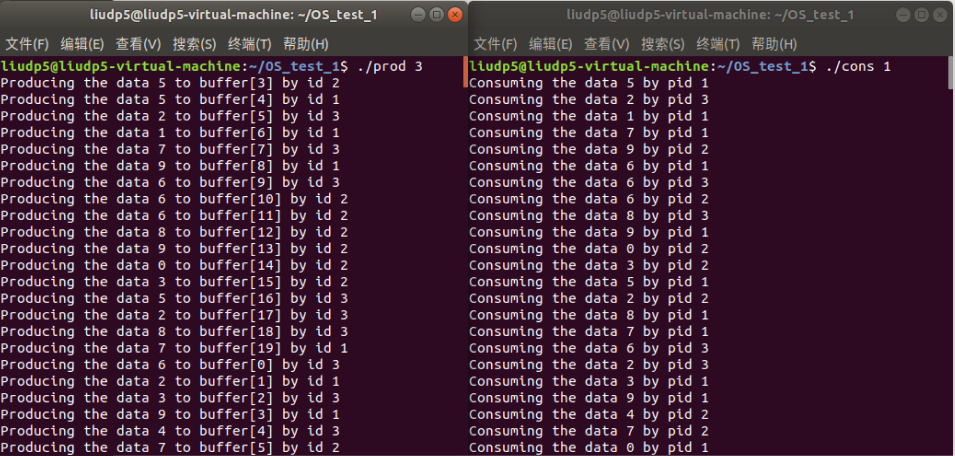
**操作系统第一次大作业**

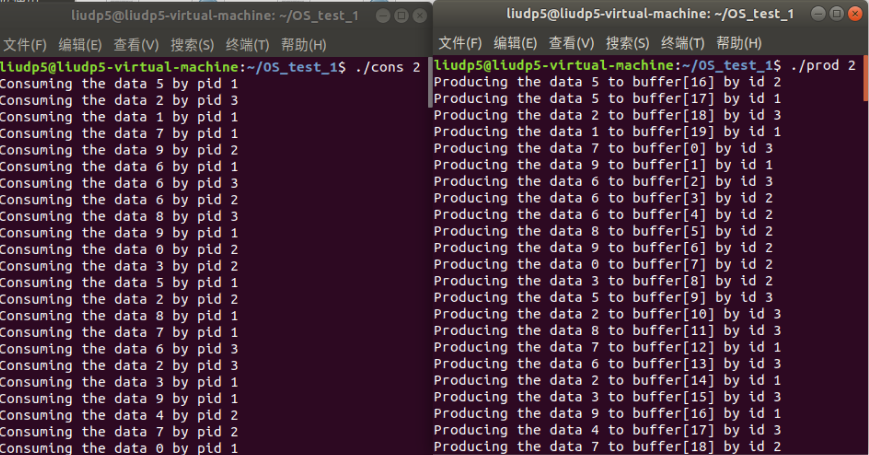
智科一班 18364059 刘德鹏

1.

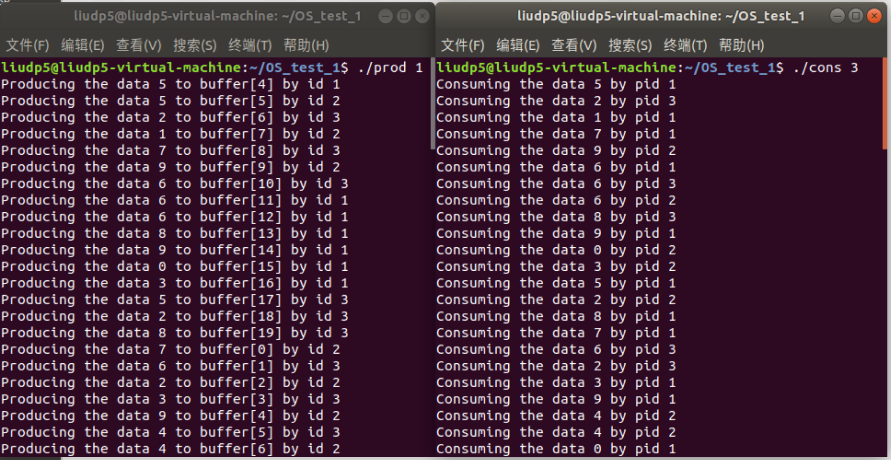
①λp = 3, λc = 1



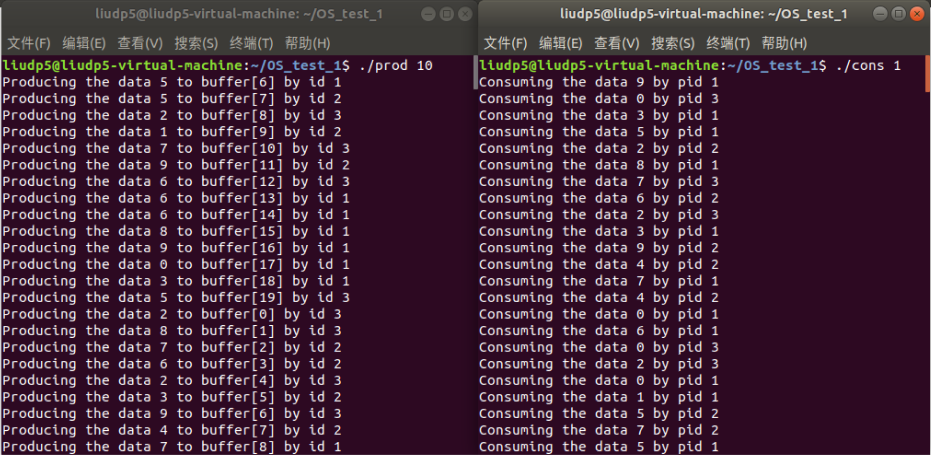
②λp = 2, λc = 2



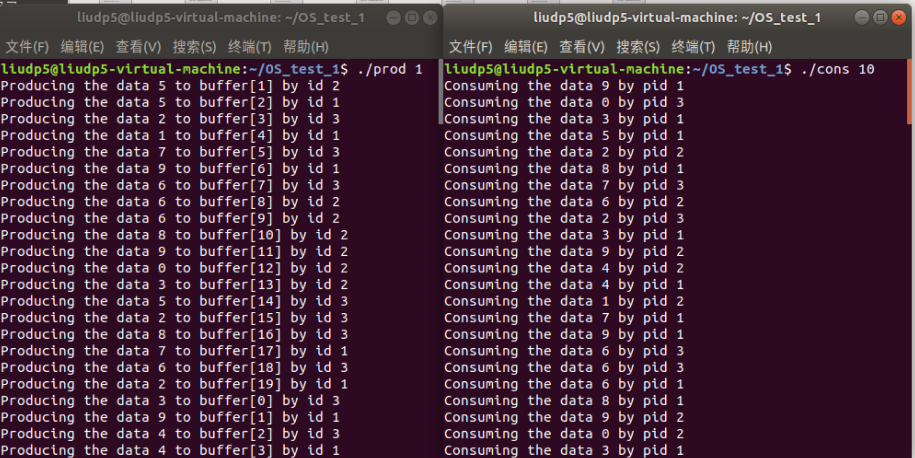
③λp = 1, λc = 3



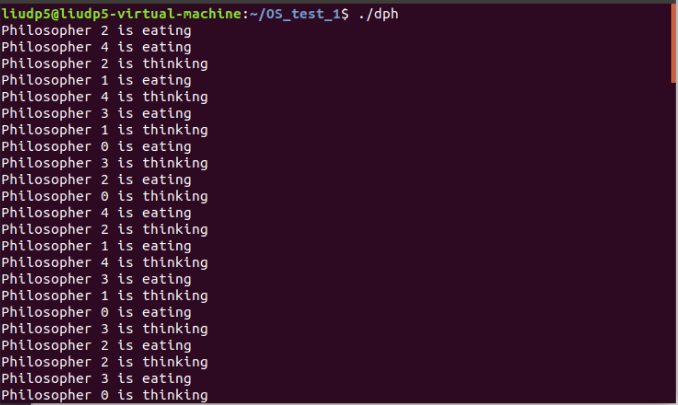
④λp = 10, λc = 1



⑤λp = 1, λc = 10



2.哲学家状态如图（部分截图）



3.1

**Sleep.c代码**

#include "types.h"

#include "user.h"

int main(int argc, char \*argv[]){

if(argc != 2)

write(2, "Error Message", strlen("Error Message"));

int x = atoi(argv[1]);

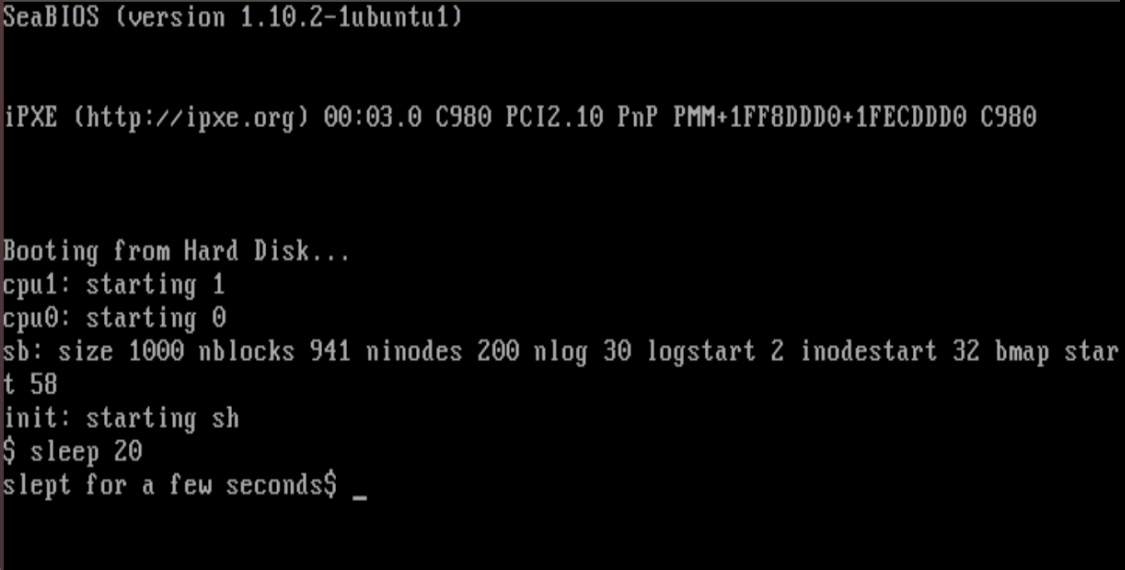
sleep(x);

write(2, "slept for a few seconds", strlen("slept for a few seconds"));

exit();

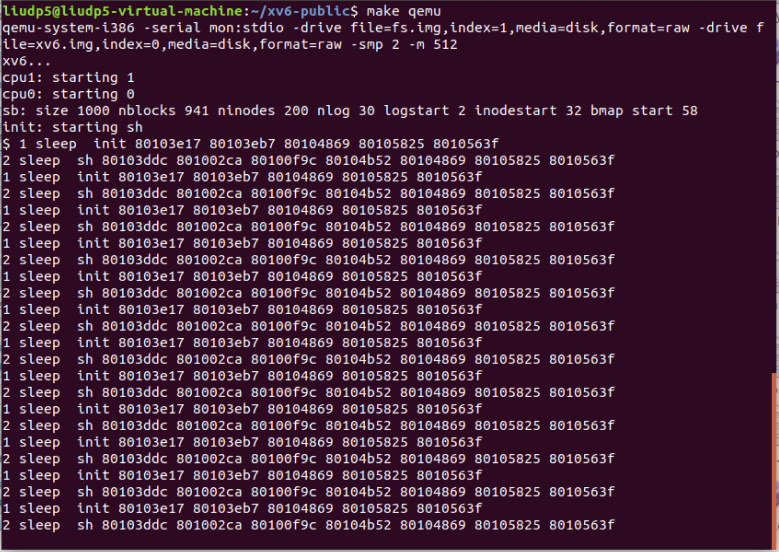
}

**Sleep运行截图**



3.2

①



②

Swtch.S

# Context switch

#

# void swtch(struct context \*\*old, struct context \*new);

#

# Save the current registers on the stack, creating

# a struct context, and save its address in \*old.

# Switch stacks to new and pop previously-saved registers.

.globl swtch

swtch:

movl 4(%esp), %eax //将寄存器eax指向栈内context\*old

movl 8(%esp), %edx //将寄存器edx指向栈内context new

# Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

pushl %edi //以上将有关上下文信息的寄存器依次push到栈中，完成对一个context的保存，此时，栈顶指针esp同时也指向context的第一个元素

# Switch stacks

movl %esp, (%eax) //保存原栈的栈顶指针，old\_proc->contex = esp;

movl %edx, %esp //新栈的栈顶指针new\_proc->contex = edx; 通过esp = edx切换栈

# Load new callee-saved registers

popl %edi

popl %esi

popl %ebx

popl %ebp //以上将new\_contex中保存的寄存器值依次弹回到对应寄存器中

ret //弹回eip，继续执行swtch函数后的下一条语句

**scheduler(void)**

void

scheduler(void)

{

struct proc \*p;

struct cpu \*c = mycpu();

c->proc = 0;

for(;;){ //死循环，持续等待/寻找可调度进程

// Enable interrupts on this processor.

sti(); //设置中断标志为1

// Loop over process table looking for process to run.

acquire(&ptable.lock); //获取锁，使以下操作成为原子操作

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //遍历进程列表

if(p->state != RUNNABLE)

continue; //跳过当前无法开始运行的进程

// Switch to chosen process. It is the process's job

// to release ptable.lock and then reacquire it

// before jumping back to us.

c->proc = p; //选取到一个可执行进程

switchuvm(p); //从用户端读取该进程的信息

p->state = RUNNING; //将当前进程状态标记为RUNNING

swtch(&(c->scheduler), p->context); //调用swtch()函数，将原栈上的内容暂存，立即开始运行当前进程

switchkvm(); //进程运行结束，内核读取内存

// Process is done running for now.

// It should have changed its p->state before coming back.

c->proc = 0; //当前无进程待执行

}

release(&ptable.lock); //释放锁

}

}

**sched(void)**

void

sched(void)

{

int intena; //声明一个变量intena

struct proc \*p = myproc();

if(!holding(&ptable.lock))

panic("sched ptable.lock"); //若当前未持有进程表锁，中止，并输出sched ptable.lock

if(mycpu()->ncli != 1)

panic("sched locks"); //若当前未关闭cpu中断，中止，并输出sched locks

if(p->state == RUNNING)

panic("sched running"); //若当前进程状态仍为RUNNING，中止，并输出sched running

if(readeflags()&FL\_IF)

panic("sched interruptible"); //判断当前进程是否可中断

intena = mycpu()->intena;

swtch(&p->context, mycpu()->scheduler); //上下文切换至scheduler

mycpu()->intena = intena;

}

**yield(void)**

void

yield(void)

{

acquire(&ptable.lock); //DOC: yieldlock //获取进程表锁

myproc()->state = RUNNABLE; //将进程状态设为可运行，以便下次遍历时可以被唤醒

sched(); //执行sched函数，准备将CPU切换到scheduler context

release(&ptable.lock); //释放进程表锁

}

③**Xv6进程调度**

综合swtch.S，scheduler(void)，sched(void)，yield(void)等函数，可以判断xv6操作系统使用的是轮转调度（RR）算法，类似于FCFS调度，但增加了抢占以切换进程。将一个较小时间单位定义为时间量，就绪队列作为循环队列。CPU调度程序循环整个就绪队列，为每个进程分配不超过一个时间片的CPU。

**Linux CFS进程调度**

基于调度类，通过每个任务的变量vruntime维护/记录虚拟运行时间，从而为每个进程定义一个友好值nice value，友好值越低，分配到的CPU运行时间越多。这种调度方式红黑树结构，能迅速（O（lgN））找到最左侧节点，亦即由最高优先级的任务。

**Xv6调度算法的不足之处：**

a.缺少优先级排序，导致紧迫任务响应慢，效率低

b.线程在其生命周期内多次更换处理器使得高速缓存的使用率很低

**改进方案：**在保留轮转机制的前提下，增加类似SJF的优先级定义，保证优先级高的进程任务得到优先处理，同时轮转机制又能在一定程度上解决SJF算法带来的无限阻塞与饥饿问题，使低优先级的进程任务也能在不长的等待时间内得到CPU分配。