Introduction to Operating Systems
Student Handbook

Matt Johnson

January 8, 2006
# Contents

1 Syllabus

2 Course Outline
   2.1 Introduction ........................................... 9
   2.2 Welcome to Linux ..................................... 9
   2.3 The Changing Computing Environment .............. 10
   2.4 Files ................................................. 10
   2.5 Processes ............................................. 10
   2.6 Threads ................................................ 11
   2.7 Concurrency, Deadlock and Starvation .......... 11
   2.8 Memory ................................................ 11
   2.9 Scheduling ........................................... 12
   2.10 More Files ........................................... 12

3 C / C++ Programming Guidelines
   3.1 Documentation ......................................... 13
   3.2 Header files .......................................... 14
   3.3 Names .................................................. 14
   3.4 Good Programming Practice ......................... 14
   3.5 Appearance and Readability ......................... 15
   3.6 Defensive Coding .................................... 15
   3.7 Classes and Structures ............................... 15

4 Technical Writing
   4.1 Motivation ............................................. 17
   4.2 Matt’s Technical Writing Guidelines .............. 17
      4.2.1 Grammar ........................................ 17
      4.2.2 Person ........................................... 18
      4.2.3 Gender in Writing ............................... 18
      4.2.4 Margins .......................................... 18
      4.2.5 Font type and size ................................ 18
      4.2.6 Section headings ................................ 18
      4.2.7 Spacing .......................................... 18
<table>
<thead>
<tr>
<th>5</th>
<th>Socket Programming</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Socket Types</td>
<td>21</td>
</tr>
<tr>
<td>5.2</td>
<td>Stream Socket Functions</td>
<td>21</td>
</tr>
<tr>
<td>5.3</td>
<td>Socket Structures</td>
<td>24</td>
</tr>
</tbody>
</table>
Chapter 1

Syllabus

Instructor: Matt Johnson
E-Mail: matt@umr.edu
Website: http://web.umr.edu/~matt
Office: CS 342A

Lecture Info: MCNT 211, MWF 1:00-1:50, Lecture A, Reference Number 70700
Office Hours: See my schedule: http://web.umr.edu/~matt/schedule.html

Text Books:

2. Linux Programming by Example - The Fundamentals, Arnold Robbins, 2004

Contacts:

Dr. Fikret Ercal: Coordinator for CS 284. E-Mail: ercal@umr.edu

Supplementary Resources: Students are required to check my web page and their campus e-mail regularly for assignments, special notes, and other information pertaining to this class. Students should also browse Dr. Ercal’s 284 web page, as it contains a variety of material and sample code.

Course objectives:

1. Gain a good working knowledge of Unix commands.
2. Gain an understanding of operating system design and some of the major operating systems concepts including processes, threads, mutual exclusion, deadlock, memory management, virtual memory, scheduling, I/O management, and file management.
3. Implement programs on the Unix platform which will allow the student to gain first hand knowledge about processes, pipes, threads, and sockets.

**Grading:**

- Exam 1: 15%
- Exam 2: 20%
- Exam 3: 25%
- Programs, papers, and quizzes: 40%

**Quizzes:** Quizzes may be given from time to time. These quizzes may be announced or unannounced.

**Programming Assignments:** Programming assignments will be posted on the web page. Homework will usually be submitted electronically. The procedures for submission will be announced in class.

**Late Work:**

- Up to 24 hours late - 10% off
- From 24 to 48 hours late - 20% off
- From 48 to 72 hours late - 30% off
- More than 72 hours late - not accepted

**Attendance:** Attendance is mandatory. Quizzes may not be made up. As a general rule, make up exams will not be allowed. In extreme situations, make up exams will be provided. However, the student must provide a typed and signed document indicating the reason at least 8 days in advance. In the case of severe illness, a doctors note must be provided; this too must be prior to the exam. A student who misses an exam without prior and sufficient notification to the instructor will receive a zero for that exam. Make up exams will be more difficult than in class exams and will be graded harder.

**Assistance:** It is very important to me that everyone in this class learn the material. If you are having problems, please come by and speak with me during my office hours. If I say something in lecture that makes no sense to you, please ask at that time. It is likely you are not the only person who has misunderstood.

**Integrity and Ethics:** It is expected by the instructor, the department, the school, and the students future employer that every individual will do his or her own work. Individuals who acquire and use material from someone else and present it as their own are guilty of plagiarism. First time offenders will receive a zero on that assignment or test; the department chair will be notified of the offense. A second occurrence will result in the student being dropped from the class. Students are required to take the following steps to protect their work:

1. Electronic data should be viewable by the student who owns the data and no one else.
2. Shred or keep any hard copies of code. Hard copies of code lying around can be very tempting to so-called “garbage can programmers”.

3. Protect your books, notes, PCs, etc. from other individuals who may frequent your work or living space.

4. Lock your PC when you leave it unattended.

**Student Interaction:** It can be helpful for students to obtain assistance from each other during the course of the semester. Discussion about concepts and difficulties is encouraged. That being said, there is a fine line between helpful discussion and too much help. BE FOREWARNED: Students who “work together” enough that I can tell who is working with whom are guilty of cheating.

**Software Development:** Due to the nature of the code written in this class, it is possible that major problems on host systems may inadvertently be caused. Students are required to develop and test their software on the dells or blades. Use of the gpunix boxes for coding or testing purposes is strictly forbidden. If a student chooses to disregard this instruction and accidentally causes problems on a gpunix machine, the problem will be handled by the IT department as a “Security Incident.”

**Drop Policy:** If a student’s average falls below 70% or attendance falls below 80%, that student is expected to initiate a meeting with the instructor to discuss their grade in the course. Students in this situation will be required to show immediate improvement or drop the class. If the student fails to take these actions, the instructor may drop the student.

**Revision:** I reserve the right to revise this syllabus as needed. Notification of such a revision will occur during the next scheduled class.

**Revision History:**

**January 8:** Created this document.
Chapter 2

Course Outline

2.1 Introduction

The student should read Chapter 1 in the Stallings text [5]. Most of this material will not be included in lecture. Students who have not written code recently should refer to an introductory text in C or C++ programming for review. Students are expected to be capable of programming on a CS 153 level.

Topics: Computer System Overview, C / C++ Coding

Reading: Chapter 1 in Stallings [5], C or C++ Programming Text

Deliverables: None

2.2 Welcome to Linux

This section is named “Welcome to Linux” because this will be the students first required exposure to the Linux prompt and Linux programming.

Topics: The Linux Prompt, GNU, Posix

Reading: Chapters 1 and 2 in Robbins [4]

Deliverables: Program 1 - First Iteration of Shell (100 points)

- Command Line Arguments / Options
- Null - Terminated Character Array Manipulation - strcmp, strcat, strncpy
- The Environment - env, getenv
- getopt_long
- printf, scanf, getline
2.3 The Changing Computing Environment

This section of the course will include a viewing of the film “Triumph of the Nerds”, which is an interesting history lesson in the development of the Personal Computer. This film will tie in nicely with the lecture discussion of the evolution of operating systems and architectures.

**Topics:** History, Evolution, and Architecture

**Reading:** Chapter 2 in Stallings [5], Film: “Triumph of the Nerds”

**Deliverables:** None.

2.4 Files

One of the most important and fundamental skills for a Linux user is the ability to work with files and file descriptors. This section will expand on the concepts of the file and the directory.

**Topics:** Files, File I/O, Directories

**Reading:** Chapters 4 and 5 in Robbins [4]

**Deliverables:** Program 2 - Second Iteration of Shell (100 points)
- File I/O, File descriptors, File Pointers
- Setting Environment variables from C
- Changing Directories from C
- Error handling
- Functions: open, fdopen, close, fclose, read, write, getline, setenv, chdir, getcwd, strerror
- Built in copy, mkdir, rmdir commands

2.5 Processes

A fundamental concept in any modern operating system is that of the process. This section is devoted to understanding the concept of the process and the various states a process may be in. Understanding of the process is crucial, because it will be required for the remainder of the semester.

**Topics:** Processes, Process States, Process Control, Process Creation, Process Groups, Pipes

**Reading:** Stallings chapter 3 [5], Robbins chapter 9 [4]

**Deliverables:**
2.6. THREADS

1. Program 3 - Third Iteration of the Shell (100 points)
   - fork
   - wait
   - execvp
   - _exit

2. Program 4 - Fourth Iteration of the Shell (100 points)
   - pipe
   - dup2
   - ParsedTokens class

2.6 Threads

The concept of the thread is a natural extension to the concept of the Process, and easily confused with the Process. The distinction between thread and process should become apparent here, as well as the different types of threads.

Topics: Threads, PThreads, Mutual Exclusion, Synchronization, Sockets

Reading: Stallings chapter 4 [5], Socket Section of this manual, PThread section of this manual

Deliverables: Semester Project. This assignment will be a 200 point group project due near the end of the semester. It will include threads, sockets, group work, and a technical write up. The overall best project will be awarded a 20 point bonus.

2.7 Concurrency, Deadlock and Starvation

Execution environments which contain more than one thread of execution run the risk of becoming deadlocked. Individual threads or processes also run the risk of starving. This section will discuss these issues and ways of dealing with deadlock and starvation.

Topics: Concurrency, Deadlock, Starvation, Signals

Reading: Stallings chapters 5 and 6 [5], Robbins chapter 10 [4]

Deliverables: Program 5 - Signal Program

2.8 Memory

Computers could not function without memory. This section looks at memory management and virtual memory.

Topics: Memory Management, Virtual Memory

Reading: Stallings chapters 7 and 8 [5]

Deliverables: None.
2.9  Scheduling

The primary focus of this section is on Uniprocessor Scheduling. The scheduling of processes has a significant impact on how a system performs.

Topics:  Uniprocessor Scheduling, Real Time Scheduling
Reading:  Stallings chapters 9 and 10 [5]
Deliverables:  None.

2.10  More Files

Topics:  I/O Management, Disk Scheduling, File Management
Reading:  Stallings chapters 11 and 12 [5]
Deliverables:  None.
Chapter 3

C / C++ Programming Guidelines

C++ programmers must follow the department programming guidelines, as well as the guidelines listed below. In the event that there is a difference between the department guidelines and the guidelines listed below, see the instructor for conflict resolution. C programmers should ignore the C++ specific components of this document.

These guidelines have been composed based on input from the department, Dr. Ercal, and Brian Sea. Further, these guidelines reflect the instructors own personal coding style, which has been shaped by both teaching and software engineering experience. This list is by no means a comprehensive treatise on programming. Rather, it is intended to summarize most widely accepted programming guidelines in a manner appropriate for the needs of the educational environment. By following these guidelines, the student will be more likely to produce well written and well documented code. Additionally, the student may develop the habit of referencing a coding standard. Such a habit will be extremely valuable in the individuals future as a programmer.

3.1 Documentation

1. The program must be well documented. All comments must be written in application terms. Technical jargon in a comment is not permitted — that’s what the code is for.

2. Comments should be written using correct grammar, correct spelling, complete sentences, and correct punctuation.

3. Each file should contain a comment at the top of the file with the programmers name, file name, course, section, instructors name, due date, and a brief program description.

4. All function prototypes must be preceded by pre and post conditions and sometimes a description. The description should be used on an as needed basis.
3.2 Header files

1. Header files should have preprocessor directives which prevent inclusion of the file more than once.

2. Header files should be written for structure definitions and in other situations in which their use will enhance readability.

3. Never \#include a .c or a .cpp file.

4. Every file should include the header files which are needed within that file.

3.3 Names

1. Constant and macro names should be ALL CAPS.

2. File names should reflect the content of the file.

3. Put names for all formal parameters in prototypes.

4. Function naming: The first letter of each word in the name should be capitalized. The remaining letters should be lower case. Use an underscore in a function name to separate uppercase acronyms. ex. LOL_Now, GetValue, SetFYI_Yesterday

5. Variable naming: Use exactly the same technique as function naming except that the first letter of a variable must always be lower case.

3.4 Good Programming Practice

1. Make good use of functions to divide and conquer the problem at hand. Try to keep these functions short and sweet.

2. Do not use global or file scope variables.

3. The return type of main should be int.

4. Always return 0 (or some other integer if necessary) at the end of main.

5. Variables should always be declared and initialized at the top of each function. Do not declare variables at random locations throughout your code.

6. Calculation functions should not have input or output in them.

7. Always check the preconditions of a function before calling it.

8. Dynamically allocated memory must always be deallocated.

9. Address arrays using array notation instead of pointer arithmetic. For example: Arr[i] instead of *(Arr + i)
10. Be extremely careful with pointers! Always initialize them to NULL or a valid value. Do not leave any dangling pointers.

11. If the size of a parameter is much larger than a pointer, make it a constant pass by reference parameter.

### 3.5 Appearance and Readability

1. I am INSANELY PICKY about indentation. Indentation is exactly three spaces. Curly braces are always lined up as shown in the example below. A curly brace is always on a line by itself. The following is an example of the only type of indentation I will accept.

   ```c
   for(ii = 0; ii < MAX; ii++)
   {
     if(ii == 5)
     {
       printf("Hello World");
     }
   }
   ```

2. Tabs will never appear in your code.

3. The width of every line of code will never exceed 75 characters.

4. Always use beginning and ending curly braces, even if the body of that control structure is only one line.

### 3.6 Defensive Coding

1. Check all meaningful return values for error conditions.

2. Do NOT use any unbounded function calls (i.e. `strcpy()`, `strcmp()`, `sprintf()`). Instead use their bounded counterparts (i.e. `strncpy()`, `strncpy()`, `snprintf()`).

3. Do NOT assume valid input unless you have verified it elsewhere.

4. If you use `malloc()` or `new`, verify that you have been given the memory. Also, for every `malloc()` there should be a corresponding `free()`, and for every `new` there should be a `delete`.

### 3.7 Classes and Structures

1. Classes and Structures should be named exactly the same way functions are named. See section 3.3 for details.
2. Use a class when you need to encapsulate data and functionality into an object.

3. The class interface should be public.

4. Helper functions used by that class should be private.

5. Data in a class should always be private.

6. In situations where a parent class and a child class are meant to be tightly coupled, it is acceptable to have some protected data and functionality. However, protected members should be minimized; tightly coupled classes should be avoided when possible.

7. Use a struct when you need to encapsulate data, but do not want to provide any functionality, except maybe a constructor.
Chapter 4

Technical Writing

4.1 Motivation

Engineers and scientists need good technical writing skills. There are times when a
scientist or engineer needs to document work done by himself or his team. There will
be other times in which the engineer needs money, support, or resources to accomplish
a goal, and needs to write in order to attain such resources.

Good writing helps a person excel, because good writing makes the author appear
intelligent and well spoken. Likewise, poor writing creates an image of ignorance and
laziness. These general rules apply not only to formal correspondence and documenta-
tion, but also to e-mail.

Because writing is so important, it is required in this class, as well as many other
classes at this university. For those students who do not feel prepared for the challenge,
help is available. First, the student should read Matt’s Technical Writing Guidelines
(see section 4.2), which are applicable to this class. Second, the student should read
“Engineered Writing”, a short technical writing manual which is on reserve in the
library. The Engineered Writing technique will help the student both now and in the
future. Finally, the student is encourage to utilize the campus writing center.

4.2 Matt’s Technical Writing Guidelines

The goal of technical writing is written communication that cannot be misunderstood [2].
While this is a lofty goal, a competent writer can get quite close to this goal; the results
will be easy to read and communicate effectively. The guidelines below are meant to
aid the student striving for top notch writing.

4.2.1 Grammar

Engineers are expected to speak and write correct English. This means correct gram-
mar is a requirement. The student is expected to proofread his work, and to utilize a
grammar handbook such as the “Harbrace College Handbook.”
4.2.2 Person

The use of second person (you, your) should be avoided in technical writing. The use of first person (I, we) may be used on a limited basis. Third person is preferred.

4.2.3 Gender in Writing

When a third person singular pronoun is needed, use the masculine form [2]. In English writing, “he” can be used to denote either a male person, or a person (male or female), based on the context. Use of “he/she” or “himself/herself” is clumsy and difficult to read. For further discussion on this subject, refer to “Engineered Writing.”

4.2.4 Margins

Left, right, top, and bottom margins should be 1.5 inches.

4.2.5 Font type and size

The font should be either Times or Times New Roman. The size should be similar to the text appearing in this document, which is about 10 pt.

4.2.6 Section headings

The use of numbered section and subsection headings is required. The headings and subheadings should represent a topic and subtopic outline of the document. The size of these headings should be slightly larger than the standard text.

4.2.7 Spacing

The document should use single spacing. However, an additional line of white space should appear between paragraphs. White space should appear above and below section and subsection headings.

Typesetting and Word Processing

These days, Microsoft Word has become the defacto standard word processor in most business environments. Reasons for this include ease of use, wide availability, and features. Thus, it is important for engineering students to learn this word processor.

In many scientific communities, particularly Computer Science, many professionals feel that \LaTeX{} is far superior to a word processor for technical writing. \LaTeX{} is a document typesetter, not a word processor. This means that the user provides information, and provides \LaTeX{} with commands that indicate the logical significance of the information provided. \LaTeX{} takes care of the actual layout and placement of the various elements.

The default layout and formatting of \LaTeX{} is most desirable for CS 284. The formatting guidelines above attempt to summarize the style; this information is provided
to benefit those who do not want to learn \LaTeX. However, students are strongly encouraged to learn the tool, not only for this class, but also for the benefit of future writing.
Chapter 5

Socket Programming

A socket is a programming tool that allows computers in the same domain to exchange data with one another. According to Wang [6], there are three basic types of sockets.

5.1 Socket Types

Stream socket: Stream sockets use TCP/IP. A connection is formed when a pair of computers connects to one another using a stream socket. Once the connection is setup, it may be used in much the same way as a pipe. According to Wang [6], the stream socket supports a “bidirectional, reliable, sequenced, and unduplicated flow of data.”

Datagram socket: Datagram sockets use the User Datagram Protocol (UDP/IP) [6]. While datagram sockets are bidirectional, they are not particularly reliable. They will not be discussed further in this document.

Raw socket: The raw socket is a low-level, datagram-oriented socket [6]. Again, this type of socket will not be discussed further in this document.

5.2 Stream Socket Functions

accept: Used to accept a connection on a socket [3].

headers: sys/types.h, sys/socket.h [3]

prototype: int accept(int s, struct sockaddr *addr, socklen_t *addrlen); [3]

args:

- The first argument is the file descriptor for the socket.
- “The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the connecting entity...” [3]
- The size of the structure.
**return:** Returns a non-negative file descriptor on success, or -1 on error. The errno value will be set appropriately [3].

**bcopy:** Copies a byte sequence.

**headers:** strings.h [3]

**prototype:** void bcopy(const void *src, void *dest, size_t n); [3]

**args:**
- The src argument stores the source data to be copied [3].
- The dest argument is where the bytes will be copied to.
- The size_t n indicates the number of bytes to copy.

**bind:** Used to associate a socket with a port on the local machine [1].

**headers:** sys/types.h, sys/socket.h [3]

**prototype:** int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen); [3]

**args:**
- The integer arg sockfd is the file descriptor for the socket.
- The pointer to the sockaddr struct should refer to a previously defined structure.
- The last argument is the size of the sockaddr structure.

**return:** Returns zero on success, or -1 on error. The errno value will be set appropriately [3].

**connect:** Called by the client to connect to the server. There should be a corresponding accept in the server.

**headers:** sys/types.h, sys/socket.h [3]

**prototype:** int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen); [3]

**args:**
- The sockfd argument should be the descriptor returned by the socket function call.
- The serv_addr pointer should refer to an instance of the structure previously defined.
- The addrlen argument should indicate the size of the structure.

**return:** Returns zero on success, or -1 on error. The errno value will be set appropriately [3].

**gethostbyname:** Used to define the h_name field of the hostent structure.

**headers:** netdb.h [3]

**prototype:** struct hostent *gethostbyname(const char *name); [3]

**args:**
5.2. STREAM SOCKET FUNCTIONS

- The **name** argument indicates the name of the machine to which the socket will be connected [3].

  **return:** The function returns a pointer to a hostent structure with the name field as specified on the function call [3].

**listen:** Specifies a willingness to receive connections across a socket, and sets the queue limit for that socket [3].

  **headers:** sys/socket.h [3]
  **prototype:** int listen(int s, int backlog); [3]
  **args:**
  - The first argument is the file descriptor for the connection.
  - The second argument sets the maximum number of connections to queue. This is limited to five or less [6].
  **return:** Returns zero on success, or -1 on error. The errno value will be set appropriately [3].

**socket:** The socket function creates a socket on the calling machine [6]. It also creates a file descriptor [1]. This function will be used by both the client and the server.

  **headers:** sys/types.h, sys/socket.h [3]
  **prototype:** int socket(int domain, int type, int protocol); [3]
  **args:**
  - The **domain** argument is used to specify the communication domain of the socket. The options are AF_UNIX, which is a UNIX socket, and AF_INET, which is an internet socket [6]. In CS 284, the internet socket will be used.
  - The **type** argument is used to indicate what kind of socket should be created. The options are SOCK_STREAM for a stream socket, SOCK_DGRAM for a datagram socket, and SOCK_RAW for a raw socket [6]. In CS 284, we will use the stream socket.
  - According to the socket man page, the **protocol** argument “specifies a particular protocol to be used with the socket” [3]. For our purposes, we will always use a zero for this argument. In situations where multiple protocols are available, the reader should refer to the protocols man page.
  **return:** The socket function returns a file descriptor, also referred to as a socket descriptor.
5.3 Socket Structures

sockaddr_in: [6]

```c
struct sockaddr_in
{
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```

hostent: [6]

```c
struct hostent
{
    char *h_name;
    char ***h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
```
Bibliography

[1] HALL, B. Beej’s guide to network programming.


