第 2 次操作系统实验报告

161810129 董世晨

一、 线程支持

让我把最关键的内容写在最前面，以便批阅。

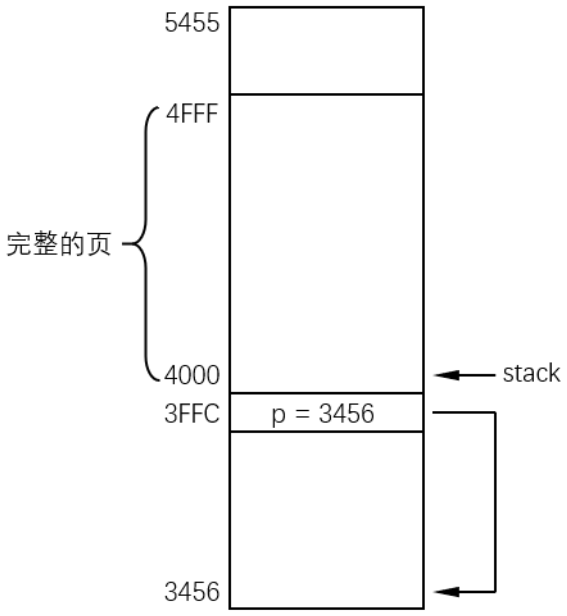
1. 如何不在PCB中新增字段来保存线程栈的地址stack？

简而言之：xthread_create中分配2*PGSIZE的空间，从中取一个完整的页作为线程栈空间，在线程栈空间的前一个4字节域中保存整个2*PGSIZE空间的首地址，用于free。join中可根据当前esp计算出stack。

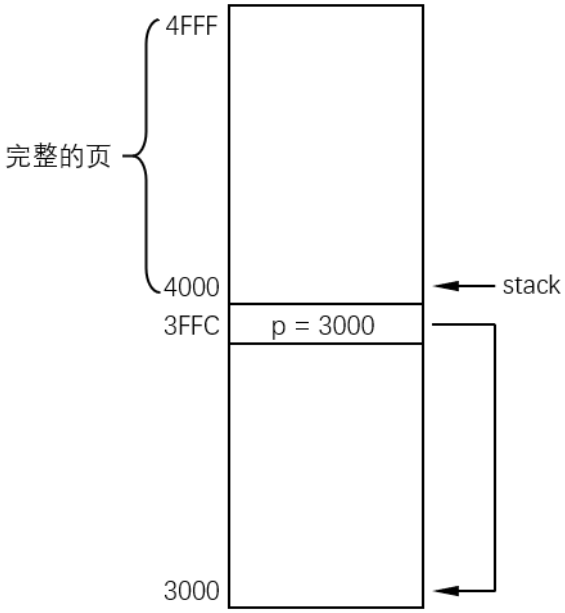
以下代码摘自xthread_create函数：

```
    // There must be a whole page in 2*PGSIZE
    void *p = malloc(PGSIZE * 2);
    if (!p)
        return -1;
    // Stack need to be aligned to page
    void *stack = (void*)PGROUNDUP((uint)p + sizeof(void *));
    // Save `p` just before stack, it's guaranteed that stack-4 >= p
    *((void **)stack-1) = p;
```

如下图所示，假设malloc分配了0x3456到0x5455这2*PGSIZE的空间，其中必然存在一个完整连续的页（即0x4000~0x4FFF），通过PGROUNDUP宏找到它，保存给stack。让stack页对齐是为了在join时可以直接将当前esp的后12位置零来找出stack，另一方面，线程栈和进程栈都实现了页对齐，较为统一。在stack的前一个4字节域（即0x3FFC）中，保存malloc分配的首地址，用于xthread_join时释放空间。



下图展示了当malloc返回值恰好页对齐时的情况。我不使用第一个页，而使用第二个页，这样子可以保证stack-4不越界，即stack-4>=p。上文摘录的代码的第6行实现了这一点。



2. 如何不在PCB中新增字段来保存线程返回值？

简而言之：将返回值压入线程的用户栈。

以下代码摘自thread_exit系统调用，解释了如何压栈：

```
// Push `ret` to thread stack
curproc->tf->esp -= sizeof(void *);
*(void **)curproc->tf->esp = ret;
```

以下代码摘自join系统调用，解释了如何获取stack和ret：

```
// `ret_p` is stored at the top of stack
*ret_p = *(void **)tidproc->tf->esp;
// `stack` is page aligned
*stack = (void **)PGROUNDDOWN(tidproc->tf->esp);
```

3. 如何捕获线程return的值？

在clone系统调用中，将线程的返回地址设为0xFFFFFFF0，该地址实际并不存在，当线程正常返回时，会触发page fault，其trapno为T_PGFLT，即14，这样就可以在trap函数中捕获。我在trap.c中添加了以下代码：

```
case T_PGFLT:
  // Return from thread
  if (myproc()->tf->eip == 0xFFFFFFF0)
    thread_exit((void *)myproc()->tf->eax);
  // Fall through
```

值得注意的是：我使用了0xFFFFFFF0而不是0xFFFFFFFF，是因为0xFFFFFFFF已经被用为进程的虚假返回地址（见exec.c的82行），为了避免冲突，改用另一个类似的虚假返回地址，以便在捕获时判断。

另一点是，如果捕获时eip不为0xFFFFFFF0，说明该page fault错误不是线程返回出发的，我没有写任何的处理程序，也不break，而是fall through到接下来的default分支，进行默认的错误处理。

我从eax寄存器中获取返回值，直接调用thread_exit进行相同的处理。

4. clone和fork的不同和相同之处

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();
  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;

  np->parent = curproc;
  *np->tf = *curproc->tf;
  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;
  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));
  pid = np->pid;

  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);
  return pid;
}
```

```
int
clone(void *(*fn)(void *), void *stack, void *arg)
{
  struct proc *np, *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from proc.

  np->sz = curproc->sz;
  np->pgdir = curproc->pgdir;
  // Set curproc as parent of new thread.
  np->parent = curproc;
  *np->tf = *curproc->tf;
  for(int i = 0; i < NOFILE; i++)

    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  // No need to `idup(curproc->cwd)`
  // because it's completely controlled by OS, accordingly:
  // `thread_exit` DO NOT call `iput(curproc->cwd)`
  // `exit` DO call `iput(curproc->cwd)`
  np->cwd = curproc->cwd;
  safestrcpy(np->name, curproc->name, sizeof(curproc->name));
  void** sp = stack + PGSIZE;
  *--sp = arg;  // push arg
  *--sp = (void*)0xFFFFFFF0;  // push return address
  np->tf->eip = (uint)fn;
  np->tf->esp = (uint)sp;
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);
  return np->pid;
}
```

我的clone函数是基于fork函数修改的，上图对比了fork和clone的差异。

a) 我复用了PCB，因此clone和fork一样，通过调用allocproc函数来初始化PCB和进程的内核栈。

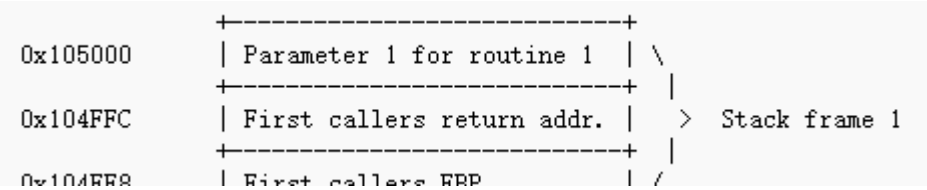
b) 线程共享地址空间，因此clone不需要copyuvm，直接复制pgdir即可。

c) 创建的线程也应隶属于创建者，因为thread_exit和join中的wakeup和sleep操作是基于父子关系的，因此同样设当前线程为新线程的parent。

d) clone和fork一样，需要调用filedup来增加所有文件的引用计数。

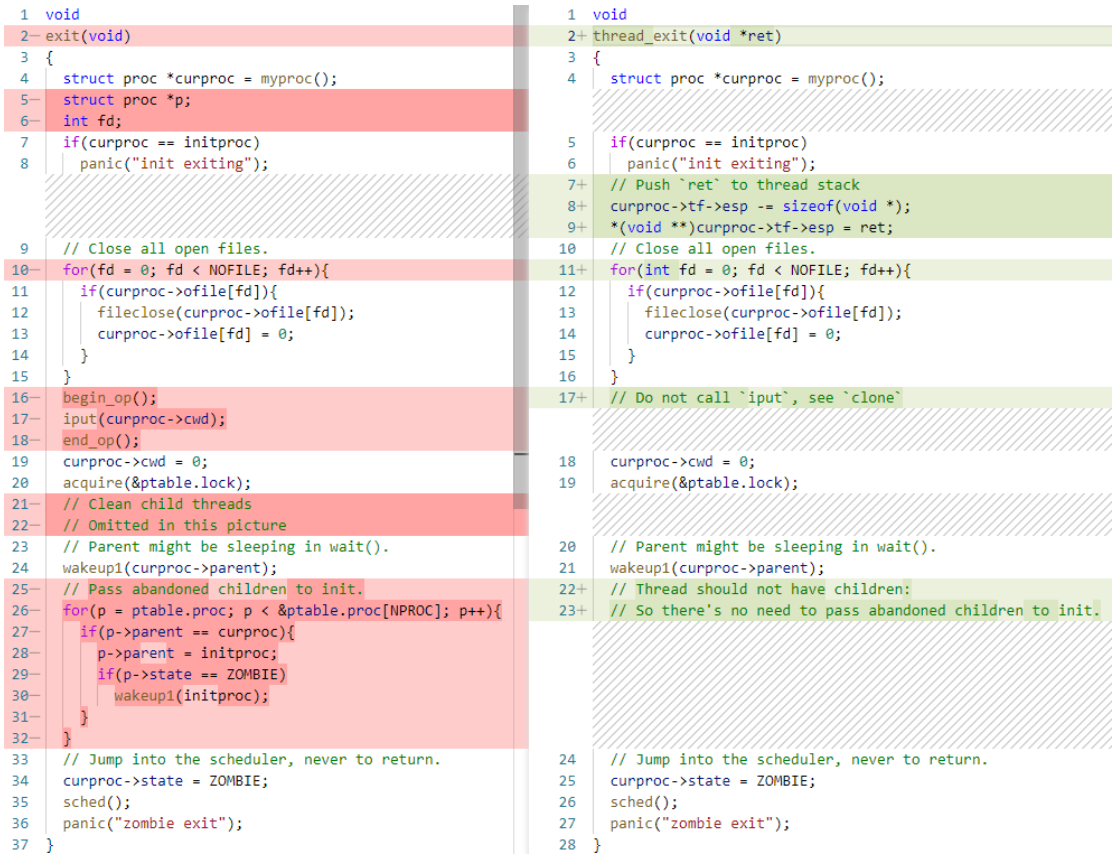
e) 相反，**不需要调用idup来增加cwd的引用计数**。因为和文件不同，对cwd的引用完全是由内核控制的：我约定，进程创建和退出时增减引用计数，而线程创建和退出时不改变引用计数。这样可以简化exit函数中释放子线程资源的难度。

f) clone需要在线程栈中构造出一个栈帧。如下图所示，只需要以此压入arg参数和一个虚假的返回地址即可。需要注意栈是往下长的，而malloc返回的是低地址，因此初始栈顶在stack+PGSIZE处。



g) 通过改变eip寄存器来改变下一条要执行的指令；通过改变esp寄存器来改变栈的地址。

5. thread_exit和exit的不同和相同之处



```
void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;
  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }
  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;
  acquire(&ptable.lock);
  // Clean child threads
  // Omitted in this picture
  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);
  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }
  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

```
void
thread_exit(void *ret)
{
  struct proc *curproc = myproc();

  if(curproc == initproc)
    panic("init exiting");
  // Push `ret` to thread stack
  curproc->tf->esp -= sizeof(void *);
  *(void **)curproc->tf->esp = ret;
  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }
  // Do not call `iput`, see `clone`

  curproc->cwd = 0;
  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);
  // Thread should not have children:
  // So there's no need to pass abandoned children to init.

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

我的thread_exit函数是基于exit函数修改的，上图对比了thread_exit和exit的差异，图中为了节省篇幅，省略了exit中清理子线程资源部分的代码，该部分将在后文详述。

a) thread_exit和exit一样需要调用fileclose减少文件引用计数，但是不需要调用iput来减少cwd的引用计数，相关原因已在前文解释。

b) 由于实验要求中说明了：“只有进程被创建时的线程（主线程）会执行 clone，即，被 clone 出来的线程不会执行 clone。”因此，线程不会有子线程，所以不需要将孤儿线程转交给 initproc。

6. join和wait的不同和相同之处

```
void join(int tid, void **ret_p, void **stack)
{
  struct proc *curproc = myproc(), *tidproc = 0;
  acquire(&ptable.lock);
  // Find tid
  for(struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->pid == tid) {
      tidproc = p;
      break;
    }
  // Should I exit silently without error code?
  if (!tidproc)
    goto exit;
  if (tidproc->parent != curproc)
    goto exit;
  // Wait until tidproc exits
  while (tidproc->state != ZOMBIE && !curproc->killed)
    sleep(curproc, &ptable.lock);
```

```
  // `ret_p` is stored at the top of stack
  *ret_p = *(void **)tidproc->tf->esp;
  // `stack` is page aligned
  *stack = (void **)PGROUNDDOWN(tidproc->tf->esp);
  kfree(tidproc->kstack);
  tidproc->kstack = 0;
  tidproc->pid = 0;
  tidproc->parent = 0;
  tidproc->name[0] = 0;
  tidproc->killed = 0;
  tidproc->state = UNUSED;
exit:
  release(&ptable.lock);
}
```

我的join函数参考了wait函数，但是代码结构进行了大幅调整和优化，进行diff没有意义，故只展示了join的代码。

a) 首先找到PID为tid的线程，这里有一个疑问：实验要求中指定的join没有返回值，如果找不到tid，或找到的tid不是当前进程的子线程，只能静默退出，无法报错。我认为将join的返回类型改为int，用于判断是否成功，会好一些。

b) 使用while循环一直等待子线程退出。这里需要用while而不是if的原因是：wakeup会唤醒所有在同一个chan上sleep的进程，因此醒来之后需要重新判断一下子线程到底有没有退出。

c) 从线程栈的栈顶取出返回值，将当前esp页对齐得到stack。

d) 释放线程资源。

7. 在exit中释放子线程资源

```
  // Clean child threads
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent != curproc)
      continue;
    // I tried my best not to add anything in PCB
    // I use esp to tell whether it's process or thread
    // All process's stack is at 0x3000
    if(PGROUNDDOWN(p->tf->esp) == 0x3000)
      continue;
    // Close all open files.
    for(fd = 0; fd < NOFILE; fd++)
      if(p->ofile[fd]){
        fileclose(p->ofile[fd]);
        p->ofile[fd] = 0;
      }
    // Do not call `iput`, see `clone`
    p->cwd = 0;
    // As does in `join`
    kfree(p->kstack);
    p->kstack = 0;
    p->pid = 0;
    p->parent = 0;
    p->name[0] = 0;
    p->killed = 0;
    p->state = UNUSED;
  }
```

以上代码需要添加到 exit 函数**获取 ptable 的锁之后**，因为其中要访问 ptable。

我首先遍历所有进程，找到当前进程的子线程。我通过子进程/线程的 esp 来判断它是进程还是线程：进程的栈是固定的 0x3000，如果不是该值，就说明是子线程。在写报告时，我意识到，可以简单地判断子进程/线程的 pgdir 和当前进程的 pgdir 是否相等来判断时子进程还是子线程，我认为这是更好的方法。

子线程占用的内核资源有：打开的文件、线程的内核栈、PCB。故在代码中我依次将它们释放。注意这里并不需要释放线程的用户栈，因为整个地址空间都会在主进程退出的时候被释放。

8. 其他一些细节

clone、join、thread_exit 这三个都是有参数的系统调用，其中需要使用 argptr 辅助函数来获取指针信息。argptr 的第三个参数的含义是所获取的指针指向的内存块的大小。令人困惑的是 clone 的第一个参数函数指针 fn，和第三个参数 arg，我不知道 fn 所指的函数有多大，也不知道参数具体有多少字节，因此我将这两个 size 都设为了 0，也许用 argint 来获取更好一些。

9. 运行结果

如下所示，连续运行两遍 threadtest，结果与预期相符。

```
$ threadtest

-------------------- Test Return Value --------------------
Child thread 1: count=3
Child thread 2: count=3
Main thread: thread 1 returned 2
Main thread: thread 2 returned 3

-------------------- Test Stack Space --------------------
ptr1 - ptr2 = 16
Return value 123

-------------------- Test Thread Count --------------------
Child process created 60 threads
Parent process created 61 threads
$ threadtest

-------------------- Test Return Value --------------------
Child thread 1: count=3
Child thread 2: count=3
Main thread: thread 1 returned 2
Main thread: thread 2 returned 3

-------------------- Test Stack Space --------------------
ptr1 - ptr2 = 16
Return value 123

-------------------- Test Thread Count --------------------
Child process created 60 threads
Parent process created 61 threads
```

10. 回答问题

Question: 在你的设计中，struct proc 增加了几个字节？（优先级调度部分增加的不算）

Answer: 没有增加字节。

二、 优先级调度

“道生一，一生二，二生三，三生万物。”

——《道德经》

既然要支持三个优先级，不如做一个通用的，支持任意多个优先级的调度算法。我在proc.h中定义了以下两个宏，前者指定了优先级的数量，后者指定了默认优先级。值得注意的是，按照惯例，优先级是从0开始计数的，当PRIORITY_LEVEL为4时，可用的优先级为0、1、2、3。这样设计符合实验要求，还多了一个0级优先级，可供将来作为实时优先级。

```
// valid priority is from 0 to PRIORITY_LEVEL-1
#define PRIORITY_LEVEL 4
#define DEFAULT_PRIORITY 2
```

1. 调度函数scheduler

我调整了scheduler函数的代码结构，让它循环做以下两件事：1）按照规则找到下一个要执行的进程；2）切换到该进程。第二件事很简单，只需要参考原先的代码即可，这里主要介绍第一件事的实现方法。

```
  // Init last chosen process as the last one,
  // so that the first process is chosen first.
  int lastChosen[PRIORITY_LEVEL];
  for (int i = 0; i < PRIORITY_LEVEL; ++i)
    lastChosen[i] = NPROC - 1;
```

首先，我定义了一个数组lastChosen，表示每个优先级上次执行的进程，用下标表示。每次搜索从上次执行的进程开始往后搜索，可以保证不会每次都选到同一个进程。将它初始化为NPROC-1，表示最后一个PCB，这样第一次调度就会从第一个PCB开始搜索。

```
    // Choose a process
    struct proc *chosen = 0;
    for (int p = 0; !chosen && p < PRIORITY_LEVEL; ++p)
      for (int i = 1; !chosen && i <= NPROC; ++i)
      {
        int idx = (i + lastChosen[p]) % NPROC;
        struct proc *t = &ptable.proc[idx];
        if (t->state == RUNNABLE && t->prior == p)
        {
          chosen = t;
          lastChosen[p] = idx;
        }
      }
```

以上的代码展示了选择进程的逻辑：从最高优先级的开始，从该优先级上次选择的进程的下一个开始往后循环搜索，遍历一遍ptable，找到第一个RUNNABLE的进程。注意循环条件中有!chosen，表示如果找到了就退出两层循环。

2. 其他一些细节

a) 在proc结构体中添加优先级字段，在allocproc函数中默认初始化为DEFAULT_PRIORITY。

b) 在sched函数中判断display_enabled是否为1，若是，输出当前pid。

c) 在sys_set_priority系统调用中遍历ptable时需要加锁。

3. 运行结果

```
$ schedtest
================================
Parent (pid=3, prior=1)
Child (pid=4, prior=1) created!
Child (pid=5, prior=2) created!
Child (pid=6, prior=3) created!
Child (pid=7, prior=1) created!
Child (pid=8, prior=2) created!
Child (pid=9, prior=3) created!
Child (pid=10, prior=2) created!
================================
3 - 4 - 7 - 4 - 7 - 4 - 7 - 4 - 7 - 4 - 7 - 4 - 7 - 4 - 7 - 4 - 7 - 3 - 7 -
 3 - 5 - 8 - 10 - 5 - 8 - 10 - 5 - 8 - 10 - 5 - 8 - 10 - 5 - 8 - 10 - 5 - 8
 - 10 - 5 - 8 - 10 - 5 - 3 - 8 - 10 - 3 - 8 - 8 - 8 - 3 - 6 - 9 - 6 - 9 - 6
 - 9 - 6 - 9 - 6 - 9 - 6 - 9 - 6 - 9 - 3 - 3 - 6 -
```

4. 回答问题

Question 1: 父进程的优先级为1，为何有时优先级低的子进程会先于它执行？

Answer 1: 父进程在第60行循环调用了7次wait函数，等待子进程运行完成，此时父进程处于SLEEPING状态，不满足被调度的条件。

Question 2: 父进程似乎周期性出现在打印列表中，为什么？

Answer 2: 每当有一个子进程运行完成，就会唤醒父进程，父进程会在第59行的循环中继续调用wait，从而放弃CPU，输出pid。

Question 3: set_priority系统调用会否和scheduler函数发生竞争条件（race condition）？

Answer 3: 会。set_priority和scheduler都会访问并修改ptable。运行set_priority时可能会被时钟中断打断，进而转入scheduler。

Question 4: 你是如何解决的？

Answer 4: 在set_priority中通过acquire获取ptable的锁。acquire函数内部会禁用中断，这样就避免了set_priority持有锁时被中断转到scheduler中造成死锁。