Project 1 measures process startup times. The idea is measurement, not processes per se, so file open or other operation could be substituted. However, too short an operation makes measurement difficult and too consistent.

1 Measure the average time it takes to start up a process, by writing a program that does it.
2 Run it several times, and present a histogram of your results.
3 What are mean and standard deviation for your observations?
4 What is the 95% confidence level interval?
5 Why shouldn't it always take exactly the same amount of time to perform this simple action?

Note: Students usually succeed in writing the fairly trivial program to start a process (with UNIX fork(), or DOS spawn(), or whatever) without any further help. Because student programs often include resource-hogging bugs, I’ve sometimes talked about the problems if many processes are created and not terminated.
In project 2 you will attempt to measure the effect on program execution speed of memory cache. Cache is a layer of the memory hierarchy that stands between main memory and the CPU. Optimizing for best use of cache is increasingly important in optimizing for best performance. A memory reference that can be satisfied from cache is typically much faster than one that must go to main memory.

First, a few words about cache memory. Semiconductor memory comes in various speeds, and the faster stuff costs more. An ad in last month's Byte magazine advertises 32KB of 12 ns memory for 12.75. For 11.50, the same ad offers 256KB of 70 ns memory. If you built an 8 Meg memory for your PC entirely with the fast stuff, you'd spend about 3000 just on the memory, whereas 8 MB of the `slow' memory would cost only 368. For some applications, you'd want to spend the money. Supercomputers, which are designed to get the biggest bang without regard for the bucks, take this approach.

Caching is one way to mix fast and slow memory to get some of the advantages of each. Nowadays it is used on almost all computers, from PCs to mainframes. The scheme is as follows: The main memory is made up of cheaper, slower chips. We use the fast, expensive memory to store copies of some recently used values and their addresses in a cache. A cache controller is interposed between the CPU and the two memory systems. When the CPU does a store, a copy of the value and the address is saved in the fast memory. When the CPU asks for the contents of an address, the cache controller checks to see if it has that address stored in fast memory, and if so, it can return the value more quickly than if the value is only stored in the slow memory.
If 90 of requests can be satisfied by reference to the cache, and a cache reference takes 12 ns, and the other 10 of requests take 80 ns to complete, the average time for a memory request will be $0.90 \times 12\text{ ns} + 0.10 \times 80\text{ ns} = 18.8\text{ ns}$ Of course, because we now have to store the address as well as the value, we'll lose big if we have to have 7MB (+28MB of address) of expensive fast memory in order to reach a 90 hit rate.

Luckily it turns out that most programs exhibit good temporal locality, that is, they tend to refer to the same words of memory over and over, either in code loops, or as data par. So a PC cache of 128K is big enough to give a 90 hit rate for most programs. In order to avoid having to store another 512KB of addresses, several different schemes are used; one frequent scheme is to organize adjacent bytes into cache blocks that can share the same address. For example, if the cache is organized into 128-byte cache blocks, each of the 128 bytes in a block will have the same high-order bits, and so we need store only $1/128$ as many addresses. For our example, we could get away with storing only 1024 addresses.

Most current operating systems pay no attention to caching; the hardware tries to hide the details from the software, and the tradeoffs change rapidly. But if you were attempting to stretch the edge of your computer's abilities, you might want to take cache properties into account. Note that if you are running a program on our hypothetical PC that never hits cache, your program can run $70/12$ times longer than a program which is carefully tuned so that it always hits cache.

The computer system you usually use probably has a cache system, although some PC's may not. In this homework, you'll try to discover what its parameters are. You'll attempt to answer the questions:

1. What is the difference in execution time between a reference to main memory and a reference to cache?
2. How big is a cache block?
3. How big is the cache?
4. How long does a reference to main memory take to complete?
5. How long does a reference that can be satisfied from cache take to complete?

The basic idea is to compare the execution time of two loops, one of which tries to force reading a new value from main memory at each iteration. Because we are talking only a few ns difference between the two read times, you'll have to execute a lot of reads in order to be able to measure the difference.

The test programs for this problem would be easier to program in assembly language. I don't expect you to do that. However, you should probably determine what machine language instructions occur in the loops you are timing, because the loops will obviously take different amounts of time to run if they include different instructions. On UNIX systems, one way to do this is with the -S switch to the C compiler. Some PC compilers will not provide a similar switch; in this case you may be able to use a debugger to examine the program.

Your final program(s) won't be very long, although you may write and discard several; they may not work. (This is a difficult exercise.) So give me a clear explanation of how you propose to perform the measurements, and what you think may have worked or gone wrong.

Many modern computer systems have several layers of cache; for example most of the current crop of microprocessors have a significant amount of cache on the chip with the CPU, but also have an off-chip cache that is significantly faster than main memory. If you are still full of vim after dealing with the first few questions, you might attempt to consider whether there seem to be two or more levels of cache in your system.
Project 3

The programming problem for this week involves adding deadlock detection/prevention to a simple transaction package. I have given you an undebugged group of transactions that play with the same set of semaphores. I claim that they are prone to deadlock, and I want you solve the problem in two distinct ways.

1. First compile and execute the distributed program. Note that it deadlocks. In order to make running it less frustrating, I made it terminate with the message `No runnable threads.'

2. Modify the program by changing the order of calls to use two-phase locking and produce a version of the program with modified threads, which doesn't deadlock. Note that it never terminates, because each thread sits in a rather pointless infinite loop.

3. Now go back and solve the deadlock problem by modifying the resource management package, instead of the thread code: Check for deadlock when a thread waits for a semaphore, and if granting it would cause deadlock, make the thread wait until things are safe. (In order to check, you'll need to define some new data structures, and perhaps insert some additional code into each thread to manage them.)

You'll turn in copies of your code for parts (2) and (3), and a wordy English explanation of what you've done and why.

The sample code follows.

/*
 * This is a simple-minded exercise in locking (and deadlock)
* it is intended to suggest some of the locking required
* for file system operations (although historically
* single-processor OS's have skimped on locking.)
*/

#include <stdio.h>
#include <errno.h>
#include <dce/pthread.h>
/* */
#define TRANS_SIGNAL(S) semaphore_signal(&S);
#define TRANS_READ_WAIT(S) TRANS_WAIT(S)
#define TRANS_READ_SIGNAL(S) TRANS_SIGNAL(S)
#define TRANS_WRITE_WAIT(S) TRANS_WAIT(S)
#define TRANS_WRITE_SIGNAL(S) TRANS_SIGNAL(S)

#define NUM_FILES 10
#define NUM_THREADS 4

define TRANS_BEGIN
#define TRANS_END \ if (pthread_self_is(DiskAllocation.owner))
semaphore_signal(&DiskAllocation); \

#define TRANS_ABORT
#define TRANS_WAIT(x) semaphore_wait(&x)
#define THREAD_SEMAPHORE(s,v) \ semaphore_t s={v};

/* some code to implement the transaction package */
pthread_mutex_t semaphore_mutex;

typedef struct {
  int count;
  pthread_cond_t queue;
  int queue_length;
  pthread_t owner;
} semaphore_t;
int RunnableThreads; /* a count for easy determination of deadlock */
int Pending=0;

semaphore_wait(semaphore_t* s){
    int thisThreadWait;

    pthread_mutex_lock(&semaphore_mutex);
    thisThreadWait = 0;
    while (s->count <= 0){/* wait for condition, freeing mutex */
        thisThreadWait = 1;
        RunnableThreads--;
        s->queuelength++;
        if (pthread_cond_wait(&s->queue, &semaphore_mutex) != -1){
            printf("call to pthread_cond_wait failed with errno = %d\n", errno);
        }
        s->queuelength--;
        RunnableThreads++;
    } /* DECthreads documentation says wait may return prematurely*/
    if (thisThreadWait) Pending--;
    s->count--;
    s->owner = pthread_self();
    pthread_mutex_unlock(&semaphore_mutex);
    /* introduce some randomness */
    if (rand()<0x40000000) pthread_yield();
}

semaphore_signal(semaphore_t* s){
    pthread_mutex_lock(&semaphore_mutex);
    if (s->queuelength > 0){ /* if anyone waiting */
        Pending++;
    }
    s->count++;
    s->owner.field1 = 0;
    s->owner.field2 = 0;
    s->owner.field3 = 0;
    pthread_cond_signal(&s->queue);
    pthread_mutex_unlock(&semaphore_mutex);
int pthread_self_is(pthread_t other)
{
    pthread_t t;

    t = pthread_self();
    if (t.field1 != other.field1) return 0;
    if (t.field2 != other.field2) return 0;
    return (t.field3 == other.field3);
}

/* these semaphores are declared static because they have initial values */

THREAD_SEMAPHORE(Directory,1);
THREAD_SEMAPHORE(DiskQ,1);
THREAD_SEMAPHORE(InodeAllocation,1);
THREAD_SEMAPHORE(DiskAllocation,1);
semaphore_t Inode[NUM_FILES];

/*
 * no non-locking code; open_file() reads directory, to find inode
 * inode is locked while file is open (exclusive access to files)
 */

open_file(F)
{
    TRANS_READ_WAIT(Directory);
    TRANS_WAIT(Inode[F]);
    TRANS_READ_SIGNAL(Directory);
}

/*
 * create_file()
 * Uses fixed file, instead of getting one from Inode table,
 * to avoid implementing Inode table.
 * frees up all locks it uses.

-9-
create_file(F)
{
TRANS_WRITE_WAIT(Directory);
TRANS_WAIT(InodeAllocation);
TRANS_WAIT(Inode[F]);
TRANS_WAIT(DiskQ)
TRANS_SIGNAL(DiskQ)
TRANS_SIGNAL(InodeAllocation);
TRANS_WRITE_SIGNAL(Directory);
TRANS_SIGNAL(Inode[F]);
}

read_byte(F)
{
TRANS_WAIT(DiskQ);
TRANS_SIGNAL(DiskQ);
}

write_byte(F)
{
TRANS_WAIT(DiskAllocation);
TRANS_SIGNAL(DiskAllocation);
TRANS_WAIT(DiskQ);
TRANS_SIGNAL(DiskQ);
}

close_file(F)
{
TRANS_SIGNAL(Inode[F]);
}

/*
* rewrite directory so no pointer to inode
* release allocated Disk storage
*/
* mark inode unallocated,
*/
unlink_file(F)
{
TRANS_WAIT(Inode[F]);
TRANS_WAIT(InodeAllocation);
TRANS_WRITE_WAIT(Directory);
TRANS_WAIT(DiskQ);
TRANS_SIGNAL(DiskQ);
TRANS_WAIT(DiskAllocation);
TRANS_SIGNAL(DiskAllocation);
TRANS_WAIT(DiskQ);
TRANS_SIGNAL(DiskQ);
TRANS_WRITE_SIGNAL(Directory);
TRANS_SIGNAL(Inode[F]);
TRANS_SIGNAL(InodeAllocation);
}

/* copy one file to another */
cp_file(A,B)
{
open_file(A);
create_file(B);
open_file(B);
read_byte(A);
write_byte(B);
close_file(A);
close_file(B);
unlink_file(A);
}

void* file_copy(void* x)
{
int A,B;

A = (int)x;
B = A + 1;
for (;;) {/*ever*/
    TRANS_BEGIN

-11-
```c
main()
{
    pthread_t T[NUM_THREADS];
    int i;

    /* initialize semaphores */
    if ( pthread_mutex_init(&semaphore_mutex,pthread_mutexattr_default))
        printf("Complaint about mutex_init");
    if (pthread_cond_init(&Directory.queue,pthread_condattr_default))
        printf("Complaint about cond_init");
    if (pthread_cond_init(&DiskQ.queue,pthread_condattr_default))
        printf("Complaint about cond_init");
    if (pthread_cond_init(&InodeAllocation.queue,pthread_condattr_default))
        printf("Complaint about cond_init");
    if (pthread_cond_init(&DiskAllocation.queue,pthread_condattr_default))
        printf("Complaint about cond_init");
    for (i=0; i<NUM_FILES; i++)
        if (pthread_cond_init(&Inode[i].queue, pthread_condattr_default))
            printf("Complaint about cond_init");
        Inode[i].count = 1;
    }

    /* start up threads */
    RunnableThreads = 0;  /* don't count main thread */
    for (i=0; i<NUM_THREADS; i++)
        RunnableThreads++;
    pthread_create(&T[i],pthread_attr_default,file_copy, (void*)(2*i));
    pthread_detach(&T[i]);
    while (RunnableThreads || pending) pthread_yield();
    printf("No runnable threads\n");

    cp_file(A,B);
    cp_file(B,A);
    TRANS_END
}
```
The goal of this project is to decide what tradeoff of storage efficiency versus transfer speed might be appropriate for the files on your system, if you were designing the software for the existing hardware and files.

You may gather data on either a timesharing system or a PC; the PC will probably be easier to find documentation for (disk parameters are usually available in the SETUP program) but the assumptions I've suggested for optimum values are probably more appropriate to multi-user timesharing systems.

The parameter we will optimize is disk block size. Disk hardware supports transfers only in units of one or more sectors; for many disks the sector size is 512 bytes, but most operating systems try to group sectors together into blocks, so that the cost of a seek (moving the disk arm to the data) may be amortized over more bytes. The maximum possible transfer rate is achieved when an entire cylinder (all the sectors at a particular arm position) is transferred before the next seek. One way to optimize transfer rate would be to make the operating system unit of allocation one cylinder, so that all files would have an integer number of cylinders allocated to them.

The problem with this scheme is that it would waste storage if files were less than a cylinder in size. (On my home PC, each cylinder is about 500KB, and there are about 1000 cylinders.) How much storage would be wasted depends on the sizes of the files happens to be on the system.

A smaller block size would waste less storage space, but reduce the maximum possible transfer rate, assuming that a seek would be required for each separate block.

If we regard waste of storage space and waste of transfer rate as equally important, we may be able to find a `sweet spot' where the two wastes are balanced.
The first step is to determine the population density and access patterns of files for various sizes. We'll skip measuring the access patterns for this project, and just do a static measurement of file sizes.

The second step is to determine the parameters for your disks.

The third step is to combine these two data items in various combinations to study the effect of varying the size of storage block allocated on storage efficiency and file transfer time. So:

1. Profile the size of files on your system. Prepare graphs of:
   a) Number of files of a given size versus size of file
   b) Number of bytes stored on disk in files of each given size.

   Although it will probably be convenient to gather this data with a program (or a shell script, or the output of a backup utility) I don't want to see the program itself, just the two graphs, and an explanation of how you found the information.

   Probably the graphs will be rather bumpy, especially for larger file sizes; you may want to try to group sizes together for a smoother graph. Maybe a log scale would be appropriate?

   If for some reason you are unable to provide the data for the whole system, provide statistics on whatever subset you can. Explain your problem, and offer an estimate of how far off your data may be.

2. Using the actual size profile of files on your system, draw a graph of disk-space waste versus size of storage block allocated.

3. Determine from documentation the size, seek time, rotational latency, and transfer rate of your disks. [Although it will be much more satisfying to use real numbers for a contemporary machine, If you are unsuccessful in your search for documentation, you may use the following figures from my home PC:
<table>
<thead>
<tr>
<th>size</th>
<th>1014 cylinders</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>500 KiloBytes</td>
</tr>
<tr>
<td>/Cylinder</td>
<td></td>
</tr>
<tr>
<td>seektime</td>
<td>10 milliseconds</td>
</tr>
<tr>
<td>rotational latency</td>
<td>8.3 milliseconds</td>
</tr>
<tr>
<td>transfer rate</td>
<td>1.2 MB / sec</td>
</tr>
</tbody>
</table>

The maximum possible throughput of the disk, in bytes per second is the same as the transfer rate. But if you assume that each transfer must be preceded by a seek, and a wait while the beginning of the data spins around on the disk to be under the head (rotational latency) you get a smaller maximum throughput:

$$\text{Throughput} = \frac{\text{Bytes Transferred}}{\text{Seek Time} + \text{Rotational Latency} + \left(\text{Bytes Transferred} \times \text{Transfer Rate}\right)}$$

To read the whole disk will require several seeks. How many? What is the maximum throughput, if you read the entire disk?

1. Assuming that each file on your system is equally likely to be accessed, and that a seek will be required for each storage block, graph the percentage of possible disk throughput versus size of storage block.
2. Take files sizes into account, i.e. if the block size is 2048 bytes, but the file size is only 10 bytes, you should assume that the transfer time is for 2048 bytes, but the effective bytes transferred is only 10 bytes, that is
Throughput = \frac{10}{\text{Seek Time} + \text{Rotational Latency} + (2048 \times \text{Transfer Rate})}

If the block size is 2048 bytes, and the file size is 2050 bytes, you'd need to read two blocks (and the last one doesn't have much data.)

3. Comment on your graphs. Can you choose an optimum blocksize?
Project 5

Project 5 is a short program to support logging and crash recovery, as used in DBMS transactions. The example we’ll work on is very much a toy; it operates in an environment where all the interesting items are single characters stored in an array; the program will present a user interface that accepts the following commands:

- **begin t**  Starts transaction t. Several transactions may be in progress at once.
- **get t m**  As part of transaction t, get the value of item m. May report
  - this item also accessed by transactions t...
  or
  - Aborting; this item written by transaction t...
- **change t m c**  As part of transaction t, change item m to value c. May report
  - Aborting; this item written by transaction t
  or
  - Aborting; this item accessed by transactions t...
- **commit t**  Finishes transaction n, releases any locks.
- **abort t**  ends transaction t, and undoes any actions done for it.
- **crash**  aborts all transactions still unended.
- **display m n ___**. Shows the value of items m, n _

At any point in the running of the program, the display command shows the value of the latest unaborted changes, even if they have not been committed.
Because the user process doesn't actually crash, and crashes can come only between other commands, not in the middle of them, it is relatively easy to program a version of this program which meets its specification but is difficult to extend to a more general case.

1. Try to write an extensible version, using a log of old and new values and a crash-recovery algorithm. In addition to your code and some sample runs, include a text description of your algorithms and data structures. If you see alternative choices of algorithms, for example, locking versus time-stamping, mention why you chose the ones you did.

2. Suppose that you were going to extend the program of the previous question so that a crash could occur at any time, and the array and the log are both to be stored on disk. What mechanisms might you use to guarantee that the log is updated before the data array? Why does this make any difference?

3. What difference might it make to your program design if the transactions were running in separate threads? What about separate processes?
There's a well-known UNIX spell-checker script which uses pipes. Here's one version of it, the file spellcheck:

```
tr -cs A-Za-z '12' | sort -u | comm -23 - /usr/dict/words
```

The write-only character of this shell script is no doubt a big contributor to the popularity of UNIX' competitors. Because this example dates to the early days of UNIX, the various commands have switches designed to make it terse.

The `tr` command reads the standard input, and substitutes a newline (ASCII 012) for every character which is not alphabetic, squeezing successive newlines into one.

The `sort` command sorts the input lines, discarding duplicates.

The `comm` command compares two files, in this case standard input and `/usr/dict/words`, and prints those lines which appear in the first file, but not the second.

When I apply the spell-checker to a short example, I notice a deficiency in the technique. I type:

```
By the rude bridge that ard'd the flood,
their flag to april's breeze unfurled,
here once embattled farmers stood,
and fired the shot heard round the world.
```

(ending my input with control-D) and I get back:

```
april
```
ard
embattled
farmers
fired
unfurled

Two of these lines are really errors on my part. For april I should have typed April, and for ard’d I should have typed arc’d. The other problems are all due to words which aren't in the /usr/dict/words list. Of course, we could fix this by expanding the list. But a different approach would be to try to remove common prefixes and suffixes for English words, and this approach would permit us to use a shorter list.

/usr/dict/words happens to have embattle, farm, and furl in it, but the compiler of the list thought we ought to be able to figure out that words ending in s are probably plurals, words ending in ed are probably the past tense of verbs, and words ending in er are nouns formed from verbs. Words beginning in un, re or in also have shorter roots.

If you were to write a short program to sit in the middle of the pipeline to apply the following modifications to words, we could make my example work:

<table>
<thead>
<tr>
<th>modification</th>
<th>sample before</th>
<th>sample after</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remove un-</td>
<td>unfurled</td>
<td>furled</td>
</tr>
<tr>
<td>Remove re-</td>
<td>remove</td>
<td>move</td>
</tr>
<tr>
<td>Remove -s</td>
<td>farmers</td>
<td>farmer</td>
</tr>
<tr>
<td>Remove -er</td>
<td>farmer</td>
<td>farm</td>
</tr>
<tr>
<td>Change -tled to -tle</td>
<td>battled</td>
<td>battle</td>
</tr>
<tr>
<td>Remove -ed</td>
<td>furled</td>
<td>furl</td>
</tr>
</tbody>
</table>
For other examples, we'd need more rules, but it seems plausible that we might be able to get by with a relatively small number, at the risk of missing a few misspellings.

1. Write a small program which applies the handful of rules for reducing English words I've given. It will take input lines with a single word, and output each root. Then include your program in the spellcheck pipeline, and try it on a handful of examples. Do you have any suggestions for more simplification rules?

  [If you're working on a non-UNIX system, like VMS or PC-DOS, you may not have the same utility programs available. That's okay; go ahead and write the program anyway, you'll be coding up substitutes in the following problem.]

If you do not have pipes available, the last exercise can easily be done with files; in fact, the PCDOS COMMAND.COM provides an interface which looks just like the UNIX shell interface for pipes, but stores the output from each stage in a file. However, the file approach gives up a chance at parallelism. Another way of obtaining the parallelism is to rewrite each of the shell commands of the UNIX script as a separate thread, and have each thread act as consumer for the thread before it, and producer for the thread after it.

2. Rewrite the spell-checker so that each of the processes (tr, sort, comm) corresponds to a single thread. (Although you could actually fork() the programs to do the work, they do so little that I suggest just coding up the C yourself.) Each thread except the first will get its input from a buffer provided by the previous one. You'll need to make sure
that buffers don't overflow, that all the output is eventually used up, etc.

When you've finished with this section, you'll have coded:
+ A routine to run as a thread which reads a file character by character, converting all non-alphabetic characters to newline characters, and squeezing all sequences of two or more newlines into a single one. The output of this routine should be handed off with place-stuff(). When it reaches end of file, it should hand off a buffer containing the string `'`.`'
+ A pair of message passing routines which can be used by threads to communicate:
  /* called by producer thread. May wait until buffer empty */
  void place_stuff(struct buffer_descriptor, char* new_stuff, int new_length);
  /* called by consumer thread. May wait until buffer filled.
  * Returns size of item transferred to local buffer*/
  int get_stuff(struct buffer_descriptor,char* local_buffer; int local_buffer_max);
+ A thread version of your word-transformation routine, which uses get-stuff() and place-stuff()
+ A sort thread which accepts its input using get-stuff(), and hands off its output using place-stuff(). Because it's difficult to know what the smallest item will mean until all items have been received, the sort thread may not produce any output until it gets the final `'`.`'`; however, it is allowed to decide that it has no room to store more data, and output a `'`,`'`; only runs of data between successive commas or period is guaranteed to be in order.
+ A match thread, which accepts input using get-stuff() and reads through a sorted word file, comparing its input to the words in the file, and
printing on the terminal whenever it is passed a word not in the file. Whenever it is handed a `\`," it may have to reopen the word file and start to read from the beginning.

Although you could get elaborate on the various components of your pipeline, I'd rather you get carried away, if at all, on adding bells and whistles to the handoff routines. Include a text description of the design of your routines, and what problems various whistles address.
Remote Procedure Call- RPC --is the name of a family of fancy wrappers for message passing.

One widely used fancy wrapper is the SUN RPC package, which SUN has released to the public domain, and has been ported to many platforms, including the SUN and DEC workstations available to you at WPI. I am handing out a description of this package, and I want you to use it for the following otherwise trivial client-server application.

You'll write a client program called TellNick and a server program called Nick. The server program is assumed to be running when the client starts; the client is run with an argument which tells it which system the server is running on, just like the example in the handout. The client takes command from the console, and uses RPC to communicate with Nick.

The client commands are:

---

**Tell string**    Passes a string to Nick to remember. Nick returns a number, which the client program prints out.

**ask number**    Passes a number to Nick to get the corresponding string. The client then prints it out.

**huh?**    Requests a range of valid numbers from Nick, and prints them out.

**exit**    Causes the client program to shut down.
---

Here's a sample run of the client program:

```bash
% TellNick banzai
>Huh?
Connected to Nick on banzai
```
Strings 1 - 53 are available.

> Tell There's a hawk in the pine in the park.
That's message 55 on banzai

> Ask 51
Message 51 from banzai:
Fire! Fire! Fire! Everybody out of the building!

> Exit
Goodbye
%

The server program keeps track of what it's been told in memory. An array should do the trick nicely; you can place small fixed limits on number of characters in a string and number of strings if you like. It should print enough verbiage on the console that it is easy to tell if it is working.

Here's a sample run of the server program:

% nick
Got a new message, number 0:
By the rude bridge that arc'd the flood,

Got a new message, number 1:
Their flag to April's breeze unfurled,

Got a new message, number 2:
Here once embattled farmers stood,
Got a new message, number 3:
I think that I shall never see,

Got a request for message count (3).
Got a request for message number 3.
Got a new message, number 4:
And fired the shot heard round the world.

^C
%

I'll want listings of sources for the server and client programs and for the protocol file (the .x file) used for input to rpcgen. Sample runs would be nice, too.
For this week, you'll do a small simulation to compare two scheduling algorithms. Simulation is a serious business, with lots of pitfalls, but we'll just rush in, and take our chances. The two algorithms will be

1. Round-Robin, with a time-slice of 3 ticks.
2. Round-Robin, with a time-slice of 10 ticks.
3. Priority, with higher priority arriving jobs preempts a lower priority running job.

(I know that looks like three)

Simulate the algorithms against the same work loads, and compare average response times and cpu usage.

For this simulation, we'll assume that events happen only at `ticks;' you don't need to simulate anything that happens between ticks, but at each tick,

+ The currently executing job, if any, completes another tick of work.
+ The currently executing job may finish, in which case we should update response time statistics.
+ A new job may arrive. Every arriving job has an expected duration and a priority, which will need to be stored until it finishes.
+ Depending on the scheduling algorithm, which job is the currently executing job may change.

In the ideal project the distribution of arrival rates, job durations, and priorities would be based on a measurement of a realistic workload, but I'm just making up the following numbers. My goal is that (average number of
arriving jobs each tick) x (average length of a job in ticks) < 1 Otherwise, there'd be too much work to ever finish, and the response time would grow steadily as the system ran.

On the other hand, unless the system is at least a little loaded, there's lots of free time, jobs never interfere with each other, and all scheduling algorithms are equivalent.

I suggest that

+ the probability of a job arriving on any tick should be .17
+ the average runtime of a job should be 5.5. One way to get this average is to choose a runtime from the set \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} with equal probability.
+ the priority is also a randomly chosen number from 1 to 10, but different from the runtime.

I don't think that the quality of the random number generator will have much effect on your simulation, but you may want to compare the documentation for rand(), srand(), random() etc.

Your simulation should run long enough to settle into a steady state if it's going to, or exhibit runaway behavior if I've messed up on the parameters. I'm not sure how long that is, but I'd think anything less than 10000 ticks is too short. One way to judge this might be to graph the response time against the number of ticks simulated when the measurement is made, and see if there's a trend.

Serious simulators regard each simulation as a measurement, and try to get enough measurements (by running the simulation again with different data) to do some statistics on their results.

If you run each simulation several times, you should be able to report confidence ranges for average response time for each scheduling algorithm.
This week I am asking you to write "solutions" to some classic process synchronization problems using threads. In order to do so, you will need a thread library. DECthreads, available on VMS, OSF, and Ultrix systems, seems like a good choice if you have access to it.

The Lightweight process facility available on Sun systems is another possibility.

Another possibility would be Posix 1003.4a threads, if you have a system which provides them. A version of Posix threads for several different systems, including alpha-osf1, hpux, irix, linux, sunos, and ultrix, is available by anonymous ftp from the Internet, from (among other places:)

Host bloom-picayune.mit.edu

Location: /pub
DIRECTORY drwxrwxr-x 512 Dec 21 22:22 pthreads

If you cannot use threads, you will probably be able to use processes, communicating with shared memory, semaphores, or files.

The comments should explain what the programs are doing in adequate detail. There should be sufficient scaffolding and output in each program so that it can be run and tested.

Do two of the following:

*The Dining Philosophers.* There are N philosophers seated at a circular table, sharing chopsticks, so that no two adjacent philosophers may eat at the same time. Each philosopher is modeled by a process executing the following code:
void philosopher(int i)
{
    while (1){
        think();
        get_hungry_and_try_to_eat(i);
    }
}

You will write the routine get_hungry_and_try_to_eat(). Your routine must obtain a chopstick from the left and right side. Once it has both chopsticks, it should eat() and then release them. Because the philosophers are concurrent threads/processes, any access they make to common data must be appropriately protected. You should avoid busy-waiting, if possible.

**Producer/Consumer.**

There is one Producer process and one Consumer process. The producer is manufacturing numbers which are passed to the consumer in one of two static buffers (which are not shown.)

```c
void producer()
{
    int i;
    while (1){
        i = produce();
        put(i);
    }
}
```
void consumer()
{
    int i;
    while (1){
        i = get();
        consume(i);
    }
}

Your solution will consist of put() and get() routines, which will use two different words as buffers, so that the producer can get ahead of the consumer.

You may use the following produce() and consume() routines:

int production_global;

int produce()
{
    return production_global++;
}

void consume(int i)
{
    printf("%i", i);
}
Readers and Writers.

There are N reader processes and M writer processes. Reader processes are executing:

```c
void reader(int i)
{
    while (1){
        read_db();
        compute();
    }
}
```

Writer processes are similar. Any number of readers may be reading the database at once, but only one process may be writing, and then only if there are no processes reading. Your solution will consist of subroutines for `read-db()` and `write-db()` and enough scaffolding to run and test them.