Lab3 Scheduling

21307110014

1 获取进程运行状态

任务一是实现具有获取进程运行时长统计的系统调用`wait_sched`。

1.1 `on_state_change`辅助函数

首先实现辅助函数`on_state_change`用于在进程状态改变时更新统计量。进程结构`struct proc`中已经加入了记录该进程分别在**RUNNABLE**、**RUNNING**和**SLEEPING**状态下运行时长的变量`runnable_time` `running_time` `sleep_time`，它们会在`allocproc`中被初始化为0。`on_state_change`的工作就是更新这三个值，系统会在将要改变进程状态时调用这个函数，把该进程在当前状态消耗的时间加到相应变量中。

具体实现如下：

```
int on_state_change(int cur_state, int nxt_state, struct proc *p) {
  switch (cur_state) {
  case UNUSED: // assert (nxt_state is RUNNABLE)
    break;
  case RUNNABLE: // assert (nxt_state is RUNNING or SLEEPING)
    p->runnable_time += ticks - p->start;
    break;
  case RUNNING: // assert (nxt_state is RUNNABLE or ZOMBIE)
    p->running_time += ticks - p->start;
    break;
  case SLEEPING: // assert (nxt_state is RUNNABLE)
    p->sleep_time += ticks - p->start;
    break;
  case ZOMBIE: // assert (nxt_state is UNUSED)
  default:
    break;
  }
  p->start = ticks;
  return 0;
}
```

其中利用进程结构体中的`start`记录进程所在状态的起始时刻，状态再次改变时当前时刻`ticks`和`p->start`的差值即进程处于当前状态的时长。

1.2 `wait_sched`系统调用

接下来实现系统调用`wait_sched`，这是一个增强版的`wait`，在等到结束子进程的同时父进程还会获得这个子进程的运行时间统计量`runnable_time` `running_time` `sleep_time`。其逻辑与`wait`基本一致：

```
int wait_sched(int *runnable_time, int *running_time, int *sleep_time) {
  struct proc *np;
  int havekids, pid;
```

```
  struct proc *p = myproc();

  acquire(&p->lock);
  for (;;) { // Scan through table looking for exited children.
    havekids = 0;
    for (np = proc; np < &proc[NPROC]; np++) {
      if (np->parent == p) {
        acquire(&np->lock);
        havekids = 1;
        if (np->state == ZOMBIE) { // Found one.
          pid = np->pid;
          *runnable_time = np->runnable_time;
          *running_time = np->running_time;
          *sleep_time = np->sleep_time;
          freeproc(np);
          release(&np->lock);
          release(&p->lock);
          return pid;
        }
        release(&np->lock);
      }
    }

    if (!havekids || p->killed) { // No children.
      release(&p->lock);
      return -1;
    }
    sleep(p, &p->lock); // Wait for any children to exit.
  }
}
```

注意到这里没有进行`copyout`，只是把值放到了`sys_wait_sched`传入的指针里。`copyout`是在后者做的（避免了更多类型转换）：

```
uint64 sys_wait_sched(void) {
  uint64 p0, p1, p2, ret;
  struct proc *p = myproc();
  int runnable_time, running_time, sleep_time;

  if (argaddr(0, &p0) < 0) return -1;
  if (argaddr(1, &p1) < 0) return -1;
  if (argaddr(2, &p2) < 0) return -1;

  ret = wait_sched(&runnable_time, &running_time, &sleep_time);
  if (copyout(p->pagetable, p0, (char *)&runnable_time, sizeof(runnable_time)) < 0) return
-1;
  if (copyout(p->pagetable, p1, (char *)&running_time, sizeof(running_time)) < 0) return
-1;
  if (copyout(p->pagetable, p2, (char *)&sleep_time, sizeof(sleep_time)) < 0) return -1;

  return ret;
```

```
}
```

接下来在几个文件中加入系统调用的声明，与上一个lab类似，这里不再赘述。

1.3 运行结果

使用提供的`stat`进行测试，结果如下：

```
$ stat 5
PID: 4 | Runnable Time: 52 ticks | Running Time: 71 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 45 ticks | Running Time: 79 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 55 ticks | Running Time: 80 ticks | Sleep Time: 12 ticks
PID: 7 | Runnable Time: 57 ticks | Running Time: 79 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 52 ticks | Running Time: 85 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 143 ticks
```

PID	Runnable Time (ticks)	Running Time (ticks)	Sleep Time (ticks)
4	52	71	12
5	45	79	12
6	55	80	12
7	57	79	12
8	52	85	12

Average Turnaround Time: 143 ticks

2 优先级调度

任务二是实现静态优先级调度模式。

2.1 `scheduler`

先增加PR分支的`scheduler`逻辑：每轮选择RUNNABLE进程中优先级最高的调度。具体如下：

```
    // inside an endless loop
#elif defined PR
    // First find the process with the highest priority and is RUNNABLE
    struct proc *mx = 0;
    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE) {
        if (mx == 0 || p->priority > mx->priority)
          mx = p;
      }
      release(&p->lock);
    }

    // If found such mx, run it.
    if (mx != 0) {
```

```
      acquire(&mx->lock);
      if (mx->state == RUNNABLE) {
        on_state_change(mx->state, RUNNING, mx);
        mx->state = RUNNING;
        c->proc = mx;
        swtch(&c->context, &mx->context);

        c->proc = 0;
        found = 1;
      }
      release(&mx->lock);
    }
#endif
```

2.2 `set_priority`系统调用

再实现系统调用`set_priority`供用户设置进程的优先级（可选0到3之间的整数）。内部实现即设置具有指定pid的进程的成员变量`priority`：

```
int set_priority(int priority, int pid) {
  struct proc *p;

  if (priority < 0 || priority > 3) return -1;

  // Scan through table looking for targeted process.
  for (p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if (p->pid == pid) {
      p->priority = priority;
      release(&p->lock);
      return 0;
    }
    release(&p->lock);
  }

  return -1; // not found
}
```

2.3 运行结果

使用提供的`priostat`进行测试，结果如下：

```
$ priostat 5
Set priority 0 to PID 4
Set priority 1 to PID 5
Set priority 2 to PID 6
Set priority 3 to PID 7
Set priority 0 to PID 8
PID: 7 | Runnable Time: 13 ticks | Running Time: 68 ticks | Sleep Time: 12 ticks
PID: 6 | Runnable Time: 31 ticks | Running Time: 66 ticks | Sleep Time: 12 ticks
PID: 5 | Runnable Time: 60 ticks | Running Time: 73 ticks | Sleep Time: 12 ticks
PID: 4 | Runnable Time: 98 ticks | Running Time: 73 ticks | Sleep Time: 12 ticks
PID: 8 | Runnable Time: 127 ticks | Running Time: 81 ticks | Sleep Time: 12 ticks
Average Turnaround Time: 150 ticks
```

PID	Runnable Time (ticks)	Running Time (ticks)	Sleep Time (ticks)
7	13	68	12
6	31	66	12
5	60	73	12
4	98	73	12
8	127	81	12

Average Turnaround Time: 150 ticks

3 不同调度模式对比

3.1 通过Round-Robin和优先级调度对于进程运行的影响对比

在`stat`和`priostat`中，程序都会创建`n`个子进程，在每个子进程中进行同样的`big_calculation`（大约1e10次运算），因此它们的RUNNING时间应该差不多，运行结果显示的确如此。同时所有子进程会调用相同次数的`sleep(1)`，因此它们的SLEEPING时间都为12ticks。RR和PR的区别主要体现在RUNNABLE时间上：RR中不同子进程公平地轮流被执行，因此RUNNABLE时间呈均匀分布，而PR严格先执行优先级高的进程，因此RUNNABLE时间单调递增（而其中位/平均数与RR的平均数大致相等，所以RR和PR的平均轮转时间也大致相等）。

3.2 Round-Robin和优先级调度分别适应场景

RR特点是响应时间短，不同进程运行机会均等，适用于需要快速响应的交互式系统或需要公平分配CPU时间的多任务系统（如多用户环境）；PR保证高优先级任务的响应时间和完成时间，适用场景如PC应该优先响应用户的交互（键盘输入/鼠标点击），再进行其他后台任务（网络传输/...）。

3.3 任务一和任务二思路

代码实现在第一第二部分有写，这里就写一下思路。

任务一代码分`on_state_change`和`wait_sched`两块。在写`on_state_change`时我本来以为需要列举所有可能的转移，后来发现其实只要看当前状态是什么即可，进程结构体中的大多数辅助变量也没有用到，就用一个`start`记一下时间，以便在下一个state change时获取当前进程是什么时候开始的。`wait_sched`大部分借鉴了`wait`的代码，只是把copyout子进程的结束状态改成了把子进程的时长统计写到传入的指针。注意到在访问进程的成员变量之前都会先`acquire(&p->lock)`，除了访问`np->parent`前，防止`np`是`p`的祖先进程造成死锁。copyout在trap handler`sys_wait_sched`中实现，避免把`argaddr`返回的参数转成`int*`传给`wait_sched`再在copyout时转成`uint64`。`wait_sched`第一遍写犯了一个愚蠢的错误，在found时把所有时间设置成父进程的相关变量，调试了半天才找出来（本来以为是copyout之类的地方出了问题）。

任务二相对比较直接，做的也比较顺利。`scheduler`里就是先找RUNNING的进程里优先级最高的进程`mx`，再调度它。`set_priority`则是找到指定pid的进程把它的`priority`设置为指定值。