Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible
- Classical problem classes of concurrent programs:
  - Races: outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - Deadlock: improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - Livelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line
- Many aspects of concurrent programming are beyond the scope of 15-213

Echo Server Operation

Iterative Servers

Iterative servers process one request at a time
**Fundamental Flaw of Iterative Servers**

Client 1 blocks waiting for user to type in data

Solution: use concurrent servers instead

- Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

---

**Concurrent Servers (approach #1): Multiple Processes**

Client 1 blocks waiting for user to type in data

User goes out to lunch

Client 2 blocks waiting to complete its connection request until after lunch!

---

**Three Basic Mechanisms for Creating Concurrent Flows**

1. Processes
   - Kernel automatically interleaves multiple logical flows
   - Each flow has its own private address space

2. Threads
   - Kernel automatically interleaves multiple logical flows
   - Each flow shares the same address space

3. I/O multiplexing with select()
   - Programmer manually interleaves multiple logical flows
   - All flows share the same address space
   - Popular for high-performance server designs

---

**Review: Sequential Echo Server**

```c
int main(int argc, char **argv)
{
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);

    listenfd = Open_listenfd(port);
    while (1) {
        connfd = Accept(listenfd, &clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

- Accept a connection request
- Handle echo requests until client terminates
int main(int argc, char **argv) {
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);
    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(port);
    while (1) {
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
    }
}

Process-Based Concurrent Server

Fork separate process for each client
Does not allow any communication between different client handlers

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

Process Execution Model

Listening server process must reap zombie children
- to avoid fatal memory leak
Listening server process must close its copy of connfd
- Kernel keeps reference for each socket/open file
- After fork, refcnt(connfd) = 2
- Connection will not be closed until refcnt(connfd) == 0

Each client handled by independent process
- No shared state between them
- When child created, each have copies of listenfd and connfd
  - Parent must close connfd, child must close listenfd
**Pros and Cons of Process-Based Designs**

+ Handle multiple connections concurrently
+ Clean sharing model
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
+ Simple and straightforward
  - Additional overhead for process control
  - Nontrivial to share data between processes
  - Requires IPC (interprocess communication) mechanisms
    - FIFO’s (named pipes), System V shared memory and semaphores

**Approach #2: Multiple Threads**

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

**Traditional View of a Process**

Process = process context + code, data, and stack

<table>
<thead>
<tr>
<th>Program context:</th>
<th>Code, data, and stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data registers</td>
<td>stack</td>
</tr>
<tr>
<td>Condition codes</td>
<td></td>
</tr>
<tr>
<td>Stack pointer (SP)</td>
<td></td>
</tr>
<tr>
<td>Kernel context:</td>
<td>shared libraries</td>
</tr>
<tr>
<td>VM structures</td>
<td>run-time heap</td>
</tr>
<tr>
<td>Descriptor table</td>
<td>read/write data</td>
</tr>
<tr>
<td>brk pointer</td>
<td>read-only code/data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process context</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
</tr>
<tr>
<td>brk</td>
</tr>
<tr>
<td>PC</td>
</tr>
</tbody>
</table>

**Alternate View of a Process**

Process = thread + code, data, and kernel context

<table>
<thead>
<tr>
<th>Code and Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared libraries</td>
</tr>
<tr>
<td>run-time heap</td>
</tr>
<tr>
<td>read/write data</td>
</tr>
<tr>
<td>read-only code/data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread (main thread)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack pointer (SP)</td>
</tr>
<tr>
<td>Program counter (PC)</td>
</tr>
<tr>
<td>Data registers</td>
</tr>
<tr>
<td>Condition codes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM structures</td>
</tr>
<tr>
<td>Descriptor table</td>
</tr>
<tr>
<td>brk pointer</td>
</tr>
</tbody>
</table>
A Process With Multiple Threads

Multiple threads can be associated with a process
- Each thread has its own logical control flow
- Each thread shares the same code, data, and kernel context
  - Share common virtual address space (inc. stacks)
- Each thread has its own thread id (TID)

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

Kernel context:
- VM structures
- Descriptor table
- brk pointer

Logical View of Threads

Threads associated with process form a pool of peers
- Unlike processes which form a tree hierarchy

Threads associated with process foo

Process hierarchy

Concurrent Thread Execution

Two threads run concurrently (are concurrent) if their logical flows overlap in time
Otherwise, they are sequential

Examples:
- Concurrent: A & B, A&C
- Sequential: B & C

Threads vs. Processes

How threads and processes are similar
- Each has its own logical control flow
- Each can run concurrently with others
- Each is context switched

How threads and processes are different
- Threads share code and data, processes (typically) do not
- Threads are somewhat less expensive than processes
- Process control (creating and reaping) is twice as expensive as thread control
- Linux/Pentium III numbers:
  - ~20K cycles to create and reap a process
  - ~10K cycles (or less) to create and reap a thread
Posix Threads (Pthreads) Interface

Pthreads: Standard interface for ~60 functions that manipulate threads from C programs

- Creating and reaping threads
  - pthread_create()
  - pthread_join()
- Determining your thread ID
  - pthread_self()
- Terminating threads
  - pthread_cancel()
  - pthread_exit()
  - exit() [terminates all threads], RET [terminates current thread]
- Synchronizing access to shared variables
  - pthread_mutex_init
  - pthread_mutex_[un]lock
  - pthread_cond_init
  - pthread_cond_[timed]wait

The Pthreads "hello, world" Program

The Pthreads "hello, world" Program

```
#include "csapp.h"

void *thread(void *vargp);

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

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world\n");
  return NULL;
}
```

Execution of Threaded "hello, world"

Execution of Threaded "hello, world"

main thread waits for peer thread to terminate

```
exit() terminates
main thread and
any peer threads
```

Thread-Based Concurrent Echo Server

Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv) {
  int port = atoi(argv[1]);
  struct sockaddr_in clientaddr;
  int clientlen = sizeof(clientaddr);
  pthread_t tid;

  int listenfd = Open_listenfd(port);
  while (1) {
    int *connfdp = Malloc(sizeof(int));
    *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    pthread_create(&tid, NULL, echo_thread, connfdp);
  }

  return NULL;
}
```

- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of Malloc()!
Thread-Based Concurrent Server (cont)

/* thread routine */
void *echo_thread(void *vargp) {
  int connfd = *((int *)vargp);
Pthread_detach(pthread_self());
Free(vargp);
echo(connfd);
Close(connfd);
return NULL;
}

- Run thread in “detached” mode
  - Runs independently of other threads
  - Reaped when it terminates
- Free storage allocated to hold clientfd
  - “Producer-Consumer” model

Process Execution Model

- Multiple threads within single process
- Some state between them
  - File descriptors (in this example; usually more)

Potential Form of Unintended Sharing

while (1) {
  int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
Pthread_create(&tid, NULL, echo_thread, (void *) &connfd);
}

Issues With Thread-Based Servers

Must run “detached” to avoid memory leak
- At any point in time, a thread is either joinable or detached
- Joinable thread can be reaped and killed by other threads
  - must be reaped (with pthread_join) to free memory resources
- Detached thread cannot be reaped or killed by other threads
  - resources are automatically reaped on termination
- Default state is joinable
  - use pthread_detach(pthread_self()) to make detached

Must be careful to avoid unintended sharing.
- For example, what happens if we pass the address of connfd to the thread routine?
  - Pthread_create(stid, NULL, thread, (void *) &connfd);

All functions called by a thread must be thread-safe
- (next lecture)
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 and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads
  - (next lecture)

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