# xv6信号量实现

1. **核心代码**

* **proc.h**

|  |
| --- |
| // semaphore  #define MAX\_NPROC 5  #define MAX\_SEMAPHORE 3  struct semaphore  {  int value;  struct spinlock lock;  struct proc \*queue[MAX\_NPROC];  int end;  int start;  }Semaphore[MAX\_SEMAPHORE]; |

MAX\_NPROC代表一个信号量最多可以容纳多少个进程同时等待，MAX\_SEMAPHORE代表系统最多同时支持多少种信号量。

* **proc.c**

|  |
| --- |
| int  sem\_init(int p,int value)  {  if (p<0 || p>=MAX\_SEMAPHORE)  return -1;  struct semaphore\* s = Semaphore+p;  s->value = value;  initlock(&s->lock, "semaphore\_lock");  s->end = s->start = 0;  return 0;  } |

信号量的初始化，指定第p个信号量初始有多少资源。

|  |
| --- |
| int  sem\_P(int p, void\* chan)  {  if (p<0 || p>=MAX\_SEMAPHORE)  return -1;  struct semaphore\* s = Semaphore+p;  acquire(&s->lock);  s->value--;  if (s->value < 0)  {  s->queue[s->end] = chan;  s->end = (s->end + 1) % MAX\_NPROC;  sleep(chan, &s->lock);  }  release(&s->lock);  return 0;  } |

P原语，指定第p个信号量，以及当前进程sleep的chan，信号量减一，若这之后信号量小于0，说明没有足够的资源给当前进程，则将当前进程加入到等待队列中，并使其sleep。

|  |
| --- |
| int  sem\_V(int p)  {  if (p<0 || p>=MAX\_SEMAPHORE)  return -1;  struct semaphore\* s = Semaphore+p;  acquire(&s->lock);  s->value++;  if (s->value <= 0)  {  wakeup(s->queue[s->start]);  s->queue[s->start] = 0;  s->start = (s->start + 1) % MAX\_NPROC;  }  release(&s->lock);  return 0;  } |

V原语，指定第p个信号量，释放一个资源，信号量加一，若在这之后信号量小于等于0，说明有进程在等待资源，将等待队列的第一个进程移出队列，并将其唤醒。

* **defs.h、syscall.h、syscall.c、sysproc.c、usys.S、user.h**

主要目的是将sem\_init、sem\_P、sem\_V注册为系统函数，以供用户调用，比较繁琐，涉及数个文件。

* **defs.h**

|  |
| --- |
| // semaphore  int sem\_init(int semaphore\_pointer, int value);  int sem\_P(int semaphore\_pointer,void\* chan);  int sem\_V(int semaphore\_pointer); |

添加三个函数的声明。

* **syscall.h**

|  |
| --- |
| #define SYS\_sem\_init 29  #define SYS\_sem\_P 30  #define SYS\_sem\_V 31 |

添加系统函数预编译。

* **syscall.c**

|  |
| --- |
| extern int sys\_sem\_P(void);  extern int sys\_sem\_V(void);  ……  [SYS\_sem\_P] sys\_sem\_P,  [SYS\_sem\_V] sys\_sem\_V, |

系统函数关联。

* **sysproc.c**

|  |
| --- |
| //semaphore  int  sys\_sem\_init(void)  {  int semaphore\_pointer,value;  // fetch the arguments  if(argint(0, &semaphore\_pointer) < 0 || argint(1, &value) < 0)  return -1;  return sem\_init(semaphore\_pointer, value);  }  int  sys\_sem\_P(void)  {  int semaphore\_pointer,chan;  // fetch the arguments  if(argint(0, &semaphore\_pointer) < 0 || argint(1, &chan) < 0)  return -1;  return sem\_P(semaphore\_pointer, (void\*)chan);  }  int  sys\_sem\_V(void)  {  int semaphore\_pointer;  // fetch the arguments  if(argint(0, &semaphore\_pointer) < 0)  return -1;  return sem\_V(semaphore\_pointer);  } |

对三个函数进行一层封装，通过argint的使用使其能接受任意数量的参数。

* **usys.S**

|  |
| --- |
| SYSCALL(sem\_init)  SYSCALL(sem\_P)  SYSCALL(sem\_V) |

用户调用系统函数定义。

* **user.h**

|  |
| --- |
| int sem\_init(int semaphore\_pointer,int value);  int sem\_P(int semaphore\_pointer,void\* chan);  int sem\_V(int semaphore\_pointer); |

用于c语言的接口，使得c语言可调用三个函数。

* **Makefile、test\_semaphore.c**

如字面意思，用于信号量的测试。

* **Makefile**

|  |
| --- |
| UPROGS=\  ……  \_test\_semaphore\  ……  EXTRA=\  ……  test\_semaphore.c\ |

UPROGS加上\_test\_semaphore\，EXTRA加上test\_semaphore.c\。

* **test\_semaphore.c**

分析几个主要函数。

|  |
| --- |
| void delay(double x)  {  volatile unsigned long long s = x\*10000000;  while (s>0)  --s;  } |

delay()函数如字面意思，根据参数，通过一个for循环延时，为了不让编译器优化掉for循环，形参加上了volatile关键字。

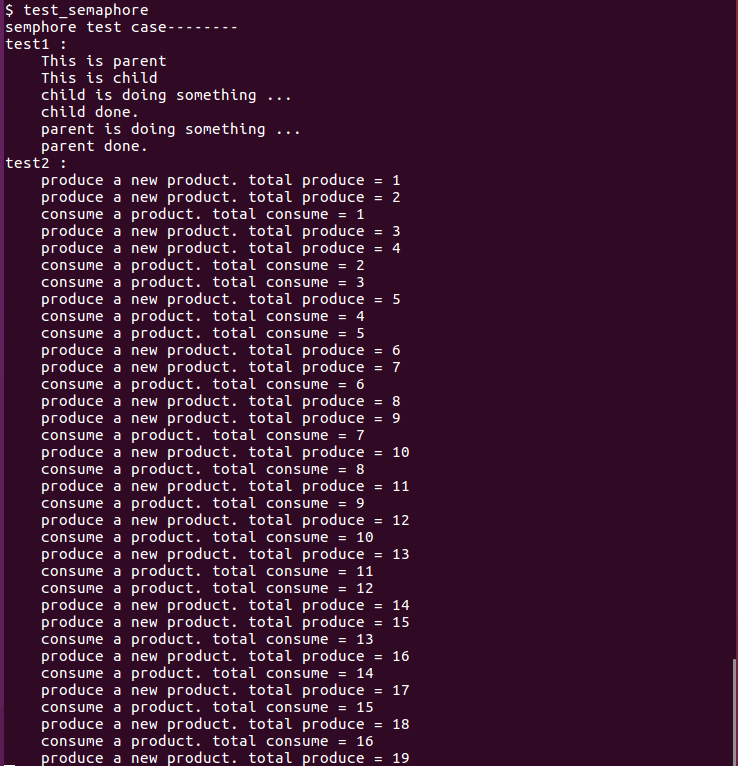
|  |
| --- |
| void test1()  {  sem\_init(0,1);  int cid = fork();  if(cid==0){  // This is child  printf(1, "This is child\n");  sem\_P(0,(void\*)getpid());  printf(1,"%s\n","child is doing something ...");  delay(1);  printf(1,"%s\n","child done.");  sem\_V(0);  exit();  }else{  // This is parent  printf(1, "This is parent\n");  sem\_P(0,(void\*)getpid());  printf(1,"%s\n","parent is doing something ...");  delay(2);  printf(1,"%s\n","parent done.");  sem\_V(0);  }  wait();  } |

test1，验证信号量的互斥访问功能。首先初始化0号信号量，使其拥有1个资源，然后通过fork()形成子进程，在子进程与父进程中分别进行调用sem\_P申请资源，并通过延时确保子进程与父进程的生命周期有交叉。

|  |
| --- |
| #define BufferNum 0  #define ProductNum 1  #define Mutex 2  double next\_pseudo\_random\_number(double now)  {  int a = 17;  int c = 9;  int m = 1387;  int x = now\*m;  double y = (x\*a+c)%m;  return y/m;  }  void test2()  {  sem\_init(BufferNum,3);  sem\_init(ProductNum,0);  sem\_init(Mutex,1);  int cid = fork();  if(cid==0){  // This is producer  double delta = 0.8;  double total = 0;  int total\_produce = 0;  while (total<3)  {  delta = next\_pseudo\_random\_number(delta);  delay(delta\*5);  total += delta;  sem\_P(BufferNum,(void\*)getpid());  sem\_P(Mutex,(void\*)getpid());  total\_produce += 1;  printf(1,"%s%d\n"," produce a new product. total produce = ",total\_produce);  sem\_V(Mutex);  sem\_V(ProductNum);  }  exit();  }else{  // This is consumer  double delta = 0.4;  double total = 0;  int total\_consume = 0;  while (total<3)  {  delta = next\_pseudo\_random\_number(delta);  delay(delta\*10);  total += delta;  sem\_P(ProductNum,(void\*)getpid());  sem\_P(Mutex,(void\*)getpid());  total\_consume += 1;  printf(1,"%s%d\n"," consume a product. total consume = ",total\_consume);  sem\_V(Mutex);  sem\_V(BufferNum);  }  }  wait();  } |

test2及相关函数，验证信号量用于生产-消费问题。将BufferNum、ProductNum、Mutex分别设为0、1、2号信号量，并分别初始化，BufferNum拥有3个资源，ProductNum拥有0个资源，Mutex拥有1个资源。随后通过fork形成两个进程，一个用于produce，一个用于consume。next\_pseudo\_random\_number通过线性同余法生成随机数，使得生产消费过程更为随机。

运行结果如下。



test1，实际运行时可明显看出，This is child之前的提示信息几乎同时出现，在This is child出现之后，会过一小段时间，再出现parent done和child is doing something，再过一段时间最后出现child done。因此，我们的信号量确实实现了进程之间互相通信的功能，可用于进程之前竞争有限资源时的协调工作。

test2，可以看到，total consume始终小于total produce，但不会比total produce小超过3。这代表我们的信号量起到了作用，一方面，只有produce进程生产出产品后consume进程才消费，另一方面，当produce进程的产品生产过多时，produce进程会等待consume进程消费掉buffer里面的产品后再生产。