

OS LAB 4 进程同步

个人信息

姓名：郑凯琳

学号：205220025

邮箱：205220025@smail.nju.edu.cn

1. 实验目的

实现操作系统的信号量及对应的系统调用，然后基于信号量解决实际问题。

(1) 内核：提供基于信号量的进程同步机制，并提供系统调用 `sem_init`、`sem_post`、`sem_wait` 和 `sem_destroy`，完成格式化输入函数

(2) 库：对上述系统调用进行封装

(3) 用户：对上述系统调用进行测试

2. 实验进度

成功完成了所有内容（包括“生产者-消费者问题”和“哲学家就餐问题”（选做），但似乎不成功）。

3. 实验过程

(1) 从标准输入读取数据

代码修改位置：kernel/kernel/irqHandle.c

syscallReadStdIn

1. 初始化和检查阻塞状态

检查设备值：

- `dev[STD_IN].value` 为 0 时，表示没有进程阻塞在标准输入上。
- 如果 `dev[STD_IN].value` 小于 0，表示已经有进程阻塞在标准输入上。

2. 阻塞当前进程

阻塞当前进程：

- 如果没有进程阻塞，将当前进程的设备值减 1，将当前进程插入到阻塞队列中。
- 设置当前进程的状态为 `STATE_BLOCKED`，并将其睡眠时间设为-1 表示在标准输入上阻塞。

3. 重置缓冲区

重置缓冲区指针：

- 将缓冲区头指针 `bufferHead` 和缓冲区尾指针 `bufferTail` 重置为相同值，表示缓冲区为空。
- 触发上下文切换以切换到其他进程。

4. 从阻塞状态恢复

恢复上下文：

- 从阻塞状态恢复后，读取段选择器 `sf->ds`，缓冲区指针 `sf->edx` 和缓冲区大小 `sf->ebx`。
- 设置段寄存器以准备存储读取到的数据。

5. 读取键盘缓冲区数据

读取键盘输入：

- 循环读取键盘缓冲区中的字符，直到读取到的字符数达到缓冲区大小-1，或缓冲区为空。
- 每次读取一个字符，将其存储到用户提供的缓冲区中，并输出到屏幕。
- 如果读取到空字符，不计入读取的字符数。

6. 终止缓冲区

添加字符串结束符：

在用户提供的缓冲区末尾添加空字符，表示字符串结束。

7. 返回结果

返回读取的字符数：

- 将读取到的字符数存储在当前进程的 `eax` 寄存器中，表示系统调用的返回值。
- 如果已有进程阻塞，返回错误码-1。

(2) 键盘缓冲区处理

代码修改位置： `kernel/kernel/irqHandle.c`

keyboardHandle

思路：处理（解锁）因等待输入而阻塞的进程，使其重新变为可运行状态。

1. 解除阻塞：

增加 `dev[STD_IN].value` 的值，表示标准输入设备有新的输入，从而可以解除一个阻塞的进程。

2. 获取被阻塞的进程表项：

使用 `offsetof` 宏计算 `ProcessTable` 结构中 `blocked` 成员的偏移量。通过从 `dev[STD_IN].pcb.prev` 减去这个偏移量，得到完整的 `ProcessTable` 结构的地址，从而获取被阻塞的进程表项。

3. 更新进程状态：

- 将被阻塞的进程状态更新为 `STATE_RUNNABLE`，表示该进程现在是可运行的。
- 同时重置进程的 `sleepTime` 为 0，表示该进程不再处于睡眠状态。

4. 移除阻塞队列：

- 更新 `dev[STD_IN].pcb.prev` 指针，指向当前阻塞队列中最后一个阻塞的进程的前一个进程。
- 更新新的最后一个阻塞进程的 `next` 指针，使其指向 `dev[STD_IN].pcb`，从而将解除阻塞的进程从队列中移除。

(3) 信号量

代码修改位置：`kernel/kernel/irqHandle.c`

syscallSemInit

功能：初始化信号量。

思路：

1. 初始化索引变量

定义一个变量 `idx` 用于记录当前遍历到的信号量槽的索引。

2. 遍历查找未使用的信号量槽

使用一个 `for` 循环来遍历所有信号量槽 (`sem` 数组)。在循环条件中，增加了对 `sem[idx].state != 0` 的检查，这样在找到未使用的信号量槽时会自动退出循环。如果遍历完所有槽都未找到，`idx` 的值将等于 `MAX_SEM_NUM`。

3. 返回索引

通过一个三元运算符将找到的索引或 -1 返回给当前进程。如果 `idx` 小于 `MAX_SEM_NUM`，表示找到了未使用的信号量槽，否则返回 -1。

4. 初始化信号量

如果找到了可用的信号量槽 (`idx` 小于 `MAX_SEM_NUM`)，则进行信号量的初始化：

- 将 state 设置为 1，标记信号量为已使用。
- 将 sf->edx 中的值赋给信号量的 value 字段，作为信号量的初始值。
- 初始化信号量的阻塞队列，将 next 和 prev 指针指向自身，表示队列为空。

syscallSemWait

功能：等待信号量。

思路：

1. 检查信号量索引的有效性

首先，检查信号量索引是否在有效范围内（0 到 MAX_SEM_NUM - 1）。

然后，检查该信号量是否被使用（sem[idx].state == 1）。如果信号量未被使用，则返回错误 -1。

2. 处理信号量值

如果信号量的值（sem[idx].value）大于 0，则表示当前进程可以直接减少信号量值并继续执行。在这种情况下，将信号量值减 1，然后设置 eax 寄存器为 0 表示操作成功。

如果信号量的值小于等于 0，则表示当前进程需要阻塞在该信号量上。在这种情况下：

- 将当前进程（pcb[current]）插入到信号量的阻塞队列中，这通过双向链表实现。
- 设置当前进程的状态为 STATE_BLOCKED，表示当前进程被阻塞。
- 将当前进程的睡眠时间设置为 -1，以指示进程被阻塞在信号量上。
- 触发一个中断（int \$0x20），进行上下文切换，允许其他进程继续执行。
- 当上下文切换完成后，恢复执行当前进程，并设置 eax 寄存器为 0 表示操作成功。

syscallSemPost

功能：发送信号量。

思路：

1. 信号量索引有效性检查：

首先，检查传入的信号量索引是否在有效范围内（0 到 MAX_SEM_NUM-1）。如果不在范围内，则返回 -1 表示错误。

2. 信号量状态检查：

检查信号量是否被使用（sem[idx].state == 1）。如果信号量未被使用（state == 0），则返回 -1 表示错误。

3. 处理无进程阻塞的情况：

如果信号量的值（sem[idx].value）大于等于 0，表示当前没有进程被阻塞在该信号量上。直接将信号量值加 1，表示释放一个资源，并返回 0 表示操作成功。

4. 释放被阻塞的进程：

如果信号量的值小于 0，表示有进程被阻塞在该信号量上。首先将信号量的值加 1，然后从信号量的阻塞队列中取出一个被阻塞的进程。

获取被阻塞进程的表项，并将其状态设置为 `STATE_RUNNABLE`，表示进程现在可以继续运行。

从信号量的阻塞队列中移除被阻塞的进程。

5. 返回操作结果：

无论是释放被阻塞进程还是增加信号量值，最后设置 `eax` 寄存器为 0 表示操作成功。

syscallSemDestroy

功能：销毁信号量。

思路：

1. 参数检查

函数首先从系统调用的参数中获取信号量的索引 `i`，然后检查该索引是否在有效范围内。如果不在范围内，将设置 `eax` 寄存器为 -1，表示出错，并直接返回。

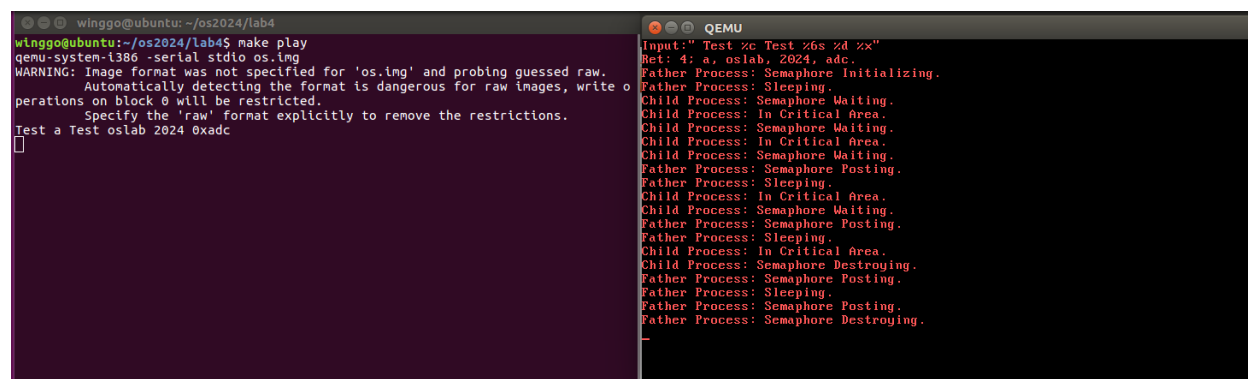
2. 检查信号量状态并销毁

接下来，函数检查信号量的状态 `sem[i].state`。如果该信号量当前未被使用 (`state == 0`)，则设置 `eax` 寄存器为 -1，表示出错，并直接返回。

如果信号量被使用 (`state == 1`)，则将其状态设置为未使用状态 (`state = 0`)，并将其值 (`value`) 和阻塞队列 (`pcb.next` 和 `pcb.prev`) 重置为指向自身，表示阻塞队列为空。

最后，将 `eax` 寄存器设置为 0，表示操作成功。

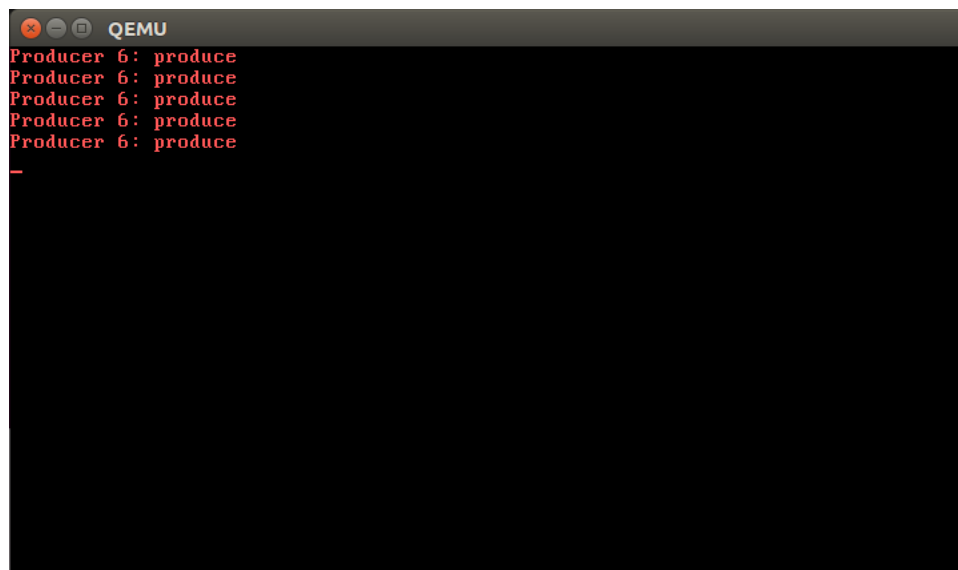
4. 实验结果



```
winggo@ubuntu:~/os2024/lab4$ make play
qemu-system-i386 -serial stdio os.img
WARNING: Image format was not specified for 'os.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write off
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
Test a Test oslab 2024 0xad
Input: "Test %c Test %6s %d %x"
Ret: 4: a, oslab, 2024, adc.
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.
```

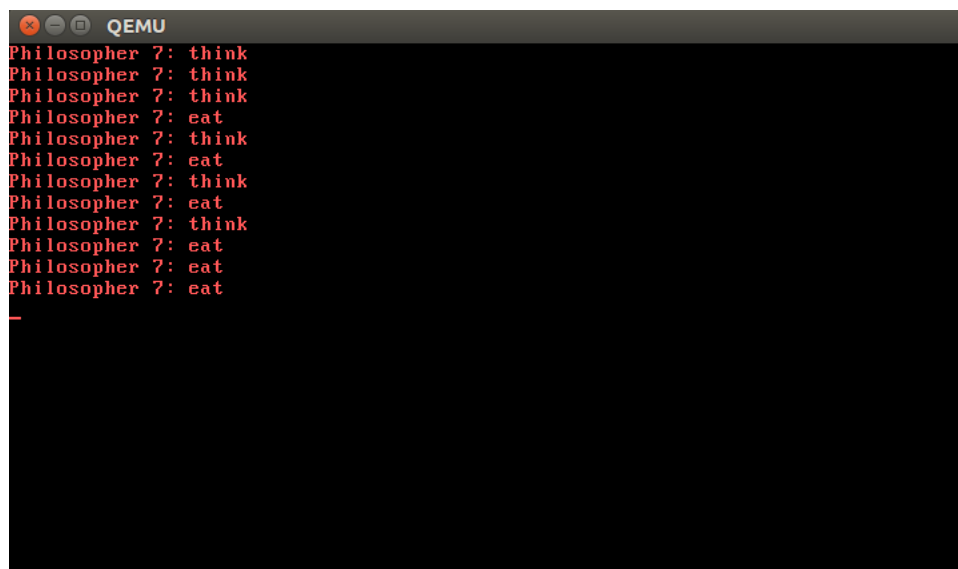
解决进程同步问题：

“生产者-消费者问题”

A screenshot of a QEMU terminal window. The title bar shows the QEMU logo and the text "QEMU". The terminal output consists of five lines of red text: "Producer 6: produce", "Producer 6: produce", "Producer 6: produce", "Producer 6: produce", and "Producer 6: produce". A single red underscore character is on the line immediately following the last output line.

```
QEMU
Producer 6: produce
Producer 6: produce
Producer 6: produce
Producer 6: produce
Producer 6: produce
_
```

“哲学家就餐问题”

A screenshot of a QEMU terminal window. The title bar shows the QEMU logo and the text "QEMU". The terminal output consists of thirteen lines of red text: "Philosopher 7: think", "Philosopher 7: think", "Philosopher 7: think", "Philosopher 7: eat", "Philosopher 7: think", "Philosopher 7: eat", "Philosopher 7: think", "Philosopher 7: eat", "Philosopher 7: think", "Philosopher 7: eat", "Philosopher 7: eat", "Philosopher 7: eat", and "Philosopher 7: eat". A single red underscore character is on the line immediately following the last output line.

```
QEMU
Philosopher 7: think
Philosopher 7: think
Philosopher 7: think
Philosopher 7: eat
Philosopher 7: think
Philosopher 7: eat
Philosopher 7: think
Philosopher 7: eat
Philosopher 7: think
Philosopher 7: eat
Philosopher 7: eat
Philosopher 7: eat
Philosopher 7: eat
_
```