We have taken a rather long journey
From the physical hardware,
To the code that manages it,
To the optimal structure of that code,
To models that describe the code,
To predictive techniques for behavior.
Elements of the story

The realities of running in constant time.
How to maintain the illusion of autonomy for processes.
How and why actions are made atomic in a concurrent environment.

Algorithms for managing concurrency, conflict, and complexity.
How the complex is made simple via knowledge boundaries and producer/consumer relationships.
Design tradeoffs between efficiency and responsiveness.
Conditions to avoid and how to avoid them.
Some design tradeoffs

- **throughput** versus **response**.
- **hashing** versus **caching**.
  - (dense versus sparse structures).
- **processes** versus **threads**.
- **semaphores** versus **mutexes**.
  - (kernel objects versus user objects).
- **states** versus **events**.
  - (regular versus transactional filesystems).
Some important principles

**Knowledge boundaries:** it's easier to write modules if one limits what knowledge they collect and/or interpret.

**Locality:** if next accesses are near prior accesses, then caching is effective in speeding up access.

**Proof:** it is not possible to debug concurrent code. It is necessary to **prove** it correct.

**Contracts:** some multi-level subsystems work properly only because the lower-level system caches content for the upper-level one.
Knowledge boundaries

The disk scheduler doesn't know the meaning of blocks.
The virtual memory subsystem doesn't know the meaning of pages.
The raw disk driver doesn't know where the superblocks are located.
The filesystem driver doesn't know how to write a block.
And if they did, they would be too complex to write.
Locality

Accesses to memory/disk tend to be close together, so caching speeds up access.

- physical memory is a cache for virtual memory.
- page cache is a cache for disk files.

The reason bit arrays "work" for keeping track of what's used in a filesystem is that they tend to stay in the LRU cache!
I hope the examples I gave show that locking, deadlock, and livelock are subtle concepts.

It is not enough to debug a concurrent program; this would not detect subtle race conditions.

One must prove that a process is deadlock-free.

=> use of fully debugged concurrent abstractions is highly recommended (e.g., bounded buffer queues)
Some subsystems only function properly because of the behavior of others. Example: the disk page cache makes EXT2 and EXT3 filesystems practical.
Some conditions to avoid

**Deadlock**: no progress is made, because everyone is blocked.

**Livelock**: no progress is made, even though no one is blocked.

**Thrashing**: the operating system overhead exceeds the useful computation.

**Starvation**: a process or processes do not have the resources and/or cycles to make progress.
Some algorithms to know

Round robin scheduling.
Batch scheduling.
(know about O(1) and completely fair scheduling)
Lock priority (anti-deadlock) algorithm.
Banker's algorithm.
Buddy system.
Least-recently used and least-frequently accessed.
Sparse structures (e.g. memory pages) are represented by hashing (inverted tables).
Dense structures (e.g., driver tables) are represented by arrays and caches.
Translation lookaside buffering: a way to speed up use of multiple-level hashes.
Scheduling

There is no such thing as perfect scheduling.
Fairness versus throughput.
Queueing models and predictions.

Little's law: predicts performance based upon steady state.
M/M/1 queues: predicts performance based upon statistical assumptions on input and processing.
Security

The concept of identity.
Users and groups.
Files and protections.
For what's on the final....

Look at the preceding slides.

Emphases:
- design tradeoffs.
- I/O subsystem.
- Filesystems.

It is impossible not to touch upon prior concepts.

Kinds of questions:
- Design tradeoffs: compare options, suggest the best one.
- Algorithms: demonstrate an algorithm and predict its results.
- General literacy: understand how the subsystems of an operating system interact.
- Expert question: answer questions on a new feature that doesn't exist in any operating system. Is it possible, is it a good idea?
This is not the end...
It is not the beginning of the end...
It is the end of the beginning...

(John Varley, "Millenium")
What next?

Networking (comp112)
Databases (comp115)
Security (comp116)
Distributed systems (comp150IDS)
Cloud computing
Not a stale discipline:

Operating systems are a very active research area. Advances are made each year. We haven't talked much about present-day innovations.
How to keep up with what's hot:

- USENIX Annual Technical Conference.
- USENIX HotOS: pre-publication results.
- Operating Systems Design and Implementation (OSDI).
- Network Systems Design and Implementation (NSDI).
- File and Storage Technologies (FAST).
What's hot:

**Virtualization**: running multiple operating systems on one physical host.

**Proactive bug avoidance**: making the operating system detect and avoid known software bugs.

**New privilege models**: the "user" is obsolete.

**High-performance storage**: rethinking the disk drive.

**Provenance-aware filesystems**: store how the file was created, as well as its contents.

**Power-aware filesystems and operating systems**: server power is the dominant cost in providing services.

**Self-management and self-organization**: can operating systems take care of themselves?
Virtualization

Allow two operating system instances to run on the same physical host.
Use an overall scheduler to manage instances.

Implementing virtualization:
A mini-OS (the "hypervisor") manages which OS runs and has its own scheduler.
The OSs talk to the hypervisor instead of devices.

Reasons for virtualization:
Configuration conflicts can be avoided.
Can utilize standard configurations without change.
Leading to higher reliability.
Proactive bug avoidance

We know that deadlocks in software are due to bugs. What if we could detect the bugs and avoid them?
A new approach: deadlock avoidance through schedule avoidance.

We know web applications are full of bugs. Deadlocks occur because affected threads execute according to a particular schedule. Detect the schedule, and prevent it from occurring via artificial delays!
Rethinking privilege

The old ideas of user/group/other protection are outmoded.
Especially in the cloud!

Example: the android phone

Runs linux!
Completely ignores linux protections!
Has its own idea of protection based upon cloud storage.
Your google account determines your privileges.
linux accounts protect applications, not users.
Often, it's not enough to just store a file. We want to know how it was created. This is called the "provenance" of the file.

New work: provenance-aware filesystems. Store actions taken to create the file. Transparently, without user intervention.
In Google AppEngine,
   There are no "files".
   There are distributed objects with multiple instances.
To store an instance of an object, one throws it into the cloud.
To reconstruct something like a file, one makes a query into the cloud to return matching instances.
So, your so-called "serial program" utilizes 10,000 or more machines to accomplish its result.
In google mail
   your messages are spread all over google's infrastructure.
   your "inbox" is a polite illusion: it's a query.
We face a future in which computers and humans are equal partners.

- Both are more complex than one person can understand.
- Both are subject to motivations and develop "personality".
Some current trends:

Multi-core applications and use of GPUs as extra "cores".
Ubiquitous interactions between tiny embedded computers.
The social power of scale.
Operating systems and social pressure

You might consider operating systems to be dramatically non-social tools.
In fact, they're very much controlled by social factors:
what people use is
where bugs get discovered, which is
what implementors debug.
There is huge social pressure to fix some things, and an equal lack of pressure to fix other things.
What have we learned?
That it is possible to build complex systems from very simple building blocks.
That subsystems can cooperate toward a common mission with only partial knowledge of the mission. That knowledge boundaries don't just make a problem easier; they make it possible to solve.
The job of programmer is obsolete. But the systems remain, and are more complex than ever. The big job is to be a lens between the users and the increasingly complex systems.
Operating systems are a metaphor for life

One often *doesn't know* precisely what's going on. One must at best use *partial knowledge* to make decisions.

One is constantly faced with systems that *no one person can completely understand.*

*Small changes have big effects.*

Nonetheless, things get done!
So, in your own personal operating system:

Don't deadlock.
Don't livelock.
Don't thrash.