# xv6消息队列

1. **核心代码**

* **proc.h**

|  |
| --- |
| struct msg\_queue{  struct spinlock lock;  char data[BUFFER\_SIZE][MSGSIZE];  int start;  int end;  int channel; // channel is used for sleep and wakeup  }MsgQueue[NPROC]; |

为每一个进程维护一个消息队列。

* **proc.c**

|  |
| --- |
| int  send\_msg(int sender\_pid, int rec\_pid, char \*msg){  int id = get\_process\_id(rec\_pid);  if(id == -1) return -1;  acquire(&MsgQueue[id].lock);  if((MsgQueue[id].end + 1) % BUFFER\_SIZE == MsgQueue[id].start){  // cprintf("Buffer is full\n");  release(&MsgQueue[id].lock);  return -1;  }  int i;  for(i = 0; i < MSGSIZE; i++){  MsgQueue[id].data[MsgQueue[id].end][i] = msg[i];  }  MsgQueue[id].end++;  MsgQueue[id].end %= BUFFER\_SIZE;  wakeup(&MsgQueue[id].channel);  release(&MsgQueue[id].lock);  return 0;  } |

发送信息，rec\_pid指定接收的进程，msg是发送的消息

|  |
| --- |
| int  recv\_msg(char\* msg){  int rec\_id = myproc()->pid;  int id = get\_process\_id(rec\_id);  if(id == -1) return -1;  acquire(&MsgQueue[id].lock);  while(1){  if(MsgQueue[id].end == MsgQueue[id].start){  sleep(&MsgQueue[id].channel, &MsgQueue[id].lock);  }  else{  int i;  for(i = 0; i < MSGSIZE; i++){  msg[i] = MsgQueue[id].data[MsgQueue[id].start][i];  }  MsgQueue[id].start++;  MsgQueue[id].start %= BUFFER\_SIZE;  release(&MsgQueue[id].lock);  break;  }  }  return 0;  } |

接收消息的函数。这是一个阻塞的函数。

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

主要目的是将send, recv注册为系统函数，以供用户调用。

* **defs.h**

|  |
| --- |
| int send\_msg(int sender\_pid, int rec\_pid, char \*msg);  int recv\_msg(char\* msg); |

添加三个函数的声明。

* **syscall.h**

|  |
| --- |
| #define SYS\_send 22  #define SYS\_recv 23 |

添加系统函数预编译。

* **syscall.c**

|  |
| --- |
| extern int sys\_send(void);  extern int sys\_recv(void);  ……  [SYS\_send] sys\_send,  [SYS\_recv] sys\_recv, |

系统函数关联。

* **sysproc.c**

|  |
| --- |
| int  sys\_send(void){  int sender\_pid, rec\_pid;  char\* msg;  // fetch the arguments  if(argint(0, &sender\_pid) < 0 || argint(1, &rec\_pid) < 0 || argptr(2, &msg, MSGSIZE) < 0)  return -1;  return send\_msg(sender\_pid, rec\_pid, msg);  }  int  sys\_recv(void){  char\* msg;  // fetch the arguments  if(argptr(0, &msg, MSGSIZE) < 0)  return -1;  return recv\_msg(msg);  } |

对两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(send)  SYSCALL(recv) |

用户调用系统函数定义。

* **user.h**

|  |
| --- |
| int send(int sender\_pid, int rec\_pid, void \*msg);  int recv(void \*msg); |

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\_unicast\  ……  EXTRA=\  ……  test\_unicast.c\ |

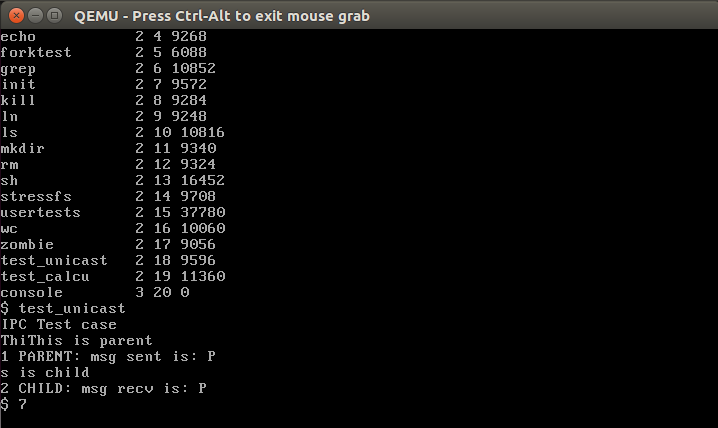
UPROGS加上\_test\_unicast\，EXTRA加上test\_unicast.c\。

* **test\_unicast.c**

测试函数，比较简单，父进程向子进程的消息队列写入消息。

|  |
| --- |
| #include "types.h"  #include "stat.h"  #include "user.h"  int main(void)  {  printf(1,"%s\n","IPC Test case");  int cid = fork();  if(cid==0){  // This is child  printf(1, "This is child\n");  char \*msg = (char \*)malloc(MSGSIZE);  int stat=-1;  while(stat==-1){  stat = recv(msg);  }  printf(1,"2 CHILD: msg recv is: %s \n", msg );  free(msg);  exit();  }else{  // This is parent  printf(1, "This is parent\n");  char \*msg\_child = (char \*)malloc(MSGSIZE);  msg\_child = "P";  send(getpid(),cid,msg\_child);  printf(1,"1 PARENT: msg sent is: %s \n", msg\_child );  free(msg\_child);  }  wait();  exit();  } |

测试结果：



# xv6基于信号的进程间通信

1. **核心代码**

* **proc.h**

|  |
| --- |
| // A signalqueue for every receiver  struct sig\_queue{  struct spinlock lock;  char sig\_arg[SIG\_QUE\_SIZE][MSGSIZE];  int sig\_num\_list[SIG\_QUE\_SIZE];  int start; // works as channel also  int end;  };  …  // Per-process state  struct proc {  …  // signal handling data:  sighandler\_t sig\_htable[NoSigHandlers];  // table of function pointers of different signal handlers  int sig\_handler\_busy;  // is the process executing the signal handler  struct sig\_queue SigQueue; // Queue for pending signals  }; |

为每一个进程维护一个信号队列。

* **proc.c**

设置信号的处理函数

|  |
| --- |
| int sig\_set(int sig\_num, sighandler\_t handler){  if(sig\_num < 0 || sig\_num >= NoSigHandlers)  return -1;  myproc()->sig\_htable[sig\_num] = handler;  return 0;  } |

发送信号

|  |
| --- |
| int sig\_send(int dest\_pid, int sig\_num, char \*sig\_arg){  if(sig\_num < 0 || sig\_num >= NoSigHandlers)  return -1;  int id = get\_process\_id(dest\_pid);  if(id == -1) return -1;  struct sig\_queue \*SigQueue = &ptable.proc[id].SigQueue;  acquire(&SigQueue->lock);  if(ptable.proc[id].sig\_htable[sig\_num] == 0){  release(&SigQueue->lock);  return 0;  }  if((SigQueue->end + 1) % SIG\_QUE\_SIZE == SigQueue->start){  // queue is full  release(&SigQueue->lock);  return -1;  }  // copy the data in queue  SigQueue->sig\_num\_list[SigQueue->end] = sig\_num;  int i;  for(i = 0; i < MSGSIZE; i++)  SigQueue->sig\_arg[SigQueue->end][i] = sig\_arg[i];  SigQueue->end++;  SigQueue->end %= SIG\_QUE\_SIZE;  wakeup(&SigQueue->start);  release(&SigQueue->lock);  return 0;  } |

阻塞当前进程直至收到信号。

|  |
| --- |
| int sig\_pause(void){  int pid = myproc()->pid;  int id = get\_process\_id(pid);  if(id == -1) return -1;  acquire(&ptable.proc[id].SigQueue.lock);  if(ptable.proc[id].SigQueue.end == ptable.proc[id].SigQueue.start){  sleep(&ptable.proc[id].SigQueue.start, &ptable.proc[id].SigQueue.lock);  }  release(&ptable.proc[id].SigQueue.lock);  return 0;  } |

系统会在从内核态返回用户态时检查信号状态和执行对应函数，这一个操作由下面的函数实现：

|  |
| --- |
| void execute\_signal\_handler(void){  struct proc \*curproc = myproc();  if(curproc == 0)  return;  if((curproc->tf->cs & 3) != DPL\_USER)  return;  struct sig\_queue \*SigQueue = &curproc->SigQueue;  // if(curproc->sig\_handler\_busy)  // return;  acquire(&SigQueue->lock);  if(SigQueue->start == SigQueue->end){  // no signal pending  release(&SigQueue->lock);  return;  }  int sig\_num = SigQueue->sig\_num\_list[SigQueue->start];  char\* msg = SigQueue->sig\_arg[SigQueue->start];  SigQueue->start++;  SigQueue->start %= SIG\_QUE\_SIZE;  // ustack\_esp : user stack pointer  uint ustack\_esp = curproc->tf->esp;  // copy the trap frame from kernel stack to user stack  // for retrieving it in the sig\_ret call  ustack\_esp -= sizeof(struct trapframe);  memmove((void \*)ustack\_esp, (void \*)curproc->tf, sizeof(struct trapframe));  // Modify the eip in tf(which is on kernel stack) to execute sig\_handler on returning from kernel mode  curproc->tf->eip = (uint)curproc->sig\_htable[sig\_num];  // Wrap and copy the sig\_ret asm code on user stack  void \*sig\_ret\_code\_addr = (void \*)execute\_sigret\_syscall\_start;  uint sig\_ret\_code\_size = ((uint)&execute\_sigret\_syscall\_end - (uint)&execute\_sigret\_syscall\_start);  // return addr for handler  ustack\_esp -= sig\_ret\_code\_size;  uint handler\_ret\_addr = ustack\_esp;  memmove((void \*)ustack\_esp, sig\_ret\_code\_addr, sig\_ret\_code\_size);  // Push the parameters for sig\_handler  // First push the char array  ustack\_esp -= MSGSIZE;  // Parameter addr(msg)  uint para1 = ustack\_esp;  memmove((void \*)ustack\_esp, (void \*)msg, MSGSIZE);  ustack\_esp -= sizeof(uint);  memmove((void \*)ustack\_esp, (void \*)&para1, sizeof(uint));  // push the return addr  ustack\_esp -= sizeof(uint);  memmove((void \*)ustack\_esp, (void \*)&handler\_ret\_addr, sizeof(uint));  curproc->tf->esp = ustack\_esp;  release(&SigQueue->lock);  return;  } |

在用户执行对应的信号处理函数过程后，会隐式调用下面函数以使得系统返回原有的系统态，之后继续执行trapret返回用户态，执行原有进程。

|  |
| --- |
| int sig\_ret(void){  // debug:  // cprintf("in sig ret\n");  struct proc \*curproc = myproc();  uint ustack\_esp = curproc->tf->esp;  uint sig\_ret\_code\_size = ((uint)&execute\_sigret\_syscall\_end - (uint)&execute\_sigret\_syscall\_start);  ustack\_esp += sizeof(uint) + MSGSIZE + sig\_ret\_code\_size;  // copy back the trapframe to kernel stack  memmove((void \*)curproc->tf, (void \*)ustack\_esp, sizeof(struct trapframe));  return 0;  } |

一个向多用户发送消息的系统调用

|  |
| --- |
| int send\_multi(int sender\_pid, int rec\_pids[], char \*msg, int rec\_length){  int i;  // debug:  // cprintf("rec\_length %d\n",rec\_length);  for(i = 0; i < rec\_length; i++){  sig\_send(rec\_pids[i], 0, msg);  }  return 0;  } |

以及一些用来做参数初始化的函数，在代码中已有标注。

* **trapasm.S**

如上文所说，返回用户态前调用信号处理函数。

|  |
| --- |
| # Return falls through to trapret...  .globl trapret  trapret:  # \*\*\*\*added\*\*\*\*\*\*\*\*\*\*\*\*\*\*  call execute\_signal\_handler  # \*\*\*\*  popal  popl %gs  popl %fs |

* **sigret\_syscall.S**

用来使信号处理函数隐式调用系统调用sig\_ret。

|  |
| --- |
| #include "syscall.h"  #include "traps.h"  .globl execute\_sigret\_syscall\_start;  .globl execute\_sigret\_syscall\_end;  execute\_sigret\_syscall\_start:  movl $SYS\_sig\_ret, %eax;  int $T\_SYSCALL;  ret  execute\_sigret\_syscall\_end: |

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

注册相关函数成为系统调用，和上节类似，此处略过

* **test\_calcu.c**

测试样例。程序大致流程如下：

1. 1..1000分成八份，八个子进程计算每部分的和。

2. 每个子进程将结果发送给原始进程，调用 sig\_pause 等待下一步信息

3. 父进程计算和和均值，调用 sig\_send 将均值发送给每个子进程。等待子进程结果。

4. 子进程的信号处理函数计算每部分的差的平方和，发送给父进程。

5. 父进程接收所有和，计算方差。

部分代码：

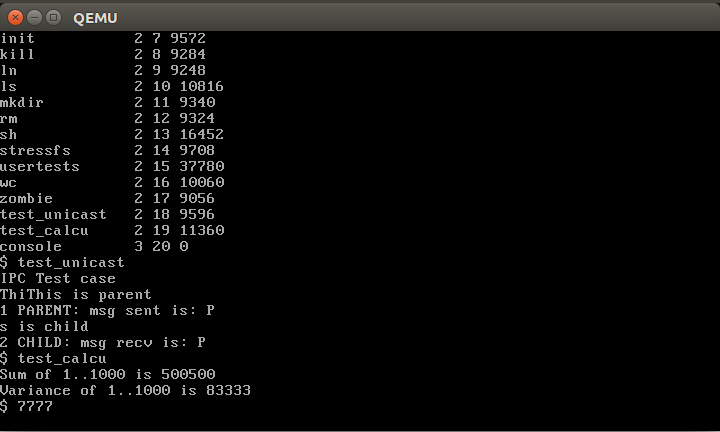
子进程：

|  |
| --- |
| int partial\_sum = 0;  for(i = start; i < end; i++)  partial\_sum += arr[i];  char \*msg = (char \*)malloc(MSGSIZE);  // pack the partial\_sum in msg  for(i = 0; i < 4; i++)  msg[i] = \*((char \*)&partial\_sum + i);  // send the partial sum to the co-oridinator process  send(pid, par\_pid, msg);  // printf(1, "\*\*\* child %d send partial\_sum %d \*\*\*\n", ind, msg);  // pause until msg is received  if(flag\_handler == 0) sig\_pause();  float partial\_var = 0.0;  for(i = start; i < end; i++){  float diff = (avg\_global - (float)arr[i]);  partial\_var += diff \* diff;  }  // pack the partial\_var in msg  for(i = 0; i < 4; i++)  msg[i] = \*((char\*)&partial\_var + i);  // send the partial var to the co-oridinator process  send(pid, par\_pid, msg);  free(cid);  free(msg);  exit(); |

父进程：

|  |
| --- |
| for(i = 0; i < NO\_CHILD; i++){  recv(msg);  tot\_sum += \*((int \*)msg);  // printf(1, "=== received tot\_sum %d: %d ===\n", i, tot\_sum);  }  float avg = (float)tot\_sum/size;  for(i = 0; i < 4; i++)  msg[i] = \*((char \*)&avg + i);  send\_multi(par\_pid, cid, msg, NO\_CHILD);  for(i = 0; i < NO\_CHILD; i++){  recv(msg);  variance += \*((float \*)msg);  // printf(1, "=== received variance %d: %d ===\n", i, variance);  }  variance /= (float)size;  for(i = 0; i < NO\_CHILD; i++)  wait();  free(msg); |

测试结果：



**参考文献：**

<https://github.com/ranxian/xv6-chinese>

<https://github.com/NatanJMai/xv6_IPC>

<https://github.com/manishtanwar/xv6-signals>

<https://github.com/techcentaur/XV6-Syscalls-IPC>