Assignment 3

Praneta Paithankar

Question 1. Compare following cases for xinu vs pointer based implementation (assume n number of nodes for each scenario)

a) Which implementation consumes more space? Why? Give an objective answer comparing your implementation vs Xinu.

b) Which implementation may take less time for basic process manipulation? A comparison for time to FIFO and non-FIFO queues with your implementation vs Xinu.

c) Advantage and disadvantage of pointer based implementation over xinu version.

Solution a) Pointer based doubly linked list consumes more memory as it stores more fields.
Xinu structure is as follow:

```
struct  qentry  {
        int32   qkey;             //4 bytes
        qid16   qnext;            //2 bytes
        qid16   qprev;            //2 bytes
};
```

Therefore, xinu requires 8 bytes to store one qentry. In case of pointer based implementation, structure of qentry is

```
struct  qentry  {
        int32   qkey;             //4 bytes
        pid32   process_id;       //4 bytes
        struct qentry*  qnext;    //4 bytes
        struct qentry*  qprev;    //4 bytes
};
```

Therefore, size of qentry is 16 bytes. Pointer based implementation stores process_id field which requires 4 bytes. Xinu implements qnext and qprev as qid16 which take 2 bytes each. In case of pointer based implementation, qprev and qnext are pointers to qentry structure. So they require 4 bytes each.

b) Pointer based implementation will take more time to execute basic process manipulation. For FIFO queue, we are removing first element of list and adding the new element before the tail node. So pointer based implementation will take more time to execute it. In case of non-FIFO queue, we are comparing the key value of each process with new process to insert it. So non-FIFO queue is relatively slower than FIFO queue. non-FIFO queue will take more time if implemented by using pointer based implementation.

c) Advantage of pointer based implementation :

- As we are using pointers to refer the qentry, we can use non-contiguous memory block to store queue table. Xinu requires contiguous memory to store queue table.By using pointer based implementation, memory blocks are used efficiently.

Disadvantage of pointer based implementation:

- It requires more memory to store the qentry structure than xinu implementation.
- It is slower than xinu implementation.

Question 2. Explain what it means to be a valid queue ID in Xinu.

Solution In xinu,queue id are assigned from NPROC to NQENT-1.In this case, NPROC = number of processes and $NQENT = NPROC + 4 + NSEM + NSEM$.

Question 3. Rewrite resched to have an explicit parameter giving the disposition of the currently executing process, and examine the assembly code generated to determine the number of instructions executed in each case.

Solution Assembly code of function resched2:

```
resched2.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <resched2>:
/*------------------------------------------------------------------------
 *  resched2  -  Reschedule processor to highest priority eligible process
 *------------------------------------------------------------------------
 */
void    resched2(int32 next_state) /* Assumes interrupts are disabled*/
{
   0:   e92d4070        push    {r4, r5, r6, lr}
        struct procent *ptold;  /* Ptr to table entry for old process   */
        struct procent *ptnew;  /* Ptr to table entry for new process   */
        /*next state should have value from 0 to 7*/
        if(!(next_state>=0 && next_state<=7))
   4:   e3500007        cmp     r0, #7
   8:   88bd8070        pophi   {r4, r5, r6, pc}
                return;
        }

        /* If rescheduling is deferred, record attempt and return */

        if (Defer.ndefers > 0) {
   c:   e59f30e4        ldr     r3, [pc, #228]  ; f8 <resched2+0xf8>
  10:   e5933000        ldr     r3, [r3]
  14:   e3530000        cmp     r3, #0
  18:   da000003        ble     2c <resched2+0x2c>
                Defer.attempt = TRUE;
  1c:   e3a02001        mov     r2, #1
  20:   e59f30d0        ldr     r3, [pc, #208]  ; f8 <resched2+0xf8>
  24:   e5c32004        strb    r2, [r3, #4]
                return ;
  28:   e8bd8070        pop     {r4, r5, r6, pc}
        }

        /* Point to process table entry for the current (old) process */

        ptold = &proctab[currpid];
  2c:   e59f30c8        ldr     r3, [pc, #200]  ; fc <resched2+0xfc>
  30:   e5933000        ldr     r3, [r3]
  34:   e0634183        rsb     r4, r3, r3, lsl #3
  38:   e1a04184        lsl     r4, r4, #3
  3c:   e59fc0bc        ldr     ip, [pc, #188]  ; 100 <resched2+0x100>
  40:   e084500c        add     r5, r4, ip
        ptold->prstate=next_state;
  44:   e18400bc        strh    r0, [r4, ip]
        if (next_state == PR_CURR) {  /* Process remains eligible */
  48:   e3500001        cmp     r0, #1
  4c:   1a00000f        bne     90 <resched2+0x90>
                if (ptold->prprio > firstkey(readylist)) {
  50:   e1d520f2        ldrsh   r2, [r5, #2]
  54:   e59f10a8        ldr     r1, [pc, #168]  ; 104 <resched2+0x104>
  58:   e1d110b0        ldrh    r1, [r1]
  5c:   e59f00a4        ldr     r0, [pc, #164]  ; 108 <resched2+0x108>
  60:   e6bf6071        sxth    r6, r1
  64:   e0806186        add     r6, r0, r6, lsl #3
  68:   e1d660f4        ldrsh   r6, [r6, #4]
  6c:   e7900186        ldr     r0, [r0, r6, lsl #3]
```

```
70:     e1520000        cmp     r2, r0
74:     c8bd8070        popgt   {r4, r5, r6, pc}
                        return;
                }

                /* Old process will no longer remain current */

                ptold->prstate = PR_READY;
78:     e3a00002        mov     r0, #2
7c:     e18400bc        strh    r0, [r4, ip]
                insert(currpid, readylist, ptold->prprio);
80:     e1a00003        mov     r0, r3
84:     e6bf1071        sxth    r1, r1
88:     ebfffffe        bl      0 <insert>
8c:     ea000007        b       b0 <resched2+0xb0>
        }
        if(next_state==PR_READY){ // Process is ready to execute
90:     e3500002        cmp     r0, #2
94:     1a000005        bne     b0 <resched2+0xb0>
                insert(currpid,readylist,ptold->prprio);
98:     e59f305c        ldr     r3, [pc, #92]     ; fc <resched2+0xfc>
9c:     e5930000        ldr     r0, [r3]
a0:     e59f305c        ldr     r3, [pc, #92]     ; 104 <resched2+0x104>
a4:     e1d310f0        ldrsh   r1, [r3]
a8:     e1d520f2        ldrsh   r2, [r5, #2]
ac:     ebfffffe        bl      0 <insert>

        }
        /* Force context switch to highest priority ready process */

        currpid = dequeue(readylist);
b0:     e59f304c        ldr     r3, [pc, #76]     ; 104 <resched2+0x104>
b4:     e1d300f0        ldrsh   r0, [r3]
b8:     ebfffffe        bl      0 <dequeue>
bc:     e59f3038        ldr     r3, [pc, #56]     ; fc <resched2+0xfc>
c0:     e5830000        str     r0, [r3]
        ptnew = &proctab[currpid];
c4:     e0600180        rsb     r0, r0, r0, lsl #3
c8:     e1a02180        lsl     r2, r0, #3
cc:     e59f302c        ldr     r3, [pc, #44]     ; 100 <resched2+0x100>
d0:     e0821003        add     r1, r2, r3
        ptnew->prstate = PR_CURR;
d4:     e3a00001        mov     r0, #1
d8:     e18200b3        strh    r0, [r2, r3]
        preempt = QUANTUM;                /* Reset time slice for process */
dc:     e3a02002        mov     r2, #2
```

```
  e0:   e59f3024        ldr     r3, [pc, #36]   ; 10c <resched2+0x10c>
  e4:   e5832000        str     r2, [r3]
#ifdef MMU
        FlushTLB();
        setPageTable();
#endif/*MMU*/

        ctxsw(&ptold->prstkptr, &ptnew->prstkptr);
  e8:   e2850004        add     r0, r5, #4
  ec:   e2811004        add     r1, r1, #4
  f0:   ebfffffe        bl      0 <ctxsw>
  f4:   e8bd8070        pop     {r4, r5, r6, pc}
        ...
```

resched function contains 50 instructions whereas resched2 contains 61 instructions to execute.

All calls will check for Defer.ndefers. If its value is not 0 then it will proceed to reschedule the processes. From instruction 0 to 18 will execute, as defer.ndefers is < 0. It will execute instruction at 2c directly. From instruction 2c to 4c is executed for all states.

There are 8 process states defined in xinu. Scheduler will perform as described below if next_state is defined as follow:

1. PR_FREE:
   - In this case, it means that process is about to finish its execution.
   - If process calls resched2 then scheduler should find highest priority process from ready list and give control to that process.
   - At 4c, as value of next_state is not PR_CURR, it will execute branch instruction at 90.
   - After executing instruction 94, it will move to b0 directly as r0 is not 2. i.e next_state is not PR_CURR. From b0 to c0, it will dequeue process from queue. It will execute all instructions till f4. These instructions assigns control to new currpid.
   - Number of instructions executed are 36.
2. PR_CURR
   - In this case, it means that next state of process is currently executing.
   - If process calls resched2 then scheduler should compare highest priority process from ready list with current process. If the priority of current process is greater than the highest priority process from ready list, scheduler will consider the current process to execute next. Otherwise it will consider a process from readylist to execute and add the current process to readylist and change its status to PR_READY.

- At instruction 48, r0 is 1. It will continue to execute instruction at 50. It will compare the priority of current process to first element in readylist. If r2 i.e. priority of current process is greater then it will pop pc, r4 , r5, r6 from stack and return from this function. Else it will execute instruction from 78 to 94.After 94, it will execute instructions from b0 to f4.
- Number of instructions executed are 26 if current process priority is greater than first key in ready queue.
- Otherwise Number of instructions executed are 50.

3. PR_READY

   - In this case, it means that next state of process is ready.That means process should be inserted into ready list. Scheduler will choose the process from ready list who has the highest priority and it will execute that process next.
   - At 4c, it will go to instruction at 90 directly. After executing instruction 94, it will move to 98.It will insert process in ready queue by using instruction from 98 to ac. From b0 to c0, it will dequeue process from queue. It will execute all instructions till f4.
   - Number of instructions executed are 42.

4. PR_RECV

   - In this case, next state of process is waiting for message.
   - Scheduler will choose the process from ready list who has the highest priority and it will execute that process next.
   - At 4c, as value of next_state is not PR_CURR, it will execute branch instruction at 90.
   - After executing instruction 94, it will move to b0 directly as r0 is not 2. From b0 to c0, it will dequeue process from queue. It will execute all instructions till f4.
   - Number of instructions executed are 36.

5. PR_SLEEP

   - In this case, next state of process is waiting for timer.
   - Scheduler will choose the process from ready list who has the highest priority and it will execute that process next.
   - At 4c, as value of next_state is not PR_CURR, it will execute branch instruction at 90.
   - After executing instruction 94, it will move to b0 directly as r0 is not 2. From b0 to c0, it will dequeue process from queue. It will execute all instructions till f4.

- Number of instructions executed are 36.

6. PR_SUSP

   - In this case, next state of process is suspended.
   - Scheduler will choose the process from ready list who has the highest priority and it will execute that process next.
   - At 4c, as value of next_state is not PR_CURR, it will execute branch instruction at 90.
   - After executing instruction 94, it will move to b0 directly as r0 is not 2. From b0 to c0, it will dequeue process from queue. It will execute all instructions till f4.
   - Number of instructions executed are 36.

7. PR_WAIT

   - In this case, next state of process is waiting for semaphore.
   - Scheduler will choose the process from ready list who has the highest priority and it will execute that process next.
   - At 4c, as value of next_state is not PR_CURR, it will execute branch instruction at 90.
   - After executing instruction 94, it will move to b0 directly as r0 is not 2. From b0 to c0, it will dequeue process from queue. It will execute all instructions till f4.
   - Number of instructions executed are 36.

8. PR_RECTIM

   - The next state of process is waiting for a timer or a message whichever occurs first.
   - Scheduler will choose the process from ready list who has the highest priority and it will execute that process next.
   - At 4c, as value of next_state is not PR_CURR, it will execute branch instruction at 90.
   - After executing instruction 94, it will move to b0 directly as r0 is not 2. From b0 to c0, it will dequeue process from queue. It will execute all instructions till f4.
   - Number of instructions executed are 36.