As of now, this course as an experiment is "at risk":
   My initial policies don't properly support the outcomes I desire.
   Evidence: the midterm.
   The course has been used for unforeseen purposes.
   Evidence: people wishing to use the course to apply to the MS program.
   In practice, the structure I proposed isn't sustainable.
   Evidence: too much grading to do, no help available.

Therefore, for the \textit{survival of the course as a concept}, it is necessary to rethink the course to address these needs:
   Need: skill building
      Problem: assignments too abstract.
      Solution: make assignments concrete.
   Need: certification of individual skills
      Problem: group assignments do not serve.
      Solution: individual assignments.
   Need: sustainability
      Problem: essays are difficult to grade.
Solution: exercises in practice are easier to complete and to grade.
Ensuring program performance

The practice of making programs execute "fast enough" (and in appropriately small memory).

Several parts:
Factoring performance into causal influences:
Measuring the effect of each component.
Exploiting performance patterns to address weaknesses.
Some important language from Service Oriented Architecture

A Service Level Objective (SLO): defines what is "fast enough".
A Service Level Agreement (SLA): defines economic penalties for not meeting SLOs.
An appropriate SLO:
"Web service requests should respond in less than 1 second of real time.

A related SLA:
"If less than 90% of requests meet the SLO in a month, the customer doesn't pay anything for the service."
Statistics show that if a travelocity search takes more than 5 seconds, the sale is lost.
The point of SLOs and SLAs
Performance isn't free.
But neither is lack of performance.
Often, performance engineering is a matter of balancing the cost of performance against its value.
Performance engineering

**Factoring:** what factors contribute to performance

**Performance measurement:** how we document how programs perform

**Performance analysis:** how we determine the performance problems for software.

**Performance patterns:** how we address common performance problems.
Factoring performance issues.
  Speed of processing.
  Speed of memory access.
  Size of caches.
  Device latency.
  Network latency.
The concept of a bottleneck

Performance problems are usually expressed in terms of bottlenecks. A **bottleneck** is a specific performance problem that -- if addressed -- most effectively speeds up total execution. You might also hear a bottleneck defined through specifying the "critical resource".
Kinds of bottlenecks:

**CPU-bound:** the critical resource is the speed of the processing unit.

**Cache-bound:** the critical resource is the size of the CPU cache.

**Memory-bound:** the critical resource is the speed of the (front-side) memory bus.

**Disk-bound:** the critical resource is the speed of (local) disk access.

**Network-bound:** the critical resource is the speed of the connected network/internet.

**Service-bound:** the critical resource is the speed of response of a remote service.
The notion of time

**Execution time:** the time spent executing in a core or CPU.

**User time:** time spent executing your program.

**System time:** time spent in the operating system on your behalf.

**Wallclock time:** the elapsed time according to an external observer.

**User time + System time = Execution time < Wallclock time** (due to external influences).
Performance measurement:

- **times** command: tells how long a program ran (in cpu time and real time).
- **getrusage** system call: indicates time used, memory used.
- **profiling**: gives detailed resource consumption of functions.
Compile a special statistics library into the code.
Keep statistics on CPU consumption in each subroutine.

gcc -pg file.c
./a.out # store results
gprof a.out # report results
The simple cases

gprof often points out obvious performance problems

Case 1: simple subroutine, high runtime
 => inline the function to avoid stack overhead.

Case 2: complex subroutine, high runtime
 => unroll loops, declare register variables, analyze memory access, or consider redesign.
Case study: loop unrolling

A common form of performance optimization: loop unrolling:

Change:

for (i=0; i<10; i++) foo(i);

To:

foo(0); foo(1); foo(2); foo(3); foo(4);
foo(5); foo(6); foo(7); foo(8); foo(0);

Why?
The actual mechanics of the loop slow down execution (i<10, i++)
Eliminating "if" statements

In truly performance-critical code, it is often advantageous to eliminate if statements and replace them with indirections.

Example:

if (something(x)) foo(x);

Becomes

foo(redirect(x))

Where

redirect(x) has two values:

x if something(x);

a random other place if ! something(x);

The trick is to do this without invoking the "if" statement at all, e.g., by making redirect(x) involve only arithmetic.

For example, in a profiler, it is cheaper to measure everything (and then throw some statistics away) than to measure selectively, because the "if" statement slows down the code.
Example of if statement elimination

```c
if (x=='a') a_count++;  
else if (x=='q') q_count++;  
else if (x=='w') w_count++;  

int counts[26];  
#define a_count counts['a'-a'];  
#define q_count counts['q'-a'];  
#define w_count counts['w'-a];  

Then the code becomes  
++counts[x-'a'];
```
The complex cases

Often gprof fails to provide anything useful
- If runtime is not solely dependent upon program design.
- If external factors contribute.
Performance measurement problems and solutions:

- Problem: clock resolution is 1/100 second. Solution: scaffold test, repeat test 1000 times, divide by 1000.
- Problem: performance measurement of inaccessible subsystems (disk, network, ...).
  Solution: **Wallclock time - Execution time = Resource latency + Concurrency latency**
  
  Resource latency: time spent waiting for resources.
  Concurrency latency: time spent waiting for concurrent (unrelated) tasks to finish.

Typical approach: isolate test on quiescent system, then concurrency latency is 0.
Performance analysis

Briefly stated, the activity of determining "why" programs run slowly.
In broader terms,
   Inferring unmeasurable behaviors.
   Hypothesis testing.
   (some sneaky tricks help)
Some (very) sneaky tricks of performance analysis

Cache ramps.
Denial of service testing.
Example: determining whether a problem is cache-bound.

Note: this only works well in an otherwise quiescent environment.
Why do cache ramps occur?

Cache is a limited resource. Actual program runs in cache at GHZ. Memory is transferred to and from cache (automagically) at MHZ. If your program "fits into cache", then the whole thing runs at GHZ. If it doesn't, there is cache thrashing: pages have to be moved in and out at MHZ.
Example: determining whether a program is resource bound

Two approaches:
- Vary the resource and do sensitivity analysis (if you can vary it).
- Synthetically slow down resource availability and do sensitivity analysis (e.g., for network requests).
Otherwise, vary the availability of the resource through controlled, \textbf{competitive} processes.
Example: competitive processes

Item under test: an intensive calculation.
Question: is the calculation disk-bound or CPU bound?
Strategy: schedule another known disk-bound problem (e.g., writing a huge file full of 0xDEADBEEF patterns to the same disk).
Inference method: plot severity of interference versus speed for a single run.

Example: a real case study
A website in Bosnia wanted to restructure its services with one new machine. Should the machine be a web server or a database server?
Analysis: competing with the database server creates no performance problem: put in a second web server.
Performance patterns

Memory access tuning
Latency hiding
Caching and lookaside buffering
Horizontal scaling
Vertical scaling

Often a simple problem, combined with a performance bound, leads to a complex solution.

Almost all of these invoke some kind of time/resource tradeoff: to decrease time, increase space or some other resource (e.g., cores, computers, etc)
Memory access tuning

Simply stated, restructuring memory access so that in sections of code, all memory fits inside processor cache.

Conceptually,

```c
for (i=0; i<BIG; i++) do something with a[i];
for (i=0; i<BIG; i++) do something else with a[i];
for (i=0; i<BIG; i++) do a third thing with a[i];
```

becomes

```c
for (j=0; j<BIG; j+=SMALL) {
    for (i=j; i<j+SMALL; i++) {
        do something with a[i];
        do something else with a[i];
        do a third thing with a[i];
    }
}
```
Latency hiding

Problem: latency in accessing multiple external resources.
Solution: threaded programming to hide latencies.

Initial condition
for (i=0; i<SIZE; i++) get a[i];

Threaded version
for (i=0; i<SIZE; i++) {
    begin thread;
    get a[i];
    end thread;
}
Caching

Problem: overload of a resource, e.g., a web service.
Observation: many queries are for the same thing.
Solution: cache reusable results for a given time, avoid queries if possible.

The problem:
get something;
return something;

The solution:
if (cached(something)) and age_in_cache(something) < timeout)
    return something;
else
    get something;
    put_in_cache something;
    return something;
Lookasides
A special case of caching.
Problem: a complex chain of calculations involving several steps, where conclusions are not time-varying.
Solution: store results in a half-hearted cache against opportunistic reuse.
The problem:
\[
x = \text{something}(a) \\
y = \text{something}_\text{else}(x) \\
z = \text{something}_\text{further}(y) \\
\text{return } z;
\]
Becomes:
\[
\text{if (lookaside}(a) \text{ exists)} \\
\quad \text{return lookaside}(a) \\
\text{else} \\
\quad x = \text{something}(a); \\
\quad y = \text{something}_\text{else}(x); \\
\quad z = \text{something}_\text{further}(y); \\
\quad \text{store}_\text{in}_\text{lookaside}(a,z); \\
\quad \text{return } z;
\]
## Differences between lookasides and caching

<table>
<thead>
<tr>
<th>Caching</th>
<th>Lookasides</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time-varying results</strong></td>
<td>Results are a deterministic function of input</td>
</tr>
<tr>
<td><strong>Timeout determines when to refresh results</strong></td>
<td>Overwrite transparently ages out old entries.</td>
</tr>
<tr>
<td><strong>Storage is deterministic</strong></td>
<td>Storage is non-deterministic; utilizes an imperfect hash of inputs with no collision resolution</td>
</tr>
<tr>
<td><strong>Copes with known patterns of reuse</strong></td>
<td>Copes with flash crowds and other unanticipated forms of reuse.</td>
</tr>
<tr>
<td><strong>Average case constant time</strong></td>
<td>Real O(1), no collision strategy</td>
</tr>
</tbody>
</table>
Scaling

Horizontal scaling: applying more cores and/or independent computers to the computation of the answer.

Vertical scaling: restructuring the computation to run on a network of computers rather than a single computer.
Example of horizontal scaling

Problem: database queries are taking too long for a constant database.
Solution: implement more database servers, redirect requests to others in round-robin style.
Example of vertical scaling

Problem: a complex eigenvalue computation takes too long on a single machine.

Solution: implement a parallel computer consisting of a cluster of similar computers, and utilize the parallel libraries LINPACK and EISPACK to parallelize the computation.

You're limited to what is known, or you have to implement your own (a whole course).