Chapter 3
Concurrency

Problem: How to handle multiple requests for service

Multiuser systems: users make requests
Networks: machines make requests

Concurrency: the ability to handle multiple requests at the same time.

All multiuser systems provide concurrency,

Most single user system provide it.

Concurrency Terms:

Multiple processes/Multiprocessing
Multiple tasks/Multi-tasking
Multiple threads

Two Kinds of Servers

Iterative

- Server handles one client at a time
- Usable if services takes a short time
- Examples: time-of-day, ping
- Disadvantage: one slow client can block others

Concurrent

- Creates a process/task/thread for each client
- Allows long running service
- Examples: telnet, ftp
- Disadvantage: process creation is a heavy-weight operation

Unix Processes

Fundamental unit of concurrency

Created by the `fork()` command.

A process is always created by another process.

Terminology:
Creating process: parent
Created process: child

A child is a separate copy of the parent
(including a separate copy of all variables)

Each runs independent of the other
(unless the programmer deliberately builds in some sort of synchronization)

Each process is given a unique id number (pid) by the operating system

Example of Process Creation

```
#include <stdio.h>
int sum;
int main () {
  int i;
  sum = 0;
  fork();  /* create a new process */
  for (i=1; i<=10000 ; i++) {
    sum += i;
    printf("sum is %d\n", sum);
    fflush(stdout);
  }
  return 0;  // or exit(0);
}
```

Both parent and child print 10000 lines.
both have separate `i` and `sum` variables

Output depends on the time slice and I/O buffering:

```
sum is 1
sum is 3
sum is 1
sum is 3
sum is 6
```

Operating System Issues

Timeslicing: each runnable process gets an allocation of CPU time.

Switching processes costs overhead so:
the bigger the time slice the less over head
the smaller the time slice the more frequently each process makes progress

Multiple CPU machine will have several processes running at once

Efficiency issue:

A design that context switches too often will use too much overhead (and will run slow).

Concurrent Server Example

```
#include <stdio.h>
int main () {
 int pid;
 char ch = 'y';
 while (ch != 'x') {
  printf("hit return for another process");
  getc(ch);
  pid = fork();  /* create a new process */
  if (pid != 0){
   printf("original/parent\n");
  } else {
   printf("new/child\n");
   exit(0);  /* don't forget this or this:
    execl("/bin/ls", NULL); */
  }
 }
 return 0;
}
```

Creates one child per iteration.

Child does something then exits.

Warning: missing exit gets you exponentially many processes.

I/O and Concurrency

Problem: Several things are/may happen at once, including I/O.

In Unix all I/O is considered file I/O.

Serial: Read and wait for the information to arrive.

Conditional: Read information if available, otherwise continue

Solution: tell read to return `-1` if nothing available, otherwise it reads normally
(use `ioctl` to set non-blocking I/O)

Solution: see what files have something available to read, read only from those with input available
(use the `select` to see what has input)