CS331 Lab Report #4

Andrew Janssen (apjansse@uiuc.edu)
Michael Tucknott (tucknott@uiuc.edu)
Terrence Janas (tjanas@uiuc.edu)

Thursday at 5:00pm
Workstation: emb6
Username: group26

Program implementation (part 1):

In part 1, we created implemented the io and display tasks as separate processes (`io.cc`, `display.cc`). To achieve communication between the two processes, we used a message queue. The display process is implemented in `display.cc`, and the io process is implemented in `io.cc`.

The display process is executed before the io process; therefore, we initialize the A/D controller in `display.cc`. After the card is initialized using `hw_out()` from previous labs, the user is prompted to calibrate the seesaw. The two sets of raw data corresponding to the extreme left & right positions of the seesaw are pulled in from the A/D controller, and normalized as 12-bit unsigned values. The shared message queue is opened as blocking and read-only, and a sighandler is created to capture SIGINT (CTRL-C) signals. This ensures that the message queue is properly unlinked and closed before the program exits. Finally, an infinite loop makes a blocking call to `mq_receive()`, which removes a set of raw data created by the io process from the shared message queue. This 12-bit voltage value is converted into its corresponding analog voltage value between -10.0 and +10.0 V. The seesaw angle that corresponds to this voltage is also computed. Finally, the raw data, voltage, and angle are printed to the screen. This infinite loop continues until the user presses CTRL-C.

Once the calibration steps have been taken in the display process, the io process should be executed. The io processs opens the shared message queue as non-blocking with write access. A timer is created that raises a SIGALRM ten times per second. When a SIGALRM is detected by the io process, it calls `timer_handler()`. This function reads in raw data from the A/D controller. The set of raw data is written to the message queue by `mq_send()`. If the message queue is full, `mq_send()` returns -1. If this happens, the message queue is unlinked and closed before the io process exits. An infinite while loop is created that waits for a SIGALRM and writes to the message queue. Similar to `display.cc`, the function `sigint_handler()` is created to capture SIGINT signals that are triggered when the user presses CTRL-C to exit.

Program implementation (part 2):

In part 2, we created implemented the io and display tasks as separate threads, within the same process (`threads.cc`). To achieve communication between the two threads, we used a global buffer variable, protected by a mutex. The `main()` thread simulates the io task from part 1, and the thread spawned by `pthread_create()` simulates the display task from part 1. Both threads have equal priority values.

First of all, the seesaw is calibrated in `main()`, just like in part 1. Afterwards, two timers are created: `io_timer` and `display_timer`. These timers generate SIGUSR1 and SIGUSR2 signals ten times per second, respectively. After both timers are started, the display thread is spawned and enters `display()`. Meanwhile, the `main()` thread (io thread) enters an infinite while loop, and performs a `sigwait()` on SIGUSR1. Once `sigwait()` detects a SIGUSR1 signal, the while loop continues and `io_timer_handler()` is executed. This function reads raw data from the A/D controller and locks the mutex. Once in the critical section, the new raw data is written to the buffer. The mutex is unlocked and the io thread returns to the `while(1)` loop inside `main()`.

The display thread begins execution inside `display()`. The thread obtains a lock on the mutex, reads the raw data from the global buffer, and unlocks the mutex. Afterwards, it performs the same steps from `display.cc` (converting raw value to voltage and angle, and printing out values to the screen). All of the steps in `display()` are contained within an infinite while loop.

Problems encountered:

Our biggest problem was figuring out how to organize the program into separate functions and threads, and following the program flow. We also encountered initial difficulty in converting raw data into corresponding angle values, and how raw data was stored in the shared buffer. However, even though program development took longer than expected, all problems were solved.

Team Member Contributions:

Michael: Developed the logic for converting raw seesaw data into voltage, angle values. Assisted in development and debugging of program code.
Andrew: Wrote final report, helped develop and debug the program.
Terrence: Developed most of the program coding and assisted in some debugging.

Lab Questions:

1. In `io.cc` we opened the message queue with as non-blocking. We did this because the io process is the producer of data, and the display process is blocking. As a result, the display process waits until data is produced by io before it removes data from the buffer and prints it to the screen. As long as the display process is running and ready before `io.cc` is executed, the program will run normally.

2. In `display.cc` we opened the message queue with a blocking parameter. Since io is non-blocking and produces data 10 times per second, the maximum rate of consumption by display is governed by the rate of the producer (`io.cc`).

3. A special handler was used in `io.cc` and `display.cc` because without it, the queue would not have been unlinked and closed properly. This would have left it in memory causing unexplainable errors the next time the program was run.

4. A mutex is required because two separate threads are reading/writing the same shared variable. Therefore, only one thread at a time can have access to the variable at once. For example, a thread cannot read from the variable while the other thread is writing to it, and vice-versa.

```
/*********************** BEGIN display.cc ***********************/

#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <mqueue.h>

int status;
mqd_t wave_q_id;   // message queue ID

extern "C"
{
  // ASM function that outputs voltage to D/A
  void hw_out(unsigned int port, unsigned char val);
  unsigned char hw_in(unsigned int port);
}



// this is called whenever there is a SIGINT (CTRL-C)
void sigint_handler()
{
  // close & unlink the shared message queue, then exit
  status = mq_close(wave_q_id);
  status = mq_unlink("WAVE_SPEC_Q");
  cout << "Ctrl-C Detected!\n";
  cout << "Queue unlinked.  Exiting.\n";
  exit(0);
}



int main(void)
{
  char prompt;
  unsigned char hiByteR, loByteR, hiByteL, loByteL;

  unsigned char buffer[2];
  struct mq_attr wave_q_attr;          // message queue attributes
  unsigned short digitalV;             // discrete 12-bit voltage
  unsigned short digitalL, digitalR;  // extreme L/R voltage bounds
  float angle, voltage;
  unsigned int prior = 5;              // arbitrary priority value

  // initialize the D/A, used in lab1
  hw_out( 0x309, 0 );   // disable int & DMA
  hw_out( 0x308, 0 );   // clear trigger flip flop for interrupt
  hw_out( 0x302, 0 );   // select analog input channel


  cout << "************************************************************\n"
       << "************************************************************\n"
       << "**                                                        **\n"
       << "**    CS 331 Lab 4 (Part 1) Display Process: Group 26    **\n"
       << "**                                                        **\n"
       << "************************************************************\n"
       << "************************************************************\n\n";
  cout << "Tilt the seesaw all the way to the right, and press ENTER ...\n";
  cin.get(prompt);

  hw_out(0x300,0x00);    // start A/D conversion
  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 !=0) { };
  loByteR = hw_in(0x300);
  hiByteR = hw_in(0x301);

  cout << "Tilt the seesaw all the way to the left, and press ENTER ...\n";
  cin.get(prompt);

  hw_out(0x300,0x00);    // start A/D conversion
  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 !=0) { };
  loByteL = hw_in(0x300);
  hiByteL = hw_in(0x301);

  // setup message queue attributes
  memset(&wave_q_attr,0,sizeof(wave_q_attr));
  wave_q_attr.mq_maxmsg = 10;
  wave_q_attr.mq_msgsize = sizeof(buffer);

  // setup buffer to store the waveform spec
  // buffer[0] = loByte, buffer[1] = hiByte
  memset(buffer,0,sizeof(buffer));
  wave_q_id = mq_open("WAVE_SPEC_Q",O_RDONLY | O_CREAT,0600,
    &wave_q_attr);

  struct sigaction my_action;              // sighandler initializations
  memset(&my_action,0,sizeof(my_action));  // clear all flags
  my_action.sa_handler = sigint_handler;
  sigaction(SIGINT,&my_action,NULL);

  digitalL = hiByteL & 0x00FF;
  digitalL = (digitalL << 4) & 0x0FF0;           // [0000HHHH HHHH0000]
  digitalL = ((loByteL >>4)& 0x0F) | digitalL;   // [0000HHHH HHHHLLLL]

  digitalR = hiByteR & 0x00FF;
  digitalR = (digitalR << 4) & 0x0FF0;           // [0000HHHH HHHH0000]
  digitalR = ((loByteR >>4)& 0x0F) | digitalR;   // [0000HHHH HHHHLLLL]

  cout << "Start the I/O process now...\n";

  while(1) {
    // pull out a chunk of data from the message queue, put into buffer
    status = mq_receive(wave_q_id, (char *)buffer, sizeof(buffer),&prior);
    digitalV = buffer[1] & 0x00FF;
    digitalV = (digitalV <<4) & 0x0FF0;            // [0000HHHH HHHH0000]
    digitalV = ((buffer[0] >>4)& 0x0F) | digitalV; // [0000HHHH HHHHLLLL]
    angle=(((float)(digitalV-digitalL)/(float)(digitalR-digitalL))*30.0)-15.0;
    voltage = (((float)digitalV/4095.0)*20.0)-10.0;
    cout << "raw=" << digitalV << "  angle=" << angle
         << "  voltage=" << voltage << "\n";
  }
}

/*********************** END display.cc ***********************/
```

```
/************************ BEGIN io.cc ************************/

#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <mqueue.h>

int status;                       // status of queue operations
unsigned char buffer[2];          // a unit of data in the message queue
mqd_t wave_q_id;                  // ID of the message queue
struct mq_attr wave_q_attr;       // message queue attributes

extern "C"
{
  // ASM function that outputs voltage to D/A
  void hw_out(unsigned int port, unsigned char val);
  unsigned char hw_in(unsigned int port);
}


void timer_handler()  // this is called whenever there is a SIGALRM
{
  hw_out(0x300,0x00);     // start A/D conversion

  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 !=0) { };
  buffer[0] = hw_in(0x300);
  buffer[1] = hw_in(0x301);
}



void sigint_handler()  // this is called whenever there is a SIGINT (CTRL-C)
{
  status = mq_close(wave_q_id);        // close & unlink the
  status = mq_unlink("WAVE_SPEC_Q");   // shared message queue,
  exit(0);                             // then exit
}



int main(void)
{
   cout << "************************************************************\n"
        << "************************************************************\n"
        << "**                                                        **\n"
        << "**   CS 331 Lab 4 (Part 1) I/O Process Example: Group 26   **\n"
        << "**                                                        **\n"
        << "************************************************************\n"
        << "************************************************************\n\n";

   cout << "The display process should already be running and calibrated.\n\n";
   /* make queue */
   memset(buffer,0,sizeof(buffer));        // clean up buffer space
   memset(&wave_q_attr, 0, sizeof(wave_q_attr));
   wave_q_attr.mq_flags = O_NONBLOCK;      // set mqueue parameters
   wave_q_attr.mq_maxmsg = 10;
   wave_q_attr.mq_msgsize = sizeof(buffer);

   // open the message queue
   wave_q_id = mq_open("WAVE_SPEC_Q",O_WRONLY | O_CREAT | O_NONBLOCK,0600,
      &wave_q_attr);

   struct sigaction my_action;   // timer_handler() is called for SIGALRM's

   // used by the timer created, and initialized for the timer handler
   timer_t timer_id;
   struct itimerspec timer_spec, old_timer_spec;
   timer_create(CLOCK_REALTIME, NULL, &timer_id);

   // set timer parameters
   timer_spec.it_value.tv_sec = 0;   // first SIGALRM will go off only 1 nsec
   timer_spec.it_value.tv_nsec = 1;  // after the timer is started
   timer_spec.it_interval.tv_sec = 0;   // our timer interval is
   timer_spec.it_interval.tv_nsec = 200000000;

   // sighandler initializations
   memset(&my_action,0,sizeof(my_action));  // clear all flags
   my_action.sa_handler = timer_handler;
   sigaction(SIGALRM,&my_action,NULL);
   my_action.sa_handler = sigint_handler;
   sigaction(SIGINT,&my_action,NULL);

   // start the timer!
   timer_settime(timer_id,0,&timer_spec,&old_timer_spec);

   while(1) {
      sigpause(SIGALRM);  // when there is a SIGALRM, run timer_handler()
      status = mq_send(wave_q_id,(char *)buffer,sizeof(buffer),5);
      if (status == -1)   // queue overflow?
      {
         status = mq_close(wave_q_id);
         status = mq_unlink("WAVE_SPEC_Q");
         return 1;
      }
   }
}

/************************ END io.cc ***********************/
```

```
/************************ BEGIN threads.cc ************************/

#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <mqueue.h>
#include <pthread.h>

pthread_mutex_t wave_spec_mutex;    // protects buffer
unsigned char buffer[2];            // loByte, hiByte
unsigned short digitalL, digitalR;  // digital left/right bounds of seesaw

sigset_t io_signals;           // signal set used by SIGWAIT for io
sigset_t display_signals;      // signal set used by SIGWAIT for display

extern "C"
{
  // ASM function that outputs voltage to D/A
  void hw_out(unsigned int port, unsigned char val);
  unsigned char hw_in(unsigned int port);
}



// function executed by display thread
void* display(void* args)
{
  int status;
  unsigned char loByte, hiByte;  // data from A/D card
  unsigned short digitalV;       // 12-bit discrete voltage
  float angle, voltage;          // seesaw angle & analog voltage

  while(1)
  {
    sigwait(&display_signals,&status);  // wait until SIGUSR2 is received

    status = pthread_mutex_lock(&wave_spec_mutex);
    /***  enter critical section, read global variable  ***/
    loByte = buffer[0];
    hiByte = buffer[1];
    /***  exit critical section  ***/
    status = pthread_mutex_unlock(&wave_spec_mutex);

    // convert loByte and hiByte into one 12-bit value
    digitalV = hiByte & 0x00FF;
    digitalV = (digitalV << 4) & (0x0FF0);
    digitalV = ((loByte >>4)&(0x000F)) | digitalV;

    // from our discrete value, we can calculate the angle and analog voltage
    angle=(((float)(digitalV-digitalL)/(float)(digitalR-digitalL))*30.0)-15.0;
    voltage = (((float)digitalV/4095.0)*20.0)-10.0;
    cout << "raw=" << digitalV << "  voltage=" << voltage
         << "  angle=" << angle << "\n";
  }
}
```

```
// function executed by main thread after SIGUSR1 interrupt occurs
void io_timer_handler()
{
  int status;
  unsigned char loByte, hiByte;

  hw_out(0x300,0x00);   // start A/D conversion
  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 != 0) { };
  loByte = hw_in(0x300);
  hiByte = hw_in(0x301);

  status = pthread_mutex_lock(&wave_spec_mutex);
  // enter critical section, write global variable
  buffer[0] = loByte;
  buffer[1] = hiByte;
  // exit critical section
  status = pthread_mutex_unlock(&wave_spec_mutex);
}



int main(void)
{
  int status;
  char prompt;
  unsigned char hiByteR, loByteR, hiByteL, loByteL;

  pthread_mutex_init(&wave_spec_mutex, NULL);  // initialize mutex
  pthread_t wave_thread_id;                    // thread ID

  // initialize the A/D, D/A controller
  hw_out( 0x309, 0 );   // disable int & DMA
  hw_out( 0x308, 0 );   // clear trigger flip flop for interrupt
  hw_out( 0x302, 0 );   // select analog input channel

  cout << "************************************************\n"
       << "************************************************\n"
       << "**                                            **\n"
       << "**    CS 331 Lab 4 (Part 2) Thread: Group 26    **\n"
       << "**                                            **\n"
       << "************************************************\n"
       << "************************************************\n\n";

  cout << "Tilt the seesaw all the way to the right, and press ENTER...\n";
  cin.get(prompt);

  hw_out(0x300,0x00);                     // start A/D conversion
  // Cheap hack for getting stable A/D input
  // input loByte from BASE+0, hiByte from BASE+1
  while(hw_in(0x308) & 0x80 !=0) { };
  loByteR = hw_in(0x300);                 // get calibration data,
  hiByteR = hw_in(0x301);                 // right-side of seesaw

  cout << "Tilt the seesaw all the way to the left, and press ENTER ...\n";
  cin.get(prompt);

  hw_out(0x300,0x00);                     // start A/D conversion
```

```
// Cheap hack for getting stable A/D input
// input loByte from BASE+0, hiByte from BASE+1
while(hw_in(0x308) & 0x80 !=0) { };
loByteL = hw_in(0x300);                // get calibration data,
hiByteL = hw_in(0x301);                // right-side of seesaw

// convert "left" calibration voltage data into 12 bit discrete value
digitalL = hiByteL & 0x00FF;
digitalL = (digitalL << 4) & (0x0FF0);
digitalL = ((loByteL >>4)&(0x000F)) | digitalL;

// convert "right" calibration voltage data into 12 bit discrete value
digitalR = hiByteR & 0x00FF;
digitalR = (digitalR << 4) & (0x0FF0);
digitalR = ((loByteR >>4)&(0x000F)) | digitalR;


struct itimerspec io_timer_spec, display_timer_spec,
                  io_old_timer_spec, display_old_timer_spec;

// create io timer
timer_t io_timer;
struct sigevent io_timer_evp;
memset(&io_timer_evp, 0, sizeof(io_timer_evp));
io_timer_evp.sigev_signo = SIGUSR1;        // io responds to SIGUSR1 signal
io_timer_evp.sigev_notify = SIGEV_SIGNAL;
timer_create(CLOCK_REALTIME, &io_timer_evp, &io_timer);

// set io timer settings (goes off 10 times per second)
io_timer_spec.it_value.tv_sec = 0;   // first SIGUSR1 will go off only 1 ns
io_timer_spec.it_value.tv_nsec = 1;   // after the timer is started
io_timer_spec.it_interval.tv_sec = 0;   // our timer interval is
io_timer_spec.it_interval.tv_nsec = 200000000;

// create display timer
timer_t display_timer;
struct sigevent display_timer_evp;
memset(&display_timer_evp, 0, sizeof(display_timer_evp));
display_timer_evp.sigev_signo = SIGUSR2;   // display responds to SIGUSR2
display_timer_evp.sigev_notify = SIGEV_SIGNAL;
timer_create(CLOCK_REALTIME, &display_timer_evp, &display_timer);

// set display timer settings (goes off 10 times per second)
display_timer_spec.it_value.tv_sec = 0;   // 1st SIGUSR2 will go off only
display_timer_spec.it_value.tv_nsec = 1;  // 1 ns after the timer is started
display_timer_spec.it_interval.tv_sec = 0;   // our timer interval is
display_timer_spec.it_interval.tv_nsec = 200000000;

sigemptyset(&io_signals);              // Create signal set for SIGWAIT,
sigaddset(&io_signals, SIGUSR1);       //  used by io.
sigemptyset(&display_signals);         // Create signal set for SIGWAIT,
sigaddset(&display_signals, SIGUSR2);  //  used by display thread.

// start io timer
timer_settime(io_timer,0,&io_timer_spec,&io_old_timer_spec);

// start display timer
timer_settime(display_timer,0,&display_timer_spec,&display_old_timer_spec);

io_timer_handler();  // store initial data in the buffer
```

```
  // spawn the display thread...
  status = pthread_create(&wave_thread_id, NULL, display, NULL);

  while(1) {                          // pass raw data to buffer
    sigwait(&io_signals,&status);     // 10 times per second, until
    io_timer_handler();               // user presses CTRL-C
  }
}
```

/*********************** END threads.cc ************************/**

/*********************** BEGIN iodas.c ************************/**

```
void hw_out(unsigned int port, unsigned char val)
{
   __asm__ __volatile__ ("outb %b0, %w1"
                         : /* no output */
                         : "a" (val), "d" (port) );
}

unsigned char hw_in(unsigned int port)
{
   unsigned char val;
   __asm__ __volatile__ ("inb %w1, %b0"
                         : "=a" (val)
                         : "d" (port) );
   return val;
}
```

/*********************** END iodas.c ************************/**