CS 422 (Park) **Assignment III** **Due: Oct. 9 (Sun.), 2011**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 11pm on the due date. You may use any number of word processing software (e.g., Framemaker, Word, LaTeX), but the final output must be in pdf format that uses standard fonts (a practical test is to check if the pdf file prints on a CS Department printer). For experiments and programming assignments that involve output to terminal, please use* `script` *to record the output and submit the output file.*

PROBLEM 1

Read Chapters 12–14 from Comer.

PROBLEM 2 (30 pts)

(a) Why is "spread spectrum" of DSSS (direct sequence spread spectrum) CDMA an appropriate description of what is happening at the sender side when bits are transmitted? For a given pseudo-random chipping sequence, are there payloads (i.e., bit patterns) for which "spread spectrum" would not apply when transmitted by the sender? Why is spread spectrum technology considered desirable in modern networks, especially in the context of CDMA and OFDMA? Why might it not be considered a positive feature in traditional transmission technologies such as FDMA and TDMA?

(b) In what way is OFDMA superior to FDMA? What benefit does orthogonality bring? How is OFDMA related to CDMA? What is inter-symbol interference (ISI) and how does it influence the speed of a communication system? Does orthogonality help mitigate ISI?

PROBLEM 3 (45 pts)

(a) Error detection using an odd parity allows detection of an odd number of bit flips. For example, for the 4-bit data 0110, the payload after adding odd parity is 01101. An odd number of bit flips would make the number of 1's in the resultant payload be even which is a sure indicator that bits have flipped. By the same token, an even number of bit flips cannot be detected. To detect bit flips in n-bit data, extra bits are needed to help with detection. Suppose our budget of extra bits is k. The original data can be any n-bit pattern but the packet payload will be an $(n+k)$-bit pattern that is transmitted by the sender. The receiver will receive a $(n+k)$-bit payload, either intact (no bit flips) or distorted by one or more bit flips. This error checking is typically done by firmware in the receiver's NIC. If error detection fails, the default action is to discard the packet (and its payload) and not tell the kernel about it, so that both OS and application are oblivious to such packet arrivals.

Recapping the above steps, the sender takes n-bit data, generates $(n+k)$-bit payload, which is then sent out using any number of tranmission technologies (e.g., TDM, OFDM). It is important to note that the $(n+k)$-bit payload need not be the original n data bits plus extra k bits. The parity bit method where an extra bit is added to the (unchanged) original data is a special case. In general, the original n-bit data is mapped to an arbitrary $(n+k)$-bit pattern that need not resemble the original data. Thus we can think of the sender having a 2-column table where the first column contains all possible n-bit patterns (for all possible n-bit data that an application may request to be sent)—there are 2^n entries or rows in the table—and the second column contains a $(n+k)$-bit pattern (called a code word) for each n-bit data. Thus although there are 2^{n+k} possible $(n+k)$-bit patterns or code words only 2^n (the number of entries or rows) of them are selected and utilized in the table. The receiver must have the same 2-column table so that when it receives a $(n+k)$-bit payload, it can do a table look-up in reverse to determine what the corresponding original n-bit data was. If this table look-up fails—i.e., the $(n+k)$-bit payload received does not match any of the $(n+k)$-bit code words in the second column—then it knows that bit flipping must have occurred, which is to say that the payload received has not passed the NIC's error detection. However, as noted in class, error detection methods used in practice only detect some but not all errors, hence passing the NIC's error detection does not guarantee that the payload forwarded to the kernel is error free.

Consider the following example where $n = 4$ and $k = 2$. Suppose the first two rows of the table contain:

0000 100101
0001 010100

Suppose the sender transmit 100101 in the payload representing 4-bit data 0000. Suppose during transmission 3 bits flip—say, first, second, and sixth—such that the payload received at the receiver is 010100. The receiver would decide that 0001 is the original data which, of course, is wrong. Noting that absence of errors is determined by whether the received payload matches any of the code words in the table, how would the 2^n code words out of a total of 2^{n+k} need to be selected so that all possible m (or less) bit flips are detected? (*Hint: We want m bit flips of the code word carried as payload to be "conspicuous," i.e., not disguise the distorted code word as another valid code word in the table. The number of bit flips needed to morph a code word into another is called their Hamming distance.*)

(b) Describe a brute-force method for creating such an error detection table. Is your method feasible in practice? Discuss your reasoning.

(c) As a continuation of part (a), consider the problem of correcting errors when arbitrary (up to) m bit flips occur. Intuitively, error correction is a more demanding task than error detection. The table based error detection framework, with minor adjustment, can be adopted to carry out error correction. Describe how to choose the code words so that all possible m (or less) bit flips are corrected, and how correction is to be performed with such a code word table.

PROBLEM 4 (50 pts)

A continuation of Problem 4(b), Assignment II, instead of using `select`() to monitor a set of I/O descriptors in a synchronous, blocking fashion, let's consider using interrupt based asynchronous, nonblocking I/O availed by signals in UNIX-like operating systems to achieve the same task. The kernel service provided by Linux is through user defined callback functions—called signal handlers—that are registered with the kernel using the `sigaction`() system call (for this assignment do not use `signal`()). A user process is asking the kernel to execute the supplied callback function when a certain "interrupt event" (aka signal) such as data being written to a FIFO occurs. For FIFOs the signal is called SIGIO. The callback function, i.e., signal handler, is said to catch the SIGIO signal, and the writing to FIFO event is said to raise the SIGIO signal. There are many non-I/O related signals such as SIGSEGV that is raised when a process references memory outside its allocated space. By registering a signal handler for SIGSEGV with the kernel, segmentation faults need not lead to default process termination. Using the signal mechanism to monitor multiple I/O interfaces is asynchronous in the sense that a process may be engaged in other tasks when a raising of a signal (such as SIGIO) causes it to branch to the registered signal handler. In our case, the signal handler for SIGIO that monitors multiple FIFOs would be written to check which of the FIFOs being monitored was written to (not necessary when a single I/O interface is being monitored), read from the FIFO in question, and return. Branching to/from signal handler is handled by the kernel transparent to the process. When dealing with signals, it is important to note that UNIX-type signals act as binary flags—an event has occurred or not—and they do not count the number of monitored events. For example, while a signal handler is reading from an I/O interface, additional write events might occur but the kernel will not indicate how many. It is the programmer's responsibility—and common practice—to check an I/O buffer for outstanding messages before returning so that I/O writes are not missed. This is especially so for network I/O interfaces that operate at high speeds where new packet arrivals can occur while a signal handler is not done reading and processing a previous packet. The signal mechanism is nonblocking in the sense that the kernel requires an I/O descriptor that is being monitored using signals be put in nonblocking mode.

Reimplement the client/server code of Problem 4(b), Assignment II, such that a signal handler for SIGIO is used to service client requests arriving on two FIFOs in place of `select`(). Perform the same run-time tests and show that the signal based implementation works correctly.

PROBLEM 5 (50 pts)

A continuation of Problem 4 in this assignment, rewrite the client so that it takes three command-line arguments, *app*, *num* and *interval*, where *app* specifies the application to be executed by the server (such as *finger*), *num* specifies how many back-to-back requests the client should send to the server, and *interval* specifies (in unit of milliseconds) how long the client should wait between successive requests. For example,

```
% client.bin date 10 200
```

would mean to ask the server to run the command `date`, make 10 such requests back-to-back, with a time interval of 200 msec between requests. Create your own revised request message format such that arbitrary commands (without their own command-line arguments) can be requested of the server to be executed. One (bad) solution to implementing the modified client that sends repeat requests is to use `usleep`() between successive requests which puts the calling process to sleep for a specified duration (i.e., kernel context-switches out the process) and resumes it after a specified sleep interval. This is not a desirable solution if the synchronous, blocking read that waits for the server's response is preserved. This is so since blocking on `usleep`() will delay doing a blocking read on `read`() which, in turn, delays the next call to `usleep`(). Thus, although functionally correct, the code is poor with respect to its performance since reading of server responses will be delayed and the wait interval between successive requests is likely to be inaccurate due to blocking on `read`(). To overcome this problem, we will consider a solution that implements sending of paced repeat requests using signals, specifically SIGALRM. By registering a signal handler for SIGALRM and calling `ualarm`() with the desired wait interval, the kernel will set an alarm (a kernel software clock event) that goes off after the user specified period. When SIGALRM is raised, your signal handler will write a new request to the server's FIFO, call `ualarm`() to set the next alarm event, and return. Since `ualarm`() is a nonblocking system call, the client can continue to block on `read`() waiting on the response from the server. If the wait interval, *interval*, is small and the command to be executed takes a bit of time to execute then it is possible that SIGALRM will be raised while the process is blocking on `read`(). This may mean that the next client request arrives at the server before a child process is done running the command requested in an earlier request. In our current single host environment, keep in mind that server and client processes are sharing the same CPU(s). As noted in class, a major improvement (historically) to the `read`() system call is its correct resumption by the kernel after `read`() has been interrupted which spared programmers from dealing with it (a source of run-time bugs in years past).

Repeat the tests of Problem 4 with *app* `finger`, *num* 10, and *interval* 200. When the client's SIGALRM handler is invoked by the kernel, make it print a message to standard output before returning so that the sequence of request transmission and response arrival events can be tracked. Rerun the test with a slower command than `finger`, and use smaller *interval* values to "overload" the server. Note that doing so, in the extreme, can be viewed as a form of denial of service attack.

PROBLEM 6 (50 pts)

As a continuation of Problem 5 of this assignment, modify the client so that it uses SIGIO to asynchronously receive responses from the server, in addition to using SIGALRM to pace its repeat requests as before. In the first version, use `pause`() to do a blocking wait on SIGALRM or SIGIO since the client doesn't have anything to do after transmitting a request until a server response arrives or it's time to send the next request. In a second version, instead of calling `pause`(), put the client in an infinite loop which emulates the client having to do other tasks besides sending requests to servers and receiving their responses. For example, a web browser is a classical client but because of its graphical user interface (among other chores) significant CPU cycles are expended on non-networking tasks. Test and compare the two versions using the "overload" test scenario of Problem 5. Are there any noticeable performance differences between the two versions? Separate from testing, look into Linux signals to determine what happens if while the SIGIO signal handler is running SIGALRM is raised by kernel, and vice versa. Is the default behavior implemented by the kernel what should happen for correct client behavior? Explain your reasoning.