Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Concurrency vs Parallelism

- **Concurrency**: At least two tasks are making progress at the same time frame.
  - Not necessarily at the same time
  - Include techniques like time-slicing
  - Can be implemented on a single processing unit
  - Concept more general than parallelism

- **Parallelism**: At least two tasks execute literally at the same time.
  - Requires hardware with multiple processing units
Sequential Echo Server

**Client**

- `socket`
- `connect`
- `write`
- `read`
- `close`

**Server**

- `socket`
- `bind`
- `listen`
- `accept`
- `read`
- `write`
- `Read EOF`
- `close`

Connection request from next client

Await connection request from next client
Sequential Echo Server

- **Process one request at a time**

Client 1 -> Server
- Connect
- Write
- Call read
- Read returns
- Close

Server -> Client 2
- Accept
- Read
- Write

Client 2
- Connect
- Write
- Call read
- Wait for server to accept() connection
- Read returns

Server
- Accept
- Read
- Write
- Close
Why not Sequential Servers?

• Increased latency
  – client2 must wait for client1 to finish before getting served

• Low utilization
  – Server is idle while waiting for client1’s requests. It could have served another client during those idle times!

• Solution: implement *concurrent servers*
  – serve multiple clients at the same time
Design of a Concurrent Server

1. Processes
   - Spawn one server process to handle each client connection
   - Kernel automatically interleaves multiple server processes
   - Each server process has its own private address space

2. Threads
   - Create one server thread to handle each client connection
   - Kernel automatically interleaves multiple server threads
   - All threads share the same address space

3. Event-based using I/O multiplexing
   - One process, one thread, but programmer manually interleaves multiple connections
   - Relies on lower-level system abstractions
Review: Sequential Echo Server

```c
main() /* pseudo-code, ignore syntax */
{
    int listenfd, connfd;

    listenfd = socket(...) /* create socket */
    bind(listenfd,...); /* bind socket to port */
    listen(listenfd,...); /* listen for incoming connections*/

    while (1) {
        connfd = accept(listenfd,...);
        while (readline(connfd,...)> 0)
            write(connfd,...);
        close(connfd);
    }
}
```
Approach #1: A Multi-Process Server

```c
int main(int argc, char **argv)
{
    int listenfd, connfd;

    listenfd = socket(...);
    bind(listenfd,...);
    listen(listenfd,...);
    while (1) {
        connfd = accept(listenfd,...);
        if (fork() == 0) {
            close(listenfd);
            while (readline(connfd,...) > 0)
                write(connfd,...);
            close(connfd); /* Child closes connection with client */
            exit(0);       /* Child exits */
        }
    }
}
```
Process Execution Model

- Each client is handled by a different server process
- No shared state between server processes
- Both parent & child have copies of listenfd and connfd
Implementation Must-dos With Process-Based Designs

• Listening server process must reap zombie children

• Listening server process must close its copy of connfd
  – Kernel keeps reference for each socket/open file
  – After fork, \texttt{refcnt(connfd)} = 2
  – Connection will not be closed until \texttt{refcnt(connfd)} == 0
Process-Based Concurrent Server

```c
int main(int argc, char **argv)
{
    int listenfd, connfd;

    signal(SIGCHLD, sigchld_handler);
    listenfd = socket(...);
    bind(listenfd,...);
    listen(listenfd,...);
    while (1) {
        connfd = accept(listenfd,...);
        if (fork() == 0) {
            close(listenfd);
            while (readline(connfd,...) > 0)
                write(connfd,...);
            close(connfd);    /* Child closes connection with client */
            exit(0);          /* Child exits */
        }
        close(connfd);
    }
}
```
Process-Based Concurrent Server (cont)

```c
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0) ;
    return;
}
```

– Reap all zombie children
Pros and Cons of Process-Based Designs

+ Handle multiple connections concurrently
+ Clean sharing model
  - No shared address space (no shared global variables etc..)
  - Shared file tables with separate file descriptors
– Additional overhead for process control
– Nontrivial to share data between processes
  - Requires IPC (interprocess communication) mechanisms
Approach #2: A Multi-threaded server

- Very similar to approach #1 (multiple processes)
  - but, with threads instead of processes
The View of a Process

- **Process** = data/code, execution context and kernel context

Thread (main thread)

- Stack

Thread (execution) context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)

Code and Data

- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data

Kernel context:
  - VM structures
  - Descriptor table
  - Brk pointer
A Process With Multiple Threads

• **Multiple threads can be associated with a process**
  – Each thread has its own logical control flow (different execution context)
  – Threads share the same code, data, and kernel context
    • Share common virtual address space

### Thread 1 (main thread)
- Stack 1
- Thread 1 context:
  - Data registers
  - Condition codes
  - SP1
  - PC1

### Thread 2 (peer thread)
- Stack 2
- Thread 2 context:
  - Data registers
  - Condition codes
  - SP2
  - PC2

### Shared code and data
- shared libraries
- run-time heap
- read/write data
- read-only code/data

### Kernel context:
- VM structures
- Descriptor table
- brk pointer
Thread Execution

- **Single Core Processor**
  - Simulate concurrency by time slicing
- **Multi-Core Processor**
  - True concurrency

Run 3 threads on 2 cores
Threads vs. Processes

• How threads and processes are similar
  – Each has its own logical control flow
  – Each can run concurrently with others (possibly on different cores)
  – Each is context switched

• How threads and processes are different
  – Threads share code and some data
    • Processes (typically) do not
  – Threads are less expensive than processes
POSIX Threads (Pthreads)

- C language programming types and procedure calls
- Implemented with a pthread.h header
- To compile with GNU compiler, 2 methods:
  - gcc/g++ programe -lpthread
  - gcc/g++ -pthread programe
- Programmers are responsible for synchronizing access (protecting) globally shared data.
- Capabilities like thread priority are not part of the core pthreads library.
POSIX Threads (Pthreads)

Source: https://computing.llnl.gov/tutorials/pthreads/
Posix Threads (Pthreads) Interface

- Creating and reaping threads
  - pthread_create()
  - pthread_join()

- Terminating threads
  - pthread_exit()

- Synchronizing access to shared variables
  - pthread_mutex_(un)lock
  - pthread_cond_wait

More than 100 subroutines in pthreads library!
The `pthread_join()` subroutine blocks the calling thread until the specified thread's thread terminates.
The Pthreads "hello, world" Program

/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *run(void *vargp);

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, run, NULL);
    pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *run(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
Execution of Threaded “hello, world”

main thread

call Pthread_create()
Pthread_create() returns

call Pthread_join()

main thread waits for peer thread to terminate

Pthread_join() returns

exit()
terminates main thread and any peer threads

peer thread

printf()
return NULL;
(peer thread terminates)
int main(int argc, char **argv)
{
    int listenfd, connfd;
    pthread_t tid;

    listenfd = socket(...);
    bind(listenfd, ...);
    listen(listenfd, ...);
    while (1) {
        connfd = accept(listenfd, ...);
        pthread_create(&tid, NULL, echo_run, (void *)connfd);
        pthread_detach(tid); /* do not plan to do pthread_join*/
    }
}

void *echo_run(void *p)
{
    int connfd = (int)p;
    while (readline(connfd, ...) > 0)
        write(connfd, ...);
    close(connfd);
    return NULL;
}
Threaded Execution Model

- Multiple threads within single process
Pros and Cons of Thread-Based Designs

• + Easy to share data structures between threads
  – e.g., logging information, file cache.
• + Threads are more efficient than processes.

• – Unintentional sharing can introduce subtle race errors!
Approach #3: An Event-Based Server using I/O Multiplexing

- OS offers support to determine which fd(s) has input among a list of fds.
  - select, epoll
- Server maintains a set of active fds
  - Listenfd + many connfd’s
- Repeat:
  - Determine which fds have pending inputs
  - If listenfd has input, then accept connection
    - Add new connfd to set of active fds
  - If a connfd has input, service that connfd
- Details in book
Pros and Cons of I/O Multiplexing

+ One logical control flow.
+ No process or thread control overhead.
  - More complex to code than process- or thread-based designs.
  - Cannot take advantage of multi-core CPUs
    - Single thread of control
Conclusions: Approaches to Concurrency

- **Processes**
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients

- **Threads**
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable

- **I/O Multiplexing**
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core