Week 12

Exercise 1: Controlling access via flock()

1/46

A.k.a. the great `flock()` debacle of 2017 ...

Consider a program that

- forks a child
- both parent and child write to stdout

Without any controls, output is arbitrarily interleaved.

Using `flock()`, control the file access

- so that each process writes a full line of output before the other

The Problem: ! RTFM ... `man 2 flock`

Concurrency

Concurrency

3/46

Concurrency = multiple processes running (pseudo) simultaneously

The alternative to concurrency ... sequential execution

- each process runs to completion before next one starts
- low throughput; not acceptable on multi-user systems

Concurrency increases system throughput, e.g.

- if one process is delayed, others can run
- if we have multiple CPUs, use all of them at once

If processes are completely independent ...

- each process runs and completes its task
- without any effect on the computation of other processes

... Concurrency

4/46

In reality, processes are often not independent

- multiple processes accessing a shared resource
- one process synchronizing with another for some computation

Effects of poorly-controlled concurrency

- *nondeterminism* ... same code, different runs, different results
    - e.g. output on shared resource is jumbled
    - e.g. input from shared resource is unpredictable
- *deadlock* ... a group of processes end up waiting for each other
- *starvation* ... one process keeps missing access to resource

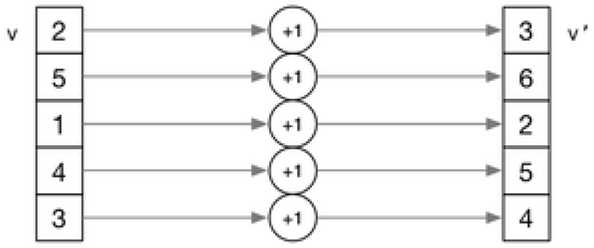
Therefore we need *concurrency control* methods.

... Concurrency

5/46

Non-problematic concurrency: parallel processing (e.g. GPU)

- mutliple identical processors
- each given one element of a data structure from main memory
- each performing same computation on that element
- results copied back to main memory data structure
- need to *synchronise* on completion of computation



... Concurrency

6/46

Example of problematic concurrency ... bank withdrawal:

```
// check balance and return amount withdrawn
1. int withdraw(Account acct, int howMuch)
2. {
3.    if (acct.balance < howMuch) {
4.       return 0; // can't withdraw
5.    else {
6.       acct.balance -= howMuch;
7.       return howMuch;
8.    }
9. }
```

Scenario:

- two processes; each withdraws $300 from account A (balance $500)
- process 1 executes up to and including line 3, then swapped out
- process 2 executes up to and including line 3, then swapped out
- process 1 continues and reduces balance by $300
- process 2 continues and reduces balance by $300
- final balance -$100;   should have been $200, and one process fails

---

Concurrency Control

7/46

Concurrency control aims to

- provide correct sequencing of interactions between processes
- coordinate semantically-valid access to shared resources

Two broad classes of concurrency control schemes

- *shared memory* based   (e.g. semaphores)
  - uses shared variable, manipulated atomically
  - blocks if access unavailable, decrements once available
- *message passing* based   (e.g. send/receive)
  - processes communicate by sending receiving messages
  - receiver can block waiting for message to arrive
  - sender *may* block waiting for message to be received
    - synchronous message passing: sender waits for ACK of receipt
    - asynchronous message passing: sender transmits and continues

---

Semaphores

8/46

Semaphore operations:

- **`init(Sem, InitValue)`**
  - set the initial value of semaphore `Sem`
- **`wait(Sem)`**   (also called `P()`)

- if current value of *Sem* > 0, decrement *Sem* and continue
- otherwise, block and wait until *Sem* value > 0

- **signal(*Sem*)** (also called `V()`)
  - increment value of *Sem*, and continue

Needs fair release of blocked processes, otherwise starvation possible

- can be achieved via a FIFO queue (fair, but maybe not optimal)

Exercise 2: Semaphores

9/46

Solve the withdrawal problem using semaphores

```
// check balance and return amount withdrawn
int withdraw(Account acct, int howMuch)
{
   if (acct.balance < howMuch) {
      return 0; // can't withdraw
   else {
      acct.balance -= howMuch;
      return howMuch;
   }
}
```

Assume that each `Account` record includes a semaphore `sem`

... Semaphores

10/46

Semaphores on Linux/Unix ...

- **`#include <semaphore.h>`**, giving **`sem_t`**
- **`int sem_init(sem_t *Sem, int Shared, uint Value)`**
  - create a semaphore object, and set initial value
- **`int sem_wait(sem_t *Sem)`** (i.e. wait())
  - try to decrement, block if *Sem* == 0
  - has variants that don't block, but return error if can't decrement
- **`int sem_post(sem_t *Sem)`** (i.e. signal())
  - increment the value of semaphore *Sem*
- **`int sem_destroy(sem_t *Sem)`**
  - free all memory associated with semaphore *Sem*

Message Passing

11/46

Message passing mechanisms often embedded in prog languages

Example: Google's Go language

- *goroutines* ... concurrently executing "functions"
- *channels* ... communication pipes between goroutines
- *select* ... manage multiple channels

```
// declare a channel
pipeline := make(chan int)

// send a value on a channel
pipeline <- 42

// receive a value from a channel
object = <- pipeline
```

... Message Passing

12/46

Go example for bank withdrawal:

```
// define 4 channels wd, dep, bal, resp
// define a variable to hold the balance
for {
   select {
   case howMuch := <- wd:
      if howMuch > balance {
         resp <- 0
      }
      else {
         balance -= howMuch
         resp <- howMuch
      }
   case howMuch := <- dep:
      balance += howMuch
   case <- bal:
      resp <- balance
   }
}
```

... Message Passing

13/46

Message passing in C is provided by *message queues*

- `#include <mqueue.h>`, giving `mqd_t`
- `mqd_t mq_open(char *Name, int Flags)`
  - create a new message queue, or open existing one
- `int mq_send(mqd_t *MQ, char *Msg, int Size, uint Prio)`
  - adds message `Msg` to message queue `MQ`
  - `Prio` gives priority; blocks if `MQ` is full
- `int mq_receive(mqd_t *MQ, char *Msg, int Size, uint *Prio)`
  - removes oldest message with priority `*Prio` from queue `MQ`
  - blocks if `MQ` is empty; can run non-blocking
- `int mq_close(mqd_t *MQ)`
  - finish accessing message queue `MQ`

Networks

Networks

15/46

Network = interconnected collection of computers

Flavours of networks:

- *local area networks* ... within an organisation/physical location
- *wide area networks* ... geographically dispersed (WAN)
- *Internet* ... global set of interconnected WANs

Why do we need networks?

- previously ... transfer data, send text-based emails
- nowadays ... communication, communication, communication
- sharing resources e.g. printers, large storage devices, ...

... Networks

16/46

What are the basic requirements for a network?

- get data from machine A to machine B
- A and B may be separated by 100's of networks and devices

How to achieve this? (using postal service analogy)

- need a unique address for destination
- identify a route (first post office)
- process at intermediate nodes (other post offices)

- follow certain protocols (envelopes, stamps fees)

Overview of Network Communication

17/46

How a file is sent over the network:

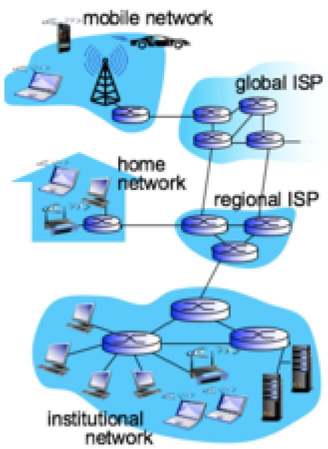
- File data divided into *packets* by source device
  - packets are small fixed-size chunks of data, with headers
- Passed across *physical link* (wire, radio, optic fibre)
- Passing through multiple *nodes* (routers, switches)
  - each node decides where to send it next (for best route)
- Packets reach destination device
- Re-ordering, error-checking, buffering
- File received by receiving process/user

The Internet

18/46

Components of the Internet ...

- millions of *connected devices*
  - e.g PC, server, laptop, smartphone
  - *host* = end system, running network apps
- *communication links*
  - e.g. fibre, copper, radio, satellite
  - bandwidth = transmission rate
- *packet switches*
  - e.g. routers, network switches
  - compute next hop, forward packets



... The Internet

19/46

Internet communications are based on a 5-layer "stack":

- *Physical layer:* bits on wires or fibre optics or radio
- *Link layer:* ethernet, MAC addressing, CSMA etc.
- *Network layer:* routing protocols, IP
- *Transport layer:* process-process data transfer, TCP/UDP
- *Application layer:* DNS, HTTP, email, Skype, torrents, FTP etc.

Typical packet encapsulates data from all lower layers

Why so many layers (of abstraction)?

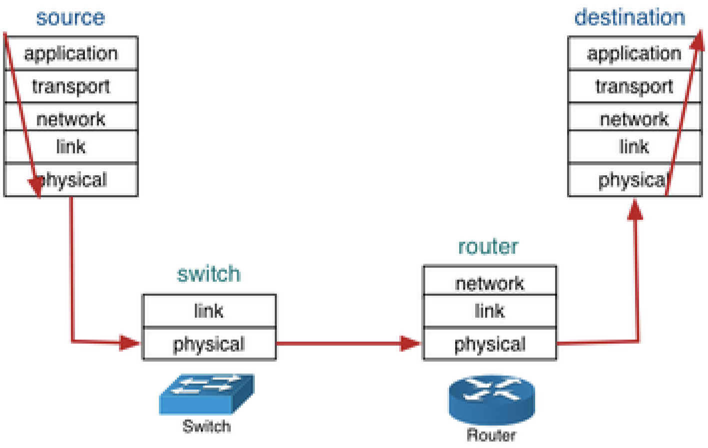
- each layer encapsulates one aspect of network transport
- provides layered *reference model* for discussion
- modularization eases maintenance/updating
  - e.g. changing implementation of one layer doesn't affect other layers

---

... The Internet

20/46

Path of data through network layers



Protocols

21/46

Network protocols govern all communication activity on the network

Protocols provide communication rules ...

- format and order of messages sent/received
- actions taken on message transmission/receipt

Protocols are defined in all of the layers, e.g.

- link layer: PPP (point-to-point protocol), ...
- network layer: IP (internet protoocol), ...
- transport layer: TCP (transmission control), UDP (user datagram)
- application layer: HTTP, FTP, SSH, POP, SMTP, ...

Typically more protocols in the higher-level layers

Networks: Application Layer

Network Apps

23/46

The application layer directly supports the apps we interact with, e.g.

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking

Client-Server Architecture

24/46

Client-server = common way of structuring network communication

Server is a data provider

- process that waits for requests
- always-on host, with permanent IP address
- possibly using data centers / multiple CPUs for scaling

Client is a data consumer

- sends requests to server; collects response
- may be intermittently connected, may have dynamic IP address
- does not communicate directly with other clients

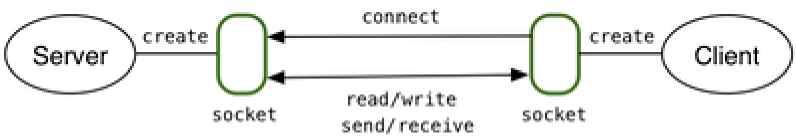
Aside: peer-to-peer (P2P) systems run both client and server processes on each host

Unix Sockets

25/46

Socket = an end-point of an inter-process communication channel

- commonly used to construct client-server systems
- either locally (Unix domain) or network-wide (Internet domain)
- server creates a socket, binds to an address, listens for connections
- client creates a socket, connects to the server, reads/writes



... Unix Sockets

26/46

`int socket(int Domain, int Type, int Protocol)`

- requires **`#include <sys/socket.h>`**, sockets are `int`s (like fds)
- creates a socket, using ...
  - ***`Domain`*** ... communications domain
    - `AF_LOCAL` ... on the local host (Unix domain)
    - `AF_INET` ... over the network (Internet domain)
  - ***`Type`*** ... semantics of communication
    - `SOCK_STREAM` ... sequenced, reliable communications stream
    - `SOCK_DGRAM` ... connectionless, unreliable packet transfer
  - ***`Protocol`*** ... communication protocol
    - many exist (see `/etc/protocols`), e.g. IP, TCP, UDP, ...
- returns a socket descriptor or -1 on error

... Unix Sockets

27/46

`int bind(int Sockfd, SockAddr *Addr, socklen_t AddrLen)`

- associates an open socket with an address
- for Unix Domain, address is a pathname in the file system
- for Internet Domain, address is IP address + port number

`int listen(int Sockfd, int Backlog)`

- wait for connections on socket *`Sockfd`*
- allow at most *`Backlog`* connections to be queued up

`SockAddr = struct sockaddr_in`

- C struct containing components of "visible" socket address

... Unix Sockets

28/46

`int accept(int Sockfd, SockAddr *Addr, socklen_t *AddrLen)`

- *`Sockfd`* has been created, bound and is listening
- blocks until a connection request is received

- sets up a connection between client/server after `connect()`
- places information about the requestor in *`Addr`*
- returns a new socket descriptor, or -1 on error

`int connect(int Sockfd, SockAddr *Addr, socklen_t AddrLen)`

- connects the socket *`Sockfd`* to address *`Addr`*
- assumes that *`Addr`* contains a process listening appropriately
- returns 0 on success, or -1 on error

... Unix Sockets

29/46

Pseudo-code showing structure of a simple client program:

```
main() {
   s = socket(Domain, Type, Protocol)
   serverAddr = {Family,HostName,Port}
   connect(s, &serverAddr, Size)
   write(s, Message, MsgLength)
   read(s, Response, MaxLength)
   close(s)
}
```

(See http://www.linuxhowtos.org/C_C++/socket.htm)

... Unix Sockets

30/46

Pseudo-code showing structure of a server program:

```
main() {
   s = socket(Domain, Type, Protocol)
   serverAddr = {Family,HostName,Port}
   bind(s, serverAddr, Size)
   listen(s, QueueLen)
   while (1) {
      int ss = accept(s, &clientAddr, &Size)
      if (fork() != 0)
         close(ss)  // server not involved
      else {
         // fork of server handles request
         close(s)
         handleConnection(ss)
         exit(0)
      }
   }
}
```

Addressing

31/46

Server processes must have a unique Internet-wide *address*

- part of address is *IP address* of host machine
- other part of address is *port number* where server listens

Example: `128.119.245.12:80`

- address of web server on gaia.cs.umass.edu

Some standard port numbers

- 22 ... ssh (Secure Shell)
- 25 ... smtp (Simple Mail Transfer Protocol)
- 53 ... dns (Domain Name System)
- 80 ... http (Web server)
- 389 ... ldap (Lightweight Directory Access Protocol)
- 443 ... https (Web server (encrypted))

- 5432 ... PostgreSQL database server

IP Addresses

32/46

IP Address = unique identifier for host on network

- given as a 32-bit identifier (dotted quad), e.g. 129.94.242.20
- special case: 127.0.0.1 (loopback address referring to local host)
- IP addresses are assigned by
    - sys admin entering into local registry (for "permanent" addresses)
    - dynamically, by getting a temporary address from DHCP server

Note: the world is runnning out of 32-bit IP addresses

- why? Internet of Things ... *every* networked device needs an IP
- IPv6 uses 128-bit addresses e.g. 2001:388:c:4193:129:94:242:20
- distinct addresses: IPv4 $4{\times}10^9$, IPv6 $3{\times}10^{38}$

Application-layer Protocols

33/46

Each application-layer protocol defines

- *types* of messages
    - different types of requests and responses
- message *syntax*
    - what fields are in messages; how fields are delineated
- message *semantics*
    - meaning of information in fields
- processing *rules*
    - when and how processes respond to messages

Protocols can be *open* (e.g. HTTP) or *proprietary* (e.g. Skype)

The HTTP Protocol

34/46

HTTP = HyperText Transfer Protocol

- an extremely important protocol (drives the Web)
- message types: URLs (requests) and Web pages (responses)
- message syntax: headers + data (see details later)

URLs are the primary type of request

Web pages are the primary type of response

- contain HTML; may contain references to other types of objects
- all objects are addressable via a URL

... The HTTP Protocol

35/46

Application-layer protocol for the Web

Client-server model:

- client = *Web browser* (e.g. Chrome)
    - sends HTTP requests
    - receives HTTP responses

    - shows response as web page
- server = *Web server* (e.g. Apache)
    - receives HTTP requests
    - sends HTTP responses

... The HTTP Protocol

36/46

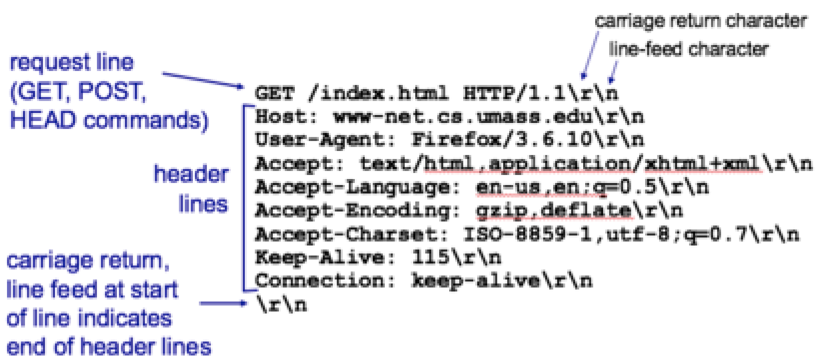
Transport layer view of HTTP application layer

- using TCP
- client initiates TCP connection (socket) to server, port 80
- server accepts TCP connection from client
- client sends HTTP request messages (e.g. GET)
- server responds with HTTP messages (e.g. HTML)
- interaction completes, connection (socket) closed

... The HTTP Protocol

37/46

HTTP request message (ascii text)



URL can also include a *query string*, e.g.

```
http://cse.unsw/course/view.php?c=COMP3231&s=verbose
```

Protocol *Host* *Path* *Query*

... The HTTP Protocol

38/46

First line of HTTP request contains (method, path, protocol), e.g.

```
GET  ~cs1521/17s2/index.html  HTTP/1.1
```

Method *Path* *Protocol*

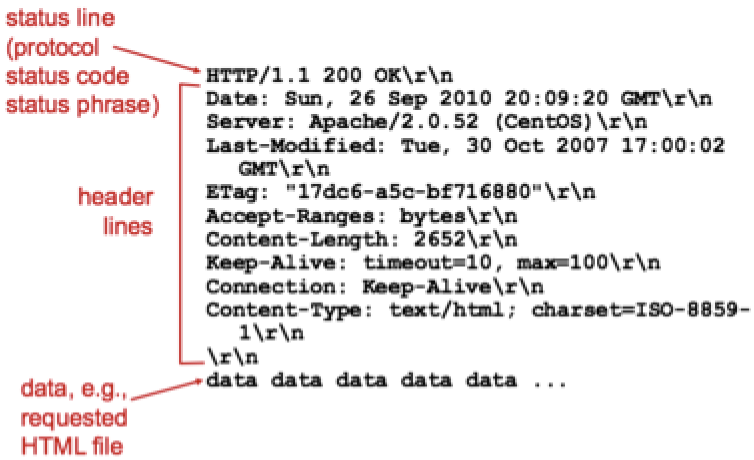
- no need to mention host, since connection already established

- **`GET`** requests data from resource specified by path
  - query string is included in the path
- **`POST`** submitd data to be processed by specified resource
  - query string is included in the body
- **`HEAD`** same as `GET`, but returns only header (no data)

... The HTTP Protocol

39/46

HTTP response message (ascii text)



... The HTTP Protocol

40/46

Response status codes appear in first line of HTTP response

- **`202 OK`** ... succesful request
- **`301 Moved Permanently`** ... requested object moved
  - returns new URL for client to use in future requests
- **`400 Bad Request`** ... request cannot be processed
  - possible reasons: bad request syntax, request size too large, ...
- **`403 Forbidden`** ... valid request cannot be processed
  - possible reasons: user does not have permission for operation
- **`404 Not Found`** ... path does not exist on server
- **`500 Internal Server Error`** ... server cannot complete request
  - possible reasons: server side script fails, database not accessible, ...

Server Addresses (DNS)

41/46

Network requests typically use server *names*, e.g.

- `http://`**`www.cse.unsw.edu.au`**`/~cs1521/17s2/`

Setting up a TCP connection needs an IP address, not a name

Domain Name System provides name→IP address mapping

Can access this on Unix/Linux via the `host` command, e.g.

```
$ host www.cse.unsw.edu.au
www.cse.unsw.edu.au has address 129.94.242.51
$ host a.b.c.com
Host a.b.c.com not found: 3(NXDOMAIN)
```

assumes that you have a network connection

... Server Addresses (DNS)

42/46

In real life, often have one object referenced by many names, e.g.

- a person: *name*, SSN, TFN, passport #, ...

On the Internet, each *host* has ...

- one or more symbolic names, unique IP address
- symbolic: `www.cse.unsw.edu.au`, IP: `129.94.242.51`

Note:

- a given IP address may be reachable via several names
- a given name may map to several IPs (e.g. for load distribution)

... Server Addresses (DNS)

43/46

Domain Name System (DNS)

- effectively a distributed database of name→IP mappings
- implemented across a hierarchy of *name servers*
- name servers cooperate to *resolve* names to IP addresses

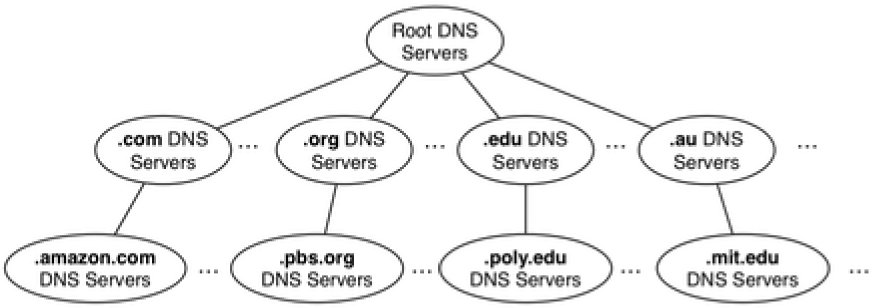
This is an extremely important core function on the Internet

Why not centralize DNS?

- central point of failure, high traffic volume
- distant database (lag), maintenance of very large DB

... Server Addresses (DNS)

44/46



Resolving `www.amazon.com` in this system

- contact a root DNS server to find `.com` DNS server
- contact `.com` DNS server to get `amazon.com` DNS server
- contact `amazon.com` DNS server to get IP of their web server

... Server Addresses (DNS)

45/46

Two styles of name resolution

- *iterated query* ... work done by client
  - client contacts name server X
  - gets response "I don't know, but ask name server Y"
  - OR gets response "Here is the IP address"
  - client repeats above steps until name resolved
- *recursive query* ... work done by name servers
  - client contacts name server X
  - X contacts name server Y, Y contacts Z, ...
  - query propagates until name resolved

... Server Addresses (DNS)

46/46

How are the various DNS servers structured/managed?

- *top-level domain* (TLD) name servers
  - **.com**, **.org**, **.edu** and all country-level domains (e.g. **.uk**)

  - Network Solutions maintains servers for **.com**
  - AusRegistry maintains servers for **.au**
- *authoritative* name servers
  - maintains mappings from names to IP within an organisation
  - all hosts within the organisation are registered here
- *local* (default) name servers
  - maintains cache of name→IP mappings
  - starting point for DNS queries, forward to TLD server (if !cached)

Produced: 19 Oct 2017