Ayush Patwari

Operating Systems [CS503]

Lab3 Answers and Discussion

Note: I have added "ayush edit" tags in the comments where changes have been made. To check the files edited please do a grep with "ayush"

Blocking Timed Message Send

1. Blocking Queue of Sending Processes: To use the efficiency of XINU queues which utilize less space compared to other queue due to relative pointers and also due to the invariant that a sender can be in the blocking queue of only one receiver I have created another queuetab called sendtab and implemented the corresponding functions for it in the queue.c, lab4.h, prototypes.h . The functions are prepended with s_ e.g instead of enqueue we use s_enqueue for sendtab. A list of queue "sendlists" is maintained which is basically a list per receiver.
2. sendb:
    a. if receiver is not able to receive
        i. if maxwait > 0 state = PR_SND, add to sleepq
        ii. if maxwait == 0 state = PR_SNDI
        iii. add to blocking queue of receiver
    b. on return if state still PR_SND then return timeout
    c. if receiver waiting then wake receiver and send msg
3. unblock implementation:

```
void unblock() {
        pid32 pid = - 1;
        struct procent *sendptr;

        // if a sender time out it will still have to removed
        while(!s_isempty(sendlists[currpid]) && isbadpid(pid))
                pid = s_dequeue(sendlists[currpid]);

        // if no process found waiting
        if(!isbadpid(pid) && pid > 1) {
                sendptr = &proctab[pid];
                sendptr->prsndflag = FALSE;
                //kprintf("\nTime %d, PID: %d unblocked PID %d", myglobalclock, currpid, pid);
                if(sendptr->prstate == PR_SND) {
                        unsleep(pid);
                        ready(pid);
                } else if (sendptr->prstate == PR_SNDI) {
                        ready(pid);
                }
        }
}
```

Since a process may timeout before it was blocked, we keep extracting from the queue until we reach end or find a valid sender process. This function is in receive.c and is called in receive(), recvclr() and rectvime() just before exiting the method to unblock any waiting processes.

4. Ripple effect: Since I used a completely different quetable the kernel modification except in rec* methods and senb is minimum. The unsleep method has to take care of the PR_SND state and rest of the code is pretty much the same.

Note: I have returned TIMEOUT on timeout of a sender which means a receiver who expected a msg will be waiting for some process to still send the message.

5. Tests: I have tun two tests where the receiver initially sleeps for 50 and 500ms respectively to see how many processes timeout and how many are able to send.

```
void receiver() {

        int numsender = 8;
        int sleeptime = 500;
        umsg32 msg[numsender];
        int i = 0;
        sleepms(sleeptime);
        for(; i < numsender; i++) {
                msg[i] = receive();
                kprintf("\nTime %d, PID: %d msg read: %d", myglobalclock, currpid, msg[i]);

        }
}
```

```
void sender(pid32 pid, int maxtime, umsg32 msg) {

        //sleepms(currpid * 1000);

        status st = sendb(pid, msg, maxtime);
        if(st == TIMEOUT) {
                kprintf("\nTime %d, PID %d Send time out", myglobalclock, getpid());
        }
}
```

```
void test_sendb() {

        pid32 pid;

        pid = create(receiver, 2048, 20, "Proc1", 0, NULL);
        resched_cntl(DEFER_START);
        resume( pid);
        resume( create(sender, 2048, 20, "Proc2", 3, pid, 100, 200));
        resume( create(sender, 2048, 20, "Proc3", 3, pid, 0, 300));
        resume( create(sender, 2048, 20, "Proc4", 3, pid, 10, 400));
        resume( create(sender, 2048, 20, "Proc5", 3, pid, 60, 500));
        resume( create(sender, 2048, 20, "Proc6", 3, pid, 50, 600));
        resume( create(sender, 2048, 20, "Proc7", 3, pid, 20, 700));
        resume( create(sender, 2048, 20, "Proc8", 3, pid, 150, 800));
        resume( create(sender, 2048, 20, "Proc9", 3, pid, 0, 900));

        resched_cntl(DEFER_STOP);
}
```

```
Time 1, PID: 4, Sent 200
Time 2, PID: 5, maxwait 0 Send Block.
Time 3, PID: 6, maxwait 10 Send Block.
Time 4, PID: 7, maxwait 60 Send Block.
Time 5, PID: 8, maxwait 50 Send Block.
Time 6, PID: 9, maxwait 20 Send Block.
Time 7, PID: 10, maxwait 150 Send Block.
Time 8, PID: 11, maxwait 0 Send Block.
Time 13, PID 6 Send time out
Time 26, PID 9 Send time out
Time 51, PID: 5, Sent 300
Time 52, PID: 3 msg read: 200
Time 53, PID: 7, Sent 500
Time 54, PID: 3 msg read: 300
Time 55, PID 8 Send time out
Time 56, PID: 10, Sent 800
Time 57, PID: 3 msg read: 500
Time 58, PID: 11, Sent 900
Time 59, PID: 3 msg read: 800
Time 60, PID: 3 msg read: 900
Time 61, PID 3 waiting to receive!!
```

Test 1: Receiver sleep time 50ms

```
Time 1, PID: 4, Sent 200
Time 2, PID: 5, maxwait 0 Send Block.
Time 3, PID: 6, maxwait 10 Send Block.
Time 4, PID: 7, maxwait 60 Send Block.
Time 5, PID: 8, maxwait 50 Send Block.
Time 6, PID: 9, maxwait 20 Send Block.
Time 7, PID: 10, maxwait 150 Send Block.
Time 8, PID: 11, maxwait 0 Send Block.
Time 13, PID 6 Send time out
Time 26, PID 9 Send time out
Time 55, PID 8 Send time out
Time 64, PID 7 Send time out
Time 157, PID 10 Send time out
Time 501, PID: 5, Sent 300
Time 502, PID: 3 msg read: 200
Time 503, PID: 11, Sent 900
Time 504, PID: 3 msg read: 300
Time 505, PID: 3 msg read: 900
Time 506, PID 3 waiting to receive!!
```

Test 2: Receiver sleep time 500ms

We see that

1. the senders which are not able to send within their maxwait are timed out
2. the senders with maxwait 0 do not timeout in both cases
3. the messages are received in order

Asynchronous message receive

1. Implementation: the pointers to callback and buffer are added to the procent entry of process.h They are names recvcb and uderbuf. This takes up only 8 bytes and hence is not space constraining. Registration method is as below. The check for an async message is made in resched.c just after it returns from being context switched.

```
syscall registercb(
        umsg32  *buf,   /* Pointer to the message buffer */
        msgcb           cb              /* Callback function to be executed */
        )
{
        /* LAB4TODO */
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */

        mask = disable();

        prptr = &proctab[currpid];
        // error if
        // 1. already registered
        // 2. NULL callback
        // 3. NULL buffer
        if(prptr->recvcb != NULL || cb == NULL || buf == NULL) {
                restore(mask);
                return SYSERR;
        }

        prptr->userbuf = buf;
        prptr->recvcb = cb;

        restore(mask);
        return OK;
}
```

```
void async_rec(struct procent *ptr) {

        if(ptr->prhasmsg && ptr->recvcb != NULL) {

                // copy to buf
                *(ptr->userbuf) = ptr->prmsg;
                // execute callback
                ptr->recvcb(ptr->prmsg);
                ptr->prhasmsg = FALSE;
                unblock();
        }
}
```

```
  /* Force context switch to highest priority ready process */
  //kprintf("\nResched: Process %d time %d", firstid(readylist), myglobalclock);
  currpid = dequeue(readylist);

  ptnew = &proctab[currpid];
  ptnew->prstate = PR_CURR;
  preempt = QUANTUM;              /* Reset time slice for process */
  ctxsw(&ptold->prstkptr, &ptnew->prstkptr);

  /* Old process returns here when resumed */
  // assuming both handlers cannot be registered at the same time
  sighandler(ptold);
  async_rec(ptold);
  return;
```

2. Testing: The testing method is similar. Only now senders send with delay to check for asynchronous delivery. To check for isolation/protection I have printed currpid in the callback method. Idea is that if the callback is not called from some other process's context then the protection is still maintained.

```
void rec3() {

        if(registercbsig(ARECV, recvsighandler, 0 ) != OK) {

                kprintf("\nRecv Signal Handler Registration failed.");
                return;
        }

        if(registercbsig(AALRM, myalarmhandler, 10000 ) != OK) {

                kprintf("\nRecv Signal Handler Registration failed.");
                return;
        }
        // while away cycles till you get a msg
        while(1) {
        }
}
```

```
void rec2() {

        umsg32 buf = 10;
        if(registercb(&buf, myrecvhandler ) != OK) {

                kprintf("\nRecv Handler Registration failed.");
                return;
        }

        if(registercbsig(AALRM, myalarmhandler, 10000 ) != OK) {

                kprintf("\nRecv Signal Handler Registration failed.");
                return;
        }
        // while away cycles till you get a msg
        while(1) {

        }
}
```

```
void sender(pid32 pid, int maxtime, umsg32 msg) {

        sleepms(currpid * 100);

        status st = sendb(pid, msg, maxtime);
        if(st == TIMEOUT) {
                kprintf("\nTime %d, PID %d Send time out", myglobalclock, getpid());
        }
}
```

```
Time 403, PID: 4, Sent 200
Time 404, PID: 3 Async msg received = 200 buf 200

Time 503, PID: 5, Sent 300
Time 504, PID: 3 Async msg received = 300 buf 300

Time 603, PID: 6, Sent 400
Time 604, PID: 3 Async msg received = 400 buf 400

Time 703, PID: 7, Sent 500
Time 704, PID: 3 Async msg received = 500 buf 500

Time 803, PID: 8, Sent 600
Time 804, PID: 3 Async msg received = 600 buf 600

Time 903, PID: 9, Sent 700
Time 904, PID: 3 Async msg received = 700 buf 700

Time 1003, PID: 10, Sent 800
Time 1004, PID: 3 Async msg received = 800 buf 800

Time 1103, PID: 11, Sent 900
Time 1104, PID: 3 Async msg received = 900 buf 900
```

We see that the callback is called as soon as a message is received and the receiver process is context switched in. The lag is so that we can implement callback from the context of the receiver. The buffer value is also shown to be updated.

BONUS:

1. Implementation: I have added two more callbacks to the procent entry in process.h The register method is as shown

```
syscall registercbsig(
        uint16  sig,            /* Signal to register */
        sigcb           cb,                     /* Callback function to be executed */
        uint32  optarg  /* Optional argument */
        )
{
        /* LAB4TODO */
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */

        mask = disable();

        prptr = &proctab[currpid];
        if(sig == ARECV && (cb != NULL && prptr->arecvcb == NULL)) {
                prptr->arecvcb = cb;
                restore(mask);
                return OK;
        }

        if(sig == AALRM && (cb != NULL && prptr->alarmcb == NULL && optarg >= 0)) {
                prptr->alarmcb = cb;
                prptr->alrmtime = (myglobalclock + optarg);
                restore(mask);
                return OK;
        }

        restore(mask);
        return SYSERR;
}
```

2. The callbacks are checked in the resched just after a process is context switched back in.

```
void sighandler(struct procent *ptr) {

        if(ptr->prhasmsg && ptr->arecvcb != NULL) {

                ptr->arecvcb();

        } else if (ptr->alarmcb != NULL) {
                if(myglobalclock >= ptr->alrmtime) {

                        ptr->alarmcb();
                        ptr->alarmcb = NULL;
                }
        }
}
```

Here I have assumed that receive will be called in the receiver callback and hene no need to unblock here.

3. Tests

```
void rec3() {

        if(registercbsig(ARECV, recvsighandler, 0 ) != OK) {

                kprintf("\nRecv Signal Handler Registration failed.");
                return;
        }

        if(registercbsig(AALRM, myalarmhandler, 1000 ) != OK) {

                kprintf("\nRecv Signal Handler Registration failed.");
                return;
        }
        // while away cycles till you get a msg
        while(1) {
        }
}
```

```
int recvsighandler() {

        umsg32 msg = receive();
        kprintf("\nPID: %d msg received = %d\n", currpid, msg);
        return OK;
}
```

```
int myalarmhandler() {
        kprintf("\nTime %d PID: %d ALARM!!!!", myglobalclock, currpid);
        return OK;
}
```

```
void test_bonus() {

        pid32 pid;

        pid = create(rec3, 2048, 20, "Rec1", 0, NULL);
        resched_cntl(DEFER_START);
        resume( pid);
        resume( create(sender, 2048, 20, "Proc2", 3, pid, 100, 200));
        resume( create(sender, 2048, 20, "Proc3", 3, pid, 100, 300));
        resume( create(sender, 2048, 20, "Proc4", 3, pid, 0, 400));
        resume( create(sender, 2048, 20, "Proc5", 3, pid, 100, 500));
        resume( create(sender, 2048, 20, "Proc6", 3, pid, 100, 600));
        resume( create(sender, 2048, 20, "Proc7", 3, pid, 100, 700));
        resume( create(sender, 2048, 20, "Proc8", 3, pid, 0, 800));
        resume( create(sender, 2048, 20, "Proc9", 3, pid, 100, 900));
        resched_cntl(DEFER_STOP);
}
```

Test scenario is same with addition that a alarmhandler is registered to be invoked after

```
Time 403, PID: 4, Sent 200
Time 404 PID: 3 Signal msg received = 200

Time 503, PID: 5, Sent 300
Time 504 PID: 3 Signal msg received = 300

Time 603, PID: 6, Sent 400
Time 604 PID: 3 Signal msg received = 400

Time 703, PID: 7, Sent 500
Time 704 PID: 3 Signal msg received = 500

Time 803, PID: 8, Sent 600
Time 804 PID: 3 Signal msg received = 600

Time 903, PID: 9, Sent 700
Time 904 PID: 3 Signal msg received = 700

Time 1001 PID: 3 ALARM!!!!
Time 1003, PID: 10, Sent 800
Time 1004 PID: 3 Signal msg received = 800

Time 1103, PID: 11, Sent 900
Time 1104 PID: 3 Signal msg received = 900
```

1000ms. We see that result is similar to async receive and also alarm handler has been invoked.