Threading, Parallelism and Networking

Slides inspired from here and the awesome book by Remzi

Plan

- Why threads
- Basics of Pthreads API
- Introducing Locks, Conditional variables and Semaphores
- A common deadlock scenario
- Basics of networking and SOCKET API
- If time permits: discussing a popular synchronization problem

Process vs. threads

- Parent P forks a child C
  - P and C do not share any memory
  - Need complicated IPC mechanisms to communicate
  - Extra copies of code, data in memory
- Parent P executes two threads T1 and T2
  - T1 and T2 share parts of the address space
  - Global variables can be used for communication
  - Smaller memory footprint
- Threads are like separate processes, except they share the same address space

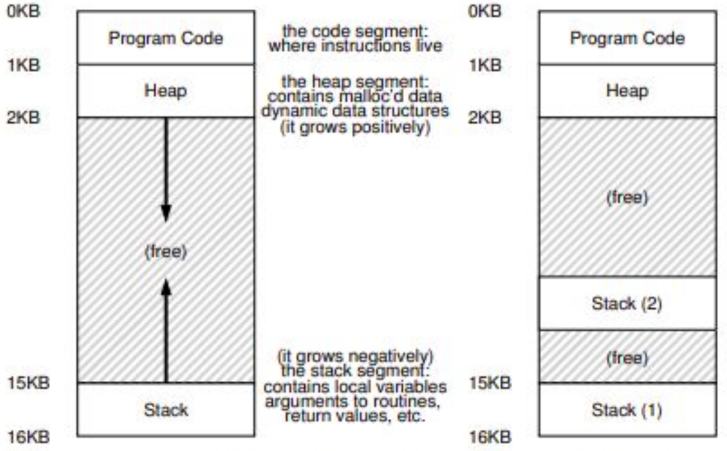


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

Why threads?

- Parallelism: a single process can effectively utilize multiple CPU cores
  - Understand the difference between concurrency and parallelism
  - **Concurrency:** running multiple threads/processes at the same time, **even on single CPU core**, by interleaving their executions
  - **Parallelism:** running multiple threads/processes in parallel over **different CPU cores**
- Even if no parallelism, concurrency of threads ensures effective use of CPU when one of the threads blocks (e.g., for I/O)

Creating a Thread

- `int pthread_create(pthread_t *tID, const pthread_attr_t *attr, void *(*thread_func)(void*), void *arg);`
- Create a new thread that will execute thread_func, passing it arg.
    - To pass the function multiple arguments, put them in a struct and pass a pointer to the struct
    - `*attr` – set thread attributes, `NULL` for default settings
- returns 0 if successful, an error number otherwise
    - updates `tID` with the thread ID of the new thread if successful

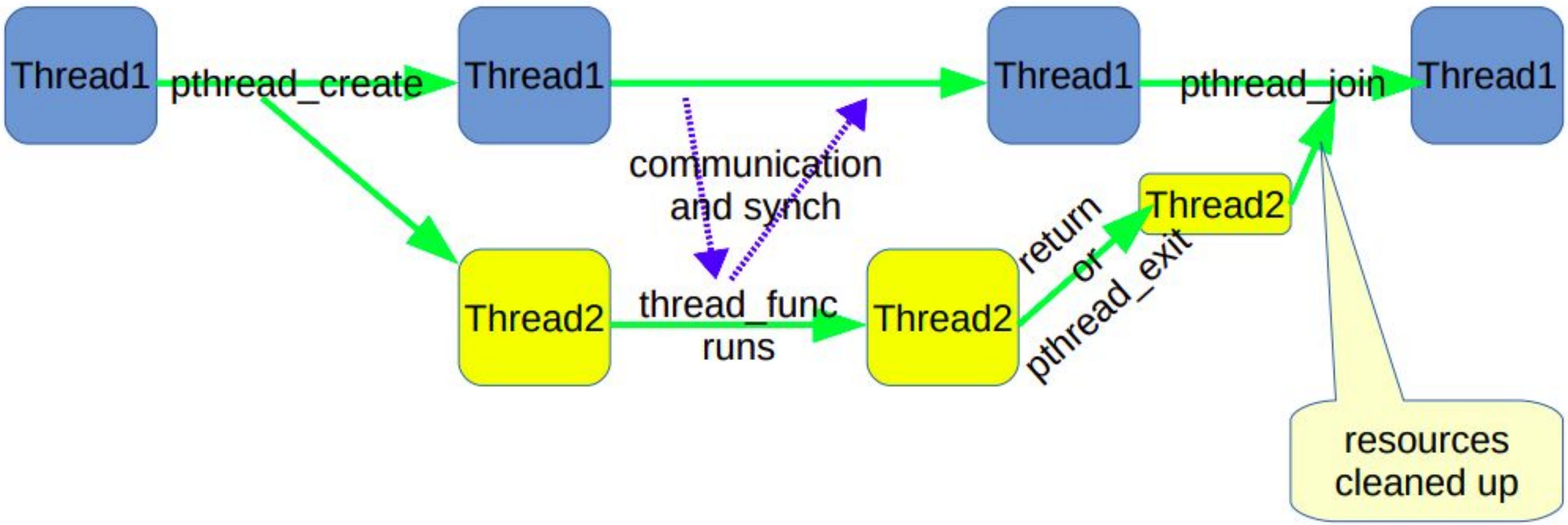
Creating threads using pthreads API

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: **Simple Thread Creation Code (t0.c)**


Thread1
pthread_create
Thread1
Thread1
pthread_join
Thread1
communication
and synch
Thread2
Thread2
thread_func
runs
Thread2
return
or
pthread_exit
resources
cleaned up

Threads incrementing a counter without mutual exclusion

```c

void *inc_counter(void *inp)
{
    int thread_idx = ((struct thread_details *)inp)->idx;

    // pthread_mutex_lock(&cnt_lock);
    counter++;
    // pthread_mutex_unlock(&cnt_lock);
    return NULL;
}

int main()
{

    const int TIMES = 1000;
    pthread_t thread_ids_arr[TIMES];

    pthread_mutex_init(&cnt_lock, NULL);

    for (int i = 0; i < TIMES; i++)
    {
        pthread_t curr_tid;
        td *thread_input = (td *)(malloc(sizeof(td)));
        thread_input->idx = i;
        pthread_create(&curr_tid, NULL, inc_counter, (void *)(thread_input));
        thread_ids_arr[i] = curr_tid;
    }
    for (int i = 0; i < TIMES; i++)
    {
        pthread_join(thread_ids_arr[i], NULL);
    }

    pthread_mutex_destroy(&cnt_lock);
    printf("Value of counter is: %d\n", counter);
    return 0;
}
```

Why is this happening ?

OS	Thread 1	Thread 2	(after instruction) PC	eax	counter
	before critical section		100	0	50
	`mov 8049a1c,%eax`		105	**50**	50
	`add $0x1,%eax`		108	**51**	50
interrupt					
save T1					
restore T2			100	0	50
		`mov 8049a1c,%eax`	105	**50**	50
		`add $0x1,%eax`	108	**51**	50
		`mov %eax,8049a1c`	113	51	**51**
interrupt					
save T2					
restore T1			108	51	51
	`mov %eax,8049a1c`		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

Locks: Basic idea

- Consider update of shared variable

  counter = counter + 1

- We can use a special lock variable to protect it

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

- All threads accessing a critical section share a lock
- One threads succeeds in locking – owner of lock
- Other threads that try to lock cannot proceed further until lock is released by the owner
- Pthreads library in Linux provides such locks

Syntax for locks

- **Declaration**
  - `pthread_mutex_t lock;`
- **Initialization**
  - `pthread_mutex_init(&lock, NULL)`
- **Locking the mutex**
  - `pthread_mutex_lock(&lock)`
- **Unlocking the mutex**
  - `pthread_mutex_unlock(&lock);`
- **Destroying the mutex**
  - `pthread_mutex_destroy(&lock);`

Another type of synchronization

- Locks allow one type of synchronization between threads - mutual exclusion
- Another common requirement in multi-threaded applications - waiting and signalling
  - Example: Thread T1 wants to continue only after Thread T2 has finished some task
- Can accomplish this task by busy waiting on some variable - Inefficient
- Need a new synchronisation primitive - Condition variables

Condition Variables

- A condition variable (CV) is a queue that a thread can put itself into when waiting on some condition
- Another thread that makes the condition true can signal the CV to wake up the waiting thread
- Pthreads provides CV for user programs
  - OS has a similar functionality of wait/signal for kernel threads
- Signal wakes up one thread, signal broadcast wakes up all waiting threads

Parent waiting for Child - Condition Variables

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Why use lock when calling wait?

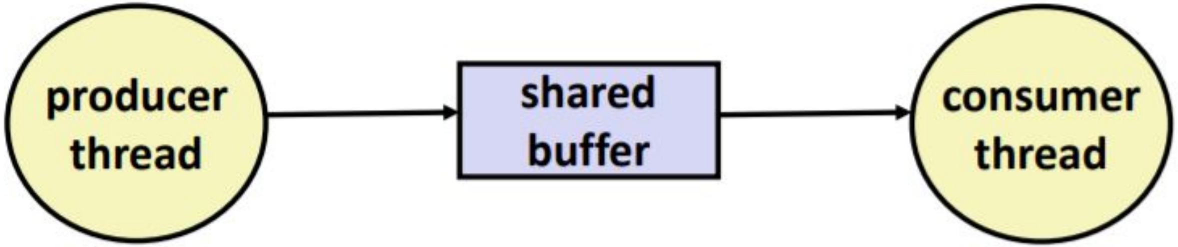
What if no lock is held when calling wait/signal?

- Race condition : missed wakeup
  - Parent checks done to be 0, decides to sleep, interrupted
  - Child runs, sets done to 1, signals, but no one sleeping yet
  - Parent now resumes and goes to sleep forever
- Lock must be held while calling wait and signal with CV
- The wait function releases the lock before putting thread to sleep, so lock is available for signaling thread

What is a semaphore?

- Synchronization primitive like conditional variables
- Semaphore is a variable with an underlying counter
- Two functions on a semaphore variable
  - Up/post increments the counter
  - Down/wait decrements the counter and blocks the calling thread if the resulting value is negative
- A semaphore with init value 1 acts as a simple lock
- **What does the current value of a semaphore mean?**
  - Positive : Number of threads that can decrement without blocking
  - Negative : Number of threads that have been blocked and are waiting
  - Zero : No threads are waiting, but if a thread tries to decrement it will block

Producer-Consumer Problem



- Producer waits for empty slot, inserts item in buffer, and notifies consumer
- Consumer waits for item, removes it from buffer, and notifies producer

Solution using Semaphores

- Need two semaphores for signaling
  - One to track empty slots, and make producer wait if no more empty slots
  - One to track filled slots, and make consumer wait if no more filled slots
- One semaphore to act as mutex for buffer

Incorrect, why?

Initial Values

- `empty` – Capacity of buffer
- `Full` – 0

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);          // Line P0 (NEW LINE)
        sem_wait(&empty);          // Line P1
        put(i);                    // Line P2
        sem_post(&full);           // Line P3
        sem_post(&mutex);          // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);          // Line C0 (NEW LINE)
        sem_wait(&full);           // Line C1
        int tmp = get();           // Line C2
        sem_post(&empty);          // Line C3
        sem_post(&mutex);          // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

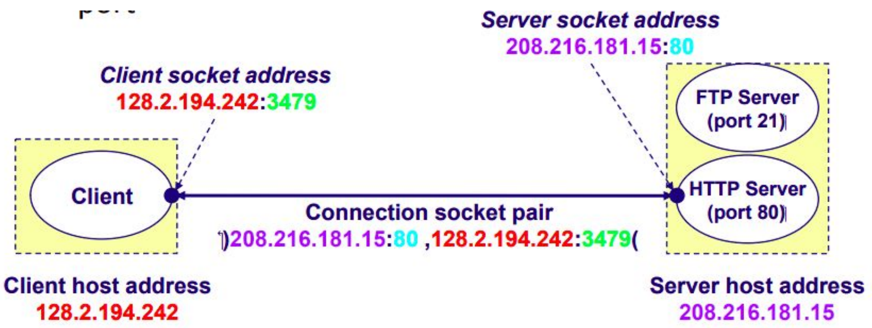
Correct Solution

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // Line P1
        sem_wait(&mutex);           // Line P1.5 (MUTEX HERE)
        put(i);                     // Line P2
        sem_post(&mutex);           // Line P2.5 (AND HERE)
        sem_post(&full);            // Line P3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);            // Line C1
        sem_wait(&mutex);           // Line C1.5 (MUTEX HERE)
        int tmp = get();            // Line C2
        sem_post(&mutex);           // Line C2.5 (AND HERE)
        sem_post(&empty);           // Line C3
        printf("%d\n", tmp);
    }
}
```

Sockets

- Socket API lets two processes in different machines to communicate with each other over the TCP/IP network stack
- Think of socket as the **door (logical pipe)** a process/thread needs to use if it needs to communicate with other processes
- **TCP sockets:** reliable delivery, congestion control
- Network socket identified by a port number on a machine – Socket bound to IP address of a network interface and a port number
- TCP socket communication: a “server” listens on a well-known port number, a “client” connects to the server, both exchange messages



Simple case when there is just one thread on the server side

TCP client

- socket: create new socket “C1” (returns socket file descriptor)
- connect: connects to a server socket using the server’s IP address and port number (initiates TCP 3 way handshake to server). This request is intercepted by server socket ‘A’ and if connection is successful, socket C1 can now be used to communicate with the server
- read: read data from a connected socket “C1” [receives what server writes on its own socket “C2”]
- write: write data from a memory buffer into socket “C1” (which server receives on “C2”)

TCP server

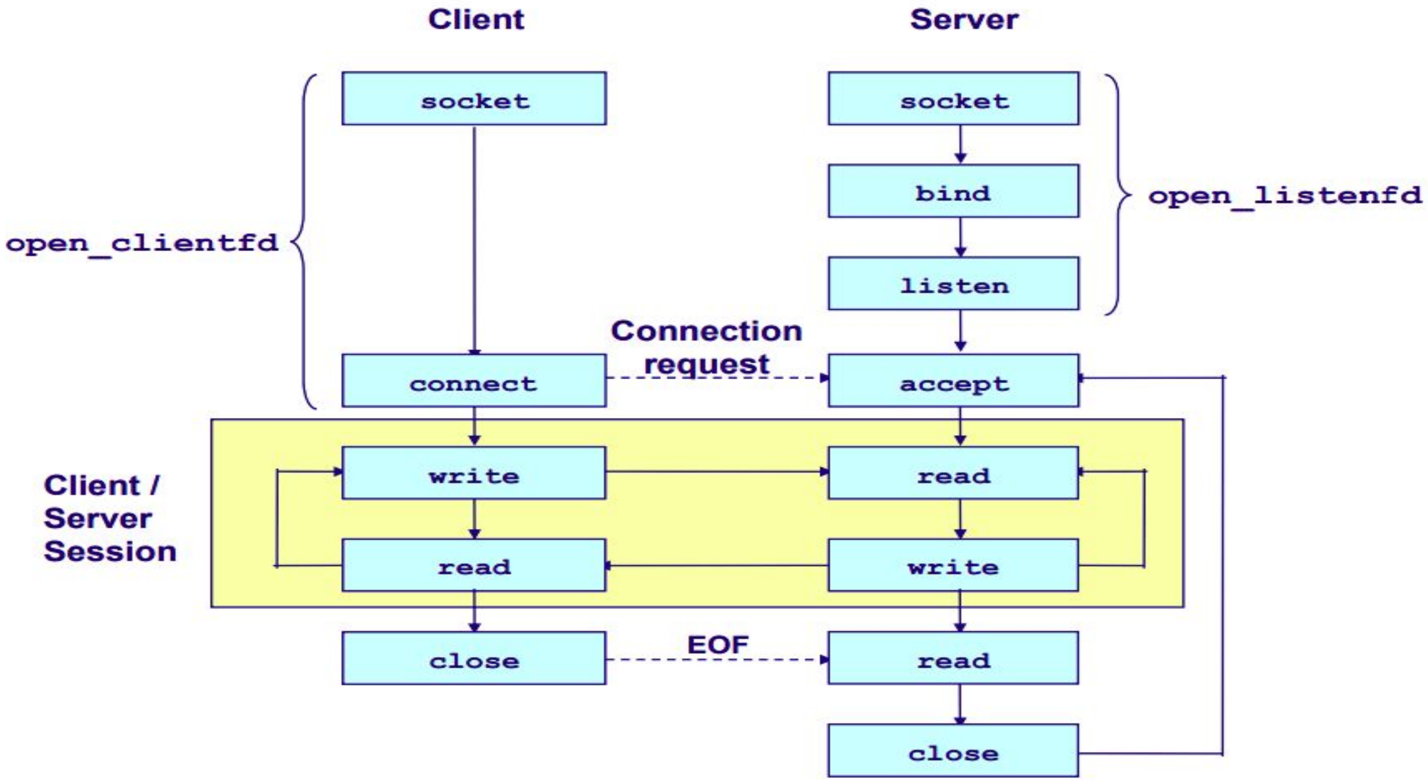
In the simple case, the server has 2 sockets:

1. Listening socket (A) where server listens for connection requests
2. Communication socket (denoted by C2 below) which is used to send and receive messages

- socket: create new socket “A” (returns socket file descriptor)
- bind: bind server socket “A” to well-known port number and IP address
- listen: start listening for new connections on socket “A”
- accept: accept a new connection on server socket “A” (returns a new socket “C2” to talk to this specific client)
- read: read data from socket “C2” into memory buffer [receives what client writes on its own socket “C1”]
- write: write data from memory buffer into socket “C” connected to client (which then client receives on “C1”)

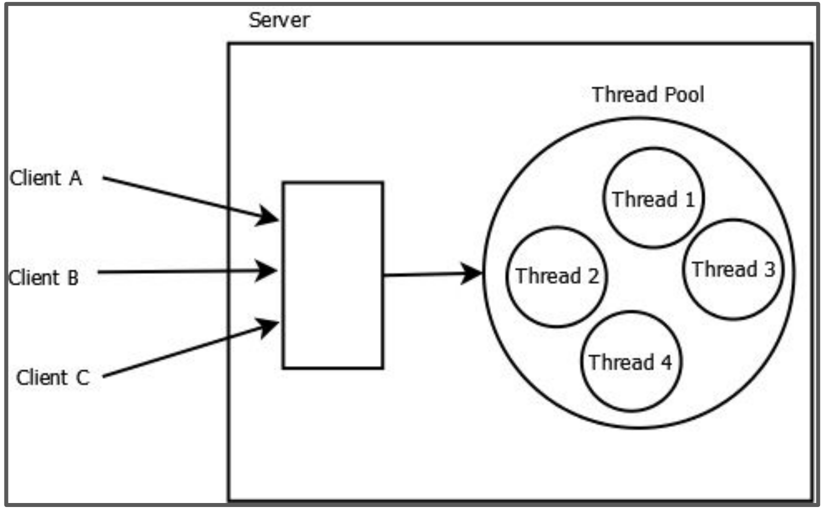
A TCP/UDP connection is identified by a tuple of five values:

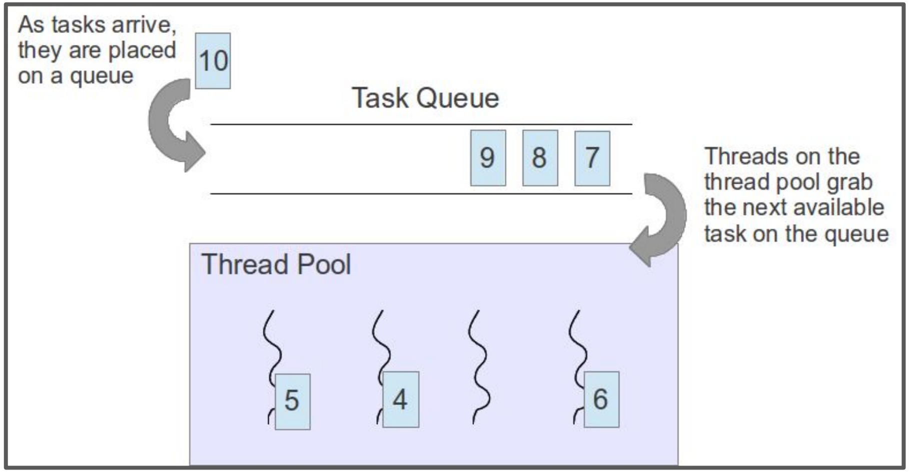
{<protocol>, <src addr>, <src port>, <dest addr>, <dest port>}

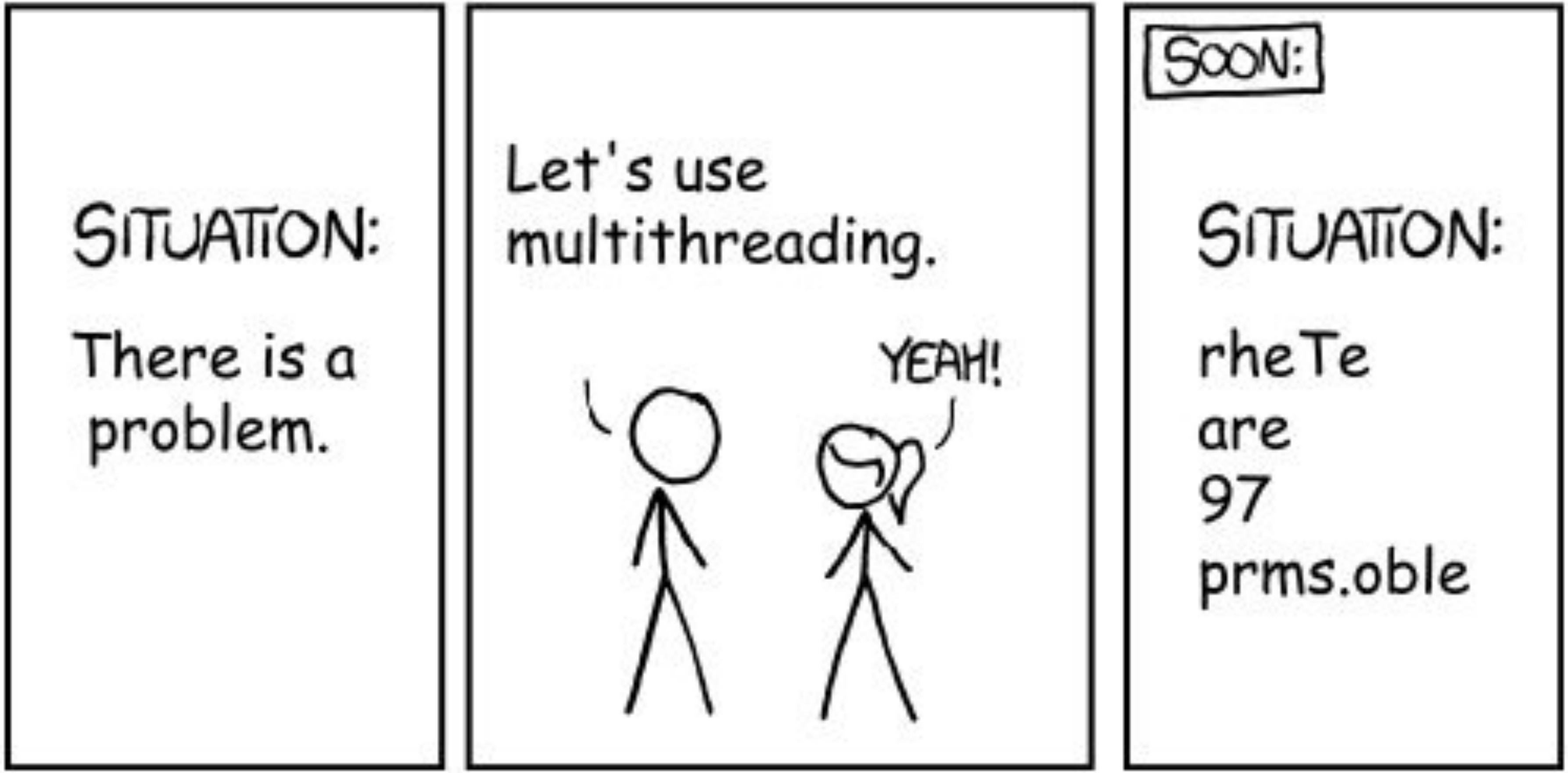


Few components of the Socket API

- int s = socket(domain, type, protocol) ; - Create a socket
  - domain: Communication domain, typically used AF_INET (IPv4 protocol)
  - type: Type of socket - SOCK_STREAM (TCP) or SOCK_DGRAM (UDP)
  - protocol : Specifies protocols - usually set to 0 [Explore !]
- int success_status = bind(sockid, &addrport, size); - Reserves a port for the socket
  - sockid: socket identifier
  - addrport: struct sockaddr_in - the (IP) address and port of the machine (address usually set to INADDR_ANY chooses a local address)
  - size : Size of the sockaddr structure
- About struct socladdr_in:
  - **sin_family :** Set this to AF_INET ; (used to designate the type of addresses that your socket can communicate with)
  - **sin_port :** The network byte-ordered bit port number
  - **sin_addr :** Source address, INADDR_ANY to choose the local address
  - Also, we need to use the htons() function to convert the PORT NUMBER from host byte order to network byte order (recall the concept of LITTLE-ENDIAN and BIG_ENDIAN taught in CSO)


Server
Thread Pool
Client A
Client B
Client C
Thread 1
Thread 2
Thread 3
Thread 4


As tasks arrive, they are placed on a queue
10
Task Queue
9
8
7
Threads on the thread pool grab the next available task on the queue
Thread Pool
5
4
6


SITUATION:
There is a problem.
Let's use multithreading.
YEAH!
SOON:
SITUATION:
rheTe are 97 prms.oble