

## 实验报告 3

221275207 喻思文

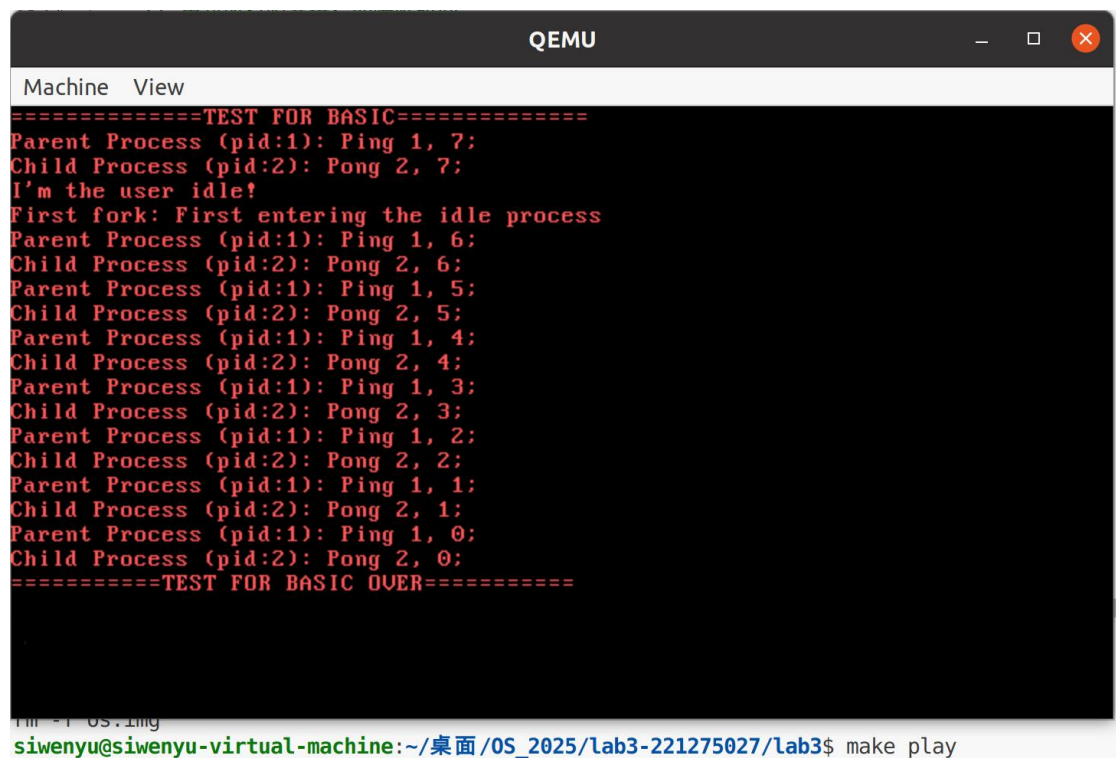
[221275027@smail.nju.edu.cn](mailto:221275027@smail.nju.edu.cn)

### 一、实验进度

实现 Fork、Exec、Sleep、Exit、GetPid 功能，未完成 wait()库函数。

### 二、运行截图

basic 测试结果如下，与网站示例调度过程相同：



```
Machine View
=====TEST FOR BASIC=====
Parent Process (pid:1): Ping 1, 7;
Child Process (pid:2): Pong 2, 7;
I'm the user idle!
First fork: First entering the idle process
Parent Process (pid:1): Ping 1, 6;
Child Process (pid:2): Pong 2, 6;
Parent Process (pid:1): Ping 1, 5;
Child Process (pid:2): Pong 2, 5;
Parent Process (pid:1): Ping 1, 4;
Child Process (pid:2): Pong 2, 4;
Parent Process (pid:1): Ping 1, 3;
Child Process (pid:2): Pong 2, 3;
Parent Process (pid:1): Ping 1, 2;
Child Process (pid:2): Pong 2, 2;
Parent Process (pid:1): Ping 1, 1;
Child Process (pid:2): Pong 2, 1;
Parent Process (pid:1): Ping 1, 0;
Child Process (pid:2): Pong 2, 0;
=====TEST FOR BASIC OVER=====
The file OS.img
siwenyu@siwenyu-virtual-machine:~/桌面/OS_2025/Lab3-221275027/Lab3$ make play
```

### 三、实验思路

#### (一) sysSleep 函数

1. 首先检查睡眠时间必须大于 0，无效参数会导致直接返回而不进入睡眠。
2. 然后从 sf->ecx 获取睡眠时间参数，设置进程状态为 STATE\_BLOCKED，设置 sleepTime 为请求的时间片数，触发时钟中断(int \$0x20)主动让出 CPU。
3. 将剩余睡眠时间通过 sf->eax 返回，允许进程查询剩余睡眠时间。

## （二）sysExit 函数

1. 设置进程状态为 STATE\_DEAD, 这样进程状态从 RUNNING/RUNNABLE 变为 DEAD, 调度器会跳过 DEAD 状态的进程, 内存等资源会在后续被回收
2. 通过 int \$0x20 触发时钟中断主动让出 CPU
3. 设置返回值 eax 为 0 表示成功退出

## （三）sysGetPid 函数

1. 从全局 pcb 数组中获取当前进程的 PCB(current 索引), 读取 pcb[current].pid 字段
2. 将 pid 值存入栈帧的 eax 寄存器作为返回值

## （四）sysFork 函数

1. 通过循环遍历找到空闲的 PCB 槽位 (STATE\_DEAD 状态)
2. 若没有空槽位, 将-1 存入栈帧的 eax 寄存器作为返回值, 表示 fork 失败。
3. 如果有, 复制父进程的内存映像到子进程空间, 设置子进程的 PCB 信息:
  - （1）内存复制: 将父进程 0x100000 开始的内存复制到子进程空间
  - （2）PCB 设置: 复制寄存器状态、栈信息等。每个进程有独立的 2MB 地址空间 (0x100000\*(pid+1)), 通过 USEL 宏生成用户态段选择子。完整复制父进程的寄存器上下文, 为子进程创建独立的段选择子。一系列复制如下:

```
341 | pcb[new_index].stackTop = pcb[current].stackTop - (uint32_t)&(pcb[current]) + (uint32_t)&(pcb[new_index]);
342 | pcb[new_index].prevStackTop = pcb[current].prevStackTop - (uint32_t)&(pcb[current]) + (uint32_t)&(pcb[new_index]);
343 |
344 | pcb[new_index].regs.edi = pcb[current].regs.edi;
345 | pcb[new_index].regs.esi = pcb[current].regs.esi;
346 | pcb[new_index].regs.ebp = pcb[current].regs.ebp;
347 | pcb[new_index].regs.xxx = pcb[current].regs.xxx;
348 | pcb[new_index].regs.ebx = pcb[current].regs.ebx;
349 | pcb[new_index].regs.edx = pcb[current].regs.edx;
350 | pcb[new_index].regs.ecx = pcb[current].regs.ecx;
351 | pcb[new_index].regs.eax = pcb[current].regs.eax;
352 | pcb[new_index].regs.iqr = pcb[current].regs.iqr;
353 | pcb[new_index].regs.error = pcb[current].regs.error;
354 | pcb[new_index].regs.eip = pcb[current].regs.eip;
355 | pcb[new_index].regs.eflags = pcb[current].regs.eflags;
356 | pcb[new_index].regs.esp = pcb[current].regs.esp;
357 |
358 | pcb[new_index].regs.cs = USEL(2*new_index + 1);
359 | pcb[new_index].regs.ss = USEL(2*new_index + 2);
360 | pcb[new_index].regs.ds = USEL(2*new_index + 2);
361 | pcb[new_index].regs.es = USEL(2*new_index + 2);
362 | pcb[new_index].regs.fs = USEL(2*new_index + 2);
363 | pcb[new_index].regs.gs = USEL(2*new_index + 2);
364 |
365 | pcb[current].regs.eax = new_index;
366 | pcb[new_index].regs.eax = 0;
```

- （3）父子关系: 设置 ppid 和 timeCount、sleepTime 父进程返回子进程 pid, 子进程返回 0。

### （五）sysExec 函数

1. 通过 loadUMain 加载新程序到内存（ $0x100000*(pid+1)$ 地址）
2. 获取新程序的入口地址，设置 eip 指向程序入口
3. 内存管理细节：每个进程有独立的 2MB 地址空间，用户栈指针 esp 和基址指针 ebp 初始化为  $0x100000*(current+1)$ ，通过 USEL 宏设置用户态段选择子。
4. 执行环境初始化：清空通用寄存器避免状态污染，设置 EFLAGS 为 0x202（启用中断+用户态）
5. 通过 contextSwitch 切换到新程序

### （六）timerHandle 函数

1. 遍历 pcb 数组更新 BLOCKED 进程的 sleepTime，检查当前进程时间片是否用完，根据进程状态决定是否需要调度。
2. 时间调度：每个进程有 MAX\_TIME\_COUNT 的时间片，sleepTime 以时钟中断为单位递减，时间片用完的进程会被设为 RUNNABLE。
3. 状态转换：（1）BLOCKED -> RUNNABLE (当 sleepTime 减至 0)；（2）RUNNING -> RUNNABLE (当时间片用完)；（3）RUNNABLE -> RUNNING (被调度选中)
4. 总体思想：
  - （1）更新阻塞进程：遍历所有进程控制块，递减 BLOCKED 进程的 sleepTime，当 sleepTime 减至 0 时唤醒进程。
  - （2）处理当前进程时间片：检查当前进程是否用完时间片，用完时间片则设为 RUNNABLE 状态。
  - （3）进程调度：优先调度非 idle 的用户进程( $pid > 0$ )，没有用户进程可运行时才调度 idle 进程( $pid = 0$ )，确保系统始终有进程可运行。通过 contextSwitch 完成切换。

### （七）schedule 函数

1. 重置当前进程的时间片计数器，将当前进程状态设为 RUNNABLE(除非

是 DEAD 状态)。

2. 用户进程调度：从 `current+1` 开始循环查找，跳过 `idle` 进程(`pid=0`)，找到第一个 `RUNNABLE` 用户进程，调用 `contextSwitch` 执行切换，返回新进程的 `pid`。

3. `idle` 进程处理：当没有用户进程可运行时，选择 `idle` 进程(`pid=0`)运行，同样执行完整的上下文切换

4. 错误处理：当系统中无任何可运行进程时，返回 -1 表示调度失败

5. **疑问**：事实上我没有发现这个函数的调用情况，当我解锁 `todo` 下面的注释掉的 `return -1` 时候仍然不影响输出，我自己没有调用这个函数。

## 四、思考题

### (一) 为什么 `idle` 会进入用户态

1. 段描述符定义：

```
68 // "virtual" segment descriptors
69 //      : they point to the same part of SEG_KCODE and SEG_KDATA
70 //      but they're with DPL_USER
71 //      so that idle becomes a USER process
72 gdt[SEG_IDLE_CODE] = SEG(STA_X | STA_R, 0, 0xffffffff, DPL_USER);
73 gdt[SEG_IDLE_DATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
```

与内核段相同基址和限制，但 `DPL=3` 允许用户态访问

2. 段寄存器初始化选择子 `RPL=3` 表示用户态请求

3. 特权级切换：通过中断返回降低特权级。`iret` 会根据 `cs` 选择子的 `RPL` 修改 `CPL`，即使代码在内核空间也进入用户态

4. 关键寄存器设置：`pcb[0].regs.eflags = 0x202`。即 `IOPL=0` 禁止用户态执行 `IO` 指令，保留内核关键操作权限。

### (二) 为什么在 `initIdle()` 中，我们移除了 `waitForInterrupt()`?

`waitForInterrupt()` 通常用于让 CPU 进入低功耗状态，等待下一个中断唤醒 CPU，常见于操作系统空闲循环中

在 `initIdle()` 中移除的原因是：`idle` 进程已经是用户态进程，不应直接使用内核级指令，用户态无法执行 `hlt` 等特权指令，简单循环足够满足需求。我们的调度器已经能正确处理 `idle` 进程，不需要额外节能机制，系统中断处理不受影响。

### （三）irqHandle 对比 lab2 也增加了部分保存与恢复的内容

#### 1. pcb[current].prevStackTop = pcb[current].stackTop:

保存当前进程的栈顶指针到临时变量 tmpStackTop，为后续恢复进程上下文做准备，确保中断处理完成后能正确返回到被中断的进程

#### 2. pcb[current].stackTop = (uint32\_t)tf:

更新 prevStackTop 为当前栈顶，记录进入中断前的栈位置，用于嵌套中断时的栈管理。

#### 3. pcb[current].stackTop = tmpStackTop:

恢复中断前的栈顶指针，保证进程上下文完整性，使 iret 能正确返回到被中断的代码。

#### • 整体:

1 保存 → 2 更新 → 处理中断 → 3 恢复,形成完整的中断上下文保存/恢复机制,支持中断嵌套和进程切换。

这种设计保证了中断处理的原子性，完整的进程上下文，正确的嵌套中断，保证了调度器切换的安全性

### （四）什么时候会触发进程的调度？

可能如下：

1. 时钟中断 → timerHandle → 检查时间片 → schedule
2. sleep() → 设置 BLOCKED → 中断 → timerHandle → schedule
3. exit() → 设置 DEAD → 中断 → timerHandle → schedule
4. fork() → 创建 RUNNABLE 进程 → 可能 schedule

### （五）考虑以下场景

P1 从时钟中断返回，顺序执行 0x100631、0x100632、0x100634、0x100636、0x100639、0x10063b、0x10063d、0x10063e、0x100644、0x10064e、0x100651、0x10062f，再次陷入时间中断，切换至 P2

P2 从时间中断返回，顺序执行 0x100631、0x100632、0x100634、0x100636、0x100639、0x10063b、0x100606、0x100609、0x10060c、0x10060f、0x100615、0x10061c

## 对全局变量 `displayRow` 的更新产生一致性问题思考

当多个进程并发地进行系统调用时，会对共享资源产生竞争，从而引发一致性问题。在给定的场景中，进程 P1 和 P2 都对共享变量 `displayRow` 进行更新，它们的执行顺序不同，可能导致 `displayRow` 的值出现不一致。

1. 临界资源：`displayRow` 是一个共享变量，由多个进程共同访问和更新，属于临界资源。
2. 竞争条件：当多个进程同时访问 `displayRow` 时，如果它们的执行顺序无法保证操作的原子性，就可能导致 `displayRow` 的值不一致。例如，P1 正在更新 `displayRow`，而 P2 在同一时间也尝试访问或修改 `displayRow`，这种情况下，`displayRow` 的最终值可能无法预期。
3. 关闭外部硬件中断：为了防止这种竞争条件，应该在系统调用过程中关闭外部硬件中断，确保临界区代码的执行不被中断。这样可以保证对共享资源的访问是原子的，避免出现数据不一致的问题。

问题的本质是共享资源未正确同步。通过在临界区关闭中断，可确保 `displayRow` 的原子更新，避免竞态条件；非临界区保持中断开启以维持系统响应性。在多进程并发执行系统调用时，为了保证共享资源的一致性，应该在临界区代码执行期间关闭外部硬件中断，确保对共享资源的访问是原子的。而对于非临界区的代码，可以开启外部硬件中断，允许中断嵌套。