CS4254
Computer Network Architecture and Programming
Dr. Ayman A. Abdel-Hamid
Computer Science Department
Virginia Tech

Outline
- Threads (Chapter 26)
  - Introduction
  - Basic Thread Functions
  - TCP echo client using threads
  - TCP Echo Server using threads
  - Thread Synchronization

Introduction
- Problems with fork approach for concurrency
  - Expensive
    - Memory copied from parent to child
    - All descriptors are duplicated in the child
    - Can be implemented using copy-on-write (don’t copy until child needs own copy)
  - IPC (Inter-process communication) required to pass information between parent and child after fork
- Threads can help!!
Introduction 3/4

• Threads

  ➢ Disadvantages
    ✓ Global variables are shared between threads →
      Inadvertent modification of shared variables can be
      disastrous (need for concurrency control)
    ✓ Many library functions are not thread-safe.
      □ Library functions that return pointers to internal static
        arrays are not thread safe
      □ To make it thread-safe → caller allocates space for the
        result and passes that pointer as argument to the function
    ✓ Lack of robustness: If one thread crashes, the whole
      application crashes

Introduction 4/4

• Threads

  ➢ State
    ✓ Threads within a process share global data: process
      instructions, most data, descriptors, etc, …
    □ File descriptors are shared. If one thread closes a file, all
      other threads can’t use the file
    ✓ Each thread has its own stack and local variables
    ✓ I/O operations block the calling thread.
    ✓ Some other functions act on the whole process. For
      example, the exit() function terminates the entire process
      and all associated threads.

Basic Thread Functions 1/6

• pthread_create function

  #include <pthread.h>
  int pthread_create (pthread_t * tid, const pthread_attr_t *attr, void
                     (*func) (void*), void *arg);
  //Returns 0 if OK, positive Exxx value on error
  ➢ When a program is started, single thread is created (main thread).
  Create more threads by calling pthread_create()
  ➢ pthread_t is often an unsigned int. returns the new thread ID
  ➢ attr: is the new thread attributes: priority, initial stack size, daemon
    thread or not. To get default attributes, pass as NULL
  ➢ func: address of a function to execute when the thread starts
  ➢ arg: pointer to argument to function (for multiple arguments,
    package into a structure and pass address of structure to function)
Basic Thread Functions

- **pthread_join function**
  ```c
  #include <pthread.h>
  int pthread_join (pthread_t tid, void ** status);
  //Returns 0 if OK, positive Exxx value on error
  
  - Wait for a given thread to terminate (similar to waitpid() for Unix processes)
  - Must specify thread ID (tid) of thread to wait for
  - If status argument non-null
  - Return value from thread (pointer to some object) is pointed to by status
  ```

- **pthread_self function**
  ```c
  #include <pthread.h>
  pthread_t pthread_self (void);
  //Returns thread ID of calling thread
  
  - similar to getpid() for Unix processes
  ```

- **pthread_detach function**
  ```c
  #include <pthread.h>
  int pthread_detach (pthread_t tid);
  //Returns 0 if OK, positive Exxx value on error
  
  - A thread is either joinable (default) or detached
  - When a joinable thread terminates -> thread ID and exit status are retained until another thread calls pthread_join
  - When a detached thread terminates -> all resources are released and can not be waited for to terminate
  - Example : pthread_detach (pthread_self());
  ```

- **pthread_exit function**
  ```c
  #include <pthread.h>
  void pthread_exit (void * status);
  //Does not return to caller
  
  - If thread not detached, thread ID and exit status are retained for a later pthread_join by another thread in the calling process
  - Other ways for a thread to terminate
    - Function that started the thread terminates, with its return value being the exit status of the thread
    - main function of process returns or any thread calls exit. In such case, process terminates including any threads
  ```
TCP Echo Client using Threads

- Recode `str_cli` function using threads
- Source code in `threads/strclithread.c`
  - Can test using `threads/tcpcli01.c`
  - `tcpcli01.c` uses `tcp_connect` function introduced in section 11.12
  - Need to pass host name and service
  - Can also test by `threads/tcpcli01_plain.c` → Pass server’s IP address
- If server terminates prematurely
  - `Readline` function returns 0 and `str_cli` function terminates
  - `main` function terminates calling `exit` → terminate all threads
- Alternative to using global data for threads to share?
  - See `threads/strclithread_args.c`, test with `threads/tcpcli01_plain_args.c`

TCP Echo Server using Threads 1/2

- One thread per client instead of one child process per client
- Source code in `threads/tcpser01.c`
- Uses `tcp_connect` function introduced in section 11.12
- Main processing loop
  ```c
  for (;;) {
    len = addrlen;
    connfd = Accept(listenfd, cliaddr, &len);
    Pthread_create(&tid, NULL, &doit, (void *) connfd);
  }
  ```
  - The casting `(void*) connfd` is OK on most Unix implementations (size of an integer is <= size of a pointer)
  - Is there an alternative approach?

TCP Echo Server using Threads 2/2

- If we pass the address of `connfd`, what can go wrong?
  ```c
  for (;;) {
    len = addrlen;
    connfd = Accept(listenfd, cliaddr, &len);
    Pthread_create(&tid, NULL, &doit, &connfd);
  }
  ```
  - A more correct approach would be to allocate space for the connected descriptor every time before the call to `accept`
  ```c
  for (;;) {
    len = addrlen;
    intptr = Malloc(sizeof(int));
    *intptr = Accept(listenfd, cliaddr, &len);
    Pthread_create(&tid, NULL, &doit, intptr);
  }
  ```
  // source code in `threads/tcpser02.c`

Thread Synchronization: Mutex

- How can a thread ensure that access/updates to shared variables are atomic?
- How can a thread ensure that it is the only thread executing some critical piece of code?
  - Need a mechanism for thread coordination and synchronization
  - `semaphores` and mutex calls
- Mutex: Mutual Exclusion
  - Threads can create a mutex and initialize it. Before entering a critical region, lock the mutex.
  - Unlock the mutex after exiting the critical region
  - See examples in `threads/example01.c` and `threads/example02.c`
Thread Synchronization: Semaphores

- A **mutex** allows one thread to enter a critical region
- A **semaphore** can allow some \( N \) threads to enter a critical region
  - Used when there is a limited (but more than 1) number of copies of a shared resource
- Can be dynamically initialized
  - Thread calls a semaphore **wait** function before it enters a critical region
- Semaphore is a generalization of a **mutex**

Thread Synchronization: Condition Variables 1/2

- A condition variable is only needed where
  - A set of threads are using a **mutex** to provide mutually exclusive access to some resource
  - Once a thread acquires the resource, it needs to wait for a particular condition to occur
- If no condition variables are available
  - Some form of *busy waiting* in which thread repeatedly acquires the **mutex**, tests the condition, and then releases the **mutex** (wasteful solution)
- A condition variable allows a thread to release a **mutex** and block on a condition atomically

Thread Synchronization: Condition Variables 2/2

- Acquire a **mutex**
- Call **pthread_cond_wait** specifying both a condition variable and the mutex being held
- Thread blocks until some other thread signals the variable
- Two forms of signaling
  - Allow one thread to proceed, even if multiple threads are waiting on the signaled variable
    - OS simultaneously unblocks the thread and allows the thread to reacquire the **mutex**
  - Allow all threads that are blocked on the variable to proceed
  - Blocking on a condition variable does not prevent others from proceeding through the critical section, another thread can acquire the **mutex**