OS第 4 次实验

同步机制

151220030 → 高子腾 : GZT@outlook.com

2017 年 5 月 26 日

https://github.com/sebgao/sebopesyslab

第一部分 必答题

1. 回忆一下整个 os 实验，什么地方是所有进程共享的呢?

   内核代码和数据

2. 你可以据此实现你的进程匿名信号量么?

   不可以，用户程序访问不到内核数据，不可以以内核数据作为共享数据．假设匿名信号量存在于内核数据中，用户程序只能间接操作信号量，这又必须要有索引，这样就不再是匿名信号量了．

3. 普通的全局变量是不是可以被用户修改?

   是的，普通的全局变量可以被任意一个线程在任意时间修改，因为它们有共同的页表项．

4. linux 进程和线程下匿名信号量的实现有什么本质区别?

   linux 下进程的匿名信号量是存放在另外申请的进程共享空间啊中； 而线程的匿名信号量是存放在全局变量中，也就是线程所在的进程的用户空间中．

5. 结合你的 os，你该怎样实现进程/线程的具名信号量?

   首先在内核数据中开一个固定信号量数组，存放具名信号量．然后在用户程序中以数组下标打开信号量，

```
sem_t *s = sem_open(key, init_value);
```

   然后就可以对 s 进行普通信号量操作． 如果多个进程一并同时执行上述语句， 只会初始化一次， 因为信号量有 used 字段示位已被使用．

第二部分 实验环境

编译虚拟机环境 Debian 32bit， 注意只能在 Debian 32bit 下编译的镜像文件才能正确运行 (不能在 Ubuntu 32bit 下编译)．运行 QEMU 的平台则没有限制．

GCC 编译版本 gcc version 4.9.2 (Debian 4.9.2-10)

第三部分 实验结果

完成了所有要求，

1. 实现线程概念
2. 实现线程间匿名信号量和同步机制
3. 实现进程间具名信号量和同步机制
4. 有演示程序，见 /app
5. 实现 sem_open, sem_init, sem_post, sem_wait, sem_trywait, sem_close

第四部分 详细实验结果

1 线程实现

线程实现其实跟 fork 实现有异曲同工之妙， 都是产生新的 PCB， 只不过地址空间是共享的， 这就意味着不需要深拷贝页表， 代码见 kernel/process.c 的 thread_current 函数． 代码做的事情跟 fork 极为相似， 但没有深拷贝 pgdir， 而是直接浅复制了 pgdir， 进而对于用户栈空间分配页表项， 指定了新线程的用户态 esp 和 eip．在这里还篡改了用户栈，使线程结束时会自动执行 exit．

有了线程的实现，自然会想到怎么实现 join，可以想到的一种方法是，把当前调用 join 的 PCB 放入 join 目标的线程的 PCB 中的 join_list 中去，相当于暂时不参加调度，而当目标线程 exit 时，就把所有在 join_list 里的线程放入 ready_list，参加调度.

```
void exit_current(){
  PCB* p;

  while(1){
    p = ll_pop(&current->join_list);
    if(p == NULL) break;
    if(p->used == 0) continue;
    ll_entail(&ready_list, p);
  }

  free_pcb(current);
  current = NULL;
  do_scheduler();
}

void join_current(int pid){
  if(pid == 0)return;
  uint32_t i;
  for(i=0; i<PCBPOOLMAX; i++){
    if(PCBPool[i].pid == pid)break;
  }
  if(i == PCBPOOLMAX) return;
  if(PCBPool[i].used == 0) return;
  PCB *p = &PCBPool[i];

  PCB* cur = current;
  ll_entail(&p->join_list, cur);
  current = NULL;
  do_scheduler();
}
```

这种阻塞于某个目标相当于把 PCB 移动到另外一个链表中去，从而不参与调度，这样的实现也可以用在下文的信号量中.

2 信号量实现

为了实现线程间通信，信号量的出现就在所难免了.首先根据理论课的知识，我们先定义信号量的结构，与 POSIX 实现可能有所不同，

```
typedef struct Semaphore {
  uint32_t used; // 是否被使用
  int count; // 信号量计数
  PCB* block_list; // 阻塞链表
} Semaphore;
```

由于我实现了具名信号量，而且由于 sem_close 的存在,used 属性不可避免的要出现了.有了这个结构，我们就可以很清晰的实现 PV 操作，阻塞的 sem_wait 操作就相当于把 count 减一，当 count 小于零时就把当前的 PCB 加入到阻塞链表里，再进行重新调度.sem_post 操作相当于 count 加一，当 count 小于等于零时就把阻塞链表里的一个 PCB 移出并加入到就绪队列. 而非阻塞的 sem_trywait 操作检测选择的信号量是否有资源，如果没有资源的话就返回 0，有资源的话就把资源减一并返回 1. 非阻塞的 sem_trywait 操作其实是把阻塞操作的阻塞部分交由用户自行处理.

具名信号量实现也不是一项特别难的问题，只要在内核中固定写好几个信号量，再依据 index 进行返回就可以. 关于信号量的函数声明如下，源代码详见 kernel/semaphore.c .

```
void sem_init_kr(Semaphore* sem, int count);
void sem_close_kr(Semaphore* sem);
void sem_post_kr(Semaphore* sem);
void sem_wait_kr(Semaphore* sem);
int sem_trywait_kr(Semaphore* sem);
Semaphore* sem_open_kr(int index, int count);
int sem_get_kr(Semaphore* sem);
```

在用户程序中，sem_t 是 Semaphore 的同义词.

第五部分 综合

为了自豪地展现程序的正确性，同时也符合本次实验的要求，我写了一个独立于 game 的演示程序，可在 app 下找到，另外 Makefile 也随之改变，详见 Makefile. 程序开始时，加载两个用户态程序，分别是 game 和 app.

演示程序的逻辑就是生产者和消费者模型，不过输出相当多废话，代码如下

```
void producer(){
  int item;
  while(1){
    sleep(1+rand()%5);
    item = rand()%10;
    printf("PRODUCER: %d produced!\n", item);
    sem_wait(&empty);
    sem_wait(&mutex);

    insert_item(item);

    printf("PRODUCER: %d sent, now %d space
        left!\n", item, N-index);

    sem_post(&mutex);
    sem_post(&full);
  }
}

void consumer(){
  int item;
```

```
  int asleep;
  while(1){
    printf("CONSUMER: ready to receive!\n");
    while(!sem_trywait(&full));
    while(!sem_trywait(&mutex));

    item = remove_item();

    sem_post(&mutex);
    sem_post(&empty);

    asleep = 1+rand()%15;
    printf("CONSUMER: %d received, now I want to
        consumer it in %ds!\n", item, asleep);
    sleep(asleep); //consume

  }
}
```

其主要逻辑是生产者以随机的时间生产随机数，放入临界区，消费者取临界区的数，并以随机的时间消费它，这里临界区的大小为 N=2，可以任意修改．此代码可以测试线程间信号量是否正确实现．

在 app/main.c 中还有一段被注释的代码 test_process_sem() 实现了测试进程间具名信号量是否正确．可以在 main 函数中调用来测试．

第六部分　一些比较好玩的东西

由于我的线程实现本质上还是独立进程加上共享页表的方法，所以在子线程里调用 fork 一点问题也没有．虽然 thread_join 有着 thread 的前缀，但本质上只要是有效的 pid 参数传进去都可以进行 join 操作，如此，可以 join 一个进程．不过当一个进程结束时，一定要调用 exit 函数，不然会出现错误．而线程就不需要 exit 了，因为在创建线程的时候用户栈被篡改当 ret 时自动跳到 exit．另外哭笑不得的一点是,fork 完紧接着 exit 操作也会造成错误，我不相信实际中还有这种操作．

第七部分　一些用户接口

```
void sem_init(semaphore *sem, int count);
semaphore* sem_open(int index, int count);
int sem_get(semaphore *sem);
void sem_post(semaphore *sem);
void sem_wait(semaphore *sem);
int sem_trywait(semaphore *sem);
void sem_close(semaphore *sem);

int thread(void* entry, void* argument);
```

第八部分　实验感想

这次实验还是比较简单的，可以说是一帆风顺了．所以没有什么比较深刻的印象．唯一一个值得说的是一个线程逻辑执行完 ret 时，栈不可以篡改为内核的 exit_current，原因是特权级，只能篡改为用户的 exit，但内核 thread 函数并不能确定用户的 exit 在哪，所以还是要加一层传一下 exit 进 thread 函数．应了那句话：软件实现靠加层!