操作系统实验报告

# Xv6 Thread

## Description

在xv6中实现线程，并实现一个小的线程池。

具体来说要求实现以下函数：

int clone(void(\*)(void\*, void\*), void\*, void\*, void\*);

int join(void\*\*);

int thread\_create(void (\*start\_routine)(void \*, void \*), void\* arg1, void\* arg2);

void lock\_init(struct lock\_t \*lk);

void lock\_acquire(struct lock\_t \*lk);

void lock\_release(struct lock\_t \*lk);

其中前两个为系统调用，后四个为用户库函数。

### Clone/join

#### System call

在xv6中实现系统调用需要以下步骤：

1. 在defs.h中添加函数原型
2. 在syscall.h中定义系统调用的序号，具体来说是

#define SYS\_clone 22

#define SYS\_join 23

1. 在syscall.c中添加sys\_clone和sys\_join函数声明，并在(\*syscalls[])(void)数组中添加对应的系统调用。
2. 在sysproc.c中实现sys\_clone和sys\_join，主要任务是处理参数并提供真正实现的包装。
3. 在user.h中添加clone和join的函数原型
4. 在usys.S中添加SYSCALL(clone)和SYSCALL(join)。

上述步骤的目的都是为了把系统调用和中断表关联起来。

#### Implement clone/join

步骤如下：

1. 在proc.h中，向程序结构体添加新的成员,void\* stack，作为线程栈。这和原来自带的kstack有着不同的作用。Kstack是内核栈，存储了进程的内核栈的起始地址。在进程切换或处理中断时，操作系统会使用这个地址来保存和恢复进程的执行上下文。当一个进程被切换出去时，它的内核栈上的内容会被保存到内存中，以便在再次切换回来时能够恢复到之前的状态。
2. 在proc.c中实现clone和join函数

int clone(void(\*fcn)(void\*,void\*), void \*arg1, void \*arg2, void\* stack)

{

  struct proc \*np;

  struct proc \*cp = myproc();

  if((np = allocproc()) == 0)

    return -1;

  np->pgdir = cp->pgdir;

  np->sz = cp->sz;

  np->parent = cp;

  \*np->tf = \*cp->tf;

  \*(uint\*)(stack + PGSIZE - 3 \* sizeof(void \*)) = 0xFFFFFFF;

  \*(uint\*)(stack + PGSIZE - 2 \* sizeof(void \*)) = (uint)arg1;

  \*(uint\*)(stack + PGSIZE - 1 \* sizeof(void \*)) = (uint)arg2;

  np->stack = stack;

  np->tf->esp = (uint)(stack + PGSIZE - 3 \* sizeof(void\*));

  np->tf->ebp = np->tf->esp;

  np->tf->eip = (uint) fcn;

  np->tf->eax = 0;

  int i;

  for(i = 0; i < NOFILE; i++)

    if(cp->ofile[i])

      np->ofile[i] = filedup(cp->ofile[i]);

  np->cwd = idup(cp->cwd);

  safestrcpy(np->name, cp->name, sizeof(cp->name));

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return np->pid;

}

这段代码是一个实现了进程（或线程）的克隆功能的函数，函数名为clone。下面是对该函数的实现逻辑的解释：

1. 首先，函数定义了两个指针变量np和cp，分别表示新创建的进程（或线程）和当前进程（或线程）。
2. 接下来，通过调用allocproc()函数来分配一个新的进程结构体，如果分配失败，则返回-1。
3. 将新进程的页目录（pgdir）、进程的大小（sz）以及父进程（parent）设置为当前进程（cp）的相应值。这样新进程就共享了当前进程的地址空间和文件描述符表。
4. 将新进程的Trapframe（tf）结构体设置为当前进程的Trapframe的副本，以保留当前进程的执行状态。这样做是为了确保新进程在开始执行时具有与当前进程相同的执行上下文。
5. 设置新进程的用户栈（stack）中的内容。这些内容包括：
   1. 在用户栈的最顶部（PGSIZE字节偏移处）存储一个特殊的标记值0xFFFFFFF。这个标记用于标识用户栈的顶部边界。
   2. 在用户栈的倒数第二个位置存储arg1参数的值。
   3. 在用户栈的倒数第一个位置存储arg2参数的值。
6. 将新进程的stack字段设置为传递给函数的stack参数，即新进程的用户栈的起始地址。
7. 更新新进程的Trapframe中的esp、ebp、eip和eax等寄存器的值，将它们设置为适当的值，以便新进程在切换到用户态时正确执行。esp和ebp被设置为用户栈的地址，eip被设置为函数指针fcn的地址，eax被设置为0。
8. 复制当前进程的文件描述符表（ofile）中的文件描述符到新进程的ofile数组中，使用filedup()函数进行文件描述符的复制。
9. 复制当前进程的当前工作目录（cwd）的引用计数，并将引用计数增加到新进程的cwd字段中。
10. 使用safestrcpy()函数将当前进程的名称（name）复制到新进程的name字段中。
11. 获取全局进程表（ptable）的锁，确保在修改新进程的状态时不会出现竞争条件。
12. 将新进程的状态设置为可运行（RUNNABLE）。
13. 释放全局进程表的锁。
14. 返回新进程的进程ID（pid）。

这样，通过调用clone函数，可以创建一个新的进程（或线程），该进程共享当前进程的地址空间、文件描述符和执行状态。

下面是join函数

int join(void\*\* stack) {

  struct proc \*p, \*cp = myproc();

  int havekids, pid;

  acquire(&ptable.lock);

  for(;;) {

    havekids = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

      if(p->parent != cp || p->pgdir != p->parent->pgdir)

        continue;

      havekids = 1;

      if(p->state == ZOMBIE){

        pid = p->pid;

        kfree(p->kstack);

        p->kstack = 0;

        p->pid = 0;

        p->parent = 0;

        p->name[0] = 0;

        p->killed = 0;

        p->state = UNUSED;

        stack = p->stack;

        p->stack = 0;

        release(&ptable.lock);

        return pid;

      }

    }

    if(!havekids || cp->killed){

      release(&ptable.lock);

      return -1;

    }

    sleep(cp, &ptable.lock);

  }

  return 0;

}

这段代码是实现了进程（或线程）等待子进程（或线程）退出的join函数。下面是对该函数的实现逻辑的解释：

1. 首先，定义了两个指针变量p和cp，分别表示当前进程（或线程）和待等待的子进程（或线程）。
2. 获取全局进程表（ptable）的锁，以确保在遍历进程表时不会出现竞争条件。
3. 进入一个无限循环，用于不断检查子进程（或线程）的退出状态。
4. 初始化变量havekids为0，用于标记是否存在子进程（或线程）。
5. 遍历进程表中的所有进程（或线程），对每个进程执行以下操作：
   * 1. 如果该进程的父进程不是当前进程或者该进程的页目录（pgdir）与父进程的页目录不相等，则继续下一个循环。
     2. 将havekids标记设置为1，表示存在子进程（或线程）。
     3. 如果该进程的状态为ZOMBIE（已退出状态），则进行以下处理：
     4. 将变量pid设置为该进程的进程ID。
     5. 释放该进程的内核栈（kstack）的内存。
     6. 将该进程的内核栈指针（kstack）置为0。
     7. 将该进程的进程ID（pid）、父进程（parent）、进程名称（name）、被杀死标记（killed）和进程状态（state）等字段重置为初始值。
     8. 将stack指针设置为该进程的用户栈（stack）的起始地址。
     9. 将该进程的用户栈指针（stack）置为0，表示该用户栈已不再使用。
     10. 释放全局进程表（ptable）的锁。
     11. 返回该子进程的进程ID（pid）。
6. 如果不存在子进程（或线程）或者当前进程（或线程）被标记为被杀死（killed），则释放全局进程表的锁并返回-1，表示等待失败。
7. 如果存在子进程（或线程），但尚未有子进程（或线程）退出，则当前进程（或线程）进入睡眠状态，释放全局进程表的锁，等待子进程（或线程）的退出。
8. 如果当前进程（或线程）被唤醒，说明有子进程（或线程）退出，继续进行下一次循环，重新检查子进程（或线程）的退出状态。
9. 如果循环结束，即没有子进程（或线程）退出，并且当前进程（或线程）没有被标记为被杀死，则释放全局进程表的锁并返回0

## Thread library

步骤如下：

1. 在user.h中定义ticket lock的结构

struct lock\_t {

    uint turn;

    uint ticket;

};

1. 在ulibc.c中实现线程创建和锁相关的函数

int thread\_create(void (\*start\_routine)(void \*, void \*), void\* arg1, void\* arg2) {

  void\* stack = malloc(PGSIZE);

  return clone(start\_routine, arg1, arg2, stack);

}

void lock\_init(struct lock\_t \*lk){

  lk->turn = 0;

  lk->ticket = 0;

}

void lock\_acquire(struct lock\_t \*lk){

  int ticket = \_\_atomic\_fetch\_add(&lk->ticket, 1, \_\_ATOMIC\_SEQ\_CST);

  while(ticket != lk->turn);

}

void lock\_release(struct lock\_t \*lk){

  lk->turn++;

}

注意我使用了\_\_atomic\_fetch\_add，这是为了原子地实现取数和自增。

1. 编写新的程序测试以上函数。这一步需要修改Makefile来创建新的make目标。

## Extra works

### Environment control tool: setting.sh

我额外编写了一个脚本用于进行控制，帮助信息如下：

    快捷地操纵环境。如果没有任何参数，将安装软件包和xv6并运行

    使用方法：

    setting.sh [options][-b|-rc backup]

    options:

    -rs, --reset           重新安装xv6

    -rc, --recover         从参数中恢复xv6，默认为.backup

    -b, --backup           备份当前xv6至backuo参数，默认为.backup

    -c, --chmod-all        赋予所有\*.sh和\*.exp权限

    -np, --no-package      不安装软件包，只安装xv6

-h, --help             显示帮助信息";

可以看到实际上我还实现了部分版本控制地功能。

### Answer generator: gen-ans

我编写了一个脚本用于直接修改文件生成结果，以下为部分代码示例

# defs.h

    sed -i "/syscall(void)/a\int\t\t\t\tclone(void(\*)(void\*, void\*), void\*, void\*, void\*);\nint\t\t\t\tjoin(void\*\*);" $src/defs.h

# syscall.c

    sed -i -e "/sys\_uptime(void)/a\extern int sys\_clone(void);\nextern int sys\_join(void);" \

           -e "/sys\_close,/a\[SYS\_clone]\t\tsys\_clone,\n[SYS\_join]\t\tsys\_join," $src/syscall.c

# syscall.h

    last\_cnt=$(tail -n 1 $src/syscall.h | grep -o '[0-9]\+')

    echo -e "#define SYS\_clone $[$last\_cnt+1]\n#define SYS\_join $[$last\_cnt+2]" >> $src/syscall.h

# usys.S

    echo -e "SYSCALL(clone)\nSYSCALL(join)" >> $src/usys.S

# user.h

    sed -i -e "/rtcdate/a\struct lock\_t {\n\tuint turn;\n\tuint ticket;\n};" \

           -e "/atoi/a\int thread\_create(void (\*start\_routine)(void \*,void\*), void \* arg1, void \* arg2);\n\

void lock\_init(struct lock\_t \*lk);\n\

void lock\_acquire(struct lock\_t \*lk);\n\

void lock\_release(struct lock\_t \*lk);" \

           -e "/uptime(void)/a\int clone(void (\*start\_routine)(void\*,void\*), void \*, void \*, void \*);\n\

int join(void \*\*);" $src/user.h

之后，再使用shc将脚本变为可执行文件。这样一来就可以方便地和被人分享代码。

# MapReduce

## Description

实现MapReduce。

MapReduce是一种用于大规模数据处理的编程模型和计算框架。它最初由Google提出，用于解决大规模数据处理和分布式计算的问题。MapReduce模型简化了分布式数据处理的复杂性，使得开发人员能够方便地编写并行处理任务。

在MapReduce模型中，数据处理任务被分为两个主要阶段：Map阶段和Reduce阶段。下面是对每个阶段的简要介绍：

1. Map阶段：

* + - * 输入数据被划分成一组大小相等的数据块。
      * Map任务并行处理这些数据块，每个数据块由一个Map函数处理。Map函数将输入数据块映射为一系列的键值对（key-value pairs）。
      * 所有Map任务的输出被收集并按照键（key）进行排序。

2. Reduce阶段：

* + - * Reduce任务并行处理Map阶段的输出。每个Reduce任务处理一组具有相同键的键值对。
      * Reduce函数将具有相同键的键值对进行聚合、合并和计算，并生成最终的输出结果。

在综上，本实验主要可以分为三个部分，key-value pairs数据结构的实现、线程池的实现和MapReduce框架的实现。

## Data structure：kernel red-black tree

对于键值对，c++中通常使用std::map来表现，而c语言中没有现成的数据结构，因此需要自己实现。由于c语言没有泛型，所以在Linux内核中的数据结构都有一个特点，那就是“不包含用户数据”，具体来说，使用时可以

struct mytype {

    struct rbt\_node my\_node;

    int             num;

};

而对于一个插入操作，则是

int my\_insert(struct rbt\_root\* root, struct mytype\* data) {

    struct rbt\_node \*\*tmp = &(root->node), \*parent = NULL;

    /\* Figure out where to put new node \*/

    while (\*tmp) {

        struct mytype\* this = rbt\_entry(\*tmp, struct mytype, my\_node);

        parent = \*tmp;

        if (data->num < this->num)

            tmp = &((\*tmp)->left);

        else if (data->num > this->num)

            tmp = &((\*tmp)->right);

        else

            return -1;

    }

    /\* Add new node and rebalance tree. \*/

    rbt\_link\_node(&data->my\_node, parent, tmp);

    rbt\_insert\_fixup(root, &data->my\_node);

    return 0;

}

这么一来，就巧妙地避开了需要构造、销毁用户数据的麻烦，从而提高了通用性。

在阅读相关代码之后，我编写了近七百行代码以实现这个部分，以下是一些细节的介绍：

节点：

struct rbt\_node {

    unsigned long    parent\_and\_color;

    struct rbt\_node\* right;

    struct rbt\_node\* left;

} \_\_attribute\_\_((aligned(sizeof(long))));

不同于常见的红黑树，我把parent指针和color field放到了同一个变量中，并使用gcc的语法扩展\_\_attribute\_\_((aligned(sizeof(long))))将结点强制对齐4字节对齐（假设32位机器）。这么一来parent指针的最后两位就总是0，那么我们就可以把最后两位用于存储额外的数据。

在这一步，我实现了两种红黑树，一种是root指针直接指向树根的，如下图。在这种结构中，所有的空指针的值为NULL。

struct rbt\_root {

    struct rbt\_node\* node;

};

还有一种是类似于C++ STL的红黑树，node.parent指向真正的树根，node.left和node.right分别指向树中最小的和最大的结点。在这一种结构中，所有的空指针都指向root\_cached（称为dummy root），并且，dummy root中的node.parent的倒数第二位用于指示是否为dummy root。在这中结构中，可以以O(1)时间获得最小和最大值，同时避免了许多空指针判断，而且还利于实现迭代器。

struct rbt\_root\_cached {

    struct rbt\_node node;

};

实现了红黑树之后，我又使用cached rbtree编写了字典结构，接口与c++保持一致，部分代码如下：

rbt\_tree\* rbt\_create(rbt\_val\_comp cmpr);

void      rbt\_destroy(rbt\_tree\*, rbt\_val\_dtor dtor);

rbt\_insert\_result\_t           rbt\_insert(rbt\_tree\*, void\*);

rbt\_insert\_result\_t           rbt\_insert\_unique(rbt\_tree\*, void\*);

rbt\_insert\_or\_assign\_result\_t rbt\_insert\_or\_assign(rbt\_tree\*, void\*);

void\* rbt\_extract(rbt\_tree\* tree, rbt\_iterator it);

size\_t       rbt\_erase(rbt\_tree\*, void\* value, rbt\_val\_dtor dtor);

rbt\_iterator rbt\_erase\_at(rbt\_tree\*, rbt\_iterator position, rbt\_val\_dtor dtor);

rbt\_iterator rbt\_erase\_range(rbt\_tree\*, rbt\_iterator first, rbt\_iterator last, rbt\_val\_dtor dtor);

void         rbt\_clear(rbt\_tree\*, rbt\_val\_dtor dtor);

rbt\_iterator rbt\_find(rbt\_tree\*, void\*);

void\*        rbt\_val\_at(rbt\_tree\*, void\*);

void\*        rbt\_val\_at\_or(rbt\_tree\*, void\*, void\*);

size\_t       rbt\_size(rbt\_tree\*);

rbt\_iterator         rbt\_lower\_bound(rbt\_tree\*, void\*);

rbt\_iterator         rbt\_upper\_bound(rbt\_tree\*, void\*);

rbt\_eqrange\_result\_t rbt\_eqaul\_range(rbt\_tree\*, void\*);

rbt\_iterator rbt\_begin(rbt\_tree\* tree);

rbt\_iterator rbt\_end(rbt\_tree\* tree);

rbt\_iterator rbt\_rbegin(rbt\_tree\* tree);

rbt\_iterator rbt\_rend(rbt\_tree\* tree);

bool rbt\_is\_empty(rbt\_tree\* tree);

void rbt\_display(rbt\_tree\* tree, rbt\_tree\_print, rbt\_val\_print);

rbt\_iterator rbt\_iter\_next(rbt\_iterator it);

rbt\_iterator rbt\_iter\_prev(rbt\_iterator it);

void\*        rbt\_iter\_val(rbt\_iterator it);

bool         rbt\_iter\_eq(rbt\_iterator lhs, rbt\_iterator rhs);

bool         rbt\_iter\_neq(rbt\_iterator lhs, rbt\_iterator rhs);

## Thread pool

这一部分的接口相当少，只有如下几个

threadpool\* tpool\_create(size\_t num\_threads);

int         tpool\_add\_work(threadpool\*, void (\*fcn)(void\*), void\* arg);

void        tpool\_wait(threadpool\*);

void        tpool\_destroy(threadpool\*);

实现上，受CPU调度一章学习的多级队列启发，我在线程池中也使用了多级任务队列。用户只需要向池中提交任务，然后线程池会根据可指定的策略（默认为随机）分配到不同的任务队列中。

以下为线程的工作循环：

static void\* thread\_loop(thread\* t) {

    threadpool\* pool = t->pool;

    pthread\_mutex\_lock(&pool->thcount\_lock);

    ++pool->num\_threads\_alive;

    pthread\_mutex\_unlock(&pool->thcount\_lock);

    size\_t index = t->id % pool->num\_queue;

    while (pool->threads\_keepalive) {

        bsem\_wait(&pool->queues[index].has\_jobs);

        if (pool->threads\_keepalive) {

            pthread\_mutex\_lock(&pool->thcount\_lock);

            pool->num\_threads\_working++;

            pthread\_mutex\_unlock(&pool->thcount\_lock);

            node\_t\* job = jobqueue\_pop(&pool->queues[index]);

            if (job == NULL) {

                for (size\_t i = 0; i < pool->num\_queue; ++i)

                    if (i != index) {

                        job = jobqueue\_steal(&pool->queues[i]);

                        if (job) {

                            job->function(job->arg);

                            free(job);

                            break;

                        }

                    }

            }

            else {

                job->function(job->arg);

                free(job);

            }

            pthread\_mutex\_lock(&pool->thcount\_lock);

            pool->num\_threads\_working--;

            if (!pool->num\_threads\_working)

                pthread\_cond\_signal(&pool->threads\_all\_idle);

            pthread\_mutex\_unlock(&pool->thcount\_lock);

        }

    }

    pthread\_mutex\_lock(&pool->thcount\_lock);

    pool->num\_threads\_alive--;

    pthread\_mutex\_unlock(&pool->thcount\_lock);

    return NULL;

}

线程会等待一个用于表示该队列是否有任务的二元信号量，如果有，就会从中取出任务并执行。

我还考虑到多个线程同时抢占任务的情况，如果发现没有获取到任务，那么会通过jobqueue\_steal函数，从其他队列中获取任务。

为了提供线程安全的随机数生成，我使用的是xorshift+算法，如下

unsigned long xorshift\_plus32(unsigned long\* seed) {

    unsigned long s = \*seed;

    s ^= s << 13;

    s ^= s >> 17;

    s ^= s << 5;

    \*seed = s;

    return s;

}

此外，我还针对CPU亲和性做了优化，通过以下函数把线程绑定到某个CPU上。

int cpu\_bind\_thread(pthread\_t t, int core) {

    if (core < 0 || core >= cpu\_get\_num())

        return EINVAL;

    cpu\_set\_t cpuset;

    CPU\_ZERO(&cpuset);

    CPU\_SET(core, &cpuset);

    return pthread\_setaffinity\_np(t, sizeof(cpu\_set\_t), &cpuset);

}

## MapReduce

在有了以上工具的帮助下，MapReduce的实现就非常轻松了。具体而言，我们需要实现以下函数：

void MR\_Emit(char \*key, char \*value);

void MR\_Run(int argc, char \*argv[],

        Mapper map, int num\_mappers,

        Reducer reduce, int num\_reducers,

        Partitioner partition);

MR\_Emit的实现如下

void MR\_Emit(char \*key, char \*value) {

  unsigned long partition\_id = partioner(key, num\_partitions);

  pair\_t\* data =(pair\_t\*)malloc(sizeof(pair\_t));

  data.key = (char\*)malloc(strlen(key));

  memcpy(data.key, key, strlen(key));

  data.parition\_id = partition\_id

  rbt\_insert(root, data);

}

主要任务是把分割出的key和value插入到字典结构中。

下面是MR\_Run

void MR\_Run(int argc, char \*argv[], Mapper map, int num\_mappers, Reducer reduce,

            int num\_reducers, Partitioner partition) {

  // init

  num\_partitions = num\_reducers;

  partioner = partition;

  mapper = map;

  reducer = reduce;

  int mapper\_index = 1;

  threadpool\* mapper\_pool = tpool\_create(num\_mappers);

  threadpool\* reducer\_pool = tpool\_create(num\_reducers);

  // run  num\_mappers mappers  concurrently

  while (mapper\_index < argc) {

    for (int i = mapper\_index; i < std::min(mapper\_index + num\_mappers, argc); i++) {

      struct MapperParameter \*parameter = new MapperParameter;

      parameter->file\_name = argv[i];

      tpool\_add\_work(mapper\_pool, MapperRunner, parameter);

    }

    tpool\_wait(mapper\_pool);

    mapper\_index += num\_mappers;

  }

  // run num\_mappers reducers  concurrently

  for (unsigned long i = 0; i < num\_reducers; i++) {

    struct ReduceParameter \*parameter = new ReduceParameter;

    parameter->get\_next = get\_next;

    parameter->partition\_number = i;

    if (rbt\_size(roots[i]) > 0) {

      tpool\_add\_work(reducer\_pool, ReduceRunner, parameter);

    }

  }

  tpool\_wait(reducer\_pool);

  tpool\_destroy(mapper\_pool);

  tpool\_destroy(reducer\_pool);

}

1. 创建了两个线程池：mapper\_pool和reducer\_pool。mapper\_pool用于并发运行num\_mappers个映射器（mappers），而reducer\_pool用于并发运行num\_reducers个减少器（reducers）。
2. 在运行映射器之前的循环中，我们使用线程池的tpool\_add\_work函数将每个映射任务添加到mapper\_pool中。这样，线程池会自动调度可用线程来执行这些任务，并行地运行映射器。注意，我们将原始的MapperRunner函数作为任务的函数，并将parameter作为参数传递给该函数。
3. 使用线程池的tpool\_wait函数来等待所有映射任务完成。这会阻塞当前线程，直到线程池中的所有任务都执行完毕。
4. 重复上述过程，直到处理完所有映射器。mapper\_index递增，并继续添加新的映射任务到线程池中。
5. 在运行减少器之前的循环中，我们使用线程池的tpool\_add\_work函数将每个减少任务添加到reducer\_pool中。这样，线程池会自动调度可用线程来执行这些任务，并行地运行减少器。注意，我们将原始的ReduceRunner函数作为任务的函数，并将parameter作为参数传递给该函数。
6. 使用线程池的tpool\_wait函数来等待所有减少任务完成。这会阻塞当前线程，直到线程池中的所有任务都执行完毕。