线程机制实习报告

姓名：李聪 学号：1800012826

日期：2020/10/24

目录

内容一：任务完成情况

任务完成列表（Y/N）

	Exercise1	Exercise2	Exercise3	Exercise4
第一部分	Y	Y	Y	Y

具体 Exercise 完成情况

Exercise1

（一）调研 Linux 中 PCB 的结构

在这部分练习中，我调研了 Linux 的 PCB 实现方式。下面让我对它进行简要的分析。

PCB 是进程描述符（进程控制块），它是操作系统专门用来表示进程的数据结构，其中记录了进程的各种属性，描述了进程在运行中的动态变化过程。操作系统通过 PCB 控制进程，PCB 是系统感知进程的唯一标志。在 Linux 中，PCB 是 task_struct 结构体。

task_struct 是 Linux 内核的一种数据结构，它被装载到 RAM 中，包含进程的各种信息，如：进程标志符（PID）、进程状态、进程优先级、进程当前的 PC、内存指针、上下文数据、I/O 信息、CPU 占用时间和时钟数等记账统计信息。所有运行中的进程对应的 task_struct 以链表的形式存于内核中，构成进程表。这个结构体位于 linux 源代码中的./include/linux/sched.h 中。下面让我们来分析一下其中的主要内容。（以 linux2.6.x 版本的内容为例）

1.进程状态

```
volatile long state;
int exit_state;
```

其中，state 的可能取值如下：

```
#define TASK_RUNNING        0
#define TASK_INTERRUPTIBLE  1
#define TASK_UNINTERRUPTIBLE    2
#define __TASK_STOPPED      4
#define __TASK_TRACED       8
/* in tsk->exit_state */
#define EXIT_ZOMBIE     16
#define EXIT_DEAD       32
/* in tsk->state again */
#define TASK_DEAD       64
#define TASK_WAKEKILL       128
#define TASK_WAKING     256
```

系统中的每个进程都处于上述状态中的一种。每种状态的含义如下：

TASK_RUNNING 表示进程要么正在执行，要么正准备执行。

TASK_INTERRUPTIBLE 表示进程被阻塞（睡眠），直到某个条件变为真，这时会把状态转换为 TASK_RUNNING。

TASK_UNINTERRUPTIBLE 类似于 TASK_INTERRUPTIBLE，但处于这个状态下的进

程不可被信号唤醒。

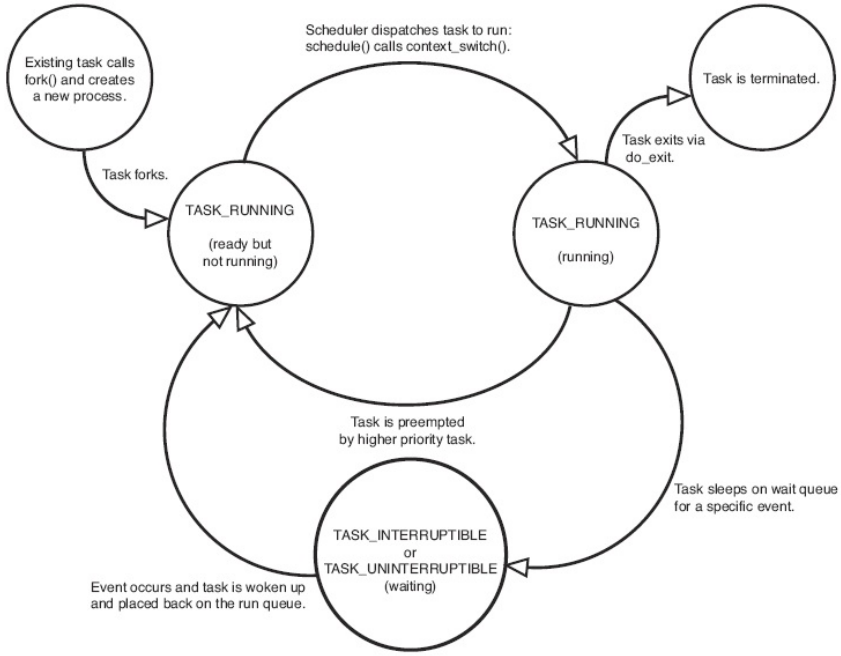
__TASK_STOPPED 表示进程被停止执行。

__TASK_TRACED 表示进程被 debugger 等进程监视。

EXIT_ZOMBIE 表示进程的执行被终止，但父进程还没有使用 wait()等系统调用获知它的终止信息。

EXIT_DEAD 表示进程执行后最终的状态。

最后两个 EXIT 类状态也可以存放于 exit_state 成员中。进程的切换大致如下图所示：



2.进程 PID

```
pid_t pid;
pid_t tgid;
```

当内核中的 CONFIG_BASE_SMALL 值为 0 时，PID 可取范围为 0 到 32767（最多 32768 个进程），如下图所示。

```
/* linux-2.6.38.8/include/linux/threads.h */
#define PID_MAX_DEFAULT (CONFIG_BASE_SMALL ? 0x1000 : 0x8000)
```

一个线程组的所有线程使用和这个线程组中的第一个线程相同的 PID，且这个值存放于 tgid 成员中，表示线程组的 id。实际上，getpid()函数返回的是当前进程的 tgid 值，我不是 pid 值

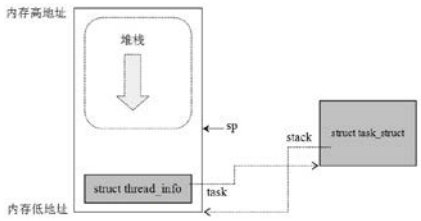
3.进程内核栈

```
void *stack;
```

这个指针指向声明内核栈信息的数据结构。进程通过 alloc_thread_info 函数分配内核栈，用 free_thread_info 函数释放分配的内核栈，这两个函数定义于 kernel/fork.c 中。进程的内核栈用 thread_union 来表示。期中的 THREAD_SIZE 宏值为 8192。

```
union thread_union {
    struct thread_info thread_info;
    unsigned long stack[THREAD_SIZE/sizeof(long)];
};
```

task_struct 与 struct_union（srtuct_info）以及内核栈的关系如下：



4.标记

```
unsigned int flags;
```

这一变量的一些可能取值如下：

```
#define PF_KSOFTIRQD    0x00000001  /* I am ksoftirqd */
#define PF_STARTING 0x00000002  /* being created */
#define PF_EXITING  0x00000004  /* getting shut down */
#define PF_EXITPIDONE   0x00000008  /* pi exit done on shut down */
#define PF_VCPU     0x00000010  /* I'm a virtual CPU */
#define PF_WQ_WORKER    0x00000020  /* I'm a workqueue worker */
#define PF_FORKNOEXEC   0x00000040  /* forked but didn't exec */
#define PF_MCE_PROCESS  0x00000080      /* process policy on mce errors */
#define PF_SUPERPRIV    0x00000100  /* used super-user privileges */
#define PF_DUMPCORE 0x00000200  /* dumped core */
#define PF_SIGNALED 0x00000400  /* killed by a signal */
#define PF_MEMALLOC 0x00000800  /* Allocating memory */
#define PF_USED_MATH    0x00002000  /* if unset the fpu must be initialized before use */
#define PF_FREEZING 0x00004000  /* freeze in progress. do not account to load */
#define PF_NOFREEZE 0x00008000  /* this thread should not be frozen */
#define PF_FROZEN   0x00010000  /* frozen for system suspend */
#define PF_FSTRANS  0x00020000  /* inside a filesystem transaction */
#define PF_KSWAPD   0x00040000  /* I am kswapd */
#define PF_OOM_ORIGIN   0x00080000  /* Allocating much memory to others */
#define PF_LESS_THROTTLE 0x00100000 /* Throttle me less: I clean memory */
#define PF_KTHREAD  0x00200000  /* I am a kernel thread */
#define PF_RANDOMIZE    0x00400000  /* randomize virtual address space */
#define PF_SWAPWRITE    0x00800000  /* Allowed to write to swap */
#define PF_SPREAD_PAGE  0x01000000  /* Spread page cache over cpuset */
#define PF_SPREAD_SLAB  0x02000000  /* Spread some slab caches over cpuset */
#define PF_THREAD_BOUND 0x04000000  /* Thread bound to specific cpu */
#define PF_MCE_EARLY    0x08000000      /* Early kill for mce process policy */
#define PF_MEMPOLICY    0x10000000  /* Non-default NUMA mempolicy */
#define PF_MUTEX_TESTER 0x20000000  /* Thread belongs to the rt mutex tester */
#define PF_FREEZER_SKIP 0x40000000  /* Freezer should not count it as freezable */
#define PF_FREEZER_NOSIG 0x80000000 /* Freezer won't send signals to it */
```

5.表示进程亲属关系的成员：

```
struct task_struct *real_parent; /* real parent process */
struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports */
struct list_head children;  /* list of my children */
struct list_head sibling;   /* linkage in my parent's children list */
struct task_struct *group_leader;   /* threadgroup leader */
```

real_parent 指向父进程。如果创建它的父进程不再存在，指向 PID 为 1 的 init 进程。

parent 指向父进程。当前进程终止时，必须向父进程发信号，这个值通常与 real_parent 相同。

children 表示子进程的链表，指向这个链表头部。

sibling 用于把当前进程插入到父进程的子进程列表中（兄弟进程列表）

group_leader 指向当前进程组所在进程的领头进程。

6.ptrace 系统调用

```
    unsigned int ptrace;
    struct list_head ptraced;
    struct list_head ptrace_entry;
    unsigned long ptrace_message;
    siginfo_t *last_siginfo; /* For ptrace use.  */
#ifdef CONFIG_HAVE_HW_BREAKPOINT
    atomic_t ptrace_bp_refcnt;
#endif
```

ptrace 值为 0 是表示不需要被跟踪。它还有以下可能的取值：

```
/* linux-2.6.38.8/include/linux/ptrace.h */
#define PT_PTRACED  0x00000001
#define PT_DTRACE   0x00000002  /* delayed trace (used on m68k, i386) */
#define PT_TRACESYSGOOD 0x00000004
#define PT_PTRACE_CAP   0x00000008  /* ptracer can follow suid-exec */
#define PT_TRACE_FORK   0x00000010
#define PT_TRACE_VFORK  0x00000020
#define PT_TRACE_CLONE  0x00000040
#define PT_TRACE_EXEC   0x00000080
#define PT_TRACE_VFORK_DONE 0x00000100
#define PT_TRACE_EXIT   0x00000200
```

7.Performance Event

```
#ifdef CONFIG_PERF_EVENTS
    struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
    struct mutex perf_event_mutex;
    struct list_head perf_event_list;
#endif
```

Performance Event 是一个性能诊断工具，用于分析进程的性能问题。

8.进程调度相关信息

```
int prio, static_prio, normal_prio;
unsigned int rt_priority;
const struct sched_class *sched_class;
struct sched_entity se;
struct sched_rt_entity rt;
unsigned int policy;
cpumask_t cpus_allowed;
```

调度器考虑的优先级保存于 prio，但某些情况下内核需要暂时提高进程优先级，因此还需要除了 static_prio 和 normal_prio 以外的变量来处理。由于这些改变不是持久的，故另外两个变量值不受影响。

static_prio 保存进程的静态优先级。它是进程启动时分配得到的优先级，可以用 nice，sched_setscheduler 系统调用修改，否则运行期间会保持恒定。

normal_prio 表示基于静态优先级和调度策略计算出的优先级。因此，即使静态优先级相同，普通进程和实时进程的 normal_prio 也会不同。调用 fork 函数时，子进程会继承这一优先级。

rt_priority 表示实时进程的优先级。即使是最低优先级的实时进程，其优先级也高于普通进程。

sched_class 表示该进程所属的调度类。大致可分为以下四种：

```
/* linux-2.6.38.8/kernel/sched_fair.c */
static const struct sched_class fair_sched_class;
/* linux-2.6.38.8/kernel/sched_rt.c */
static const struct sched_class rt_sched_class;
/* linux-2.6.38.8/kernel/sched_idletask.c */
static const struct sched_class idle_sched_class;
/* linux-2.6.38.8/kernel/sched_stoptask.c */
static const struct sched_class stop_sched_class;
```

se 是普通进程的调用实体。为了让调度器可以实现“组调度”，因此需要调度器处理“可调度实体”。sched_entity 作为普通进程实体标识，储存在 PCB 结构中。

类似地，rt 是实时进程的实体标识。

policy 表示调度策略，在这一版本的实现中有以下五种：

```
#define SCHED_NORMAL		0
#define SCHED_FIFO		1
#define SCHED_RR		2
#define SCHED_BATCH		3
/* SCHED_ISO: reserved but not implemented yet */
#define SCHED_IDLE		5
```

cpus_allowed 则用于控制进程可以在哪个处理器上运行。

9.进程地址空间

```
    struct mm_struct *mm, *active_mm;
#ifdef CONFIG_COMPAT_BRK
    unsigned brk_randomized:1;
#endif
#if defined(SPLIT_RSS_COUNTING)
    struct task_rss_stat    rss_stat;
#endif
```

mm 指向进程拥有的内存描述符，active_mm 指向进程运行时使用的内存描述符。对于普通进程而言，二者指向相同的内容。但是，内核进程不拥有任何内存描述符，所以它们的 mm 成员总为 NULL。当内核线程得以运行时，它的 active_mm 成员被初始化为前一个运行进程的 active_mm 值。

brk_randomized 用于确定对随机堆内存的预测。

rss_stat 用来记录缓冲信息。

10.判断标志

```
int exit_code, exit_signal;
int pdeath_signal;  /*  The signal sent when the parent dies  */
/* ??? */
unsigned int personality;
unsigned did_exec:1;
unsigned in_execve:1;   /* Tell the LSMs that the process is doing an
            * execve */
unsigned in_iowait:1;


/* Revert to default priority/policy when forking */
unsigned sched_reset_on_fork:1;
```

exit_code 用于设置进程的终止代号，要么是_exit()或_exit_group()的系统调用参数（正常终止），要么是由内核提供的一个错误代号（异常终止）。

exit_signal 被置为-1 时表示是某个线程组中的一员。只有当线程组的最后一个成员终止时，才会产生一个信号，以通知线程组领头进程的父进程。

pdeath_signal 用于判断父进程终止时发送信号。

personality 用于处理不同的 ABI（运行程序接口）。

did_exec 用于记录进程代码是否被 execve()函数所执行。

in_execve 用于通知 LSM 是否被 do_execve()函数所调用。

in_iowait 用于判断是否进行 iowait 计数。

sched_reset_on_fork 用于判断是否恢复默认的优先级或调度策略。

11.时间

```
    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
#ifndef CONFIG_VIRT_CPU_ACCOUNTING
    cputime_t prev_utime, prev_stime;
#endif
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;         /* monotonic time */
    struct timespec real_start_time;    /* boot based time */
    struct task_cputime cputime_expires;
    struct list_head cpu_timers[3];
#ifdef CONFIG_DETECT_HUNG_TASK
/* hung task detection */
    unsigned long last_switch_count;
#endif
```

utime，stime 用于记录进程在用户态/内核态下所经过的节拍数（定时器）

prev_utime，prev_vtime 是先前的运行时间。

utimescaled，stimescaled 也用于记录进程在用户态/内核态的运行时间，但它们以处理器的频率为刻度。

nvcsw/nivcsw 是自愿（voluntary）/非自愿（involuntary）上下文切换计数。last_switch_count 是 nvcsw 和 nivcsw 的总和。

start_time 和 real_start_time 都是进程创建时间，real_start_time 还包含了进程睡眠时间。

cputime_expires 用来统计进程或进程组被跟踪的处理器时间，其中的三个成员对应着 cpu_timers[3]的三个链表。

12.信号

```
struct signal_struct *signal;
struct sighand_struct *sighand;

sigset_t blocked, real_blocked;
sigset_t saved_sigmask; /* restored if set_restore_sigmask() was used */
struct sigpending pending;

unsigned long sas_ss_sp;
size_t sas_ss_size;
int (*notifier)(void *priv);
void *notifier_data;
sigset_t *notifier_mask;
```

signal 指向进程的信号描述符。

sighand 指向进程的信号处理程序描述符。

blocked 表示被阻塞信号的掩码，real_blocked 表示临时掩码。

pending 存放私有挂起信号的数据结构。

sas_ss_sp 是信号处理程序备用堆栈的地址，sas_ss_size 表示堆栈的大小。

设备驱动程序常用 notifier 指向的函数来阻塞进程的某些信号（notifier_mask 是这些信号的位掩码），notifier_data 指的是 notifier 所指向的函数可能使用的数据。

除了以上的主要内容，PCB 中还包括了自旋锁、stack_canary、缺页统计、文件统计、死锁检测、I/O 计数、管道、socket 控制消息等内容。

（二）Nachos 中的线程结构

观察 nachos 的代码结构，我们可以发现，nachos 中只有线程的实现（对应于 thread 文件夹），没有真正意义上的进程。线程类被声明于 thread.h 中，其成员如下：

1.int* stackTop：指明当前的栈指针（栈顶）

2.void *machineState[MachineStateSize]：保存除了栈指针以外的其他所有寄存器

3.int* stack：指明栈的位置，值为栈底地址（若为主线程，则指针为空）

4.ThreadStatus status：线程状态，一共有四种：创建、就绪、运行、阻塞，对应于 JUST_CREATED，RUNNING。READY，BLOCKED

5.char* name：线程的名称

6.多个线程级别的处理函数：Fork()创建进程、Yield()让出 CPU、Sleep()让线程睡眠、Finish()终止执行、CheckOverflow()检查栈溢出、setStatus()设置线程状态、getName()获取线程名称、Print()输出名称到命令行上。StackAllocate()分配栈空间。

7.如果要考虑用户级程序的运行，还需要添加：space 地址空间地址；userRegisters 数组，记录所有用户级寄存器的状态；SaveUserState()保存用户级寄存器的状态；RestoreUserState()恢复用户级寄存器的状态。

Exercise2

这一部分要求阅读四份线程相关的源代码，接下来我将逐文件分析每份代码的内容和功能。

（一）code/threads/main.cc

这份代码是一份引导程序，用于初始化操作系统内核。主要函数是 main 函数，用于解析命令行输入。首先，main 函数进行初始化。初始化完毕后，main 分为四个部分：线程处理、用户程序处理、文件系统处理、网络处理。分别处理完毕后，程序调用 currentThread->Finish()，结束 main 函数，完成本次执行。

对于线程处理，如果命令行输入-q 参数，会把定义在 threadtest.cc 的 testnum 参数设为-q 后紧接的数字，否则，testnum 会设为 1。然后，main 调用 ThreadTest 函数，进行线程功能的测试。

然后，程序会输出 copyright。

如果需要进行用户程序的处理，若输入-x 参数，main 函数执行 StartProcess 函数，运行一个用户级程序；若输入-c 参数，会根据命令行参数的个数，分情况调用 ConsoleTest 函数进行控制台的测试。测试完成后，或者输入-s 参数，则会调用 interrupt->Halt()函数等待命令行输入命令。

对于文件系统测试，如果输入-cp 参数，则调用 Copy()函数，从 UNIX 系统向 Nachos 系统拷贝文件；如果输入-p 参数，则调用 Print()函数，打印一个 Nachos 文件到标准输出流；如果输入-r 参数，调用 fileSystem->Remove()函数，删除一个 Nachos 系统中的文件；如果输入-l 参数，调用 fileSystem->List()函数，输出文件列表；如果输入-D 参数，则打印整个文件系统中的内容；如果输入-t 参数，则调用 PerformanceTest()函数，执行性能测试。

对于网络处理，如果输入了-o 参数，首先延迟 2 秒，以启动另外一个 Nachos 系统，，然后调用 MailTest()函数，进行网络测试。

（二）codes/threads/threadtest.cc

这个文件中主要包含了一些测试函数，用于测试线程功能是否正常。其中的 ThreadTest() 函数在 codes/threads/main.cc 中的 main 函数被调用，作为主体的测试函数。

首先，全局变量 testnum 指明了测试时的线程个数，默认为 1，在 main.cc 的 main 函数中由命令行参数-q 指明。

接下来，代码中包含了三个函数：SimpleThread，ThreadTest1，ThreadTest。

ThreadTest 为测试的主体函数。首先判断测试用的进程数量。如果线程数为 1，函数执行 ThreadTest1()的测试函数，执行完毕后退出函数；如果线程数大于 1，输出”No test specified”，然后退出（多个线程的测试函数尚未实现）。

总地来说，前两行的代码用于测试我们的实现效果。main.cc 包含为主入口，threadtest.cc 包含测试函数的实现。

（三）codes/threads/thread.h

这一部分是线程头文件，其中声明了 ThreadStatus 的枚举类型，Thread 线程类，以及一些宏定义和函数的声明。

线程状态 ThreadStatus 与线程类 Thread 在上一题中已经介绍。

在 Nachos 中，MachineStateSize 值为 18，栈的大小 StackSize 为 4096 个四字。

ThreadPrint 函数在 thread.cc 中完成。ThreadRoot 和 SWITCH 为实现于 switch.c 的两端汇编代码，ThreadRoot 在 SWITCH 中调用，用于启动第一个作为 root 的线程；SWITCH 储存原来的线程的寄存器状态，加载要调度的新线程的寄存器状态。可以看出，这两段汇编语言（转为 extern C 语言）用于上下文切换，对于不同的体系结构，采取不同的实现方法。

（四）codes/threads/thread.cc

这份文件包含着 thread.h 中的函数声明的实现，用于管理线程。

1.Thread 类的构造函数: Thread::Thread(char* threadName)

参数赋给 name 成员，初始化时，状态设置为 JUST_CREATED，其余指针全部设置为 NULL。

2. Thread 类的析构函数：Thread::~Thread()

释放一个已有线程。如果为这个线程分配了栈空间，那么同时释放这部分空间。规定当前线程不能自己删除自己，主线程不分配栈空间（Nachos 启动时会自动获取空间）。

3.void Thread::Fork(VoidFunctionPtr func, void *arg)

用于启动创建好的新线程。首先为线程分配栈空间，然后设置中断状态，并让当前线程处于就绪状态，准备运行。函数和参数的位置与 SWITCH 函数的要求保持一致。

4. void Thread::CheckOverflow()

检查栈是否溢出。Nachos 每次分配栈空间时，总会在最后一字节放置一个宏定义数值 STACK_FENCEPOST，这个函数检查这个值是否改变，来判断栈是否溢出。

5. void Thread::Finish ()

当线程执行完毕 fork 的函数后，ThreadRoot 会调用这个函数来终止这个进程。这个函数把待终止进程复制到 threadToBeDestroyed 变量中，再调用 Sleep()函数。当设置好 threadToBeDestroyed 后，Sheduler::Run()函数会调用这个变量的析构函数来释放栈空间。除

此之外，这个函数不会响应中断，这样保证了设置 threadToBeDestroyed 与 Sleep()位于同一时间片内。

6. void Thread::Yield ()

这个函数用于让当前线程主动放弃 CPU 占用，并调用调度算法来寻找下一个线程，进行主动的线程切换。

7. void Thread::Sleep ()

这个函数把当前进程设置为睡眠模式，线程状态设置为 BLOCKED，然后寻找下一个要执行的线程，在找到之前，当前线程不会响应中断；找到之后，调度算法会运行下一个进程。这样，当前进程就处于睡眠状态，等待唤醒或者回收。

8. static void ThreadFinish()

调用当前线程的 Finish 函数以结束其运行。

9. static void InterruptEnable()

让当前线程可以响应中断。

10. void ThreadPrint(int arg)

输出当前线程的信息。

11.void Thread::StackAllocate (VoidFunctionPtr func, void *arg)

为线程分配栈空间，根据不同的体系结构进行不同的栈指针处理。然后设置寄存器的状态（PCState、StartupPCState、InitialPCState、InitialArgState、WhenDonePCState）

12. void Thread::SaveUserState()

储存当前的线程上下文，在线程切换之前保护现场。

13. void Thread::RestoreUserState()

恢复线程的上下文，用于线程调度后的为新线程重新设置现场。

Exercise3

添加用户 ID 和线程 ID 两个变量，并维护。首先，这两个变量应该作为 Thread 类的私有变量，不能让其他对象直接修改。但我们还需要获取这两个变量，因此在 public 标号下应该添加两个 getter 类型的方法。除此之外，考虑是否需要改动这两个 ID。对于 TID，由于它作为线程的唯一表示，故不可以修改；对于 UID，如果考虑多用户的可能情况，做用户切换的话可能会需要修改 UID。因此，仅为 UID 提供 setter 类型的方法。这一部分的改动如下：

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int* stackTop;                 // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop

    /******************************   added by Li Cong 1800012826   ******************************/

    int UID; // User ID
    int TID; // Thread ID

  public:
    int getUID();
    int getTID();
    void setUID(int newUID);

    /******************************   added by Li Cong 1800012826   ******************************/
```

```
/*******************************   added by Li Cong 1800012826   *******************************/

//----------------------------------------------------------------------
// Thread::getUID
//      Get thread's userID.
//----------------------------------------------------------------------

int
Thread::getUID(){return UID;}

//----------------------------------------------------------------------
// Thread::getTID
//      Get thread's threadID.
//----------------------------------------------------------------------

int
Thread::getTID(){return TID;}

//----------------------------------------------------------------------
// Thread::setUID
//      Set thread's userID.
//----------------------------------------------------------------------

void
Thread::setUID(int newUID){UID = newUID;}

/*******************************   added by Li Cong 1800012826   *******************************/
```

除此之外，我们还应该考虑构造函数和析构函数中的处理。构造函数中应该为新生成的线程指定 UID 和 TID。对于 TID，由于每次创建的进程都需要寻找到一个未被占用的 ID 值，且系统应该有最大线程数的限制，故应该在全局变量中设置一个标记数组。翻阅代码后知，所有的全局变量都存放在 system.h 中，因此应把这个全局变量声明于 system.h，且在 system.cc 中初始化。注意，extern 只是声明，还需要定义。这一部分的改动如下：

```
// system.h
//      All global variables used in Nachos are defined here.
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#ifndef SYSTEM_H
#define SYSTEM_H

#include "copyright.h"
#include "utility.h"
#include "thread.h"
#include "scheduler.h"
#include "interrupt.h"
#include "stats.h"
#include "timer.h"

// Initialization and cleanup routines

/*******************************   added by Li Cong 1800012826   *******************************/
#define MAX_THREAD_NUM 128
extern bool used_TID[MAX_THREAD_NUM];
/*******************************   added by Li Cong 1800012826   *******************************/
```

```
bool used_TID[MAX_THREAD_NUM];

void
Initialize(int argc, char **argv)
{
    int argCount;
    char* debugArgs = "";
    bool randomYield = FALSE;

/*******************************   added by Li Cong 1800012826   *******************************/

    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        used_TID[i] = false;
    }

/*******************************   added by Li Cong 1800012826   *******************************/
```

添加完标记数组后，我们便可以用遍历的方法为新进程寻找 TID 了。对于 UID，我选择在构造函数中赋初值为 0。这样，构造函数便可完成修改，效果如下：

```
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif

/*******************************   added by Li Cong 1800012826   *******************************/

    this->TID = -1;
    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        if(!used_TID[i])
        {
            this->TID = i;
            used_TID[i] = true;
            break;
        }
    }
    this->UID = 0;
    if(this->TID==-1)
    {
        printf("Assertion failure: reached max thread number!\n");
    }
    ASSERT(this->TID>=0 && this->TID<MAX_THREAD_NUM);

/*******************************   added by Li Cong 1800012826   *******************************/
}
```

接下来再来修改析构函数。当线程终止时，应该释放它占用的 TID，故需要对全局的标记数组进行操作，代码如下：

```
Thread::~Thread()
{
    DEBUG('t', "Deleting thread \"%s\"\n", name);

    ASSERT(this != currentThread);
    if (stack != NULL)
        DeallocBoundedArray((char *) stack, StackSize * sizeof(int));

/*******************************   added by Li Cong 1800012826   *******************************/

    used_TID[this->TID] = false;

/*******************************   added by Li Cong 1800012826   *******************************/

}
```

这样，对线程 UID、TID 的添加和维护便添加完成，下面尝试编写代码来进行测试。我占用了 ThreadTest 中的 case 3。首先仿照 case 0 创建并 Fork 一定数量的线程，不断打印每个线程的 UID 和 TID 等信息。代码大致如下：

```
//----------------------------------------------------------------------
// ThreadTest
//      Invoke a test routine.
//----------------------------------------------------------------------

void
ThreadTest()
{
    switch (testnum) {
    case 1:
        ThreadTest1();
        break;
/*******************************   added by Li Cong 1800012826   *******************************/
    case 2:
        printf("Just a test for make!\n");
    case 3:
        Exercise3Test();
        break;
/*******************************   added by Li Cong 1800012826   *******************************/
    default:
        printf("No test specified.\n");
        break;
    }
}

/*******************************   added by Li Cong 1800012826   *******************************/

//----------------------------------------------------------------------
// CheckIDs
//      used for fork
//----------------------------------------------------------------------

void
CheckIDs(int which)
{
    int num;

    for (num = 0; num < 5; num++) {
        printf("*** thread %d with TID %d and UID %d looped %d times\n", which, currentThread->getTID(), currentThread->getUID(), num);
        currentThread->Yield();
    }
}

//----------------------------------------------------------------------
// Excercise3Test
//      test function for question 3
//----------------------------------------------------------------------

void
Exercise3Test()
{
    DEBUG('t', "Entering Excercise3Test");

    const int n_threads = 5;
    const int tid = 233;

    for(int i=0; i<n_threads; ++i)
    {
        Thread *t = new Thread("forked thread");
        t->setUID(tid+i);
        t->Fork(CheckIDs, (void *)t->getTID());
    }
    CheckIDs(0);
}

/*******************************   added by Li Cong 1800012826   *******************************/
```

测试结果如下：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos -q 3
*** thread 0 with TID 0 and UID 0 looped 0 times
*** thread 1 with TID 1 and UID 233 looped 0 times
*** thread 2 with TID 2 and UID 234 looped 0 times
*** thread 3 with TID 3 and UID 235 looped 0 times
*** thread 4 with TID 4 and UID 236 looped 0 times
*** thread 5 with TID 5 and UID 237 looped 0 times
*** thread 0 with TID 0 and UID 0 looped 1 times
*** thread 1 with TID 1 and UID 233 looped 1 times
*** thread 2 with TID 2 and UID 234 looped 1 times
*** thread 3 with TID 3 and UID 235 looped 1 times
*** thread 4 with TID 4 and UID 236 looped 1 times
*** thread 5 with TID 5 and UID 237 looped 1 times
*** thread 0 with TID 0 and UID 0 looped 2 times
*** thread 1 with TID 1 and UID 233 looped 2 times
*** thread 2 with TID 2 and UID 234 looped 2 times
*** thread 3 with TID 3 and UID 235 looped 2 times
*** thread 4 with TID 4 and UID 236 looped 2 times
*** thread 5 with TID 5 and UID 237 looped 2 times
*** thread 0 with TID 0 and UID 0 looped 3 times
*** thread 1 with TID 1 and UID 233 looped 3 times
*** thread 2 with TID 2 and UID 234 looped 3 times
*** thread 3 with TID 3 and UID 235 looped 3 times
*** thread 4 with TID 4 and UID 236 looped 3 times
*** thread 5 with TID 5 and UID 237 looped 3 times
*** thread 0 with TID 0 and UID 0 looped 4 times
*** thread 1 with TID 1 and UID 233 looped 4 times
*** thread 2 with TID 2 and UID 234 looped 4 times
*** thread 3 with TID 3 and UID 235 looped 4 times
*** thread 4 with TID 4 and UID 236 looped 4 times
*** thread 5 with TID 5 and UID 237 looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 410, idle 0, system 410, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

可以看到，TID 前后的值一致，UID 也可以得到设置的值，可以认为上述实现完成了目标。

Exercise4

首先把最大线程个数设置为 128。在上一问中，我们设置了 MAX_THREAD_NUM 这一宏变量为 128，来限制合法 TID 个数。除此之外，我们在构造函数中首先初始化 TID 为-1，如果遍历 used_TID 数组发现都为 true，那么 TID 仍然为-1.这时，利用 ASSERT 语句，我们可以检测出线程超出最大约定个数这一错误，并退出程序。因此，最大线程数为 128 的要求已经完成。测试如下：

```
            case 4:
                Exercise4_1Test();
                break;


void
Exercise4_1Test()
{
    DEBUG('t', "Entering Excercise4_1Test");

    const int n_threads = 514;
    const int tid = 233;

    for(int i=0; i<n_threads; ++i)
    {
        Thread *t = new Thread("forked thread");
        t->setUID(tid+i);
        t->Fork(CheckIDs, (void *)t->getTID());
    }
    CheckIDs(0);
}
```

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos -q 4
Assertion failure: reached max thread number!
Assertion failed: line 62, file "../threads/thread.cc"
已放弃 (核心已转储)
```

下面来考虑 TS 功能的实现。

为了实现这一功能，我们首先应该试图获取所有的线程对象。在翻阅了 thread 文件夹中的代码后，我发现，Scheduler 类中存在一个 List*变量，储存了指向线程队列的指针，但这个变量是私有变量，为了避免每次都调用 getter 方法，故考虑添加一个新的全局指针数组，每个元素指向 TID 为当前数组下标的线程，同时修改构造函数，修改如下：

```
/*******************************   added by Li Cong 1800012826   *******************************/
#define MAX_THREAD_NUM 128
extern bool used_TID[MAX_THREAD_NUM];
extern Thread *Thread_Pointer[MAX_THREAD_NUM];
/*******************************   added by Li Cong 1800012826   *******************************/
```

```
bool used_TID[MAX_THREAD_NUM];
Thread *Thread_Pointer[MAX_THREAD_NUM];

void
Initialize(int argc, char **argv)
{
    int argCount;
    char* debugArgs = "";
    bool randomYield = FALSE;

/*******************************   added by Li Cong 1800012826   *******************************/

    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        used_TID[i] = false;
        Thread_Pointer[i] = NULL;
    }

/*******************************   added by Li Cong 1800012826   *******************************/
```

```
Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif

/*******************************   added by Li Cong 1800012826   *******************************/

    this->TID = -1;
    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        if(!used_TID[i])
        {
            this->TID = i;
            used_TID[i] = true;
            Thread_Pointer[i] = this;
            break;
        }
    }
    this->UID = 0;
    if(this->TID==-1)
    {
        printf("Assertion failure: reached max thread number!\n");
    }
    ASSERT(this->TID>=0 && this->TID<MAX_THREAD_NUM);

/*******************************   added by Li Cong 1800012826   *******************************/

}
```

由于 Thread 的成员变量均为私有，故为 Thread 继续完善 getter 成员函数，以获取所有必需的信息。补充如下：

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int* stackTop;                 // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop

    /*******************************   added by Li Cong 1800012826   *******************************/

    int UID; // User ID
    int TID; // Thread ID

  public:
    int getUID();
    int getTID();
    void setUID(int newUID);

    ThreadStatus getStatus();

    /*******************************   added by Li Cong 1800012826   *******************************/
```

```
//----------------------------------------------------------------------
// Thread::getStatus
//      Get thread's ThreadStatus.
//----------------------------------------------------------------------

ThreadStatus
Thread::getStatus(){return status;}
```

现在，我们可以获取所有的 Thread 对象，也可以获取每个线程的必需信息，故 TS 函数实现如下：

```
//----------------------------------------------------------------------
// TS
//      PS-like print function
//----------------------------------------------------------------------

void
TS()
{

    DEBUG('t', "Entering TS.");

    char *State2String[] = {"JUST_CREATED", "RUNNING", "READY", "BLOCKED"};

    printf("UID\tTID\tNAME\tSTAT\n");
    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        if(used_TID[i])
        {
            printf("%d\t%d\t%s\t%s\n", Thread_Pointer[i]->getUID(), Thread_Pointer[i]->getTID(), Thread_Pointer[i]->getName(), State2String[Thread_Pointer[i]->getStatus()]);
        }
    }
}
```

然后编写测试程序来测试实现的正确性，为了测试不同状态，我编写了 SetStatus 函数来设置线程的不同状态。测试程序及实验结果如下：

```
//----------------------------------------------------------------------
// Excercise4_2Test
//      test function for question 4_2
//----------------------------------------------------------------------

void
Exercise4_2Test()
{
    DEBUG('t', "Entering Excercise4_2Test");

    Thread *t1 = new Thread("fork0");
    t1->Fork(SetStatus, (void *)0);
    Thread *t2 = new Thread("fork1");
    t2->Fork(SetStatus, (void *)1);
    Thread *t3 = new Thread("fork2");
    t3->Fork(SetStatus, (void *)2);

    Thread *t4 = new Thread("forked");

    SetStatus(0);

    printf("\n\n-------------------this is TS result-------------------\n");
    TS();
    printf("------------------- finish TS test  -------------------\n\n");

}
```

```
//----------------------------------------------------------------------
// SetStatus
//      Set the status of thread, used for test TS
//----------------------------------------------------------------------

void
SetStatus(int which)
{
    switch(which)
    {
        case 0:
            scheduler->Print();
            currentThread->Yield();
            break;
        case 1:
            IntStatus oldlevel = interrupt->SetLevel(IntOff);
            currentThread->Sleep();
            (void) interrupt->SetLevel(oldlevel);
            break;
        case 2:
            currentThread->Finish();
            break;
        default:
            currentThread->Yield();
            break;
    }
}
```

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos -q 5
Ready list contents:
fork0, fork1, fork2, Ready list contents:
fork1, fork2, main,

------------------this is TS result------------------
UID     TID     NAME    STAT
0       0       main    RUNNING
0       1       fork0   READY
0       2       fork1   BLOCKED
0       4       forked  JUST_CREATED
------------------ finish TS test  ------------------

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 90, idle 0, system 90, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

最后，为了能够像 PS 那样直接输入命令行运行测试，再修改 main 函数，使之能响应 TS 命令，修改和测试如下：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos TS
Ready list contents:
fork0, fork1, fork2, Ready list contents:
fork1, fork2, main,

------------------this is TS result------------------
UID     TID     NAME    STAT
0       0       main    RUNNING
0       1       fork0   READY
0       2       fork1   BLOCKED
0       4       forked  JUST_CREATED
------------------ finish TS test  ------------------

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 90, idle 0, system 90, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

```
    for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
      argCount = 1;
      switch (argv[0][1]) {
      case 'q':
        testnum = atoi(argv[1]);
        argCount++;
        break;
/*******************************   added by Li Cong 1800012826   *******************************/
      case 'S':
        if(argv[0][0]=='T')
        {
           testnum = 5;
           break;
        }
/*******************************   added by Li Cong 1800012826   *******************************/
```

至此，所有练习全部完成，Lab1 完结撒花！

内容二：遇到的困难以及收获

首先，通过调研作业的完成，我大致了解了 Linux 的 PCB 结构。

其次，通过阅读代码和补全代码，我初步掌握了应该如何去维护全局变量来控制结构体中的某些变量。在实验过程中，我在 extern 的使用上遇到了一些问题。exercise3 中，一开始我只是在 system.h 中声明了一个全局的 bool 数组，但是我没有意识到声明不等于定义，因此我在 make 的过程中出现了问题，产生了“未定义的引用”类型的错误。后来，我搜索了 stackover flow，发现原来是因为我没在.cc 文件中定义这个布尔数组。改正这一错误后，我也顺利完成了 exercise3。

另外，在实现 TS 的时候，一开始我有点不知道从何下手，因为我并不是很明确应该去哪里获取所有已分配的线程对象。在翻阅的所有的代码后，我发现它封装在 Scheduler 的一个私有成员中，如何简易地处理它也是我遇到的一个问题。由于练习 3 通过全局数组维护 TID，因此我认为在创建线程时也可以维护一个指向线程对象的全局数组，这便是我的实现方式。这一练习训练了我分析代码结构的能力，也让我对全局对象的使用有了更清晰的了解。除此之外，现在我使用\t 来生成较工整的输出格式，但是当线程名较长时，输出不够工整，这可能还需要我去调研一下 ps 类命令的输出规范。

内容三：对课程或 Lab 的意见和建议

希望可以把引导手册进一步完善，现在每次阅读代码的时候需要我们递归阅读和寻找其他额外的代码，但我们往往不知道应该去哪里能找到，这样浪费了许多不必要的时间。我觉得可以把某些常用函数（或者重要辅助函数）的位置标注出来，这样可以节省遍历搜索的时间。

也希望可以为调研类的作业提供更多的提示和说明。现在我们只能漫无目的地去遍历搜索百度、google 的结果，我们无法确认博客等网站表述内容的正确性和严谨性，而且有的调研题目很难找到相关博客。让我们在短时间内查询所有体系结构/操作系统的手册，对于还有许多其他课程的本科生来说在时间上不是很现实。所以希望课程可以提供更多相关资源，或者提示我们应该去看手册的哪些章节，去找什么样的权威资料。这样既可以让我们学到准确的知识，也可以提升我们的学习效率。

内容四：参考文献

1. Linux 下的 task_struct 结构体
（https://blog.csdn.net/weixin_38239856/article/details/82112597）
2. Linux 进程管理之 task_struct 结构体（上）
（https://blog.csdn.net/npy_lp/article/details/7292563）
3. Linux 进程管理之 task_struct 结构体（下）
（ https://tanglinux.blog.csdn.net/article/details/7335187?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase）
4. Linux-进程描述符 task_struct 详解
（https://www.cnblogs.com/JohnABC/p/9084750.html）
5.extern 相关
https://stackoverflow.com/questions/7670816/create-extern-char-array-in-c
6.Nachos 英文介绍
https://www.ida.liu.se/~TDDI04/material/begguide/roadmap/node1.html