15-213
“The course that gives CMU its Zip!”

Concurrent Programming
November 13, 2008

Topics

- Limitations of iterative servers
- Process-based concurrent servers
- Threads-based concurrent servers
- Event-based concurrent servers
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

Client

- socket
- connect
- rio_readlineb
- rio_writen
- close

Server

- socket
- bind
- listen
- accept
- rio_readlineb
- rio_writen
- close

open_clientfd

open_listenfd

Connection request

Await connection request from next client

Client / Server Session
Iterative Servers

Iterative servers process one request at a time

client 1

- call connect
- ret connect
- call write
- ret write
- close

server

- call accept
- ret accept
- read
- close
- call accept
- ret accept
- read
- close

client 2

- call connect
- ret connect
- call write
- ret write
- close
- call write
- ret write
- close
Fundamental Flaw of Iterative Servers

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

Concurrent servers handle multiple requests concurrently

Client 1

- Call connect
- Ret connect
- Call fgets

User goes out to lunch

Client 1 blocks waiting for user to type in data

Server

- Call accept
- Ret accept
- Fork
- Call accept
- Ret accept

Child 1

- Call read

Child 2

- Call read
- Write
- Close
- End read
- Close

Client 2

- Call connect
- Ret connect
- Call fgets
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
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, (SA *)&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 */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
Process-Based Concurrent Server (cont)

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

- Reap all zombie children
Process Execution Model

- 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
Implementation Must-dos With Process-Based Designs

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, \texttt{refcnt(connfd)} = 2
- Connection will not be closed until \texttt{refcnt(connfd)} == 0
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

**Process context**
- Program context:
  - Data registers
  - Condition codes
  - Stack pointer (SP)
  - Program counter (PC)
- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer

**Code, data, and stack**
- stack
- shared libraries
- run-time heap
- read/write data
- read-only code/data

[Diagram showing memory segments and pointers]
Alternate View of a Process

Process = thread + code, data, and kernel context

Thread (main thread)

- Stack

Thread 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
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 (main thread)  Shared code and data  Thread 2 (peer thread)

- Data registers
- Condition codes
- SP1
- PC1
- shared libraries
- run-time heap
- read/write data
- read-only code/data
- 0
- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer

- Data registers
- Condition codes
- SP2
- PC2
- stack 1
- stack 2
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:
- T1
- T2
- T3
- T4
- T5

Shared code, data, and kernel context

Process hierarchy:
- P0
  - P1
    - foo
    - sh
    - sh
    - sh
    - bar
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

/*
 * hello.c - 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”

main thread

- call Pthread_create()
  Pthread_create() returns

peer thread

- call Pthread_join()

main thread waits for peer thread to terminate

- printf()
  return NULL;
  (peer thread terminates)

- Pthread_join() returns

- exit()
  terminates
  main thread and
  any peer threads

main thread waits for peer thread to terminate
Thread-Based Concurrent Echo Server

```c
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);
    }
}
```

- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of Malloc()!
  - Without corresponding Free()
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);
}

Why would both copies of vargp point to same location?
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(&tid, 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