456: Network-Centric Programming

Yingying (Jennifer) Chen

Web: https://www.winlab.rutgers.edu/~yychen/
Email: yingche@scarletmail.rutgers.edu

Department of Electrical and Computer Engineering
Rutgers University

Threads & Thread Synchronization

Threads vs. Processes

- Threads are “lightweight” processes
  - Represent an execution path through a program
- One process can contain many threads
  - If process terminates, all threads terminate
  - Process resources (e.g. file descriptors area shared among threads)
- Threads
  - Creation, context switching is faster
  - Share same address space

Advantages over Sequential Program

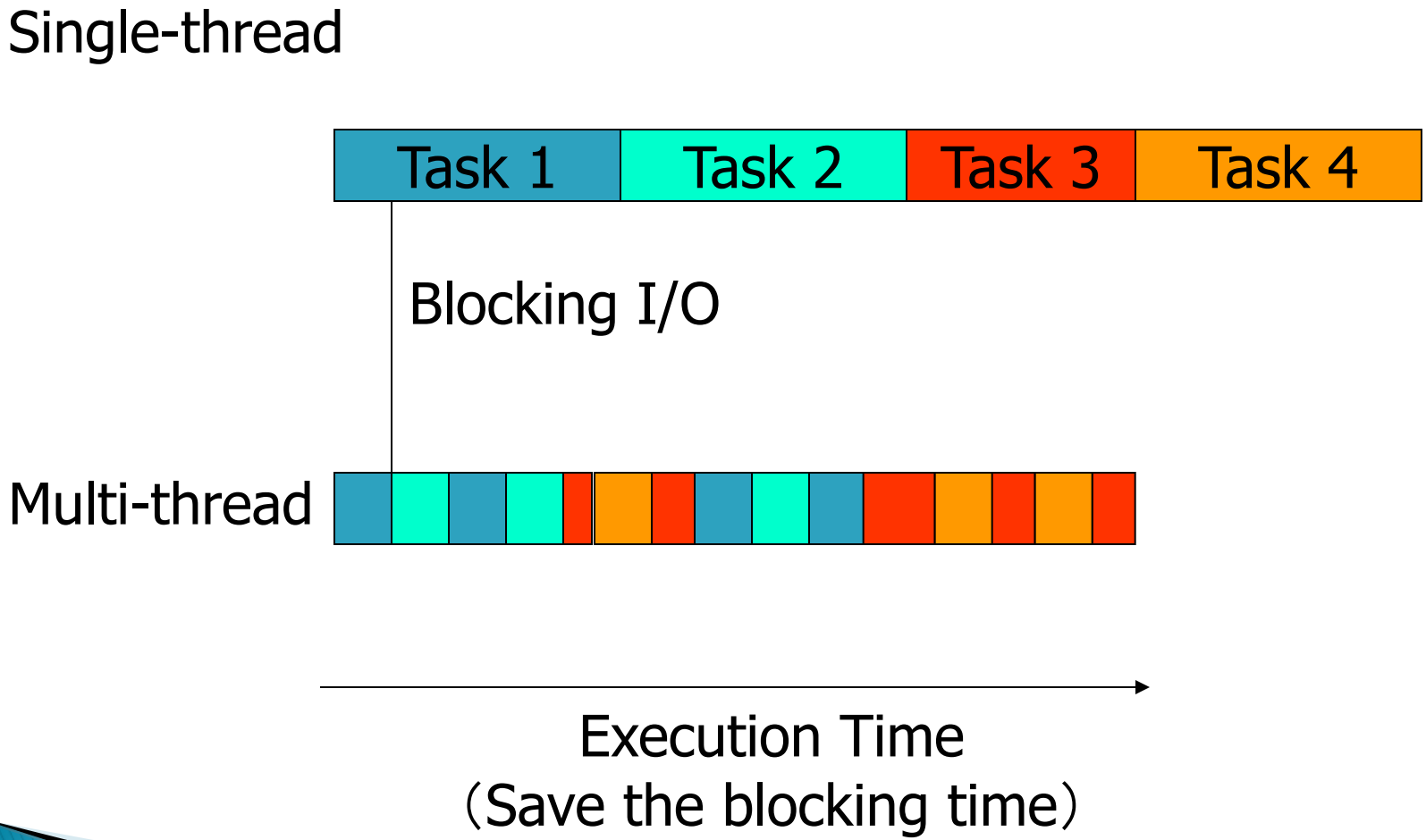
- Improve application responsiveness
  - Webserver answering requests
  - GUI responding to user input
- More efficient use of resources
  - Multiprocessor machines (e.g., Macbook Pro with 4-core CPU)
  - “Hyperthreading”
    - Intel's proprietary simultaneous multithreading (SMT) implementation used to improve parallelization of computations performed on x86 microprocessors.

Using Thread or Using Process

- A thread can do anything a process can do, and a thread could be considered a ‘lightweight’ process.
  - Since process can consist of multiple threads, a thread could be considered a ‘lightweight’ process.

- The essential difference between a thread and a process:
  - Threads are used for small tasks, whereas processes are used for more ‘heavyweight’ tasks – basically the execution of applications.

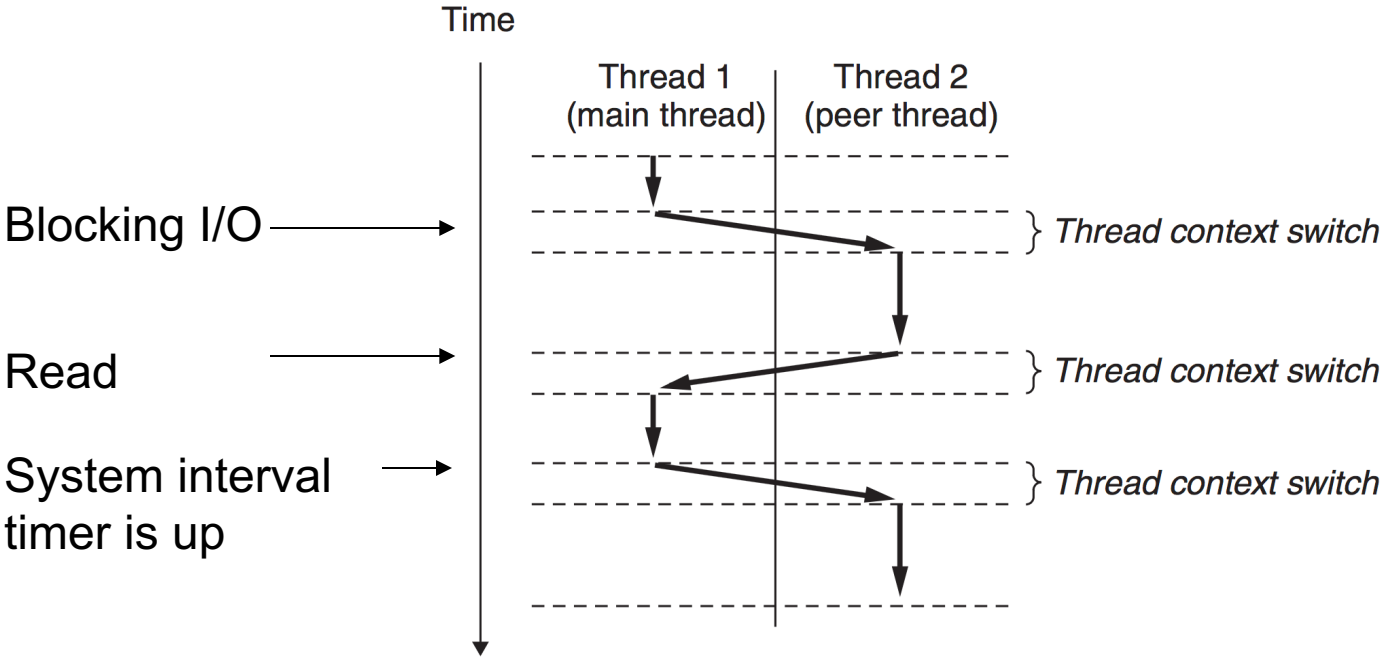
Interleaved Execution



Thread Context Switch

- A procedure that CPU follows to change from one thread to another while ensuring no conflicts.
  - Store the state of a thread (e.g., registers and memory maps) so that it can be restored and execution resumed from the same point later
  - Pause the thread and resume another thread
  - Essential feature of a multitasking operating system
- When to Switch
  - Multi-threading
  - When a thread is interrupted

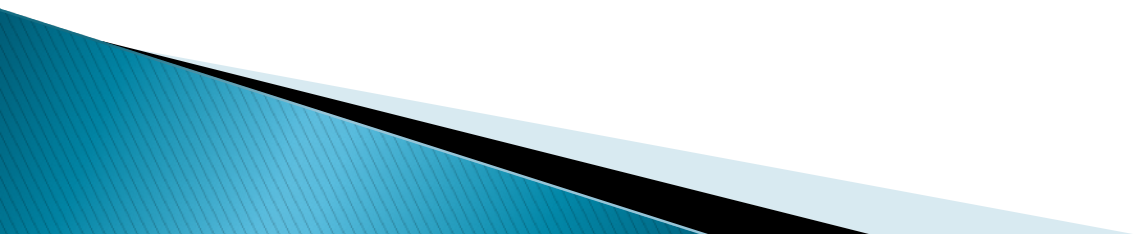
Concurrent Thread Execution



- Each process begins life as a single thread called the *main thread*.
- At some point, the main thread creates a *peer thread*, and the two threads run concurrently.
- Thread context switch
  - Control passes to the peer thread via a context switch, because the thread executes a blocking I/O (or slow system call such as read, sleep or system's interval timer)

Posix threads (Pthreads)

- Posix threads (Pthreads) is a standard interface for manipulating threads from C programs.
- It was adopted in 1995 and is available on most Unix systems.



Creating Threads

- int pthread_create(tid, attr, function pointer, arg)
  - *tid*: Thread id (also accessible within thread by *pthread_self*())
  - *attr*: Can be used to change the default attributes of the newly created thread (we will always call pthread_create with a NULL *attributes* argument)
  - *function pointer*: Expects function to execute
  - *arg*: Arguments passed as void pointer

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                   func *f, void *arg);
```

Returns: 0 if OK, nonzero on error

Returning thread ID

- When pthread_create returns, argument tid contains the ID of the newly created thread. The new thread can determine its own thread ID by calling the pthread_self function

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns: thread ID of caller

Terminating Threads

- When function returns
- Explicit call to *pthread_exit()*
- When process terminates

```
#include <pthread.h>

void pthread_exit(void *thread_return);
```

Returns: 0 if OK, nonzero on error

Waiting for Thread Termination

- thread_join(pthread_t tid, void **thread_return)
  - The pthread_join function blocks until thread *tid* terminates, assigns the generic (void *) pointer returned by the thread
  - Can only wait for specific thread for which id is known

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```

Returns: 0 if OK, nonzero on error

Waiting for Thread Termination

- Detached thread: exit status not retained
  - pthread_detach(tid)
  - Detaches an existing thread

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

Returns: 0 if OK, nonzero on error

pthread_join() vs. pthread_detach()

- **pthread_join()**
  - It is called from another thread (usually the thread that created it)
  - Wait for a thread to terminate and obtain its return value

- **pthread_detach()**
  - Can be called from either the thread itself or another thread
  - Indicate that you do not want the thread's return value or wait for it to finish.

Example (Thread prints hello world): lecture_8_code/pthread_hello/helloworld.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *function_hello(void *vargp) /* Thread routine */
{
    printf("Hello, world!\n");
    printf("Thread id = %d \n", (int)pthread_self()); //print out the current thread ID
    return NULL;
}

int main(){
    pthread_t tid;
    //create a thread running thread function
    pthread_create(&tid, NULL, function_hello, NULL);
    printf("Waiting for the thread %d to terminate.\n", (int)tid);
    pthread_join(tid, NULL); //waits for the termination
    exit(0);
}
```

Terminal:

$./main

Example (Thread returns value): lecture_8_code/pthread_return_values/main.c

```
#include <pthread.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void *function_integer(void *arg) {
    printf("Entering the function_integer() via thread: %d.\n", (int)pthread_self());
    printf("function_integer() returns.\n");
    return((void *)1);
}

void *function_string(void *arg) {
    printf("Entering the function_string() via thread: %d.\n", (int)pthread_self());
    printf("function_string() returns.\n");
    return((void *)"Hello");
}
```

Example (Thread returns value): lecture_8_code/pthread_return_values/main.c

```
int main()
{
    pthread_t tid1,tid2;
    void *tret;
    pthread_create(&tid1,NULL,function_integer,NULL); //create thread 1
    printf("Created thread: %d\n",(int)tid1);
    pthread_create(&tid2,NULL,function_string,NULL); //create thread 2
    printf("Created thread: %d\n",(int)tid2);

    pthread_join(tid1,&tret); //waiting for the thread 1 termination
    printf("Main process ID: %d. Thread 1's return value: %d\n",(int)getpid(),(int)tret);
    pthread_join(tid2,&tret); //waiting for the thread 2 termination
    printf("Main process ID: %d. Thread 2's return value: %s\n",(int)getpid(),(char *)tret);

    exit(0);
}
```

Terminal:

$./main

Example (pthread_exit returns value): lecture_8_code/pthread_exit_return/main.c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void *function_integer(int s)
{
	printf("Entering function_integer() via thread: %d.\n", (int)pthread_self());
	printf("%d\n",s);
	pthread_exit("Terminate the current thread inside of the function_integer().");
	//terminate the current thread
}

void *function_string(char s[])
{
	printf("Entering function_string() via thread: %d.\n", (int)pthread_self());
	printf("%s\n",s);
	pthread_exit("Terminate the current thread inside of the function_string().");
	//terminate the current thread
}
```

Example (pthread_exit returns value): lecture_8_code/pthread_exit_return/main.c

```
int main(void) {
        pthread_t id1,id2;
        void *a1,*a2;
        int i,ret1,ret2;
        int s1=1000;                    //input arguments
        char s2[]="Hello world!"; //input arguments
        ret1=pthread_create(&id1,NULL, function_integer,(void *)s1);
        printf("Created thread: %d\n", (int)id1);
        ret2=pthread_create(&id2,NULL, function_string,(void *) s2);
        printf("Created thread: %d\n", (int)id2);

        if(ret1!=0){
                printf ("Create pthread1 error!\n");
                exit (1);
        }
        printf("Main thread is waiting for the thread %d to terminate.\n", (int)id1);
        pthread_join(id1,&a1);
```

Example (pthread_exit returns value): lecture_8_code/pthread_exit_return/main.c

```
        printf("return value from thread 1: %s\n",(char*)a1);

        if(ret2!=0){
                printf ("Create pthread2 error!\n");
                exit (1);
        }
        printf("Main thread is waiting for the thread %d to terminate.\n", (int)id2);
        pthread_join(id2,&a2);
        printf("return value from thread 2: %s\n",(char*)a2);
        return (0);
}
```

Terminal:

$./main

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/server.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    int server_fd, *new_socket;
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    pthread_t tid;
```

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/server.c

```
// Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0){
        perror("socket failed");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = htonl (INADDR_ANY);
    address.sin_port = htons( PORT );
    // Attaching socket to the port 8080
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))<0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 1024) < 0) { //listen to the socket
        perror("listen");
        exit(EXIT_FAILURE);
    }
```

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/server.c

```
while(1){
        printf("Waiting for a client to connect...\n");
        new_socket = malloc(sizeof(int));
        if ((*new_socket = accept(server_fd, (struct sockaddr *)&address,
                        (socklen_t*)&addrlen))<0)
        {
            perror("accept");
            exit(EXIT_FAILURE);
        }

        pthread_create(&tid, NULL, thread_handle, new_socket);
        pthread_detach(tid);
    }

    return 0;
}
```

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/server.c

```
void *thread_handle(void *vargp){
    printf("Entering the thread: %d\n", (int)pthread_self());
    char buffer[1024] = {0};
    char *response = "Response from server";
    int connfd = *((int *)vargp);
    int valread = read( connfd , buffer, 1024); //receive the message from client
    printf("%s\n",buffer );
    write(connfd, response, strlen(response)); //send response from server
    printf("Response msg sent from the thread: %d.\n", (int)pthread_self());
    free(vargp);
    close(connfd);
    return NULL;
}
```

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <unistd.h>
 #include <arpa/inet.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    struct sockaddr_in address;
    int sock1, sock2, valread;
    struct sockaddr_in serv_addr;
    char *hello1 = "Hello from client 1";
    char *hello2 = "Hello from client 2";
    char buffer[1024] = {0};
```

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/client.c

```
if ((sock1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }
    if ((sock2 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }
    // initialize serv_addr
    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);
```

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/client.c

```
if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)  {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }
    if (connect(sock1, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }
   // send messages via two sockets: sock1
    write(sock1 , hello1 , strlen(hello1));                    //send msg via sock1
    printf("Hello message sent from client 1\n");
    valread = read(sock1, buffer1, 1024);                      //receive msg from server
    printf("%s\n", buffer1);
```

Example (Concurrent server via pthread): lecture_8_code/concurrent_server_thread/client.c

```
    if (connect(sock2, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0){
        printf("\nConnection Failed \n");
        return -1;
    }
    // send messages via two sockets: sock2
    write(sock2 , hello2 , strlen(hello2));                    //send msg via sock2
    printf("Hello message sent from client 2\n");
    valread = read(sock2, buffer2, 1024);                      //receive msg from server
    printf("%s\n", buffer2);
    return 0;
}
```

Terminal 1
$ ./server

Terminal 2
$ ./client

Terminal 3
$ ./client

The server can respond to multiple clients' requests concurrently through creating threads

Race Conditions

- Unpredictable interleaved execution can lead to race conditions when accessing shared resources
  - Between processes (e.g., file access)
  - Between threads (global, static variables)
- Thread libraries provide solutions for threads

Writing Thread-Safe Functions

- Problem: Function refers to global or static variables

Solutions?

- Local variables are safe, because each Thread has own stack

Mutex

- Mutual exclusion lock – only one thread can acquire the lock at any given time
  - Other threads that try to lock will block
- int pthread_mutex_lock(pthread_mutex_t *mutex)
- int pthread_mutex_unlock(pthread_mutex_t *mutex)

The macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutex that are statically allocated.

pthread_mutex_t *mutex* = **PTHREAD_MUTEX_INITIALIZER;**

Example (mutex example): lecture_8_code/mutex/main.c

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define NLOOP 500
/*
Two threads: each thread will increase the global counter NLOOP times, and
print it out with the thread ID.
*/
int counter;     /* this is incremented by the threads */
//mutex initialization
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Example (mutex example): lecture_8_code/mutex/main.c

```
void *doit(void *vptr) {
  printf("Entering the thread %d\n", (int)pthread_self());
  int   i, val;
  for (i = 0; i < NLOOP; i++) {
    pthread_mutex_lock(&counter_mutex);
    val = counter;
    printf("%d: %d\n", (int)pthread_self(), val + 1);
    counter = val + 1;
    pthread_mutex_unlock(&counter_mutex);
  }
  return(NULL);
}
```

Example (mutex example): lecture_8_code/mutex/main.c

```
int main(int argc, char **argv) {
 pthread_t tidA, tidB;
 pthread_create(&tidA, NULL, doit, NULL);//create thread 1
 printf("Created thread %d\n", (int)tidA);
 pthread_create(&tidB, NULL, doit, NULL);//create thread 2
 printf("Created thread %d\n", (int)tidB);

 /*wait for both threads to terminate */
 pthread_join(tidA, NULL);
 printf("Thread %d is terminated.\n", (int)tidA);
 pthread_join(tidB, NULL);
 printf("Thread %d is terminated.\n", (int)tidB);

 exit(0);
}
```

Terminal

```
$ ./main
```

Example (non_mutex example): lecture_8_code/non_mutex/main.c

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define NLOOP 500
/*
Two threads: each thread will increase the global counter NLOOP times, and
print it out with the thread ID.
*/
int counter;     /* this is incremented by the threads */
//mutex initialization
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Example (non_mutex example): lecture_8_code/non_mutex/main.c

```
void *doit(void *vptr) {
  printf("Entering the thread %d\n", (int)pthread_self())
  int   i, val;
  for (i = 0; i < NLOOP; i++) {
    //pthread_mutex_lock(&counter_mutex);
    val = counter;
    printf("%d: %d\n", (int)pthread_self(), val + 1);
    counter = val + 1;
    //pthread_mutex_unlock(&counter_mutex);
  }
  return(NULL);
}
```

Example (non_mutex example): lecture_8_code/non_mutex/main.c

```
int main(int argc, char **argv) {
 pthread_t tidA, tidB;
 pthread_create(&tidA, NULL, doit, NULL);//create thread 1
 printf("Created thread %d\n", (int)tidA);
 pthread_create(&tidB, NULL, doit, NULL);//create thread 2
 printf("Created thread %d\n", (int)tidB);

 /*wait for both threads to terminate */
 pthread_join(tidA, NULL);
 printf("Thread %d is terminated.\n", (int)tidA);
 pthread_join(tidB, NULL);
 printf("Thread %d is terminated.\n", (int)tidB);

 exit(0);
}
```

Terminal

$ ./main

Homework Readings

- CSAPP: Computer Systems – A Programmer's Perspective
  - Chapter 12.1, 12.3, 12.5
- ALP: Advanced Linux Programming
  - Chapter 8.3

Assignment 3

Concurrent Web Proxy

3 weeks

Due on 4/22

Problem

- You are required to upgrade your sequential Basic Web Proxy of Assignment 2 so that it uses threads or processes to deal with multiple clients concurrently.

- You will be required to implement **two versions** of the concurrent web proxy:
  - **A threaded version and a multi-process version**

Implementation

- Real proxies do not process requests sequentially. They deal with multiple requests concurrently. Once you have a working sequential logging proxy (Assignment 2), you should alter it to handle multiple requests concurrently. Following are two approaches to achieve concurrency:

- **Process based** using *file locking* for synchronization: Create a new process to deal with each new connection request that arrives.
- **Thread based** using *mutexes* for synchronization: Create a new thread to deal with each new connection request that arrives.