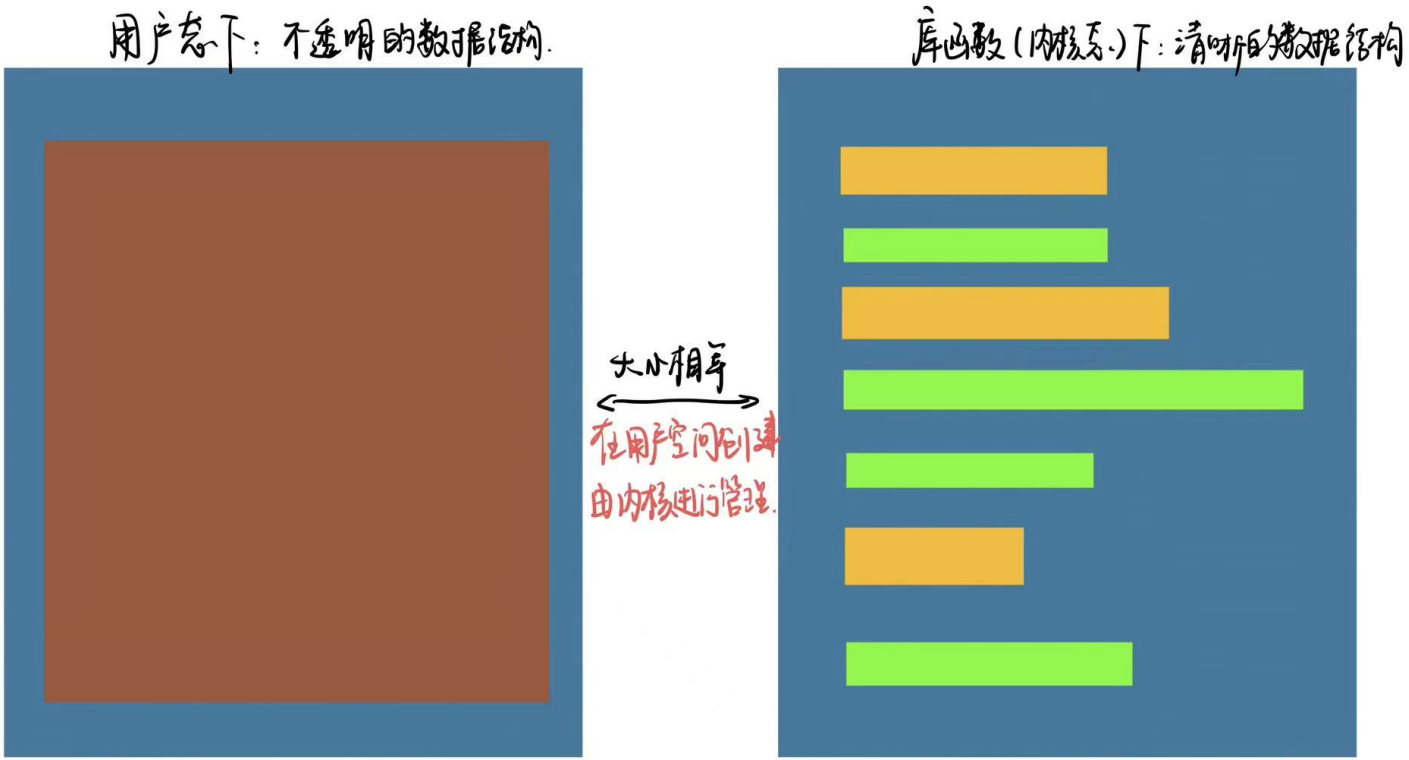

图: 由操作系统接管对于用户进程空间的一些关键变量的管理

目录

线程基本测试

threadtest.c

- 能不能顺利地创建线程
- 能不能顺利地继承父进程的文本
- 能不能顺利地执行指定的函数并返回
- 能不能分配不同的栈空间

result

```
pthread c01, local addr of a = 7effeff0
pthread 1402, local addr of a = 7dffeff0
```

线程退出（和join）测试

threadexittest.c

- 能不能顺利地退出线程
- 能不能顺利地通过retval传递信息
- 能不能通过join得到这个返回的信息

result

Exit Message

线程撤销

threadcanceltest.c

- 能不能顺利撤销自己
- 能不能为别的线程加上撤销信号
- 能不能通过testcancel检查自己是否被撤销

result

thread1 running → thread3 joins thread1 → thread2 kills thread1 → thread1 testcancel and exit → wake up thread2

信号量测试

sem.c

- 能不能顺利使用信号量的相关调用解决实际的并行问题

result

producer put 0 → customer take 0 → producer put 1 → customer take 1

...

基本符合要求

目录

谢谢!

感谢老师助教批评指正

崔怿恺 20373866 姜博老师班