深 圳 大 学 实 验 报 告

课程名称：操作系统

实验项目名称：实验 2 处理机调度

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：谭舜泉

报告人：黎浩然 冯海月 学号：2018112061 2018191116

实验时间：2021 年 4 月 20 日-2021 年 5 月 9 日

实验报告提交时间：2021 年 5 月 9 日

教务部制

实验目的与要求：

实验目的：

(1)、掌握计算机操作系统管理进程、处理机、存储器、文件系统的基本方法。

(2)、了解进程的创建、撤消和运行，进程并发执行；自行设计解决哲学家就餐问题的并发线程，了解线程（进程）调度方法；掌握内存空间的分配与回收的基本原理；通过模拟文件管理的工作过程，了解文件操作命令的实质。

(3)、了解现代计算机操作系统的工作原理，具有初步分析、设计操作系统的能力。

(4)、通过在计算机上编程实现操作系统中的各种管理功能，在系统程序设计能力方面得到提升。

实验要求：

(1) 题目 1：

试解释一下 yield 函数、scheduler 函数和 sched 函数的用途。

结合书本，确定 xv6 使用的是哪种调度算法。给出你的理由（通过分析代码证明你的观点）。

(proc.c L268) 有一个疑问，似乎每次 xv6 都是从进程表开头开始查找 Runnable 的进程。如果刚从 CPU 切换下来的进程恰好是进程表的第一个 PCB，会不会调度器永远都选择它进行调度？

(2) 题目 2：

在 xv6 界面上按 Ctrl+P，你会看到终端上会显示当前系统中的进程。

仿照相关代码，实现以下功能：在 xv6 界面上按 Ctrl+R，打印出当前系统中 sleeping 的进程，并确定它们对应的等待队列是什么（chan/waiting channel）。

说明：

（1）本次实验课作业满分为 100 分，占总成绩的比例（待定）。

（2）本次实验课作业截至时间 2021 年 5 月 9 日（周日）23:59。

（3）报告正文：请在指定位置填写，本次实验不需要单独提交源程序文件。

（4）个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的姓名和学号；实验报告的首页，请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”等信息。

（5）提交方式：请在 BLACKBOARD 平台中按时提交；延迟提交不得分。

（6）发现抄袭（包括复制&粘贴整句话、整张图），该次作业记零分。

（7）期末考试阶段补交无效。

题目 1：

试解释一下 yield 函数、scheduler 函数和 sched 函数的用途。

结合书本，确定 xv6 使用的是哪种调度算法。给出你的理由（通过分析代码证明你的观点）。

(proc.c L268) 有一个疑问，似乎每次 xv6 都是从进程表开头开始查找 Runnable 的进程。如果刚从 CPU 切换下来的进程恰好是进程表的第一个 PCB，会不会调度器永远都选择它进行调度？

（1） 试解释一下 yield 函数、scheduler 函数和 sched 函数的用途。

① yield 函数源码如下（proc.c L311）。

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  proc->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

作用：使一个进程在一次调用中放弃 CPU 占用。

使用了带锁的机制来避免多个 CPU 同时切换进程导致的竞争。

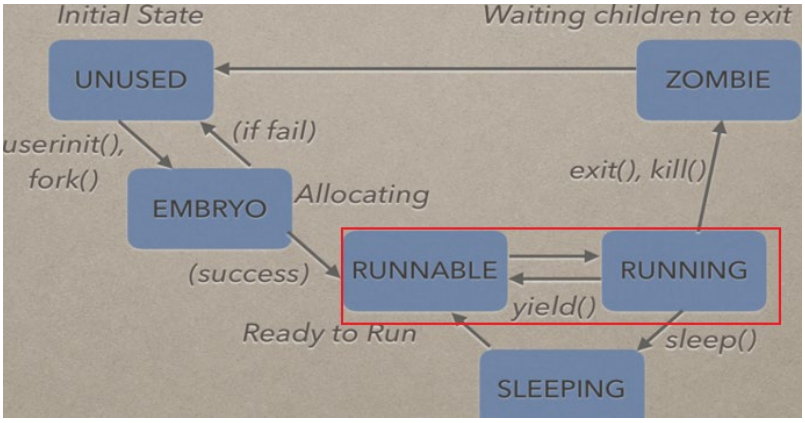
相关源码解读：

yield 在哪里被调用？

在 trap.c L185，有如下一段代码。

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

这里体现了部分 xv6 的多路复用机制——“当一个进程耗尽了它在处理器上运行的时间片（100 毫秒）后，xv6 使用时钟中断强制它停止运行，这样调度器才能调度运行其他进程”，这里就是进程状态处于 RUNNING 时，调用 yield 函数，时钟中断停止运行，然后通过 sched 函数的调用，切换上下文，运行其他进程。



Sched 函数源码如下。

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```

切换上下文

② Scheduler 函数源码如下（proc.c L257）。

在 main.c main 函数中第一个进程的状态被设置好后，mpmain 调用 scheduler 开始运行进程（main.c L62）。每个 CPU 在初始化好自身后，调用 scheduler（调度器），执行以下调度过程，永不返回，不断循环执行以下步骤：

a. 选择一个可执行的进程

b. 切换到该进程的执行

c. 执行完该进程后，通过 swtch 将控制交还给调度器，返回步骤 a.

```
//PAGEBREAK: 42
// Per-CPU process scheduler.
// Each CPU calls scheduler() after setting itself up.
// Scheduler never returns.  It loops, doing:
//  - choose a process to run
//  - swtch to start running that process
//  - eventually that process transfers control
//      via swtch back to the scheduler.
void
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;     // 切换到选定的进程
      switchuvm(p);  // 切换到该进程的页表
      p->state = RUNNING;  // 改变进程状态为RUNNING
      swtch(&cpu->scheduler, proc->context); // 切换到该进程运行
      switchkvm(); // 当没有进程在运行时，将页表寄存器切换到仅内核页表

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

③ Sched 函数源码如下，其作用主要是切换上下文（从进程的内核线程切换到当前 CPU 的调度器线程）。

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock)) // 未持有进程表锁，不应进入调度，引发内核错误
    panic("sched ptable.lock");
  if(cpu->ncli != 1)          // 处于锁状态
    panic("sched locks");
  if(proc->state == RUNNING) // 进程处于运行态
    panic("sched running");
  if(readeflags()&FL_IF)      // 开中断
    panic("sched interruptible");
  intena = cpu->intena;       // 暂存当前状态
  swtch(&proc->context, cpu->scheduler);    // 切换上下文
  cpu->intena = intena;       // 恢复原有状态
}
```

其中，体现其主要功能的 swtch 函数声明如下所示。

```
void swtch(struct context **old, struct context *new);
```

swtch 简单地保存和恢复寄存器集合，即上下文。当进程让出 CPU 时，进程的内核线程调用 swtch 来保存自己的上下文，然后返回到调度器的上下文中。

它将当前 CPU 的寄存器压入栈中并将栈指针保存在 *old 中。然后 swtch 将 new 拷贝到 %esp 中，弹出之前保存的寄存器，然后返回。具体体现在 swtch.S 汇编代码中，如下所示。

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

对于 swtch 函数的具体执行步骤，中文文档中如此描述：

swtch（2702）一开始从栈中弹出参数，放入寄存器 %eax 和 %edx（2709-2710）中；swtch 必须在改变栈指针以及无法获得 %esp 前完成这些事情。然后 swtch 压入寄存器，在当前栈上建立一个新的上下文结构。仅有被调用者保存的寄存器此时需要被保存；按照 x86 的惯例即 %ebp %ebx %esi %ebp %esp。swtch 显式地压入前四个寄存器（2713-2716）；最后一个则是在 struct context* 被写入 old（2719）时隐式地保存的。要注意，还有一个重要的寄存器，即程序计数器 %eip，该寄存器在使用 call 调用 swtch 时就保存在栈中 %ebp 之上的位置上了。保存了旧寄存器后，swtch 就准备要恢复新的寄存器了。它将指向新上下文的指针放入栈指针中（2720）。新的栈结构和旧的栈相同，因为新的上下文其实是之前某次的切换中的旧上下文。所以 swtch 就能颠倒一下保存旧上下文的顺序来恢复新上下文。它弹出 %edi %esi %ebx %ebp 然后返回（2723-2727）。由于 swtch 改变了栈指针，所以这时恢复的寄存器就是新上下文中的寄存器值。

（2） 结合书本，确定 xv6 使用的是哪种调度算法。给出你的理由（通过分析代码证明你的观点）。

使用时间片轮转法。

先来看看 scheduler 函数中的一段代码：

```
262   for(;;){
263     // Enable interrupts on this processor.
264     sti();
265
266     // Loop over process table looking for process to run.
267     acquire(&ptable.lock);
268     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
269       if(p->state != RUNNABLE)
270         continue;
271
272       // Switch to chosen process.  It is the process's job
273       // to release ptable.lock and then reacquire it
274       // before jumping back to us.
275       proc = p;
276       switchuvm(p);
277       p->state = RUNNING;
278       swtch(&cpu->scheduler, proc->context);
279       switchkvm();
280
281       // Process is done running for now.
282       // It should have changed its p->state before coming back.
283       proc = 0;
284     }
285     release(&ptable.lock);
```

从上面的代码中我们可以看到：系统循环地遍历 proc table。如果发现有进程处于 RUNNABLE 状态，那么就将被选中的进程设置为 RUNNING 状态，并且切换到对应的进程中。因为外层循环是无条件循环，所以一遍遍历完 proc table 后又会进行下一次的遍历。

通过 swtch 切换到被选中的进程的上下文执行该进程。并且，当进程从 swtch 返回之后，就会调用 switchkvm 函数切换到内核页表。然后继续 proc table 之前的位置选择下一个进程执行。

```
// Switch h/w page table register to the kernel-only page table,
// for when no process is running.
void
switchkvm(void)
{
  lcr3(v2p(kpgdir));   // switch to the kernel page table
}
```

因此推测 xv6 使用的是时间片轮转法的调度算法。

我们知道进程是轮流被调度的，那么时间是如何处理的呢？

要实现时间片轮转法这种调度算法，一是需要有计时器来计时，二是需要有一定的机制在当某个进程的时间片用完之后让进程让出 CPU。显然不可能指望进程本身会主动让出 CPU（至少我写程序的时候不会这么做），那么最终能让进程让出 CPU 的就只有 CPU 本身能做到。

CPU 需要决定在什么时候将当前的进程映像（如寄存器）保存起来。然后调用 yield 之类的函数进入到内核让调度器选择下一个待调度到进程。

```
290  // Enter scheduler.  Must hold only ptable.lock
291  // and have changed proc->state.
292  void
293  sched(void)
294  {
295    int intena;
296
297    if(!holding(&ptable.lock))
298      panic("sched ptable.lock");
299    if(cpu->ncli != 1)
300      panic("sched locks");
301    if(proc->state == RUNNING)
302      panic("sched running");
303    if(readeflags()&FL_IF)
304      panic("sched interruptible");
305    intena = cpu->intena;
306    swtch(&proc->context, cpu->scheduler);
307    cpu->intena = intena;
308  }
309
310  // Give up the CPU for one scheduling round.
311  void
312  yield(void)
313  {
314    acquire(&ptable.lock);  //DOC: yieldlock
315    proc->state = RUNNABLE;
316    sched();
317    release(&ptable.lock);
318  }
```

如果 CPU 调用 yield 函数释出当前进程，那么最终会重新回到我们上面提到的 scheduler 调度下一个进程。我们在 trap.c 中留意到如下的代码：

```
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(proc && proc->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

这段代码应该就是 CPU 调用 yield 函数让出时间片的地方了。当一个进程的时间片用完了之后，外部计时器就会发送一个中断信号给 CPU。一个与对应的中断信号对应的 trapno 会在 trapframe，而最终被在 trap.c 中被检查。如下。我们可以看到 lapiceoi 就是对中断信号的确认。

```
118  // Acknowledge interrupt.
119  void
120  lapiceoi(void)
121  {
122    if(lapic)
123      lapicw(EOI, 0);
124  }
```

我们一直沿着调用链往上追溯：

```
3     # vectors.S sends all traps here.
4   .globl alltraps
5   alltraps:
6     # Build trap frame.
7     pushl %ds
8     pushl %es
9     pushl %fs
10    pushl %gs
11    pushal
12
13    # Set up data and per-cpu segments.
14    movw $(SEG_KDATA<<3), %ax
15    movw %ax, %ds
16    movw %ax, %es
17    movw $(SEG_KCPU<<3), %ax
18    movw %ax, %fs
19    movw %ax, %gs
20
21    # Call trap(tf), where tf=%esp
22    pushl %esp
23    call trap
24    addl $4, %esp
```

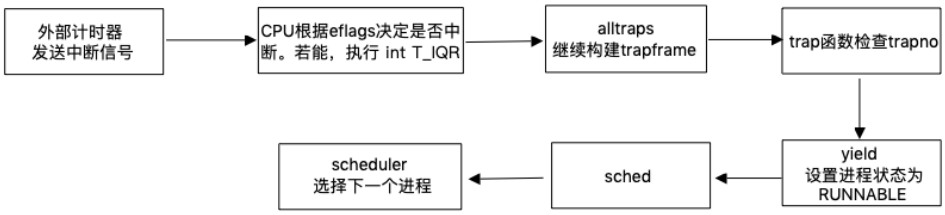
在进入 trap 函数之前，alltraps 会继续构建 trapframe。也就是我们在 trap 函数中要用到的结构体。将一些状态寄存器、段寄存器等保存到 tramframe 中，最终准备 trapframe 的指针作为参数进入到 trap（这取决于 xv6 的函数调用规范）。当然 trapno 在 alltraps 之前就已经准备好了，它仍然在 stack 中，属于 trapframe 到一部分。

我们在 trapno 的宏定义的注释中可以看到，CPU 在接受到外部计时器到中断信号后，应该是通过 int T_IRQ 中进入到 alltraps 的。

```
29
30  #define T_IRQ0          32      // IRQ 0 corresponds to int T_IRQ
31
```

不过我使用 Xcode 的字符串搜索工具未能找到这一段汇编指令。

因此硬件终端引起的进程上下文切换调用链如下：




```
52  void
53  lapicinit(void)
54  {
55    if(!lapic)
56      return;
57
58    // Enable local APIC; set spurious interrupt vector.
59    lapicw(SVR, ENABLE | (T_IRQ0 + IRQ_SPURIOUS));
60
61    // The timer repeatedly counts down at bus frequency
62    // from lapic[TICR] and then issues an interrupt.
63    // If xv6 cared more about precise timekeeping,
64    // TICR would be calibrated using an external time source.
65    lapicw(TDCR, X1);
66    lapicw(TIMER, PERIODIC | (T_IRQ0 + IRQ_TIMER));
67    lapicw(TICR, 10000000);
```

计时器及内部中断信号的处理和管理由 Local APIC 管理，我们可以在 lapicinit 看到对 TIMER 的初始化代码。由注释可以看到，计时器根据总线频率来从某个数值倒计时(counts down)。然后会发出一个中断信号。从宏定义可以看出，这个信号应该就是我们前面提到的计时器外部中断信号。

因此 xv6 的就是使用的时间片轮转法。

（3） (proc.c L268) 有一个疑问，似乎每次 xv6 都是从进程表开头开始查找 Runnable 的进程。如果刚从 CPU 切换下来的进程恰好是进程表的第一个 PCB，会不会调度器永远都选择它进行调度？

并不会，如下：

```
for(;;){
  // Enable interrupts on this processor.
  sti();

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;

    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&cpu->scheduler, proc->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state before coming back.
    proc = 0;
  }
  release(&ptable.lock);

}
```

决定了从 swtch 函数最终就是进程返回内核的 scheduler 的地方。这个时候指针 p 的值和离开 scheduler 进入进程的时候是一样的。因此会继续离开前指针所指向的条目继续搜索进程，而不是进程表的开头进行查找。

题目 2：

在 xv6 界面上按 Ctrl+P，你会看到终端上会显示当前系统中的进程。

仿照相关代码，实现以下功能：在 xv6 界面上按 Ctrl+R，打印出当前系统中 sleeping 的进程，并确定它们对应的等待队列是什么（chan/waiting channel）。

按 Ctrl+P 查看进程信息，打印如下。

```
$ 1 sleep  init 80104804 801045e9 80105ec2 80105106 801062a4 80106095
2 sleep  sh 80104804 80100987 80101dc2 80101160 801052e0 80105106 801062a4 80106095
```

进程号　进程状态　进程名称　当前调用栈

查看源码对应实现过程：

```
//PAGEBREAK: 36
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void
procdump(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  int i;
  struct proc *p;
  char *state;
  uint pc[10];

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    cprintf("%d %s %s", p->pid, state, p->name);
    if(p->state == SLEEPING){
      getcallerpcs((uint*)p->context->ebp+2, pc);
      for(i=0; i<10 && pc[i] != 0; i++)
        cprintf(" %p", pc[i]);
    }
    cprintf("\n");
  }
}
```

列出所有进程

按序打印--进程的pid, state, name

如果进程的状态是sleep，打印当前调用栈。

下面查看 getcallerpcs 函数是如何得到进程调用栈的。

```
// Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}
```

函数作用：打印出%ebp链，即栈的轨迹

循环打印出10个

查看进程对上下文的定义，如下。这可以解释上图的 ebp[0]和 ebp[1]的来源。

%ebp 一般指向当前进程的栈底，%eip 存放下一个执行的内存地址。

```
//PAGEBREAK: 17
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom of the stack they
// describe; the stack pointer is the address of the context.
// The layout of the context matches the layout of the stack in swtch.S
// at the "Switch stacks" comment. Switch doesn't save eip explicitly,
// but it is on the stack and allocproc() manipulates it.
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

对应ebp[0]

对应ebp[1]

下一步，我们查看控制台是如何调用 procdump 函数来实现终端的进程信息输出的。

Console.c 的 consoleintr 函数调用了 procdump()，如下。

```
void
consoleintr(int (*getc)(void))
{
  int c;

  acquire(&input.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      procdump();
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
```

Ctrl+P执行流

这提示我们可以通过增加一项“R”来实现系统 sleeping 进程的打印，并确定他们的等待队列。对于 sleeping 进程的打印，只需遍历一次进程表打印出状态为 SLEEPING 的进程即可。而为了确定等待队列，我们在 proc 中发现了 chan 的定义，是一个无类型指针（void*）。

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

对于每一个进程，如果 chan 不为 0，则该进程处于睡眠状态并被挂载在 chan 起始的地址上。因此我们只需要打印出 chan 对应的内存地址。

实现如下。

修改的部分用红色标明

```
void
consoleintr(int (*getc)(void))
{
  int c;

  acquire(&input.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      procdump();
      break;
    // modified on 5/3/2021
    case C('R'):
      procprintsleep();
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
```

console.c

```
// Modified on 5/3/2021
// Print any process that is sleeping.
// And their corresponding waiting channel.
// Runs when user types ^R on console.
void procprintsleep(void)
{
  static char *states[] = {
  [UNUSED]    "unused",
  [EMBRYO]    "embryo",
  [SLEEPING]  "sleep ",
  [RUNNABLE]  "runble",
  [RUNNING]   "run   ",
  [ZOMBIE]    "zombie"
  };
  struct proc *p;
  char *state;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state == SLEEPING){
      state = states[p->state];
      cprintf("%d %s %s waiting channel:%x\n", p->pid, state, p->name, p->chan);
    }
  }
}
```

打印进程信息以及对应的waiting channel

proc.c

在 make 之前，要记得在 defs.h 中添加声明。

```
//PAGEBREAK: 16
// proc.c
struct proc*     copyproc(struct proc*);
void             exit(void);
int              fork(void);
int              growproc(int);
int              kill(int);
void             pinit(void);
void             procdump(void);
void             scheduler(void) __attribute__((noreturn));
void             sched(void);
void             sleep(void*, struct spinlock*);
void             userinit(void);
int              wait(void);
void             wakeup(void*);
void             yield(void);
void             procprintsleep(void); // Modified on 5/3/2021
```

运行结果如下。

```
$ 1 sleep  init waiting channel:8010ff54
2 sleep  sh waiting channel:8010de54

$
```

除此之外，我们进一步探索了 chan 在 sleep 函数和 wakeup 函数中的作用。

将进程设置为 SLEEPING 状态并挂载在 chan 的代码位于 proc.c 的 sleep 函数中，如下。

```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

将进程挂载在chan上，
并设置状态为SLEEPING，
切换进程上下文

睡眠结束，将chan置为0

Wakeup 函数唤醒所有挂在 chan 上的进程，代码如下。

```
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

chan是一个内存地址

唤醒挂在特定chan上的进程

++

其他（例如感想、建议等等）。

通过这次实验，加深了我们对 xv6 的进程管理、调度算法的理解。这次实验也使得我们对相关的函例如 yield、sheduler 和 sched 等函数的实验的理解。最后通过编写程序让我们更加熟悉了 xv6 环境下开发应用程序，以及 xv6 的信号机制的理解。

指导教师批阅意见：   成绩评定：   指导教师签字：谭舜泉 2021 年 5 月 16 日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。