Welcome and Introduction

Lecture 1

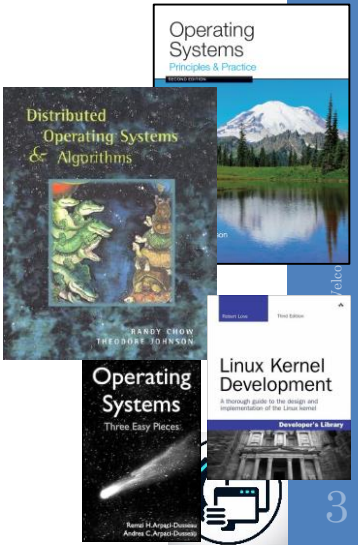
Hartmut Kaiser

https://teaching.hkaiser.org/fall2026/csc7103/

Administrativia

Infrastructure, Textbook & Readings

- Infrastructure
  - Website: https://teaching.hkaiser.org
  - Discord: https://discord.gg/yN3gGfwpfE
- Textbooks:
  - Distributed Operating Systems & Algorithms, Chow and Johnson
    - Not required, get a copy if you feel that lectures are not sufficient
    - Suggested readings posted along with lectures
    - Try to keep up with material in book as well as lectures
- Supplementary Material
  - Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Operating Systems: Three Easy Pieces, by Remzi and Andrea Arpaci-Dusseau, available for free online
  - Linux Kernel Development, 3rd edition, by Robert Love
- Homework, project, quizzes, grading, honesty
  - Use Github/Classroom50/Moodle for assignments
  - Use VSCode as a tool



Class Expectations

- Lectures
  - Come! Cannot guarantee content will be identical to previous years
  - Will explain many things related to assignments not otherwise explained
  - If you decide to come, please be on time
  - Meet your fellow students, they are your future colleagues!
  - Electronic devices used only for note taking
  - Attendance not mandatory but highly advised
    - I may take attendance (especially for student presentations)
- Office Hours
  - Come and ask for help early. There are no stupid questions!
  - We like teaching and want to meet you!
- Communicate with course staff through Discord and Email.

This Course

- This is a revised curriculum with new goals:
  - Understanding and exploiting Distributed OS and services
  - Foundational concepts and principles
  - Common problems that have been solved in distributed OS
  - Evolving directions in system architecture
- Overarching idea:
  - How to build (software) systems that don't crumble under their own weight

Important Dates and Details

- Lectures
  - Tuesday and Thursday, 1:30pm to 2:50pm, 1236 PFT
- Grading
  - Assignments 20%
  - Projects 50%
  - Reading assignments 20%, presentations 10%
- Exams
  - There might be take-home examinations

Class Structure

- Tuesdays will be lecture days
  - Slides will be posted on web site
- On Thursdays you present the papers you have read
  - Weekly reading assignments are due at that lecture
  - Will take attendance for these
  - One-question quiz about the papers
- Discussion notes may also be posted
- Will sometimes cite papers
  - Look up with search engine or ask us for more info
- If you have questions on papers you read, email us by 9am on day of lecture
  - I will try to address your questions in discussion
  - Good questions may earn participation points

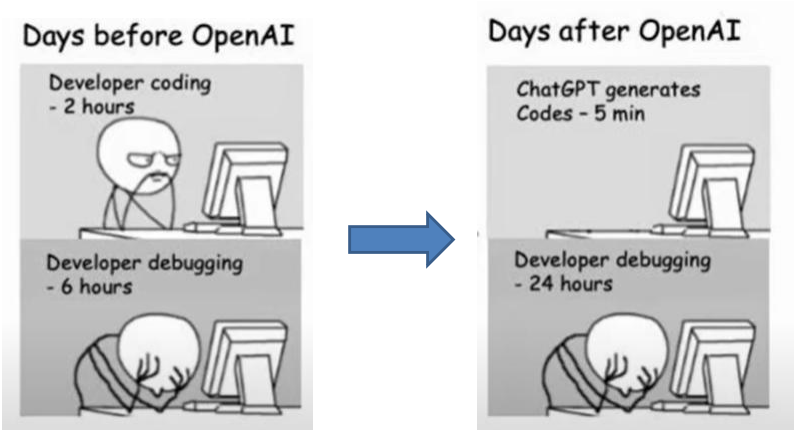
Class Structure

- Lectures
  - Discuss advanced topics
  - Every week will have mandatory readings due on Thursday
    - You are expected to have read those, submit written review for two of the papers
    - Discuss readings, ask questions
- Reading assignments
  - Everyone reads the papers and submits (hardcopy) review for two of the papers every week
  - Up to three students will present one of the papers each during Thursday's lecture time
  - Discussions will earn extra points
  - One-question quiz about the papers you have read
- Assignments
  - Programming against OS APIs
  - Assignment0 is simple 'readiness test' for you, no submission, one question quiz on Sep 1
  - Assignment1 is asking you to write a simple Linux command line tool
- Projects
  - PintOS, a toy OS
  - Implement new features: threading, scheduling, memory management

Honesty

- The LSU *Code of Student Conduct* defines plagiarism in Section 5.1.16:
  - "Plagiarism is defined as the unacknowledged inclusion of someone else's words, structure, ideas, or data. When a student submits work as his/her own that includes the words, structure, ideas, or data of others, the source of this information must be acknowledged through complete, accurate, and specific references, and, if verbatim statements are included, through quotation marks as well. Failure to identify any source (including interviews, surveys, etc.), published in any medium (including on the internet) or unpublished, from which words, structure, ideas, or data have been taken, constitutes plagiarism;"
- Plagiarism will not be tolerated and will be dealt with in accordance with and as outlined by the LSU Code of Student Conduct: https://www.lsu.edu/saa/students/codeofconduct.php

ChatGPT (et.al.)?



ChatGPT, etc.

- The goal for this course is to train our brains, not ChatGPT
  - So, do yourself a favor and don't use it for code generation (asking for recommendations or explanations is fine)
  - There are too many software 'developers' out there who copy & paste their way to the next paycheck
- However, if you do use it:
  - Never use anything without carefully reviewing it
    - Assume what you got is wrong! Prove to yourself it is correct!
  - The skill of reading (and understanding) code becomes more important than ever
  - Cite and annotate your information sources and what you took from there
- Whatever you do, remember:
  - You are responsible for whatever you submit
  - It's plagiarism if you submit the same code as your neighbor
  - No matter where you got it from, be it Google, ChatGPT, or your neighbors computer

Introduction to Operating Systems

Introduction to Operating Systems

- Why study Operating Systems Research?
  - Why care about OS/systems research?
  - What are the fundamental problems?
- Not really just about building OSes
  - Any large code base converges on becoming an OS
  - Manages memory, programming for space/performance, abstract hardware details
  - Database, JVM, browser, parallel/distributed systems, internet services
- How to structure systems and deal with complexity
  - Modularize and encapsulate
  - Choose interfaces/abstractions carefully
- Themes in this class
  - Abstractions for managing/accessing resources: storage, replication, naming
  - Correctness: concurrency and sharing
  - Guarantees in real systems: security, fault tolerance, etc.

Prerequisites & Goal

- We assume some familiarity with C
  - There will be a lecture about 'Refreshing C', though
- Some familiarity with Unix may help
- An undergraduate "textbook" OS class (e.g., CSC4103)
  - Familiar with concepts like Virtual Memory, processes, etc.
  - Enough to read papers that employ these concepts
  - Assignment 0 will give you a chance to catch up
- Goals of class:
  - Get to the point of being able to read and understand contemporary OS papers
  - Provide background if you are interested in doing (distributed) OS research

Course Topics

- Distributed aspects of:
  - Concurrency & synchronization
  - Scheduling
  - Virtual memory
  - Virtual machines
  - File systems & storage
  - Network stack implementation
  - Storage systems
  - Kernel architectures
  - OS Security and reliability

OS Research as a fundamental Discipline of Computer Science

- What is Computer Science?
  - Computer Science is the science of managing complexity
  - Modularization and Encapsulation
  - Building abstractions
- What is OS Research?

Advanced Operating Systems

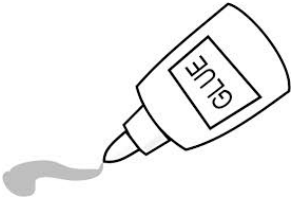
Why study Operating Systems?

- Operating systems are a maturing field
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- High-performance of servers are an OS issue
  - Face many of the same issues as OSes
- Resource consumption is an OS issue
  - Battery life, radio spectrum, etc.
- Security is an OS issue
  - Hard to achieve security without a solid foundation
- New “smart” devices need new OSes

What is an Operating System?

- Makes hardware useful to the programmer
- Provides abstractions for applications
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- Provides protection
  - Prevents one process/user from clobbering another

Three Main Hats



Referee
Manage protection, isolation, and sharing of resources

Illusionist
Provide clean, easy-to-use abstractions of physical resources

Glue
Provides a set of common services

Referee:
What does this Program do?

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  char *str = argv[1];
  while (1) {
     printf("%s\n", str);
  }
  return 0;
}
```

```
laptop> gcc –o cpu cpu.c -Wall

laptop> ./cpu A
A
A
A
A
…

laptop> ./cpu A & ./cpu B & ./cpu C
```

a)
```
A
A
A
A
…
```

b)
```
A
B
C
A
B
C
…
```

c)
```
A
B
B
A
C
B
C
…
```

```
laptop> ./cpu & ./cpu B
Segmentation Fault
B
…
```

Illusionist: What does this Program do?

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  int *p = malloc(sizeof(int));
  printf("(%d)  p: %p\n", getpid(), p);
  *p = 0;
  while (1) {
    *p = *p + 1;
    printf("(%d) *p: %d\n", getpid(), *p);
  }
  return 0;
}
```

a)

```
(120) *p: 1
(120) *p: 2
(254) *p: 1
(254) *p: 2
(254) *p: 3
(120) *p: 3
…
```

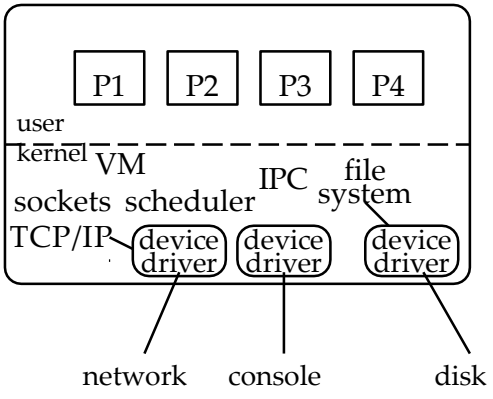
```
laptop> gcc -o memory memory.c -Wall
laptop> ./memory
(120)  p: 0x200000
(120) *p: 1
(120) *p: 2
(120) *p: 3
(120) *p: 4

laptop> ./memory & ./memory
(120)  p: 0x200000
(254)  p: 0x200000
```

b)

```
(120) *p: 1
(254) *p: 1
(120) *p: 2
(254) *p: 2
(120) *p: 3
(254) *p: 3
…
```

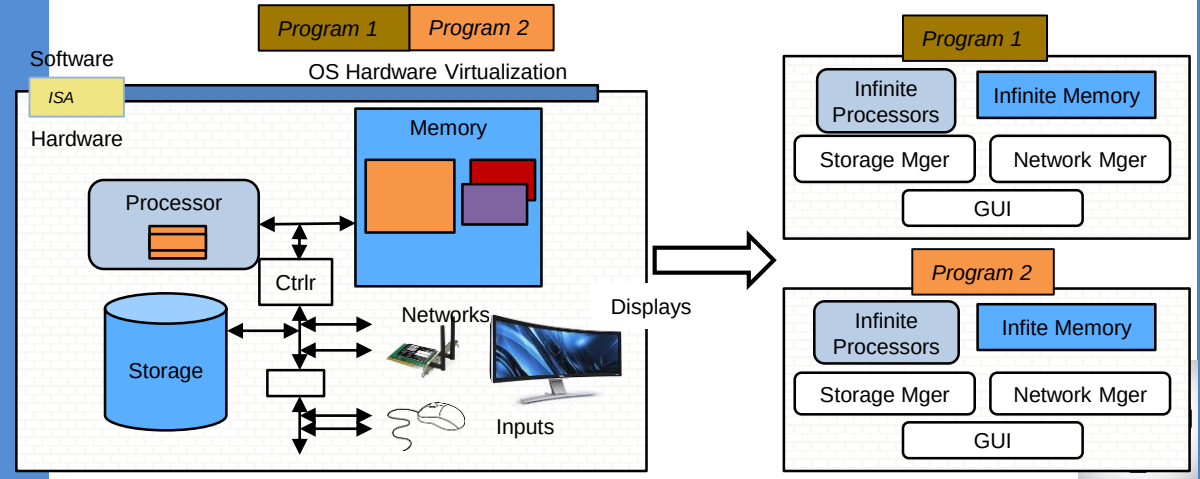
Glue: Typical OS structure

- Most software runs as user-level processes
- OS kernel handles "privileged" operations
  - Creating/deleting processes
  - Access to hardware



Putting it all together

- Referee + Illusionist + Glue => Easy to use virtual machine



*nix Architecture

- User-level
- Kernel “top half”
  - System call, page fault handler, kernel-only process, etc.
- Interrupts
  - Software interrupt
  - Device interrupt
  - Timer interrupt
- Context switch code

Transitions between Contexts

- User → kernel: syscall, page fault
- User/kernel → device/timer interrupt: hardware
- Kernel → user/context switch: return
- Kernel → context switch: sleep
- Context switch → user/kernel

System Calls

- Goal: invoke kernel from user-level code
  - Like a library call, but into more privileged OS code
- Applications request operations from kernel
- Kernel supplies well-defined system call interface
  - Applications set up syscall arguments and trap to kernel
  - Kernel performs operation and returns result
- Higher-level functions built on syscall interface
  - `printf`, `scanf`, `gets`, etc. all user-level code

Example: POSIX/Unix interface

- Applications "open" files/devices by name
  - I/O happens through open files

```
int open(char *path, int flags, ...);
```

- Returns file descriptor—used for all I/O to file

```
int read (int fd, void *buf, int nbytes);

int write (int fd, void *buf, int nbytes);

int close (int fd);
```

System Interfaces

- System call interface is interface to kernel
- Historically also interface “most programmers” need
  - But most programmers != all programmers
  - Standard OSes support many types of application
- No inherent reason for two interfaces to be the same
  - Exokernel possibly most extreme example
- Philosophy: provide protection in kernel, but abstraction only in user-level libraries

General Perception of OS Research

OS Research Actually

- Not really just about building OSes
- Any large code base converges on becoming an OS
  - Manages memory, programming for space/performance, hardware details
  - database, JVM, browser, parallel/distributed systems, internet services
- How to structure systems and deal with complexity
  - Modularize and encapsulate
  - Choose interfaces/abstractions carefully
- Themes in this class
  - Abstractions for managing/accessing resources: storage, replication, naming
  - Correctness: concurrency and sharing
  - Guarantees in real systems: security, fault tolerance, etc.

What is OS Research?

- New kernel architectures
  - Typically <10% of papers in top OS conferences!
- New kernel components
  - E.g., new VM or network stack implementation, process or disk scheduling algorithm, buffer cache, file system, etc.
  - Also only a small fraction of papers
- User-level code running on an existing kernel
  - Challenging applications often face OS issues
  - Either hit your head against OS because it does the wrong thing
  - Or have to re-implement OS-like abstractions (for better performance, distributed operation, etc.)

Today's Readings

Today's Readings: Motivation & Professional Development

- Levin & Redell: "How (and How Not) to write a Good Systems Paper"
  - Summary:
    - Rules of thumb for writing good systems papers
    - Most papers are limited
    - A few researchers consistently publish at top conferences
    - Learn from them or risk a second-rate career
    - A good framework to use for thinking about the papers we read in this class
- Haldane '28: On being the right size
  - Summary: scale and use case are determinants for the fitness of a system
- Hamming: "You and Your Research"
  - Summary:
    - Stay open to people and ideas
    - Research is both time-consuming and exciting
    - Research inspiration comes from unexpected directions

How (and How Not) to write a Good Systems Paper (Levin & Redell)

- Categories
  - Real system
  - Unimplemented system
  - Theoretical topic
- Most SOSP papers are about prototypes
  - Some differences by field
    - E.g. architecture: more simulation (why?)
- Prototype is not enough
- Why build a system?
  - "The purpose of (scientific) computing is…
  - insight, not numbers." – Richard Hamming
- Adapt old techniques to new technology

		Effort	SOSP Accept	SOSP Reject
Real system	Used by others	Huge	Occasional	Occasional
Worked once	Benchmarks ran by deadline	Large	Common	Occasional
Simulated	Simulations run	Large/med	Occasional	Common
Paper design	*Sounds* good	minimal	Rare	Common

On being the Right Size (Haldane)

- Why should we read this paper?
  - Bold statements that are unusual in scientific literature
  - Intellectual pursuits require breadth and depth
- Incommensurate scaling: eye size, jumping height
  - Hardware constraints influence system design: strength of femur limits height
- Distribution of complexity and even basic assumptions change as a function of scale
  - E.g. MapReduce algorithm is a handful of lines of code
  - Apache Hadoop: 2,422,127 SLoC (as of 9/5/17)
- What lessons relevant to computer systems?
  - Systems are not becoming larger because they are more complicated but
  - Systems become more complicated when becoming larger

Example: Video Server

Example: Video Server

- Buy hardware with the following capabilities:
  - 40 MByte/sec SCSI disk (more for NVM, solid state disks)
  - 1 Gbit/sec Ethernet (could be more in modern data centers)
- Application requirements:
  - Serve 200 Kbit/sec video streams
  - Many users spread around the Internet
  - Access control
- Maximum capacity: 500 clients???

Reality: Much lower Capacity

- CPU bottleneck
  - Software structure may impose many context switches
  - Concurrency may introduce lock contention
- Disk I/O limitations
  - Multiple streams introduce disk seeks: E.g., 5ms/8K read = 1.6 MByte/sec
  - Must pipeline disk requests—requires prefetching
  - But then OS buffer cache may fill memory and induce paging
- Network stack limitations
  - OS may buffer stale data (dropped frames)
  - Congestion control improperly prioritizes packets

Maximizing Throughput

- Goal: Service maximum number of clients over time
- Avoid letting resources stay idle
  - Different resources: CPU, disk, network
  - E.g., don't leave disk idle while waiting for network
- Key technique: Concurrency
  - Concurrency ensures each resource can be utilized
- Example: while waiting for a disk read for one client
  - . . . can use CPU to compute data for another client
  - . . . can transmit data to a third client over network

Impact of Disks

Disks

- For many systems, disks are the limiting factor
- Can limit throughput far below network capacity
  - E.g., cut naïve video server throughput by factor of 7–8
- Can also limit capacity of a system
- Example: build a system to index CSC7103 lecture notes
  - Scan all files, create index file
  - Rely on kernel for low-level details of storage management
- Example: build a system to index the web
  - Can't stuff enough disks in one computer to hold index
  - Need a user-level distributed storage system

Anatomy of a Disk

- Stack of magnetic platters
  - Rotate together on a central spindle @3,600-15,000 RPM
  - Drive speed drifts slowly over time
  - Can't predict rotational position after 100-200 revolutions
- Disk arm assembly
  - Arms rotate around pivot, all move together
  - Pivot offers some resistance to linear shocks
  - Arms contain disk heads–one for each recording surface
  - Heads read and write data to platters

Disk Positioning System

- Move head to specific track and keep it there
  - Resist physical shocks, imperfect tracks, etc.
- A seek consists of up to four phases:
  - speedup–accelerate arm to max speed or half way point
  - coast–at max speed (for long seeks)
  - slowdown–stops arm near destination
  - settle–adjusts head to actual desired track
- Very short seeks dominated by settle time (~1 ms)
- Short (200-400 cyl.) seeks dominated by speedup of arm
  - Accelerations of 40g

Seek Details

- Head switches comparable to short seeks
  - May also require head adjustment
  - Settles take longer for writes than reads
- Disk keeps table of pivot motor power
  - Maps seek distance to power and time
  - Disk interpolates over entries in table
  - Table set by periodic "thermal recalibration"
  - 500 ms recalibration every 25 min, bad for throughput
- "Average seek time" quoted can be many things
  - Time to seek 1/3 disk, 1/3 of time to seek whole disk,

What does this mean for Systems?

- More concurrency can mean higher throughput
  - If you have N requests, you can sort them
  - Ask disk/controller to perform them in optimal order
  - Can vastly reduce seek times, increase throughput
- Cache data you will use again in faster memory
- More efficient to access nearby data than far away data
  - E.g., good idea to put related data close together (e.g., put file near metadata)
- Larger requests mean higher throughput
  - Sequential throughput much higher than random
- Note: Last three may require bypassing the OS

Preview of Disk Techniques

- Make many disks seem like one large disk (RAID)
- Structure storage for large writes (Logs)
- Expose information/control to applications
  - What pages are in virtual memory
  - What pages have been accessed
  - How should OS manage cache
  - What threads are blocked waiting for disk I/O

Impact of Network

Network

- Many machines serve clients over the Internet
  - Speed of light means typical round trip time (RTT) in 10's of milliseconds
- Often have to wait for client
  - E.g., send back HTML page, wait for image requests
  - Congestion control may require limiting send rate
- Again, concurrency achieves good throughput
- Also may care about space
  - E.g., if it takes one second to service and requires 10 MByte of memory, memory will be limiting factor
- Many kernels impose other limitations:
  - Number of file descriptors or processes
  - Size of hash table for TCP connections (mostly fixed today)

Achieving Concurrency

- Can use OS processes
  - Heavy weight (expensive to create) and memory intensive
- Can use non-blocking I/O operations
  - read/write, but return immediately if not possible
  - Single process handles many clients
  - Not good for disk concurrency
- Can use threads implemented at user level
  - Build on non-blocking I/O primitives
- Can use kernel-level threads
  - But more heavy weight than user-level threads
- More on this later in the course. . .

System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address
	`listen` – listen for clients
`socket` – make socket	
`bind` – assign address (optional)	
`connect` – connect to listening socket	
	`accept` – accept connection
`write` – send data	`read` – receive data
`read` – receive data	`write` – send data

- Anything red might block, waiting for network
  - Obviously bad for applications that need concurrency

Non-blocking I/O

- Use `fctnl` to set `O_NONBLOCK` flag on descriptor
- Non-blocking semantics of system calls:
  - `read` immediately returns `-1` with `errno` `EAGAIN` if no data
  - `write` may not write all data, or may return `EAGAIN`
  - `connect` may "fail" with `EINPROGRESS` (or may succeed, or may fail with real error such as a `ECONNREFUSED`)
  - `accept` may fail with `EAGAIN` if no pending connections

How to know when to read/write?

```
struct pollfd {
  int fd;         /* file descriptor */
  short events;   /* Events you are interested in */
  short revents;  /* Events that have happened (results) */
};

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

/* Some possible events: */
#define POLLIN     0x0001  /* Can read fd without blocking */
#define POLLOUT    0x0004  /* Can write fd without blocking */
#define POLLERR    0x0008  /* Error on fd (only in revents) */
#define POLLHUP    0x0010  /* ''Hangup'' has occurred on fd */
```

- Note: BSD used `select` to achieve same thing
  - Most OSes support both `select` and `poll` today

epoll

- Newer Linux provides `epoll`
- Interface allows more efficient implementation
  - Register interest with `epoll_create`, `epoll_ctl`
  - Wait with `epoll_wait` syscall
  - Kernel doesn't have to re-scan pollfd array on each wait
- New option bits reduce calls to `epoll_ctl`
  - `EPOLLONESHOT` – only wait for event once
  - `EPOLLET` - "edge triggered" (as opposed to level triggered)
- `epoll` is Linux specific
  - But BSD has `kqueue/kevent` which is similar idea

epoll Interface

```
typedef union epoll_data {
    int fd;
    /* ... */
} epoll_data_t;

struct epoll_event {
    __uint32_t events;      /* Epoll events */
    epoll_data_t data;      /* User data variable */
};

int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd,
              struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

Asynchronous Programming Model

- **Many non-blocking file descriptors in one process**
  - Wait for pending I/O events on file many descriptors
  - Each event triggers some *callback* function
- **E.g., build "callback harness":**

```
/* Register callback for when fd is readable or writable */
void cb_add (int fd, int write, void (*fn)(void *), void *arg);

/* Unregister callback */
void cb_free (int fd, int write);

/* Loop forever checking callbacks */
void cb_check (void);
```

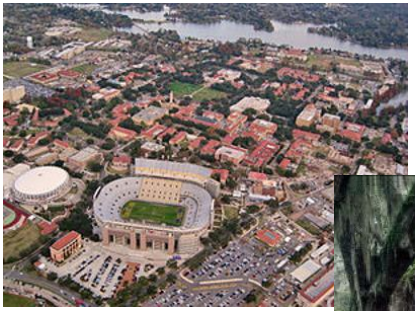
- **Often called *event-based programing***

Simplified example

```
struct state {
  int fd;
  /* ... */
};

void doit (void) {
  struct state *st = malloc (sizeof (*st));
  st->fd = create_new_tcp_socket ();
  connect (st->fd, &someplace, sizeof (someplace));
  cb_add (st->fd, 1, doit_2, st);
}
static void doit_2 (void *_st) {
  struct state *st = _st;
  write (st->fd, "request\n", 8);
  cb_free (st->fd, 1);
  cb_add (st->fd, 0, doit_3, st);
}
static void doit_3 (void *_st) {
  struct state *st = _st;
  /* read more from st->fd until you get full response */
}
```

Pros & Cons of Event-based code

- Advantages
  - Fewer nasty bugs than threads (will discuss later)
  - No locking, so no coarse- vs. fine-grained locking issues
  - Works with legacy, non-reentrant code (e.g., `strtok`, `getpwnam`)
  - Very efficient in terms of memory per client
  - Callbacks usually more efficient than thread switches
- Disadvantages
  - "Stack ripping" makes code ugly
  - Long running events make program unresponsive
  - Harder to take advantage of multiprocessors
  - Harder to do non-blocking disk I/O with existing OSes



LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP