CSC 374: Computer Systems II: Final (2010 Spring)

Joe Phillips
Last modified 2010 June 10

A. Short Answer (12 points each)

1. Compiler Optimization

Optimize the following code:

```
int silliestFunction  (const int array1[], const int array2[], int array2Len)
{
  int i;
  int j;
  int k;
  int accumulator = 0;

  for  (i = 1024;  i >= 1;  i = i / 2)
  {

    for  (j = i;  j < i*2-1;  j++)
    {

      for  (k = 0;  k < array2Len;  k++)
        accumulator += someFunction(10) * array2[array1[i/2] - array1[j*2]];

    }

  }

  return(accumulator);
}
```

(It is not important what `someFunction()` does. You only need to know that it always takes integer 10 and returns an integer back.)

2. Memory segments

You are writing an assembly-level debugger for Linux. This debugger should "look over the shoulder" of the program it is debugging, and therefore has access to registers (like %`esp`) and special memory pointers (like `break`) used by the program it is debugging.

The user has requested that the memory at a particular address be printed. What would be appropriate course of action for each of the following?

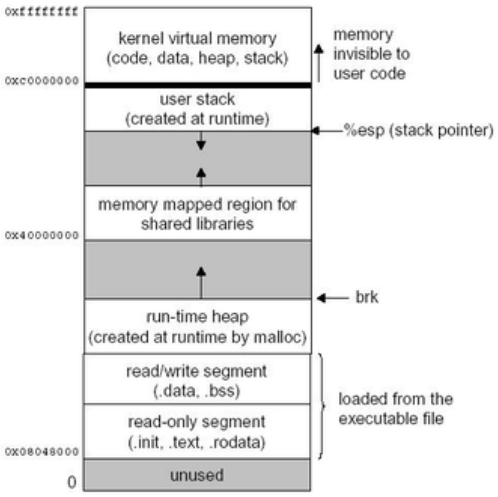
1. A value less than 0x4000,0000 but greater than the `break` pointer.
2. A value less than 0xC000,0000 but greater than the `%esp` pointer.
3. 0x4080,0000
4. 0x1880,8000

Your choices include:

A. *DON'T DO IT!* It's memory to which the process being debugged does not have access.
   Instead, *warn the user sternly* that it is **illegal** to see what is at that address.
B. *Disassemble it* and print it out as assembly language
C. *Print it*: it *definitely* is a variable
D. *Print it or display it as a return address or stored* `%ebp` *value. Maybe* it is a variable and *maybe* it is not.

Assume

- there are 0x1000,0000 bytes of executable code (not including shared libraries and not including global vars)
- there are 0x0100,0000 bytes of global vars
- there are 0x0100,0000 bytes of shared libraries linked in at run time



3. **Exceptions and Signals**

A. Write the code for a *mama process* to make 16 baby processes 0-15 to run `"childProgram"`. It should send the integer `i` (as a string) to each as a parameter and it should *wait for each child to finish before starting the next.*
The parameter to send to the children is in string `parameter`. *You may need to write other functions too.*

```
void doMama ()
{

  //  1: Any code here?

  for  (int i = 0;  i < 16;  i++)
  {
    const int PARAM_LEN = 10;
    char      parameter[PARAM_LEN];

    snprintf(parameter,PARAM_LEN,"%d",i);

    //  2: Any code here?

  }

  //  3: Any code here?

}


//  4: Any other functions?
```

B. Write the code for a *mama process* to make 16 baby processes 0-15 to run `"childProgram"`. It should send the integer `i` (as a string) to each as a parameter and it should *not wait at all.* It should, however, reap each child as it finishes.
The parameter to send to the children is in string `parameter`. *You may need to write other functions too.*

```
void doMama ()
{
```

```
    //  1: Any code here?

    for  (int i = 0;  i < 16;  i++)
    {
      const int PARAM_LEN = 10;
      char      parameter[PARAM_LEN];

      snprintf(parameter,PARAM_LEN,"%d",i);

      //  2: Any code here?

    }

    //  3: Any code here?

  }


  //  4: Any other functions?
```

This might be useful:

`int fork()`	Parent process to make a child process. Child process gets copy of complete parent process, differing only in return value. The return value is: Negative: Fork failed (process table full?) 0: This is the value the child process gets on success Positive: This is parent process. The actual number is the id of the child
`execl(const char* programPath, const char* programName, const char* parameter, NULL);`	This process is to quit running the current program and start running the program *programPath*. This program is to be given its own name (*programName*, so it knows itself) and the parameter *parameter*.
`int wait (int* statusPtr)`	This process is to wait for some child process (any process to finish). When it does it returns the process id of the child that did finish, and sets the integer pointed to by *statusPtr* to the status integer returned by the child process. If there are no more children for which to `wait()` it returns **-1**.
`signal(SIGCHLD,fncName);`	When this process is informed that a child has finished it should run the function `void fncName(int signal)` to reap it.

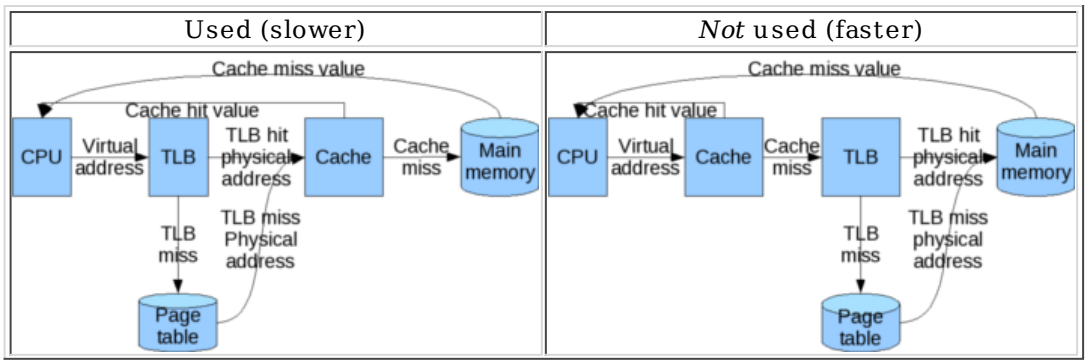
4. **Processes and Threads**

   A. When are *processes* more appropriate than *threads*?
   What is it about how processes share memory that make *pthread mutexes* inadequate for handling mutal exclusion between processes?

   B. When are *threads* more appropriate than *processes*?
   What is it about how threads share memory that make *pthread mutexes* adequate for handling mutal exclusion between threads?

5. **Memory**

   A. Two possible configurations for the TLB and the cache are shown below. The configuration on the right would be faster at getting requested memory back to the CPU, but it is the configuration on the left that is actually used.
   Why? *Hint:* There is more than one process on modern computers.



B. *Why does the TLB even exist?* Why not always go to the page table? (After all, it keeps track of all of the pages used by a process.)
Hint: If the page table were implemented in a naive, simplistic fashion, about how many entries would it hold? (Assume Linux pages are 4KB big, and that each process can access about 4GB of virtual addresses.)

6. **Networking**

   A. Servers are almost always multi-threaded or have multiple processes.
   Why? What are the different tasks that the threads/processes do?

   B. Packages like *ncurses* require the programmers to *say* when they want the users' screens to be updated. This is, of course, a pain in the a** for programmers.
   Why is it done like this anyway?

B. Long Answer (28 points)

He *warned* you . . .

You *knew* this was coming . . .

You *can't say* that this was a **complete surprise** . . .

Write the *breakoutClient*!

What to do:

1. Give me the *missing code* that goes inside the following functions:
   a. `startGame()` // 10 points
   b. `playGame()` // 16 points
   c. `endGame()` // 2 points

   You may write the code on the exam sheet, if you like

2. *Do not panic!* because:
   - Some code is truly *easy* 1-line affairs
   - Some code deals with windowing and other with sockets.
   - I did the *really nasty* stuff myself.
3. I suggest you read the overview, then do the functions from easiest to hardest.
4. Each "*YOUR CODE HERE TO*" comment tells you what you need to do.

How to:	Usage:
Send bytes	`int write(int fileDes,const void* bufferPtr, int numBytes)` Writes *numBytes* bytes pointed to by *bufferPtr* to file

	descriptor *fileDes*. Returns number of bytes written (0 means "none"), or -1 which means "error".
Read bytes (I)	`int read(int fileDes,void* bufferPtr, int bufferLen)` Reads up to *bufferLen* bytes into the buffer pointed to by *bufferPtr* from file descriptor *fileDes*. Waits until something is available. Returns number of bytes read, or returns -1 on error.
Read bytes (II)	`int recv(int fileDes,void* bufferPtr, int bufferLen, int flags)` Reads up to *bufferLen* bytes into the buffer pointed to by *bufferPtr* from file descriptor *fileDes*. *flags* tells how to read, where *MSG_DONTWAIT* means "non-blocking". Returns number of bytes read, or returns -1 and sets `errno` to `EAGAIN` if the flag was *MSG_DONTWAIT* and there was nothing to read.
Start ncurses	`initscr()`
Stop ncurses	`endwin()`
Clear the screen	`clear()`
Refresh the whole screen	`refresh()`
Turn off echoing of typed chars	`noecho()`
Allow non-blocking keyboard input	`nodelay(stdscr,TRUE)`
Allow usage of keypad chars	`keypad (stdscr,TRUE)`
Disallow scrolling	`scrollok(windowPtr, FALSE)`
Convert a 32-bit integer from network's endian to host's endian	`uint32_t ntohl(uint32_t networkInt)` Returns 32-bit integer *networkInt* so that it is in the endian of the current computer instead of for the network.
Convert a 16-bit integer from network's endian to host's endian	`uint16_t ntohs(uint16_t networkInt)` Returns 16-bit integer *networkInt* so that it is in the endian of the current computer instead of for the network.
Convert a 32-bit integer from host's endian to network's endian	`uint32_t htonl(uint32_t hostInt)` Returns 32-bit integer *hostInt* so that it is in the endian of the network instead of for the current computer.
Convert a 16-bit integer from host's endian to network's endian	`uint16_t htons(uint16_t hostInt)` Returns 16-bit integer *hostInt* so that it is in the endian of the network instead of for the current computer.
Move the cursor on the whole screen	`move(int row, int col)` Moves the cursor to row *row*, column *col* within the whole screen. 0,0 is the upper left corner.
Write a char to the whole screen	`addch(chtype character)` Writes character *character* to the current cursor position.
Write a string to the whole screen	`addstr(const char* toPrintPtr)` Writes the C-string pointed to by *toPrintPtr* to the current cursor position

Get a character from the keyboard	int getch()

```
/*-------------------------------------------------------------------------*
 *---                                                                   ---*
 *---           breakoutClient.cpp                                      ---*
 *---                                                                   ---*
 *---    This file defines the client for the breakout program.         ---*
 *---                                                                   ---*
 *---   ----    ----    ----    -----    ----    ----    ----    ----   ---*
 *---                                                                   ---*
 *---   Version 1.0             2010 May 28      Joseph Phillips       ---*
 *---                                                                   ---*
 *-------------------------------------------------------------------------*/


/*
 * Compile with:
 *  g++ -O2 -o breakoutClient breakoutCommon.cpp breakoutClient.cpp -lncurses
 */


#include "headers.h"
#include <pthread.h>



/*  PURPOSE:  To initialize the communication parameters 'hostName' and
 *      'portNumber' from the command line argument parameters 'argc' and
 *      'argv', and from whatever else the user enters.  No return value.
 *
 *      Already done, 0 Points
 */
void    initializeCommParams    (int            argc,
                                 char*          argv[],
                                 const char*&   hostName,
                                 int&           portNumber
                                )
                                throw()
{

  //  I.  Parameter validity check:


  //  II.  Initialize 'portNumber' and 'hostName':

  char*       newLinePtr;
  static char hostNameSpace[STRING_LENGTH];

  portNumber    = (argc > 1) ? atoi(argv[1]) : INITIAL_PORT;
  hostName      = (argc > 2) ? argv[2] : INITIAL_HOST;

  if  (argc <= 2)
  {
    printf("Hostname [%s]? ",hostName);
    fgets(hostNameSpace,STRING_LENGTH,stdin);
    newLinePtr = strchr(hostNameSpace,'\n');

    if  (newLinePtr != NULL)
      *newLinePtr = '\0';

    if  (hostNameSpace[0] != '\0')
      hostName = hostNameSpace;

  }

  if  (argc <= 1)
  {
    char        portSpace[STRING_LENGTH];

    printf("Port [%d]? ",portNumber);
    fgets(portSpace,STRING_LENGTH,stdin);
```

```
    newLinePtr = strchr(portSpace,'\n');

    if  (newLinePtr != NULL)
      *newLinePtr = '\0';

    if  ( isdigit(portSpace[0]) )
      portNumber = atoi(portSpace);

  }


  //  III.  Finished:

}



/*  PURPOSE:  To initialize text window from the connection with a server at
 *      'hostName':'portNumber'.  If successful sets 'numRows' to the number
 *      of rows and 'numCols' to the number of columns.  Returns pointer to
 *      WINDOW on success or NULL otherwise.
 *
 *      10 Points
 */
void    startGame       (const char*    hostName,
                         int            portNumber
                        )
                        throw (const char*)
{
  char            textToUser[STRING_LENGTH];


  //  I.  Parameter validity check:


  //  II.  Initiailize window:

  // Start ncurses
  // Clear the screen
  // Make getch() non-blocking
  // Move cursor to row = 20, column = 1
  // Allow usage of keypad chars in WINDOW* stdscr


  // Generate welcome message (optionally telling port to which to connect)

  snprintf
        (textToUser,
         STRING_LENGTH-1,
         "Welcome to breakout on %s:%d!  Press 'Esc' to end.",
         hostName,portNumber
        );
  textToUser[STRING_LENGTH-1] = '\0';


  // Write welcoming message in textToUser to screen
  // Don't echo chars when typed

  // Turn off scrolling in WINDOW* stdscr
  // Send all changes so user may see them


  //  III.  Finished:

}



/*  PURPOSE:  To play the game.  The server is contacted at file descriptor
 *      'connectDescriptor'.  'hostName' and 'portNumber' are passed so user
 *      may know server to which they are connected.  No return value.
 *
 *      16 Points
```

```
 */
void    playGame        (int            connectDescriptor,
                         const char*    hostName,
                         int            portNumber
                        )
                        throw(const char*)
{

  //  I.  Parameter validity check:

  //  II.  Play game:

  char  requestBuffer[REQUEST_LENGTH];
  bool  shouldContinueGame               = true;
  bool  hasReceivedWholescreenUpdate     = false;
  short temp16;
  int   temp32;
  short ballRow = ILLEGAL_ROW;
  short ballCol = ILLEGAL_COL;
  char* bufferCursor;
  char  paddle[PADDLE_WIDTH+1];
  char  onBlock [COLS_PER_BLOCK+1];
  char  offBlock[COLS_PER_BLOCK+1];
  char  update[MAX_UPDATE_LEN];
  int   remoteLen;

  memset(paddle,'=',PADDLE_WIDTH);
  paddle[PADDLE_WIDTH] = '\0';

  memset(onBlock,'*',COLS_PER_BLOCK);
  onBlock [COLS_PER_BLOCK] = '\0';

  memset(offBlock,'*',COLS_PER_BLOCK);
  offBlock[COLS_PER_BLOCK] = '\0';

  memset(update,       '\0',MAX_UPDATE_LEN);
  memset(requestBuffer,'\0',REQUEST_LENGTH);

  while  (shouldContinueGame)
  {

    //  II.A.  Get request from user:

    int key = // Set key to key read from keyboard without having to press Enter

    switch  (key)
    {
    case ERR :
      break;

    case KEY_LEFT :
      requestBuffer[0] = LEFT_REQUEST;
      // Send requestBuffer of length REQUEST_LENGTH to connectDescriptor
      break;

    case KEY_RIGHT :
      requestBuffer[0] = RIGHT_REQUEST;
      // Send requestBuffer of length REQUEST_LENGTH to connectDescriptor
      break;

    default :

      if  ((char)key == QUIT_REQUEST)
      {
        requestBuffer[0] = DISCONNECT_REQUEST;
        // Send requestBuffer of length REQUEST_LENGTH to connectDescriptor
      }
      else
      {
        beep();
      }

    }
```

```
//  II.B.  Handle update from server:

//  II.B.1.  Get update from server:

update[0] = '\0';
remoteLen = rio_recv(connectDescriptor,update,MAX_UPDATE_LEN,MSG_DONTWAIT);


//  II.B.2.  Ignore when nothing was sent:

if  ( (remoteLen == -1) && (errno == EAGAIN) )
  continue;


//  II.B.3.  Do update:

switch  (update[0])
{
case CONNECTION_DENIED_UPDATE :

  //  II.B.3.a.  Tell that connection was denied:

  snprintf(errorText,STRING_LENGTH,
           "%s:%d is alive but refused our request to connect, sorry.",
           hostName,portNumber
          );
  // Print errorText to row 10, column 0.
  break;


case DISCONNECT_UPDATE :

  //  II.B.3.b.  Tell that disconnect was acknowledged:

  shouldContinueGame = false;
  break;


case BEEP_UPDATE :

  //  II.B.3.c.  Make beep sound:

  beep();
  break;


case BEGIN_WHOLE_BOARD_UPDATE :

  //  II.B.3.d.  Update the whole board:

  hasReceivedWholescreenUpdate = true;
  // Clear the screen

  bufferCursor  = update + 2*sizeof(BEGIN_WHOLE_BOARD_UPDATE);


  //  II.B.3.d.I.  Get 'isBlockPresent[][]' from 'buffer[]':

  for  (int rowIndex = 0;  rowIndex < NUM_BLOCK_ROWS;  rowIndex++)
  {
    int bitArray;
    int currentBitPosition  = 0x1;

    memcpy(&temp32,bufferCursor,SIZE32);
    bufferCursor += SIZE32;
    bitArray = ntohl(temp32);

    for  (int blockIndex = 0; blockIndex < NUM_BLOCKS_PER_ROW; blockIndex++)
    {

      if  ( (bitArray & currentBitPosition) != 0 )
```

```
      {
        // Write C-string onBlock to the screen at
        //  row blockRowArray[rowIndex]
        //  column LEFT_BORDER_COL+1+blockIndex*COLS_PER_BLOCK
      }

      currentBitPosition <<= 1;
    }

  }


  //  II.B.3.d.II.  Get 'ballRow' and 'ballCol' from 'buffer[]':

  if  (ballRow != ILLEGAL_ROW)
  {
    // Write a space character (' ') to row ballRow, column ballCol
  }

  memcpy(&temp16,bufferCursor,SIZE16);
  bufferCursor += SIZE16;
  ballRow = // Set ballRow to the 16 bit integer temp16, but converted from network to host endian

  memcpy(&temp16,bufferCursor,SIZE16);
  bufferCursor += SIZE16;
  ballCol = // Set ballCol to the 16 bit integer temp16, but converted from network to host endian

  // Write the character 'O' to row ballRow, column ballCol


  //  II.B.3.d.III.  Get 'leftMostCols[MOST_CURRENT_PADDLE_POSITION_INDEX]'
  //                 from 'buffer[]':

  for  (int paddleIndex = 0;  paddleIndex < MAX_NUM_PADDLES;  paddleIndex++)
  {
    short col;

    memcpy(&temp16,bufferCursor,SIZE16);
    bufferCursor += SIZE16;
    col =  // Set col to the 16 bit integer temp16, but converted from network to host endian

    if  (col == ILLEGAL_COL)
      continue;

    if  ( col == (LEFT_BORDER_COL+1) )
    {
      // Write the C-string paddle followed by a space character (' ')
      // to row paddleRowArray[paddleIndex], column col.
    }
    else
    {
      // Write a space character (' ') followed by the C-string paddle
      // to row paddleRowArray[paddleIndex], column col-1.

      if  ( col < (RIGHT_BORDER_COL-PADDLE_WIDTH) )
      {
        // Write a space character (' ')
      }

    }

  }

  // Update the user's screen so they see the changes we made
  break;

case BEGIN_DIFFERENTIAL_BOARD_UPDATE :

  //  II.B.3.e.  Update the board based on differences from last time:

  if  (!hasReceivedWholescreenUpdate)
  {
    // Write the text "Waiting for update" to row 10, column 20
```

```
}
else
{
  short   deletedBlockRow;
  short   deletedBlockCol;

  bufferCursor  = update + 2*sizeof(BEGIN_DIFFERENTIAL_BOARD_UPDATE);


  //  II.B.  Get 'ballRow' and 'ballCol' from 'buffer[]':

  if  (ballRow != ILLEGAL_ROW)
  {
    // Write a space character (' ') to row ballRow, column ballCol
  }

  memcpy(&temp16,bufferCursor,SIZE16);
  bufferCursor += SIZE16;
  ballRow = // Set ballRow to the 16 bit integer temp16, but converted from network to host endian

  memcpy(&temp16,bufferCursor,SIZE16);
  bufferCursor += SIZE16;
  ballCol = // Set ballCol to the 16 bit integer temp16, but converted from network to host endian

  // Write the character 'O' to row ballRow, column ballCol


  //  II.C.  Get 'leftMostCols[MOST_CURRENT_PADDLE_POSITION_INDEX]'
  //         from 'buffer[]':

  for  (int paddleIndex = 0; paddleIndex < MAX_NUM_PADDLES; paddleIndex++)
  {
    short col;

    memcpy(&temp16,bufferCursor,SIZE16);
    bufferCursor += SIZE16;
    col = // Set col to the 16 bit integer temp16, but converted from network to host endian


    if  (col == ILLEGAL_COL)
      continue;

    if  ( col == (LEFT_BORDER_COL+1) )
    {
      // Write the C-string paddle followed by a space character (' ')
      // to row paddleRowArray[paddleIndex], column col.
    }
    else
    {
      // Write a space character (' ') followed by the C-string paddle
      // to row paddleRowArray[paddleIndex], column col-1.

      if  ( col < (RIGHT_BORDER_COL-PADDLE_WIDTH) )
      {
        // Write a space character (' ')
      }

    }

  }


  //  II.D.  Get 'deletedBlockRow' and 'deletedBlockCol' from 'buffer[]':

  memcpy(&temp16,bufferCursor,SIZE16);
  bufferCursor += SIZE16;
  deletedBlockRow = // Set deletedBlockRow to the 16 bit integer temp16, but converted from network to host endian

  memcpy(&temp16,bufferCursor,SIZE16);
  bufferCursor += SIZE16;
  deletedBlockCol = // Set deletedBlockCol to the 16 bit integer temp16, but converted from network to host endian

  if  (deletedBlockRow != ILLEGAL_ROW)
```

```
        {
          // Write the C-string offBlock to row blockRowArray[deletedBlockRow],
          // column LEFT_BORDER_COL+1+deletedBlockCol*COLS_PER_BLOCK.
        }

      }

      // Update the user's screen so they see the changes we made
      break;

    }

  }


  //  III.  Finished:

}



/*  PURPOSE:  To end ncurses and the game.  No parameters.  No return value.
 *
 *        4 Points
 */
void      endGame       ()
                        throw(const char*)
{
  // Stop ncurses
}



/*  PURPOSE:  To play attempt to connect to the breakout server program and
 *      to let a client play the game.  The hostname and port number are
 *      optionally given in the command line parameters 'argc' and 'argv'.
 *      Returns 'EXIT_SUCCESS' on success or 'EXIT_FAILURE' otherwise.
 *
 *      Already done, 0 Points
 */
int     main (int argc, char* argv[])
{

  //  I.  Parameter validity check:


  //  II.  Do breakout client:

  //  II.A.  Get connection parameters:

  int         portNumber;
  const char* hostName;

  initializeCommParams(argc,argv,hostName,portNumber);

  //  II.B.  Attempt to connect and to play the game:

  try
  {
    int     connectDescriptor;
    int     playerNum;
    int     socketDescriptor = getSocketDescriptor(hostName,portNumber);

    if  (socketDescriptor != ERROR_DESCRIPTOR)
    {
      startGame(hostName,portNumber);
      playGame(socketDescriptor,hostName,portNumber);
      endGame();
    }

    close(socketDescriptor);
  }
  catch  (const char* errMsgPtr)
```

```
  {
    fprintf(stderr,"%s\n",errMsgPtr);
    return(EXIT_FAILURE);
  }


  //  III.  Finished:

  return(EXIT_SUCCESS);
}
```