Lab 6: the big picture

- You add
  - YKQCreate()
  - YKQPend()
  - YKQPost()
- You modify tick handler so that it posts one message to a queue each time it runs. (The queue is emptied by a task.)
- Higher priority task can hog CPU for 5-cycle intervals, letting messages queue up.
  - Execution of high priority task triggered by any key (keypress ISR sets flag)


```
/*
    File: lab6app.c
    Description: Application code for ECEn 425 lab 6 (Message queues)
*/

#include "clib.h"
#include "yakk.h"          /* contains kernel definitions */
#include "lab6defs.h"      /* contains user's definitions */

#define TASK_STACK_SIZE  512    /* stack size in words */
#define MSGQSIZE         10

struct msg MsgArray[MSGARRAYSIZE]; /* buffers for message content */

int ATaskStk[TASK_STACK_SIZE];  /* a stack for each task */
int BTaskStk[TASK_STACK_SIZE];
int STaskStk[TASK_STACK_SIZE];

int GlobalFlag;

void *MsgQ[MSGQSIZE];      /* space for message queue */
YKQ *MsgQPtr;              /* global handle for queue */
```

```
/* File: lab6defs.h */

#define MSGARRAYSIZE  20

struct msg
{
    int tick;
    int data;
};
```

Big enough that queue should not overflow.


```
void ATask(void)      /* empties msg queue */
{
    struct msg *tmp;
    int min, max, count;

    min = 100;
    max = 0;
    count = 0;

    while (1)
    {  /* get next msg */
        tmp = (struct msg *) YKQPend(MsgQPtr);

        /* check sequence count in msg;
            were msgs dropped? */
        if (tmp->tick != count+1)
        {
            print("!  Dropped msgs: tick ", 21);
            if (tmp->tick - (count+1) > 1) {
                printInt(count+1);
                printChar('-');
                printInt(tmp->tick-1);
                printNewLine();
            }
            else {
                printInt(tmp->tick-1);
                printNewLine();
            }
        }
```

```
        /* update sequence count */
        count = tmp->tick;

        /* process data; update statistics
            for this sample */
        if (tmp->data < min)
            min = tmp->data;
        if (tmp->data > max)
            max = tmp->data;

        /* output min, max, tick values */
        print("Ticks: ", 7);
        printInt(count);
        print("\t", 1);
        print("Min: ", 5);
        printInt(min);
        print("\t", 1);
        print("Max: ", 5);
        printInt(max);
        printNewLine();
    }
}
```

This task empties queue of messages generated by the tick handler, one message per tick.

BTask is a CPU hog that keeps ATask from running. Its busy bursts are triggered by the press of any key and last for five clock ticks each.

```
void BTask(void)  /* saturates CPU for 5 ticks */
{
    int busycount,curval,j,flag,chcount;
    unsigned tickNum;

    curval = 1001;
    chcount = 0;
    while (1)
    {
        YKDelayTask(2);
        if (GlobalFlag == 1)
        {  /* flag set -- loop for 5 ticks */
            YKEnterMutex();
            busycount = YKTickNum;
            YKExitMutex();
```

```
        while (1) {
            YKEnterMutex();
            tickNum = YKTickNum;
            YKExitMutex();

            if (tickNum >= busycount + 5) break;

            curval += 2; /* evaluate next number */
            flag = 0;
            for (j = 3; (j*j) < curval; j += 2) {
                if (curval % j == 0) {
                    flag = 1;
                    break;
                }
            }
            if (!flag) {
                printChar('.'); /* output marker for each # */
                if (++chcount > 75) {
                    printNewLine();
                    chcount = 0;
                }
            }
        }
        printNewLine();
        chcount = 0;
        GlobalFlag = 0; /* clear flag */
    }
  }
}
```



```
void STask(void) /* tracks statistics */
{
    unsigned max, switchCount, idleCount;
    int tmp;

    YKDelayTask(1);
    printString("Welcome to the YAK kernel\r\n");
    printString("Determining CPU capacity\r\n");
    YKDelayTask(1);
    YKIdleCount = 0;
    YKDelayTask(5);
    max = YKIdleCount / 25;
    YKIdleCount = 0;
    YKNewTask(BTask, (void *) &BTaskStk[TASK_STACK_SIZE], 10);
    YKNewTask(ATask, (void *) &ATaskStk[TASK_STACK_SIZE], 20);
    while (1) {
        YKDelayTask(20);
        YKEnterMutex();
        switchCount = YKCtxSwCount;
        idleCount = YKIdleCount;
        YKExitMutex();
        printString("<<<<< Context switches: ");
        printInt((int)switchCount);
        printString(", CPU usage: ");
        tmp = (int) (idleCount/max);
        printInt(100-tmp);
        printString("% >>>>>\r\n");
        YKEnterMutex();
        YKCtxSwCount = 0;
        YKIdleCount = 0;
        YKExitMutex();
    }
}
```

The stat task, just like lab 5.
Starts tasks A and B.
Prints statistics every 20 ticks.


```
void main(void)
{
    YKInitialize( );

    /* create queue, at least one user task, etc. */
    GlobalFlag = 0;
    MsgQPtr = YKQCreate(MsgQ, MSGQSIZE);
    YKNewTask(STask, (void *) &STaskStk[TASK_STACK_SIZE], 30);

    YKRun( );
}
```

Lab 6: Changes to interrupt handlers

- Keypress handler
  - No longer generates output – just sets **GlobalFlag** to 1.
- Reset handler
  - No modification from previous labs.
- User code tick handler
  - No longer generates "--TICK X--" output
    - Puts a message into the message queue each time it runs.
  - You are given the code for this. (See next slide.)


```
/*  File: lab6inth.c
    Description: Sample interrupt handler code  for EE 425 lab 6 (Message queues) */

#include "lab6defs.h"
#include "yakk.h"
#include "clib.h"

extern YKQ *MsgQPtr;
extern struct msg MsgArray[];
extern int GlobalFlag;

void myreset(void) {
   exit(0);
}

void mykeybrd(void) {
   GlobalFlag = 1;
}

void mytick(void) {
   static int next = 0;
   static int data = 0;
   /* create a message with tick (sequence #) and pseudo-random data */
   MsgArray[next].tick = YKTickNum;
   data = (data + 89) % 100;
   MsgArray[next].data = data;
   if (YKQPost(MsgQPtr, (void *) &(MsgArray[next])) == 0)
      printString("  TickISR: queue overflow! \n");
   else if (++next >= MSGARRAYSIZE)
      next = 0;
}
```

```
/* File: lab6defs.h */

#define MSGARRAYSIZE  20

struct msg
{
   int tick;
   int data;
};
```



```
Welcome to the YAK kernel
Determining CPU capacity
Ticks: 1      Min: 89 Max: 89
Ticks: 2      Min: 78 Max: 89
Ticks: 3      Min: 67 Max: 89
Ticks: 4      Min: 56 Max: 89
Ticks: 5      Min: 45 Max: 89
Ticks: 6      Min: 34 Max: 89
Ticks: 7      Min: 23 Max: 89
Ticks: 8      Min: 12 Max: 89
Ticks: 9      Min: 1  Max: 89
Ticks: 10     Min: 1  Max: 90
Ticks: 11     Min: 1  Max: 90
Ticks: 12     Min: 1  Max: 90
Ticks: 13     Min: 1  Max: 90
Ticks: 14     Min: 1  Max: 90
Ticks: 15     Min: 1  Max: 90
Ticks: 16     Min: 1  Max: 90
Ticks: 17     Min: 1  Max: 90
Ticks: 18     Min: 1  Max: 90
Ticks: 19     Min: 1  Max: 91
Ticks: 20     Min: 1  Max: 91
.....
Ticks: 21     Min: 1  Max: 91
Ticks: 22     Min: 1  Max: 91
Ticks: 23     Min: 1  Max: 91
Ticks: 24     Min: 1  Max: 91
Ticks: 25     Min: 1  Max: 91
Ticks: 26     Min: 1  Max: 91
Ticks: 27     Min: 1  Max: 91
<<<<< Context switches: 62, CPU usage: 11% >>>>>
Ticks: 28     Min: 1  Max: 92
Ticks: 29     Min: 1  Max: 92
```

Sample output:
default tick rate

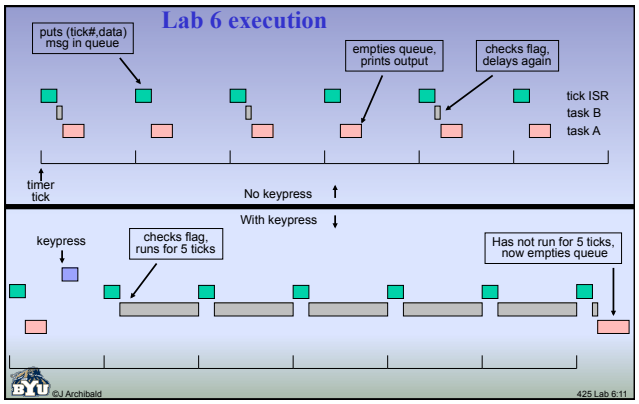
Requirements at default tick rate:
- No messages dropped
- Normal Stat task output

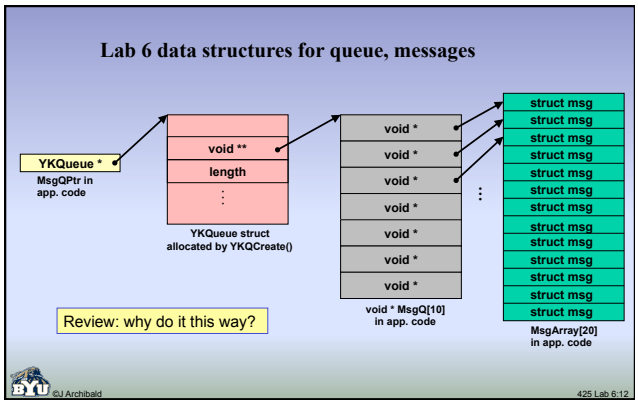
Key is pressed,
Keypress ISR sets GlobalFlag,
BTask hogs CPU for 5 tick intervals


```
Ticks: 175     Min: 0  Max: 99
Ticks: 176     Min: 0  Max: 99
Ticks: 177     Min: 0   TickISR: queue overflow!
Max: 99
! Dropped msgs: tick 178
Ticks: 179     Min: 0  Max:   TickISR: queue overflow!
99
! Dropped msgs: tick 180
Ticks: 181     Min: 0  Max:   TickISR: queue overflow!
99
! Dropped msgs: tick 182
Ticks: 183     Min: 0  Max: 99
Ticks: 184     Min: 0  Max: 99
  TickISR: queue overflow!
! Dropped msgs: tick 185
Ticks: 186     Min: 0  Max: 99
Ticks: 187     Min: 0  Max:   TickISR: queue overflow!
99
! Dropped msgs: tick 188
Ticks: 189     Min: 0  Max:   TickISR: queue overflow!
99
! Dropped msgs: tick 190
Ticks: 191     Min: 0  Max: 99
Ticks: 192     Min: 0  Max: 99
Ticks: 193     Min: 0   TickISR: queue overflow!
Max: 99
! Dropped msgs: tick 194
Ticks: 195     Min: 0  Max:   TickISR: queue overflow!
99
! Dropped msgs: tick 196
```

Sample output:
750 instr/tick

Requirements at 750 instr/tick:
- Code does not crash
- Task A still gets *some* entries from queue if CPU hog not running, but some entries may be dropped.
- May not see *any* stat task output





YAK functions for Lab 6

YKQ *YKQCreate(void **start, unsigned size)

- Initializes data structure used to maintain queue, returns pointer to it
- Parameters:
  - **start:** specifies base address of the queue itself, an array of void pointers
  - **size:** specifies number of entries in queue (size of the array)
- Return value:
  - pointer to initialized queue management struct allocated for this new queue
- YKQ is typedef defined in kernel header file; defines struct that records
  - values to manage queue: base address, queue size, head + tail indices, etc.
- Must be called exactly once per message queue, ideally in main()

YAK functions for Lab 6

void *YKQPend(YKQ *queue)

- If message queue is not empty, removes and returns the oldest message.
- If message queue is empty, calling task is suspended by the kernel until a message is placed in queue.
- Parameter:
  - **queue:** specific queue to use (pointer to queue management struct)
- Called only by tasks, never by interrupt code.

YAK functions for Lab 6

int YKQPost(YKQ *queue, void *msg)

- Inserts a new message into message queue
- Parameters:
  - **queue:** specific queue to use (pointer to queue management struct)
  - **msg:** pointer value to insert
- Return value:
  - 1 if queue is not full and message insertion was successful.
  - 0 if queue is full and message was not inserted.
- If one or more blocked tasks are waiting for a message from this queue, the highest priority waiting task is unblocked when new message is inserted
- This function may be called from *both* task and interrupt code.
  - If called from interrupt code, *it must not call the scheduler*.