15213 Recitation Section C

Shimin Chen
Dec. 2, 2002

Outline

• Threads
• Synchronization
• Thread-safety of Library Functions
Important Dates

• Lab 7 Proxy: due on Thursday, Dec 5

• Final Exam: Tuesday, Dec 17
Concurrent Servers

- Iterative servers can only serve one client at a time
- Concurrent servers are able to handle multiple requests in parallel
- Required by L7 Part II
Three Ways for Creating Concurrent Servers

1. Processes
   - Fork a child process for every incoming client connection
   - Difficult to share data among child processes

2. Threads
   - Create a thread to handle every incoming client connection
   - Our focus today

3. I/O multiplexing with Unix `select()`
   - Use `select()` to notice pending socket activity
   - Manually interleave the processing of multiple open connections
   - More complex!
     - ~ implementing your own app-specific thread package!
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
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
A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow (instruction flow)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread ID (TID)
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.
    • 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 to create and reap a thread.
Posix Threads (Pthreads) Interface

- Standard interface for ~60 functions
  - 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`
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

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() returns

return NULL;

(peer thread terminates)
int main(int argc, char **argv)
{
    int listenfd, *connfdp, port, clientlen;
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}

Thread-Based Concurrent Server (cont)

```c
/* thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);

    echo_r(connfd); /* thread-safe version of echo() */
    Close(connfd);
    return NULL;
}
```
Issue 1: Detached Threads

- At any point in time, a thread is either \textit{joinable} or \textit{detached}.
- \textit{Joinable} thread can be reaped and killed by other threads.
  - must be reaped (with \texttt{pthread_join}) to free memory resources.
- \textit{Detached} thread cannot be reaped or killed by other threads.
  - resources are automatically reaped on termination.
- Default state is joinable.
  - use \texttt{pthread_detach(pthread_self())} to make detached.

- \textit{Why should we use detached threads?}
  - \texttt{pthread_join} blocks the calling thread
Issue 2: Avoid Unintended Sharing

```c
cnnfdp = Malloc(sizeof(int));
*connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
Pthread_create(&tid, NULL, thread, connfdp);
```

- For example, what happens if we pass the address of `connfd` to the thread routine as in the following code?

```c
cnnfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
Pthread_create(&tid, NULL, thread, (void *)&connfd);
```
Issue 3: Thread-safe

- Easy to share data structures between threads
- But we need to do this correctly!
- Recall the shell lab:
  - Job data structures
  - Shared between main process and signal handler
- Need ways to synchronize multiple control of flows
Threads Memory Model

- Conceptual model:
  - Each thread runs in the context of a process.
  - Each thread has its own separate thread context.
    - Thread ID, stack, stack pointer, program counter, condition codes, and general purpose registers.
  - All threads share the remaining process context.
    - Code, data, heap, and shared library segments of the process virtual address space.
    - Open files and installed handlers
Shared Variables in Conceptual Model

- global variables are shared
- stack variables are private

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

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
Caveats of Conceptual Models

- In practice, any thread can read and write the stack of any other thread.
- So one can use a global pointer to point to a stack variable. Then all threads can access the stack variable.
- But this is not a good programming practice.
- More details in this Thursday’s lecture.
Synchronization

• If multiple threads want to access a shared global data structure, we need to synchronize their accesses.

• Ways to do synchronization:
  – Semaphores
  – Mutex and conditions
  – Etc.
Synchronizing With Semaphores

- **Classic solution**: Dijkstra's P and V operations on *semaphores*.
  - *semaphore*: non-negative integer synchronization variable.
    - P(s): \[ \text{while} \ (s == 0) \ \text{wait}(); \ s--; \]
      - Dutch for "Proberen" (test)
    - V(s): \[ s++; \]
      - Dutch for "Verhogen" (increment)
  - OS guarantees that operations between brackets [ ] are executed indivisibly.
    - Only one P or V operation at a time can modify s.
    - When while loop in P terminates, only that P can decrements.
  - **Semaphore invariant**: \((s >= 0)\)
/* initialize semaphore sem to value */
/* pshared=0 if thread, pshared=1 if process */
void Sem_init(sem_t *sem, int pshared, unsigned int value) {
    if (sem_init(sem, pshared, value) < 0)
        unix_error("Sem_init");
}

/* P operation on semaphore sem */
void P(sem_t *sem) {
    if (sem_wait(sem))
        unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
    if (sem_post(sem))
        unix_error("V");
}
#include "csapp.h"
#define NITERS 10000000

unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;

    Sem_init(&sem, 0, 1);

    /* create 2 threads and wait */
    ......

    exit(0);
}

/* thread routine */
void *count(void *arg)
{
    int i;

    for (i=0;i<NITERS;i++){
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
Thread-safety of Library Functions

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe.
  - Examples: `malloc`, `free`, `printf`, `scanf`
- Most Unix system calls are thread-safe, with a few exceptions:

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Class</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>3</td>
<td>asctime_r</td>
</tr>
<tr>
<td>ctime</td>
<td>3</td>
<td>ctime_r</td>
</tr>
<tr>
<td>gethostbyaddr</td>
<td>3</td>
<td>gethostbyaddr_r</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>3</td>
<td>gethostbyname_r</td>
</tr>
<tr>
<td>inet_ntoa</td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td>localtime</td>
<td>3</td>
<td>localtime_r</td>
</tr>
<tr>
<td>rand</td>
<td>2</td>
<td>rand_r</td>
</tr>
</tbody>
</table>
Thread-Unsafe Functions: Fixes

- Return a ptr to a static variable.
- Fixes:
  1. Rewrite code so caller passes pointer to struct.
  - Issue: Requires changes in caller and callee.

```c
struct hostent
*gethostbyname(char name)
{
    static struct hostent h;
    <contact DNS and fill in h>
    return &h;
}

hostp = Malloc(...));
gethostbyname_r(name, hostp);
```
Thread-Unsafe Functions: Fixes

- Return a ptr to a `static` variable.
- Fixes:
  2. **Lock-and-copy**

  - Issue: Requires only simple changes in caller
  - However, caller must free memory.

```c
struct hostent
*gethostbyname(char name)
{
    static struct hostent h;
    <contact DNS and fill in h>
    return &h;
}

struct hostent
*gethostbyname_ts(char *p)
{
    struct hostent *q = Malloc(...);
    P(&mutex); /* lock */
    p = gethostbyname(name);
    *q = *p;   /* copy */
    V(&mutex);
    return q;
}
```
Summary

- Threading is a clean and efficient way to implement concurrent server
- We need to synchronize multiple threads for concurrent accesses to shared variables
  - Semaphore is one way to do this
  - Thread-safety is the difficult part of thread programming