Solutions to Assignment 1

Programming Exercises

P1. Read the UNIX manual page for the system program cat. Write your own version, called mycat, that performs the basic functionality of cat (i.e., it does not need to take option switches). Copy filecopy.cc to a new file called mycat.cc, and adapt it as necessary. You *must* use the copyout function as provided—do not modify it.

The code is identical to the filecopy program with the exception of the main function, which is rewritten as follows:

```
const int STDIN_FD = 0;
const int STDOUT_FD = 1;

int main(const int argc, const char *argv[]) {
        if (argc < 2)
                copyout(STDIN_FD, STDOUT_FD);
        else
                for (int i = 1; i < argc; ++i) {
                        const int in_fd = open(argv[i], O_RDONLY);
                        if (in_fd < 0)          // open failed, die
                                err_sys("Can't open input file");
                        copyout(in_fd, STDOUT_FD);
                        if (close(in_fd) < 0)   // close failed, die
                                err_sys("Error closing input file");
                }
        return 0;
}
```

P2. The filecopy program does not include any handling for interrupt signals. If the user presses Control-C while the file is being copied, the program will exit and leave a partial copy behind.

Adjust the code to catch the SIGINT signal and remove the partial copy (if the file has not been completely copied) before exiting.

The code for copyout and err_sys can remain the same. We just need to add signal handling to the existing code (code in grey is unchanged):

```
#include <signal.h>

const char *out_name;       // destination file (used by main & handle_sigint)

void handle_sigint(int signo) {
        unlink(out_name);
        cerr << "Interrupted!" << endl;
        exit(2);
}
```

```
int main(const int argc, const char *argv[]) {
        if (argc != 3)
                err_sys("Need exactly two arguments");
        const char *const in_name = argv[1];
        out_name = argv[2];

        const int in_fd = open(in_name, O_RDONLY);
        if (in_fd < 0)                  // open failed, die
                err_sys("Can't open input file");

        signal(SIGINT, handle_sigint);

        const int out_fd =
                open(out_name, O_WRONLY | O_CREAT | O_TRUNC | O_EXCL, 0666);
        if (out_fd < 0)                 // open failed, die
                err_sys("Can't open output file");

        copyout(in_fd, out_fd);         // copy the file

        close(in_fd);
        if (close(out_fd) < 0)
                err_sys("Error closing output file");

        return 0;
}
```

In fact, while I gave full credit for the above code, it is actually flawed, containing a race condition. Eliminating the race condition requires slightly more complex code:

```
#include <signal.h>

const char *filename = 0;    // destination file (used by main & handle_sigint)
bool interrupted = false;    // set if interrupted before out_name is set

void interrupt_terminate() {
        unlink(filename);
        cerr << "Interrupted!" << endl;
        exit(2);
}

void handle_sigint(int signo) {
        if (filename != 0)
                interrupt_terminate();
        else
                interrupted = true;
}

int main(const int argc, const char *argv[]) {
        if (argc != 3)
                err_sys("Need exactly two arguments");

        const char *const in_name = argv[1];
        const char *out_name = argv[2];
```

```
    const int in_fd = open(in_name, O_RDONLY);
    if (in_fd < 0)                  // open failed, die
        err_sys("Can't open input file");

    signal(SIGINT, handle_sigint);

    const int out_fd =
        open(out_name, O_WRONLY | O_CREAT | O_TRUNC | O_EXCL, 0666);
    if (out_fd < 0)                 // open failed, die
        err_sys("Can't open output file");

    filename = out_name;
    if (interrupted)
        interrupt_terminate();

    copyout(in_fd, out_fd);         // copy the file

    close(in_fd);
    if (close(out_fd) < 0)
        err_sys("Error closing output file");

    return 0;
}
```

Other solutions are possible, including modifying copyout to check for an interrupted flag that is set by the interrupt handler.

P3. The filecopy program incurs the overheads of the C++ iostream library even though cerr is only used to print a short error message in the event of failure.

Adjust the code to print the same error message using a single invocation of the write system call.

There are a number of ways of answering this question. I've chosen an elegant-looking solution that fits the requirements of the question:

```
#include <string>

const int stderr_fd = 3;

void err_sys(const char *const message) {
    string output(message);
    output += " (";
    output += strerror(errno);
    output += ")\n";
    write(stderr_fd, output.c_str(), output.length());
    exit(1);
}
```

Alternatives include using sprintf, using strlen and strcpy, and writing your own code to copy the messages into a buffer.

All solutions must create some kind of buffer to store the message before writing it with write. An important issue is choosing the correct buffer size. C and C++ are notorious for carelessly written code that can be tricked into producing buffer overflows; a technique often exploited by crackers.

P4. Read the UNIX manual page for the system program wc.Write your own version, called mywc, that performs the basic functionality of wc (i.e., it does not need to take option switches or command-line arguments and only reads from standard input).

Your program may not (explicitly) call any functions in the standard C or C++ libraries except for isspace—everything else must be accomplished through system calls or functions you have written yourself.

```
// BUFFER_SIZE is the size of the buffer used when reading/writing chunks with read & write
const int BUFFER_SIZE = 8192;

// MAX_INT_LENGTH is the length of the ASCII rendition of an integer (the calculation
// below is an integer version of
//          ceil( sizeof(int) * log(256)/log(10) ),
// where 53 / 22 is used as an approximation of log(256)/log(10).
const int MAX_INT_LENGTH = (sizeof(int) * 53 + 52) / 22;

// Constants for standard input and standard output file descriptors
const int STDIN_FD = 0;
const int STDOUT_FD = 1;

// uitostr — Modelled after the UNIX function ulltostr, this function converts unsigned integer
//     to a null-terminated string in the supplied buffer (endptr points to one location beyond
//     the end of the buffer). Optional arguments provide additional control as to whether it
//     null-terminates the string and whether it pads it out with spaces.
char *uitostr(unsigned int value, char *endptr,
                int padding = 0, bool null_terminate = true) {
    if (null_terminate)
        (--endptr) = '\0';
    do {
        --padding;
        (--endptr) = '0' + (value % 10);
        value = value / 10;
    } while (value != 0);
    while (padding-- > 0)
        (--endptr) = ' ';
    return endptr;
}
```

continued over

```
int main(const int argc, const char *argv[]) {
    if (argc != 1) {
        write(2, "No arguments allowed for mywc\n", 30);
        exit(1);
    }

    char buffer[BUFFER_SIZE];
    bool in_word = false;
    unsigned int words = 0;
    unsigned int lines = 0;
    unsigned int chars = 0;

    for ( ; ; ) {
        const int read_count = read(STDIN_FD, buffer, BUFFER_SIZE);
        if (read_count < 0) {                   // read failure
            write(2, "Read failure\n", 13);
            exit(2);
        }
        else if (read_count == 0)
            break;                              // end of file — exit loop

        chars += read_count;

        for (int i = 0; i < read_count; ++i) {
            const char c = buffer[i];
            if (isspace(c)) {                   // get ready for a new word
                if (in_word)
                    in_word = false;
                if (c == '\n')
                    ++lines;
            } else if (!in_word) {              // start of a new word
                in_word = true;
                ++words;
            }
            // else                             middle of a word, do nothing
        }
    }

    const int MAX_OUTPUT_SIZE = 3 * (MAX_INT_LENGTH + 1);
    char output[MAX_OUTPUT_SIZE];
    char *outp = &output[MAX_OUTPUT_SIZE];
    *(--outp) = '\n';
    outp = uitostr(chars, outp, 7, false);
    *(--outp) = ' ';
    outp = uitostr(words, outp, 7, false);
    *(--outp) = ' ';
    outp = uitostr(lines, outp, 7, false);
    *(--outp) = ' ';
    write(STDOUT_FD, outp, &output[MAX_OUTPUT_SIZE] - outp);

    return 0;
}
```

Written Exercises

W1. If you modify BUFFER_SIZE in filecopy.cc you will find that increasing the buffer size does not make much of an impact on performance, but reducing it can make a significant difference (try setting BUFFER_SIZE to 8 or even 1).

(a) Why does a small buffer size cause such poor performance?

(b) Why doesn't increasing the buffer size beyond 4096 help much?

(c) Explain why a program performing one system call and 1000 subroutine calls should perform better than a similar program that performs no subroutine calls and 1000 system calls.

(a) With a small buffer size, the number of mode switches increases and begins to dominate the run time. Copying data from user space to kernel space is expensive, and some of the overheads are independent of the amount of data copied. Thus, copying very small chunks frequently is more expensive than copying larger chunks infrequently.

(b) As the buffer size increases, mode switch and data copying overheads become a less and less significant fraction of the runtime of the program, and other factors begin to dominate instead.

In fact, increasing to a very large buffer can actually *slow the program down*, because the program will have a larger memory footprint and decreased potential for parallelism. We can see that latter issue if we imagine copying a 1 MB file using a 1 MB buffer. In this case, the file won't begin to be written until it has been read in its entirety, and thus the operating system's facilities for read-ahead and write-behind may go relatively unused.

(c) System calls incur a mode switch overhead, which can be significant. Similarly, data may need be copied from user space to kernel space and back. Finally, the kernel must be diligent in checking parameters for validity. Subroutine calls typically do not incur these overheads.

W2. Recently, some operating systems have added a system call identical in functionality to the copyout function in filecopy.cc. Give the tradeoffs involved in providing this functionality as a system call versus providing it as a subroutine in a programmer's library.

Consider a library implementation of copyout with buffer size B copying N bytes of data. In this case, there will be $\lceil N/B \rceil$ reads and the same number of writes. This contrasts with a system-call implementation where there would be one mode switch. Clearly, the system-call implementation has reduced mode-switch overhead.

Similar arguments apply to data copying and parameter checking. The system-call implementation does not need to copy data to user space and back.

The downside of this approach is that it violates the simplicity principle. In class we said that the kernel of the operating system should provide only those facilities that the program cannot be trusted to provide for itself. Clearly, a program can provide the copyout function if read and write are available.

Counting words in a file would also be faster if implemented as a system call, but should it be? Where will it end? With a run_a_web_server system call?[1] Keeping the kernel simple makes the operating system manageable and makes it easier to debug.

1. Microsoft is apparently going to include a Web server inside the Windows 2000 kernel so it would seem that they are prepared to go to "crazy" extremes in search of small performance improvements.

w3. You should have been reading the *Kernel Traffic* Web site during the first few weeks of the course. Pick one topic discussed there, describe the issue briefly and make constructive comments of your own. (Include the date of the original discussion and the issue number of *Kernel Traffic*).

I gave an example in class from Apple's darwin-development mailing list. Discussions for the past week have centred around whether the HFS+ filesystem should remain case insensitive in Mac OS X (a forthcoming UNIX-based operating system that will be the successor to the current MacOS 8.x line).

The participants in the discussion overlooked a key point: policy should be separated from mechanism when possible and practical. How filenames compare for equality is a matter of policy, and there is no reason a file system could not support several policies.

For the full discussion, including my own contribution, see http://webx.lists.apple.com/.