Concurrent Programming

Prof. Jin-Soo Kim(jinsookim@skku.edu)
TA – Jinhong Kim(jinhong.kim@cs1.skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
int main (int argc, char *argv[]) {
    ...
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero((char *)&saddr, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port = htons(port);
    bind(listenfd, (struct sockaddr *)&saddr, sizeof(saddr));

    listen(listenfd, 5);
    while (1) {
        connfd = accept(listenfd, (struct sockaddr *)&caddr, &clen);
        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }
        close(connfd);
    }
}
Iterative Servers (1)

- One request at a time

client 1

- call connect
- call read
- close

server

- call accept
- write
- close

client 2

- call connect
- call read
- ret read
- close

- call accept
- ret accept
- write
- close
Iterative Servers (2)

- Fundamental flaw

- Solution: use concurrent servers instead
  - Use multiple concurrent flows to serve multiple clients at the same time.
Concurrent Programming

Thread-based
Traditional View

- Process = process context + address space

**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

SP

brk

PC

0
Alternate View

- Process = thread context + kernel context + address space

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
A Process with Multiple Threads

- Multiple threads can be associated with a process.
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread id (TID)

Thread 1 (main thread)

- Data registers
- Condition codes
- SP1
- PC1

Thread 2 (peer thread)

- 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

stack 1

stack 2
Logical View of Threads

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

Threads associated with process foo

Process hierarchy
Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow.
  - Each can run concurrently.
  - 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.
    - Linux 2.4 Kernel, 512MB RAM, 2 CPUs
      -> 1,811 forks()/second
      -> 227,611 threads/second (125x faster)
Pthreads Interface

- POSIX Threads Interface
  - Creating and reaping threads
    - pthread_create()
    - pthread_join()
  - Determining your thread ID
    - pthread_self()
  - Terminating threads
    - pthread_cancel()
    - pthread_exit()
    - exit (terminates all threads), return (terminates current thread)
  - Synchronizing access to shared variables
    - pthread_mutex_init()
    - pthread_mutex_[un]lock()
    - pthread_cond_init()
    - pthread_cond_[timed]wait()
    - pthread_cond_signal(), etc.
"hello, world" Program (1)

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "pthread.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;
}
```

Thread attributes (usually NULL)
Thread arguments (void *p)
return value (void **p)
Execution of threaded "hello, world"

- 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

- printf()
- return NULL;
- (peer thread terminates)
```c
int main (int argc, char *argv[]) {
    int *connfdp;
pthread_t tid;
    . . .

    while (1) {
        connfdp = (int *) malloc(sizeof(int));
        *connfdp = accept (listenfd, (struct sockaddr *)&caddr, &caddrlen));

        pthread_create(&tid, NULL, thread_main, connfdp);
    }
}

void *thread_main(void *arg) {
    int n;
    char buf[MAXLINE];

    int connfd = *((int *)arg);
pthread_detach(pthread_self());
free(arg);

    while((n = read(connfd, buf, MAXLINE)) > 0)
        write(connfd, buf, n);

    close(connfd);
    return NULL;
}
```
**Implementation Issues (1)**

- 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.
    - Exit state and return value are not saved.
  - Default state is joinable.
    - Use `pthread_detach(pthread_self())` to make detached.
Implementation Issues (2)

- Must be careful to avoid unintended sharing
  - For example, what happens if we pass the address connfd to the thread routine?

```c
int connfd;
...` pthread_create(&tid, NULL, thread_main, &connfd);
...```

- All functions called by a thread must be thread-safe.
  - A function is said to be thread-safe or reentrant, when the function may be called by more than one thread at a time without requiring any other action on the caller’s part.
Thread-based Designs

- **Pros**
  - Easy to share data structures between threads.
    - e.g., logging information, file cache, etc.
  - Threads are more efficient than processes.

- **Cons**
  - 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.
Concurrent Programming

Examples
Example

- Make chatting server
  - N client + 1 server
  - Server read messages from 1 client, send messages all client

- Use pthread_mutex