Webserver
needs to deal w/ concurrent clients

listenfd is kinda like a file descriptor for a client
accepts messages from internet/network card

connectfd → talk to specific client

okay, but we can also just do this in a child process

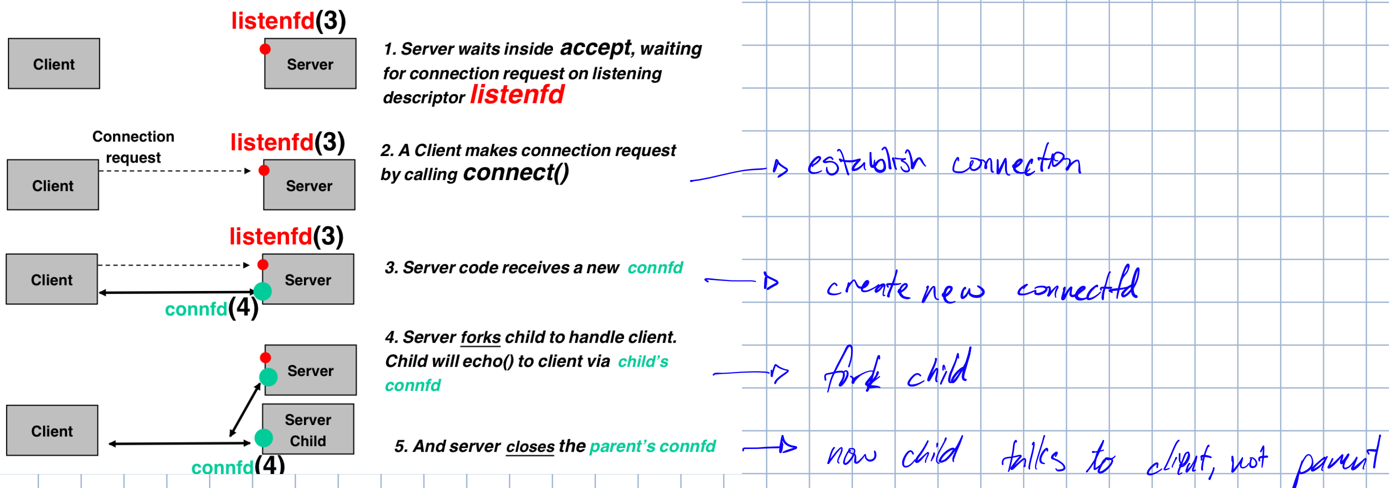
```
// 1st approach:
// Without fork (simple!)

while(1) {

  connfd = accept(listenfd, ...);

  echo(connfd); //write(connfd, "hello", ...)
  close(connfd);

}
```

FDs are I/O handlers (disk or network)

```
while (1) {
    connfd = Accept(liste
    if (Fork() == 0) {
        Close(listenfd);
        echo(connfd);
        Close(connfd);
        exit(0);
    }
    Close(connfd); /* Par
}
```

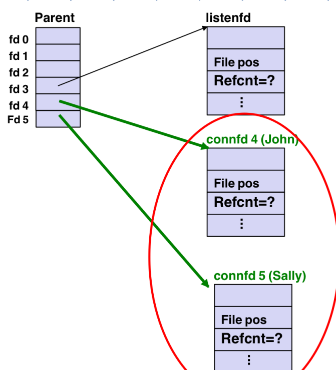
echo needs to be waiting until client acknowledges
can't do anything for other incoming requests

running echo in child process can run while parent accepts other connections



Memory leak!

- FD structures are not removed by the OS
  - (even if the childrens already exit)
  - Refcnt > 0
- Can't accept new connections
  - Process runs out of file descriptors
  - Mac: 256 FDs/process
  - Linux: 1024 FDs/process



gotta close child processes ... or else!

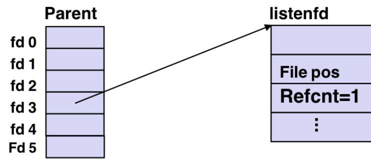
solution: close connectfd for each new connection. doesn't need pointers to its children

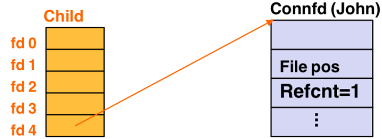
Parent only points to listenfd

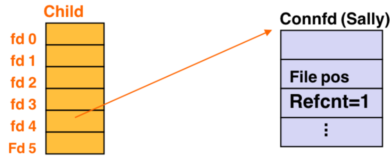
In this example, fd 4 is reused again and again

Each child points to its own connfd

connfd=4







cons of process-based concurrency

① can be overkill if children are short lived

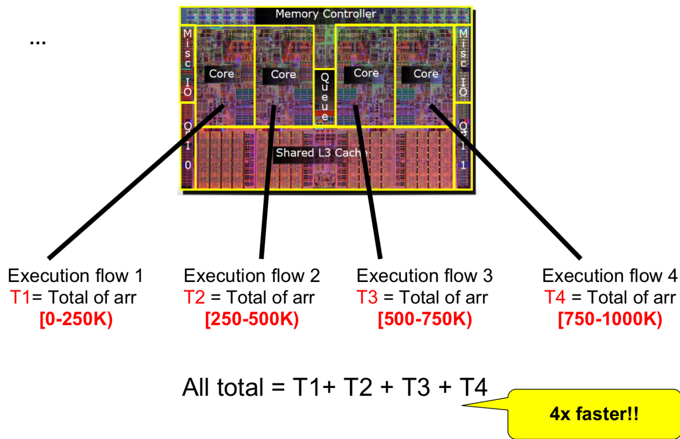
② very hard to communicate between children

Better to concurrency? → Threads!

parallelization! isn't concurrency → needs management

Exploiting Parallelism

...



All total = T1+ T2 + T3 + T4

4x faster!!

no sharing

thread allows many processes in a process
allows sharing

thread: lightweight process

process spawns multiple threads
thread only belongs to a process

A thread is a "lightweight process"

A process can spawn multiple threads

A thread only belongs to a process

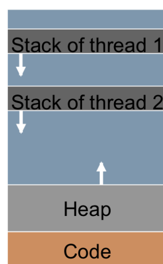
Sharing model:

Threads in a process can share:

- Global variables
- Heap
- File descriptor table
- ...

Each thread has its own:

- Program counter
- Stack
- ...

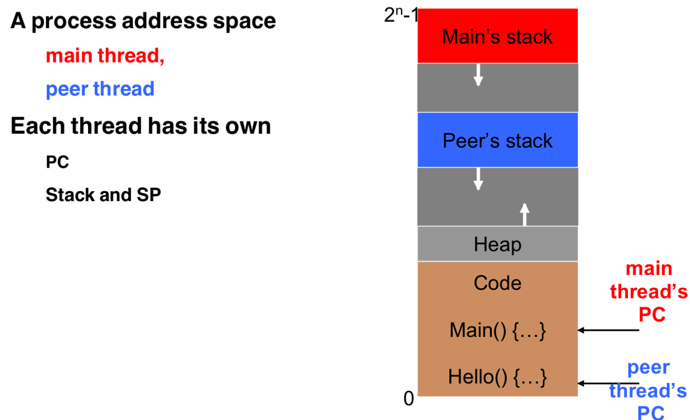


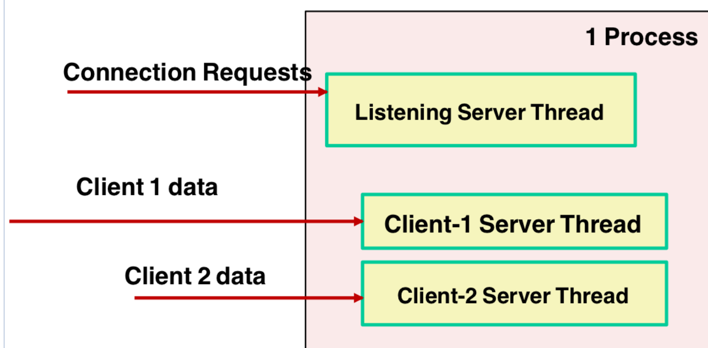
posix threads interface (pthreads)

pthread_create(&thread_id NULL function NULL)

hey OS, create thread for this process

Main & peer stack seperated





Multiple threads within single process

Some state between them

File descriptor table

How to pass connection fd argument to peer thread? 3 ways

① pass pointer to connectfd into f^n from main stack — not reliable
↳ *connectfd can change over time. data race

② use malloc to put info onto heap — good

③ pass connectfd value — pass via registers — good

Synchronization

Threads are mini processes created by process to use parallelism

- each have own stack
- PC's point to different address within same codesegment
- share data & heap

code races can mess things up

How to think about concurrency

1. Understanding shared variables
2. Understanding atomicity

Identifying shared variables

not just global variables

variables connected still shared, not global

→ Which lines in virtual memory space is accessible by which threads?

→ Follow the pointers!

3 types of variables

- global - created outside f^n
  - accessed by main + peer threads
  - reside in data segment, not stack
- local - created inside each f^n
  - reside in stack
  - accessed within f^n
    - → new for each f^n
- local static - declared within a f^n
  - resides in data segment
  - accessed by all threads running same f^n
    - → only 1 instance

```
#include <stdio.h>

void func() {
        static int x = 0;  // x is initialized only once
              // across three calls of func()
         x = x + 1;
        print(x);
}

int main(int argc, char * const argv[]) {
        func();     // x=1
        func();     // x=2
        func();     // x=3
        print(x);   // Doable??
         return 0;
}
```

NO, can't access it

a variable instance is shared iff multiple threads reference it

@ 20:00 for example

pointers can share thru their values

A variable instance x is shared iff multiple threads reference it

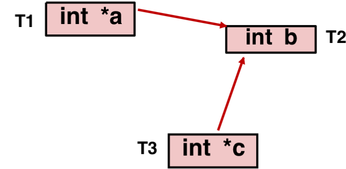
Example: "int b" in T2

Must track all pointers that point to the same data (the same memory location)

How about "a" and "c"?

Again, if a pointer, follow the pointer to the destination (the data/memory line)!!

.. then ask if the pointed variable is shared or not?



all of a.t1, b.t2, & c.t3 are shared by multiple threads

T_1 & T_2 can't see an int y = *c

Concurrency i++ problem

i++ is not atomic operation

example @ 33:00

race condition

solution: atomicity
certain segments of code won't be interrupted

"critical section"

mutual exclusion: only 1 thread in critical section at a time

Use some form of "locks" → semaphores

threads + sharing → locks

① allocate & initialize (lock)

② acquire the lock if available lock(lock)
put in sleep if lock is busy by OS

③ unlock(lock)

After lock has been allocated and initialized:

```
void increment() {
    lock(L);
    cnt++; // updating shared var, in CS
    unlock(L);
```

can create lock for each user

can use multiple locks update many variables

Synchronization vs Parallelism

Synchronization w/ semaphores

sem_* are the real functions

P() and V() are wrappers

Last lecture:	Real Usage in C:	This Lecture: (simplified)
lock *L;	sem_t *s = malloc(sem_t);	(same)
init(L)	sem_init(s, 0, value);	s = val;
lock(L)	sem_wait(s);	P(s)
unlock(L)	sem_post(s)	V(s)

Semaphore: non-negative global integer synchronization variable

① allocate space for global variable

② initialize (semaphore, 0 , 1)
idk ↑ ↑ available flag , only 1 thread available in critical section

③ P(semaphore) V(semaphore)

```
sem_t *s = malloc(sem_t);

sem_init(s, 0, 1);    // initialize s→val=1;
            // "1" means the lock is available
```

Semaphore: **sem_t s;**

non-negative global integer synchronization variable

(see man pages)

sem_init(s, ~~pshared~~, value)

Semaphore initially contains a non-negative integer value

User cannot read or write value directly after initialization

sem_wait(s) or P(s)

P() for "test" in Dutch (proberen)

Wait/block until value of sem is > 0, then decrement sem value

Details: put current thread to s→waitqueue;

(OS manage the wait and wake)

if val == 0, put in queue & tell it to sleep

decrement an available val to not available to execute stuff

sem_post(s) or V(s)

V() for "increment" in Dutch (verhogen)

Increment value of semaphore

Details:

wake(s) checks if someone is waiting on the s→waitqueue

set val available & awake someone from queue

binary semaphore good for mutual exclusion

Deadlock

every entity is waiting for resource held by another entity

overlapping locks

gist all pairs of mutexes
① is there an overlap? → not in same order? → deadlock possible

Users: faster!

old days: single threaded programs

Parallelism (multiprocessors) & concurrency (overlapping I/Os)

process is too heavy → threads

shared data + read/write conflicts → data races

Locks! atomicity & mutual exclusion

embarrassingly parallelize subtasks, don't synchronize/lock too often

wrong order of locks → deadlocks
use locks in right order