

操作系统大作业 1 实验报告

1. 生产者消费者 (producer-consumer) 问题:

生产者进程 prod.c 中核心部分:

缓冲区结构体 buf 与信号量定义:

```
typedef int buffer_item;
#define BUFFER_SIZE 20
struct buf{
    int rear;
    int front;
    buffer_item buffer[BUFFER_SIZE];
};

//producer
void *producer();
void *ptr;
double produce_time(double lambda_p);
sem_t *full;
sem_t *empty;
sem_t *s_mutex;
```

共享内存开辟与信号量初始化:

```
full = sem_open("full",O_CREAT,0666,0); //sem_open returns a sem_t pointer
empty = sem_open("empty",O_CREAT,0666,0);
s_mutex = sem_open("mutex",O_CREAT,0666,0);
sem_init(full,1,0); //第二个参数1表示共享信号量,第三个为初始值
sem_init(empty,1,BUFFER_SIZE);
sem_init(s_mutex,1,1);

lambda_p = atof(argv[1]);

int shm_fd = shm_open("buffer",O_CREAT | O_RDWR,0666); //创建共享内存
ftruncate(shm_fd,4096);

ptr = mmap(0,sizeof(struct buf),PROT_WRITE,MAP_SHARED,shm_fd,0);

struct buf *temp=((struct buf*)ptr); //将结构体指针初始化
temp->front=0;
temp->rear=0;
```

生产者函数:

```
void *producer(){
    do{
        double interval_time = lambda_p;
        //printf("sss");
        usleep((unsigned int)(produce_time(interval_time)*1e6)); // sleep
        buffer_item item = rand() % 260;
        struct buf *shm_ptr = ((struct buf *)ptr); // read the round queue's information from shared memory.
        sem_wait(empty); //缓冲区满则上锁
        sem_wait(s_mutex); //锁定二进制互斥锁s_mutex, 并在关键区域执行代码
        printf("PID: %d Producing the data %d to buffer[%d] by id %ld\n",getpid(),item,shm_ptr->rear,pthread_self());
        shm_ptr->buffer[shm_ptr->rear] = item; // Put the data to round queue.
        shm_ptr->rear = (shm_ptr->rear+1) % BUFFER_SIZE;
        sem_post(s_mutex); //Unlock s_mutex
        sem_post(full); // full+1
    }while(1);
    pthread_exit(0);
}
```

类似，消费者函数：

```
void *consumer(){
    do{
        double interval_time = Lambda_c;
        sleep((unsigned int)produce_time(interval_time));
        struct buf *shm_ptr = ((struct buf *)ptr);
        sem_wait(full); //Wait for a full buffer
        sem_wait(s_mutex);

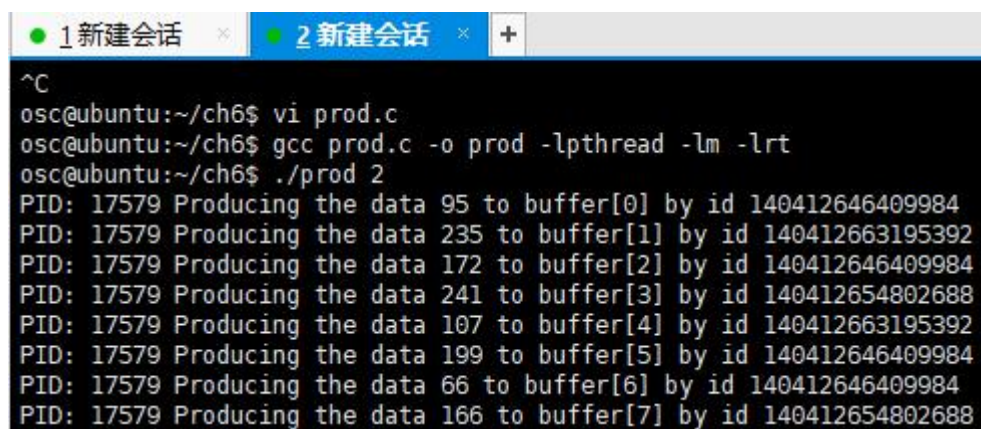
        buffer_item item = shm_ptr->buffer[shm_ptr->front];
        printf("PID: %d Consuming the data %d from buffer[%d] by pid %ld\n", getpid(), item, shm_ptr->front, pthread_self());
        shm_ptr->front = (shm_ptr->front+1) % BUFFER_SIZE;
        sem_post(s_mutex);
        sem_post(empty); //empty+1
    }while (1);
    pthread_exit(0);
}
```

两者共用的控制生产时间函数：

```
double produce_time(double lambda_c){
    double z;
    do
    {
        z = ((double)rand() / RAND_MAX);
    }
    while((z==0) || (z == 1));
    return (-1/lambda_c * log(z));
}
```

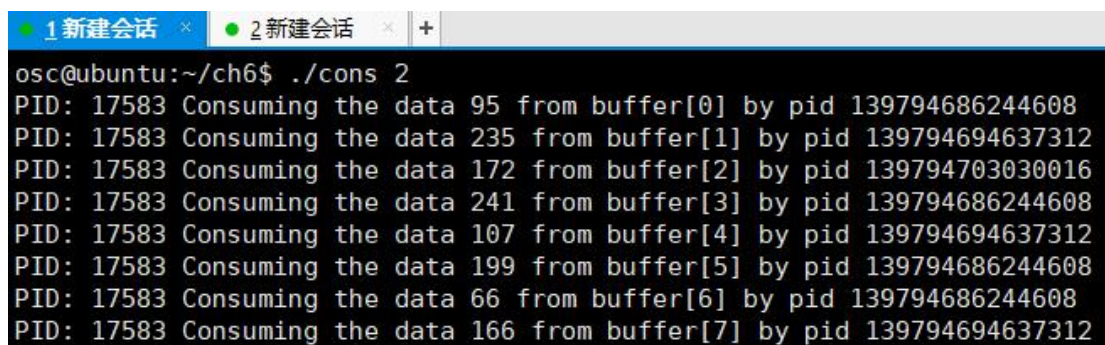
运行测试结果：

生产者：



```
^C
osc@ubuntu:~/ch6$ vi prod.c
osc@ubuntu:~/ch6$ gcc prod.c -o prod -lpthread -lm -lrt
osc@ubuntu:~/ch6$ ./prod 2
PID: 17579 Producing the data 95 to buffer[0] by id 140412646409984
PID: 17579 Producing the data 235 to buffer[1] by id 140412663195392
PID: 17579 Producing the data 172 to buffer[2] by id 140412646409984
PID: 17579 Producing the data 241 to buffer[3] by id 140412654802688
PID: 17579 Producing the data 107 to buffer[4] by id 140412663195392
PID: 17579 Producing the data 199 to buffer[5] by id 140412646409984
PID: 17579 Producing the data 66 to buffer[6] by id 140412646409984
PID: 17579 Producing the data 166 to buffer[7] by id 140412654802688
```

消费者：



```
osc@ubuntu:~/ch6$ ./cons 2
PID: 17583 Consuming the data 95 from buffer[0] by pid 139794686244608
PID: 17583 Consuming the data 235 from buffer[1] by pid 139794694637312
PID: 17583 Consuming the data 172 from buffer[2] by pid 139794703030016
PID: 17583 Consuming the data 241 from buffer[3] by pid 139794686244608
PID: 17583 Consuming the data 107 from buffer[4] by pid 139794694637312
PID: 17583 Consuming the data 199 from buffer[5] by pid 139794686244608
PID: 17583 Consuming the data 66 from buffer[6] by pid 139794686244608
PID: 17583 Consuming the data 166 from buffer[7] by pid 139794694637312
```

2. 哲学家就餐问题

只有当其两根筷子都可用时才可拿起筷子

```
void *philo(void *param){
    do{
        int id = *( (int *)param);
        /* Try to pickup a chopstick*/
        pickup_forks(id);
        printf("The philosopher %d is eating...\n",id);
        /* Eat a while*/
        srand((unsigned)time(NULL));
        int sec = (rand()%((3-1)+1)) +1; // 将sec控制在[1,3]
        sleep(sec);
        /* Return a chopstick */
        return_forks(id);
        printf("The philosopher %d is thinking...\n",id);

        srand((unsigned)time(NULL)); //思考一定的时间
        sec = (rand()%((3-1)+1)) +1;
        sleep(sec);
    }while(TRUE);
    pthread_exit(NULL);
} « end philo »
```

其中使用到主要的函数:

```
void pickup_forks(int i){
    state[i] = HUNGRY;
    test(i); // 检查是否正在吃饭
    pthread_mutex_lock(&mutex[i]);
    while (state[i] != EATING){
        pthread_cond_wait(&self[i],&mutex[i]); //等待相邻吃完
    }
    pthread_mutex_unlock(&mutex[i]);
}

void return_forks(int i){
    state[i] = THINKING;
    //告诉相邻哲学家正在吃
    test((i+4)%5);
    test((i+1)%5);
}

void test(int i){
    //个哲学家可以在他想吃的时候吃，同时他的邻居们没有在吃。
    if ( (state[(i+4)%5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i+1)%5] != EATING)
    ){
        pthread_mutex_lock(&mutex[i]);
        state[i] = EATING;
        pthread_cond_signal(&self[i]);
        pthread_mutex_unlock(&mutex[i]);
    }
}
```

编写 Makefile:


```

C=gcc
CFLAGS=-Wall
RT=-lrt
POSIXT=-lpthread
MATH=-lm
all:
    $(CC) $(CFLAGS) prod.c -o prod $(RT) $(POSIXT) $(MATH)
    $(CC) $(CFLAGS) cons.c -o cons $(RT) $(POSIXT) $(MATH)
    $(CC) $(CFLAGS) dph.c -o dph $(RT) $(POSIXT)

prod:
    $(CC) $(CFLAGS) prod.c -o prod $(RT) $(POSIXT) $(MATH)

cons:
    $(CC) $(CFLAGS) cons.c -o cons $(RT) $(POSIXT) $(MATH)

dph:
    $(CC) $(CFLAGS) dph.c -o dph $(RT) $(POSIXT)

clean:
    rm -rf prod
    rm -rf cons
    rm -rf dph

```

运行结果:

因为已经生成过 dph.o 所以 make dph 未产生变化

```

osc@ubuntu:~/ch6$ make dph
make: 'dph' is up to date.
osc@ubuntu:~/ch6$ ./dph
The philosopher 4 is eating...
The philosopher 2 is eating...
The philosopher 4 is thinking...
The philosopher 0 is eating...
The philosopher 2 is thinking...
The philosopher 3 is eating...
The philosopher 3 is thinking...
The philosopher 2 is eating...
The philosopher 4 is eating...
The philosopher 0 is thinking...
The philosopher 2 is thinking...
The philosopher 1 is eating...
The philosopher 4 is thinking...
The philosopher 3 is eating...
The philosopher 3 is thinking...
The philosopher 4 is eating...
The philosopher 1 is thinking...
The philosopher 2 is eating...
^C
osc@ubuntu:~/ch6$

```

make all 结果:

```

osc@ubuntu:~/ch6$ make all
cc -Wall prod.c -o prod -lrt -lpthread -lm
cc -Wall cons.c -o cons -lrt -lpthread -lm
cc -Wall dph.c -o dph -lrt -lpthread

```

3. MIT 6.S081 课程实验

1. 完成 sleep(easy):

sleep.c 函数:

```

zxk@zxk-virtual-machine: ~/XV6/xv6-riscv/user
#include "kernel/types.h"
#include "user/user.h"

int main(int argc, char* argv[]){
    if(argc<2){
        printf("No arguments!!!\n");
        exit(0);
    }

    int n = atoi(argv[1]);
    if(n>0){
        sleep(n);}
    else{
        printf("invalid time:%d\n",argv[1]);
    }

    exit(0);
}
~
~
19,16 全部

```

在 Makefile 文件中添加 sleep 部分后运行结果如下:

输入 make qemu 后: 进行 sleep 函数的运行测试

```

?
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ sleep 10
$
其他位置
kernel.sym  kernelvec.o  kernelvec.S  log.c  log.d

```

2. (a)

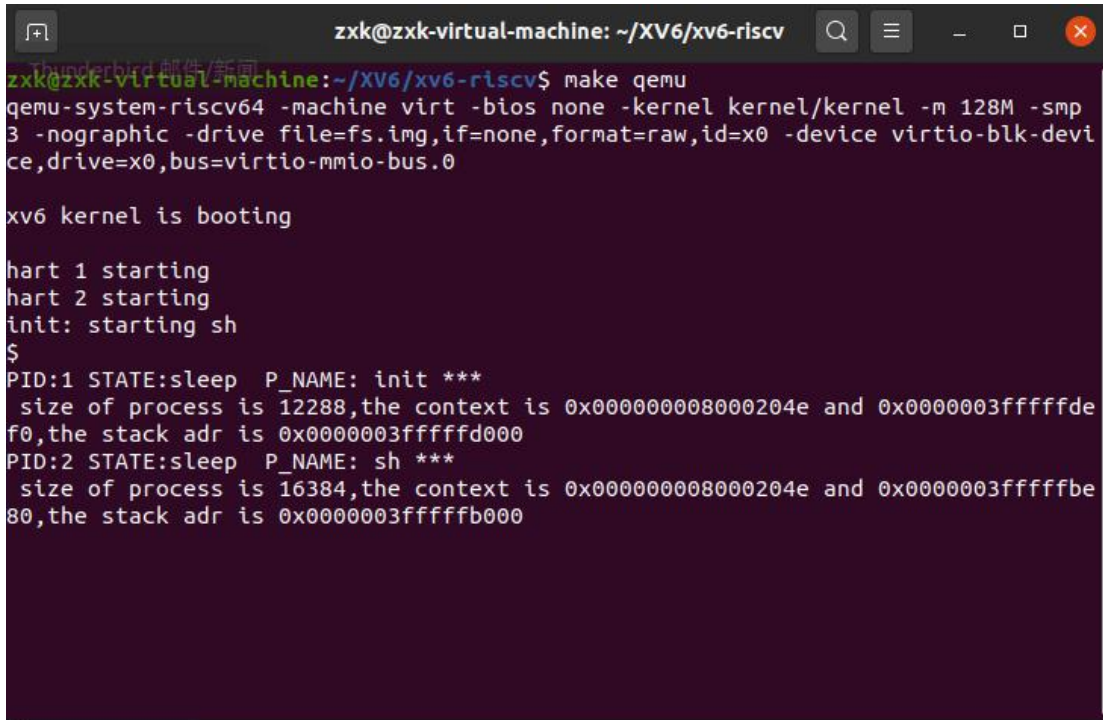
修改 prodump 函数:

```

printf("PID:%d STATE:%s P_NAME: %s ***\n", p->pid, state, p->name);
printf(" size of process is %d,the context is %p and %p,the stack adr is %p",p->sz,p->context.ra,p->context.sp,p->kstack);
printf("\n");
}

```

Ctrl+p 后打印结果如下



```

zxc@zxc-virtual-machine: ~/XV6/xv6-riscv
zxc@zxc-virtual-machine:~/XV6/xv6-riscv$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-devi
ce,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
PID:1 STATE:sleep P_NAME: init ***
size of process is 12288,the context is 0x000000008000204e and 0x0000003fffffd0
f0,the stack adr is 0x0000003fffffd000
PID:2 STATE:sleep P_NAME: sh ***
size of process is 16384,the context is 0x000000008000204e and 0x0000003fffffb0
80,the stack adr is 0x0000003fffffb000

```

(b)

Swch. S:

```

.globl switch
|switch:
    sd ra, 0(a0)      #uint64 通过八字节保存, 将old_context返回地址指针ra, 栈顶指针sp放入寄存器a0中
    sd sp, 8(a0)
    sd s0, 16(a0)     #将old_context文本内容s0~s11依次保存寄存器a0中
    sd s1, 24(a0)
    sd s2, 32(a0)
    sd s3, 40(a0)
    sd s4, 48(a0)
    sd s5, 56(a0)
    sd s6, 64(a0)
    sd s7, 72(a0)
    sd s8, 80(a0)
    sd s9, 88(a0)
    sd s10, 96(a0)
    sd s11, 104(a0)

    ld ra, 0(a1)      #将a1中新_context寄存器内容加载
    ld sp, 8(a1)      #new_context内容包括ra, sp与s0~s11依次加载
    ld s0, 16(a1)
    ld s1, 24(a1)
    ld s2, 32(a1)
    ld s3, 40(a1)
    ld s4, 48(a1)
    ld s5, 56(a1)
    ld s6, 64(a1)
    ld s7, 72(a1)
    ld s8, 80(a1)
    ld s9, 88(a1)
    ld s10, 96(a1)
    ld s11, 104(a1)

    ret #将执行switch前保存的ra设置为pc, 新进程从上一条执行switch函数的下一条指令开始运行

```

Scheduler 函数:

```

void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // 打开中断, 防止当所有进程都在等待IO时, 由于关闭中断而产生的死锁问题。
        intr_on();

        // Loop over process table looking for process to run.
        // 为进程加锁, 从进程表中寻找进程执行, 防止其他CPU更改其进程表
        // 同时防止CPU闲置时, 由于当前CPU一直占有锁, 其他CPU无法调度运行导致的死锁,
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            c->proc = p; // 找到runnable进程后进入运行将进程状态变为running
            p->state = RUNNING;

            swtch(&c->scheduler, p->context); // swtch 把当前的硬件寄存器保存在 per-cpu 的存储中, 处理器已经运行在进程 p 的内核栈上了, 返回控制权到scheduler函数
            // Process is done running for now. It should have changed its p->state before coming back.
            c->proc = 0;
        }
        release(&ptable.lock); // 释放进程锁
    }
}
// end for ;;
// end scheduler

```

sched 与 yield 函数:

```

void
sched(void)
{
    int intena;
    struct proc *p = myproc();

    if(!holding(&p->lock)) // 检查进程是否持有锁
        panic("sched p->lock");
    if(mycpu()->noff != 1) // 是否执行过pushcli
        panic("sched locks");
    if(p->state == RUNNING) // 执行的程序是否处于结束或者睡眠状态
        panic("sched running");
    if(intr_get()) // 判断中断是否可以关闭
        panic("sched interruptible");

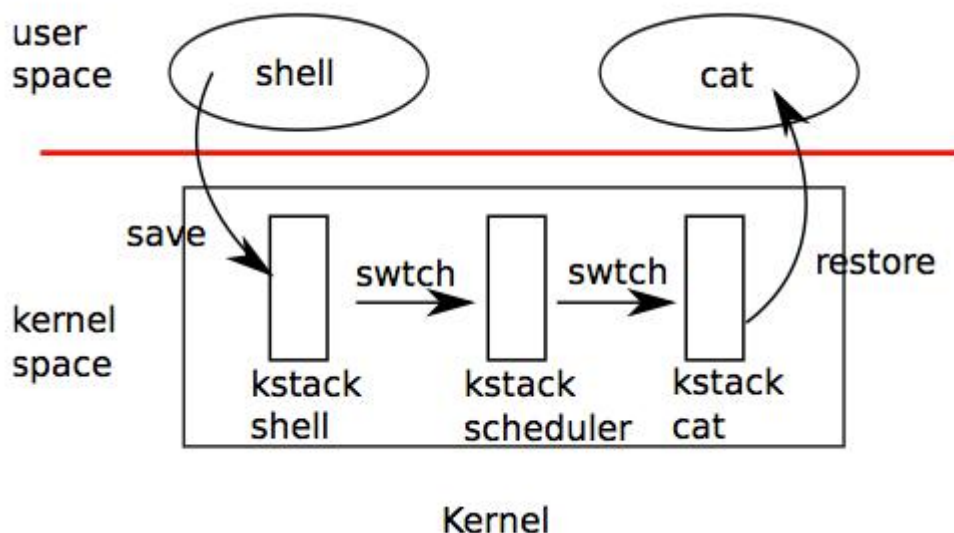
    intena = mycpu()->intena;
    swtch(&p->context, &mycpu()->context); // sched通过调用swtch切换线程
    mycpu()->intena = intena;
}

// Give up the CPU for one scheduling round.
void
yield(void) // 当进程通过调用yield进行切换时
{
    acquire(&ptable.lock); // DOC: 将当前进程上锁, 改变当前进程状态, 调用sched函数
    myproc()->state = RUNNABLE;
    sched();
    release(&ptable.lock);
}

// A fork child's very first scheduling by scheduler()
// will swtch here. "Return" to user space.

```


xv6 的进程调度框架：xv6 不会直接从用户态进程切换到另一个用户态进程；这种切换是通过用户态-内核态切换（系统调用或中断）、切换到调度器、切换到新进程的内核线程、最后这个线程返回实现的。



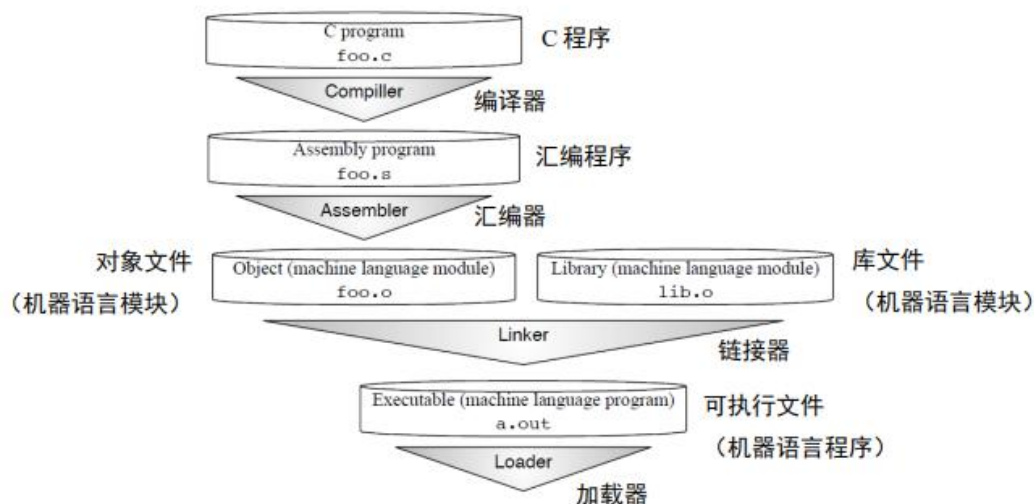
xv6 的进程调度算法中：当 CPU 初始化之后，即调用 `scheduler()`，循环从进程队列中选择一个进程执行；当进程结束时，将控制权通过 `swtch()` 移交给 `scheduler`。在每次 Loop 之后，都要及时释放进程表锁，这样可以避免当进程表中暂时没有可以运行的程序时，进程表会一直被该 CPU 锁死，其他 CPU 便不能访问。其中一种情况是，当进程等待 I/O 时，不是 `RUNNABLE` 的，而 CPU 处于 `idle` 状态，一直在占有进程表锁，I/O 信号无法到达。

`sched()` 切换至 CPU context，在切换 context 之前，进行一系列判断，以避免出现冲突。`yield()` 函数将 CPU 主动让出一个调度周期 (scheduling round)，这个函数在 xv6 的当前版本中，仅在 `trap()` 中调用。实际应用在于当一个进程正在使用 CPU，同时中断处于打开状态。`sleep` 和 `wakeup` 是两个互补的函数，共同作用实现改变进程执行顺序

每个 xv6 进程都有自己的内核栈以及寄存器集合。每个 CPU 都有一个单独的调度器线程，这样调度就不会发生在进程的内核线程中，而是在此调度器线程中。线程的切换涉及到了保存旧线程的 CPU 寄存器，恢复新线程之前保存的寄存器；CPU 会切换运行栈与运行代码。

通过阅读代码我们可以看到 xv6 中进程调度最底层 `swtch` 函数便是将文本信息放入寄存器中，而同时也是最重要的与硬件直接接触的部分，而我们阅读的相关汇编指令为 RISC-V 指令集，在不同的指令集架构中不同的指令集会给为操作系统带来不同的性能及其他特性。我们在网上见到的大部分 xv6 代码都采用了 x86 指令集。

其中 C 程序翻译成为可以在计算机上执行的机器语言程序的四个经典步骤如下图所示：



而 riscv 指令在很大程度上对之前的 arm32, x86-32, MIPS-32 指令集进行了优化, 从成本, 简洁性, 性能, 可增长性, 易编程性等方面进行了优化, 在最底层的与硬件接触的过程中采用不同的汇编指令集将发挥不同的作用。

xv6 进程调度设计中同时完成了进程间切换, 让一切切换变得透明同时通过锁机制来避免竞争并且进行了内存、资源的自动释放。xv6 所实现的调度算法非常朴素, 仅仅是让每个进程轮流执行。这种算法被称作轮转法 (round robin)。但 xv6 不支持信号量操作同时单纯的轮转执行很大程度上当进程变多变复杂时会影响整个操作系统的性能。

(c)

Linux 设计了一套本质上基于多个进程以“时间多路复用”的方式来调度所有进程的分时技术调度器——即将使用 CPU 的时间分片, 每一个进程获得一片或多片时间, 在某个进程消耗完自己的分得的时间资源后, 由时间中断中来检查, 然后强制的切换到另一个进程, 通过这样的快速切换, 从表面上达到所有进程都同时运行的效果, 其中何时切换进程并选择哪一个进程来运行即为调度策略。

Linux 的调度方式: CFS 完全公平算法: CFS 算法从 RSDS / SD 中吸取了完全公平的思想, 不再跟踪进程的睡眠时间, 也不再企图分叉: 分交互式进程。它将所有的进程都统一对待, 这就是公平的含义。但是所谓的公平并不是绝对公平, 而是相对公平。在以往的调度算法中, 当给一个进程分配 40ms 的 CPU 计算时间时, 它会一直占有 CPU, 其它进程必须等待, 这就产生不公平。但是对于 CFS 算法, 它会将 40ms 的 CPU 计算时间根据优先级比例分配给不同的进程, 使每个进程都能相对公平地获得 CPU 的执行时间。

CFS 算法的核心思想是围绕着虚拟时钟展开的。虚拟时区别于硬件的实际时钟, 虚拟时钟可以根据权重调节步调, 按照不同的步调前进。而实际时钟是按着周定的步调前进, 不能改变。每个处理器不仅维护实际的时钟, 还在对应的运行队列维护着一个虚拟时钟, 它的前进步伐与该处理器上的进程总权重成反比-总权重越大, 虚拟时钟前进的步伐越慢。同时, 每个进程也有自己的虚拟时钟, 它记录着进程已经占用 CPU 的虚拟时间。它的前进步伐与进程的权重成反比, 进程的权重对应进程的优先级, 对于不同优先级的进程, 一般来说, 优先级低的进程其虚拟时钟的步调比较快, 而优先级高的步调比较慢。CFS 调度算法会选择最落后于运行

队列虚拟时钟的进程来执行。如果某个进程的虚拟时钟快于运行队列的虚拟时钟,则在将来的调度过程受到惩罚”。相反,如果比运行队列虚拟时钟慢的进程,将在调度过程受到奖励。

通常, I/O 密集型任务在运行很短时间后就会阻塞以便等待更多的 I/O; 而 CPU 密集型任务只要有在处理器上运行的机会, 就会用完它的时间片。因此, I/O 密集型任务的虚拟运行时间最终将会小于 CPU 密集型任务的, 从而使得 I/O 密集型任务具有更高的优先级。这时, 如果 CPU 密集型任务在运行, 而 I/O 密集型任务变得有资格可以运行(如该任务所等待的 I/O 已成为可用), 那么 I/O 密集型任务就会抢占 CPU 密集型任务。

Linux 也实现了实时调度。采用 SCHED_FIFO 或 SCHED_RR 实时策略来调度的任何任务, 与普通(非实时的)任务相比, 具有更高的优先级。

xv6 的 round robin 调度算法相较于 CFS 算法没有进程间的优先级概念, 只是单纯的进行进程间的轮换, 所以当执行进程复杂度很高时其他进程将长时间无法被执行, 系统性能将严重受到影响。

在我看来, 相较于 CFS 算法, 在一个混杂着大量计算型进程和 IO 交互进程的系统中, CFS 调度器对待 IO 交互进程要比 O(1) 调度器更加友善和公平, CFS 的时间片是动态的, 是系统负载均衡以及其优先级的函数, 这便可以把进程调度动态适应到系统最佳, 以节省切换开销。

而我们可以设计一个更加适用于当今 CPU 多核时代的调度算法, 更多的关心如何将不同进程复杂度的进程放在多个 CPU 中最合适的 CPU 中执行, 我们可以采用类似于 CFS 的算法来衡量进程的复杂度, 而更多地关心如何将不同的复杂度的不同密集型进程放在合适的 CPU 上执行, 其中需要考虑不同 CPU 的占用情况同时以更多的多核并行执行为目标, 让进程的执行更多地体现出现今多核 CPU 的优势, 将目光放在如何将一些本针对于少量核心的进程运行在更多的 CPU 核心上。提高并行性同时让最常运行部分运行得更快应该是算法需要改进的方向。