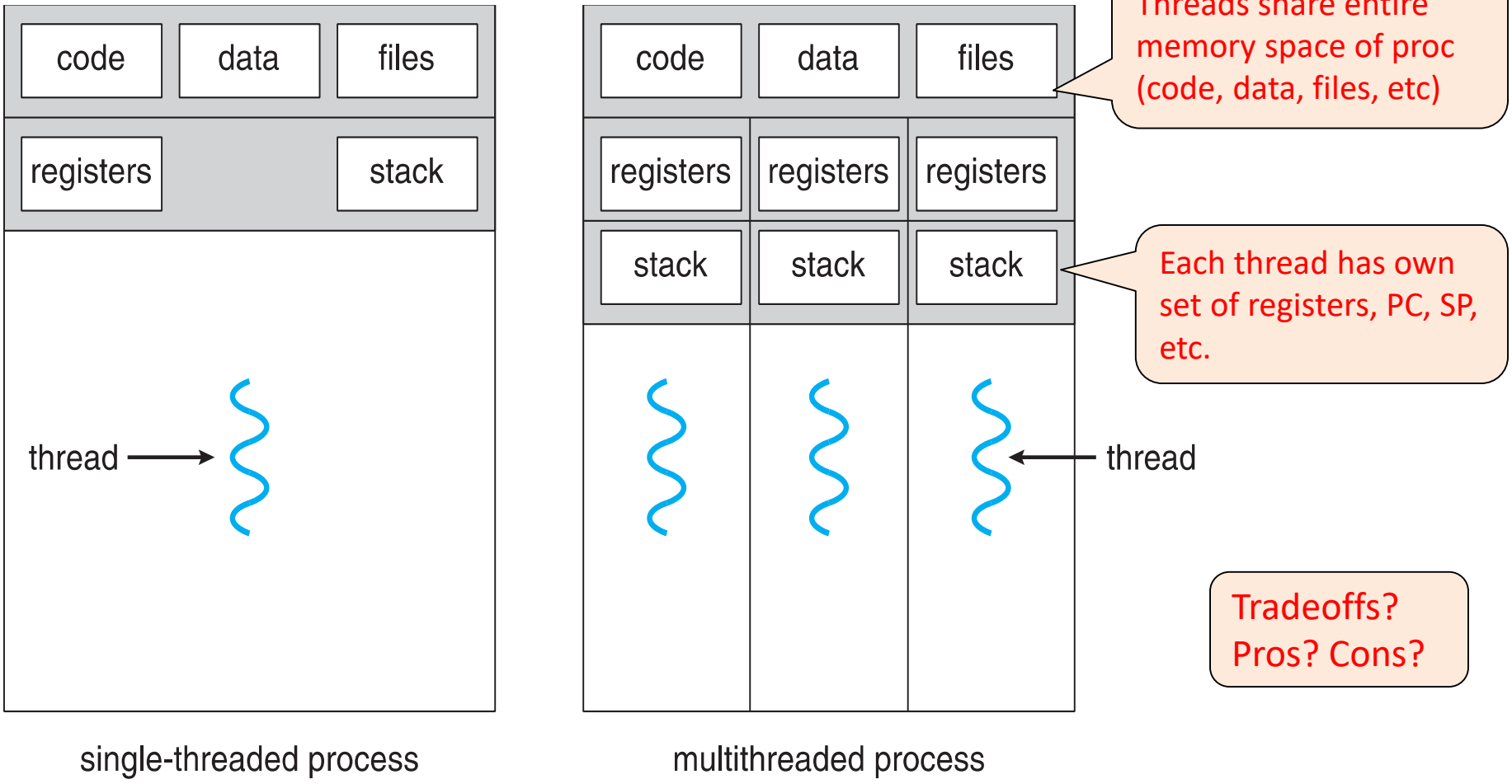
CS 149 Operating Systems
Threads

Instructor: Kong Li

Content

- What & benefits
- Multicore Programming
- Thread Programming
- Thread Libraries
  - Pthread
  - Java
- Semantics: fork, exec, signal

Single and Multithreaded Processes

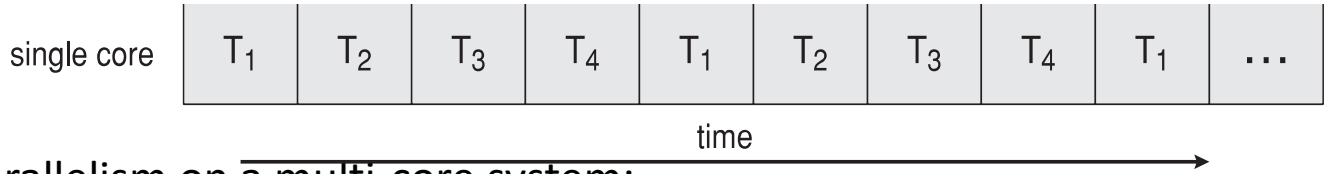


Thread Pros and Cons

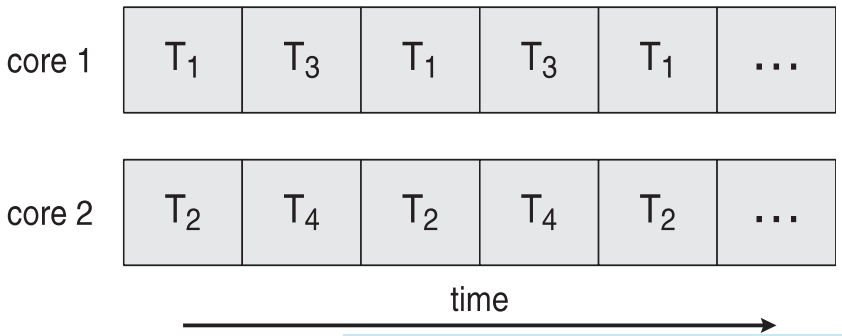
- Advantages of using threads:
  - Responsiveness
    - allow continued execution if part of proc is blocked
    - important for user interfaces
  - Resource sharing
    - easier than shared memory or message passing
  - Economy
    - cheaper than proc creation
    - overhead(thread switching) < overhead(proc context switching)
  - Scalability
    - take advantage of multiprocessor architectures
- Disadvantages of using threads
  - Synchronization overhead
  - Many library functions not thread-safe. E.g., random vs random_r
  - Lack of robustness (one thread can impact all threads)

Concurrency vs. Parallelism

- **Parallelism**: perform more than one task simultaneously
- **Concurrency**: more than one task making progress
  - Either interleaved or in parallel or hybrid
  - Concurrency w/o parallelism is possible
  - Single processor / core: scheduler providing concurrency
- Concurrent execution on single-core system:



- Parallelism on a multi-core system:

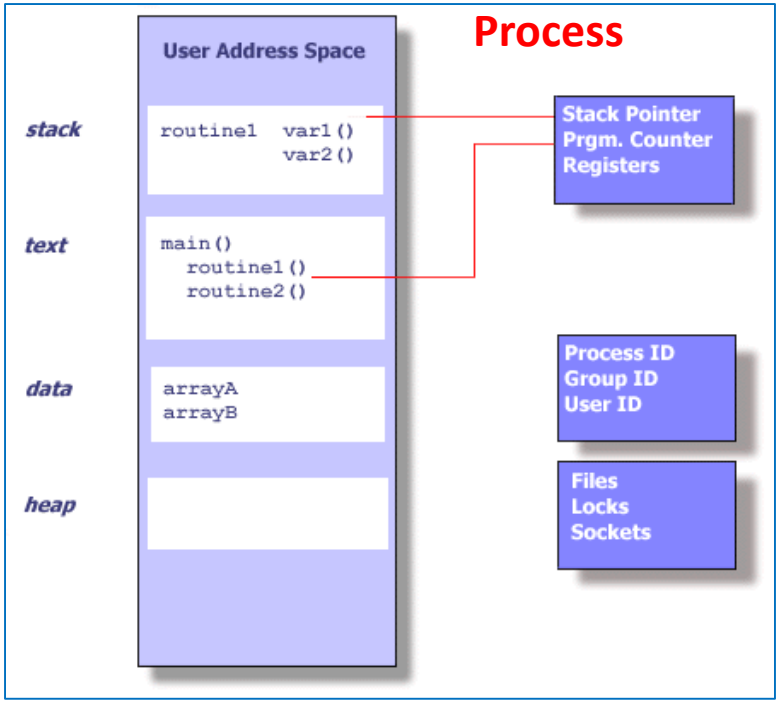


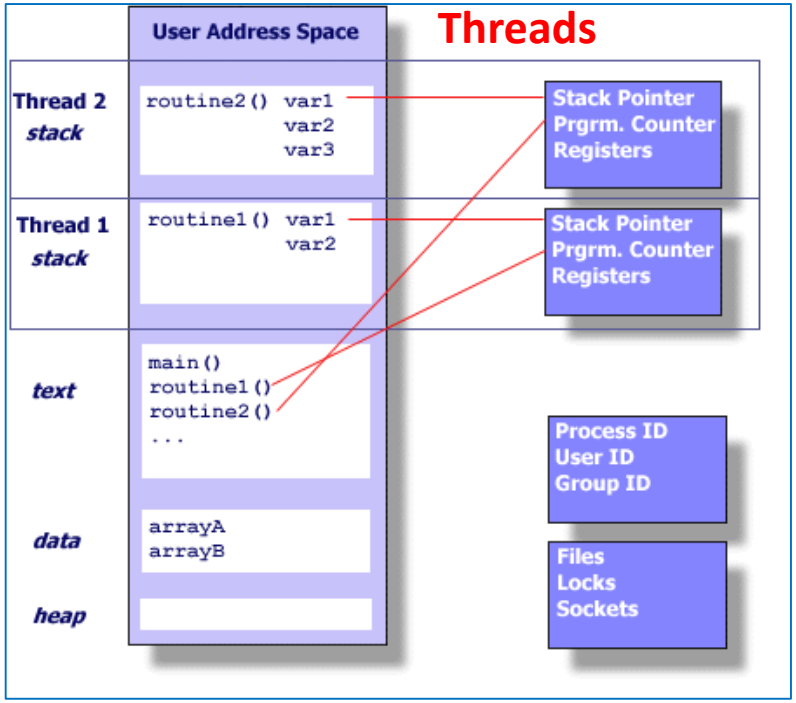
- Amdahl's Law

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- *S*: serial portion of app
- As $N \rightarrow \infty$, speedup $\rightarrow 1/S$
- App's serial portion becomes the dominating factor for performance

Proc vs Thread





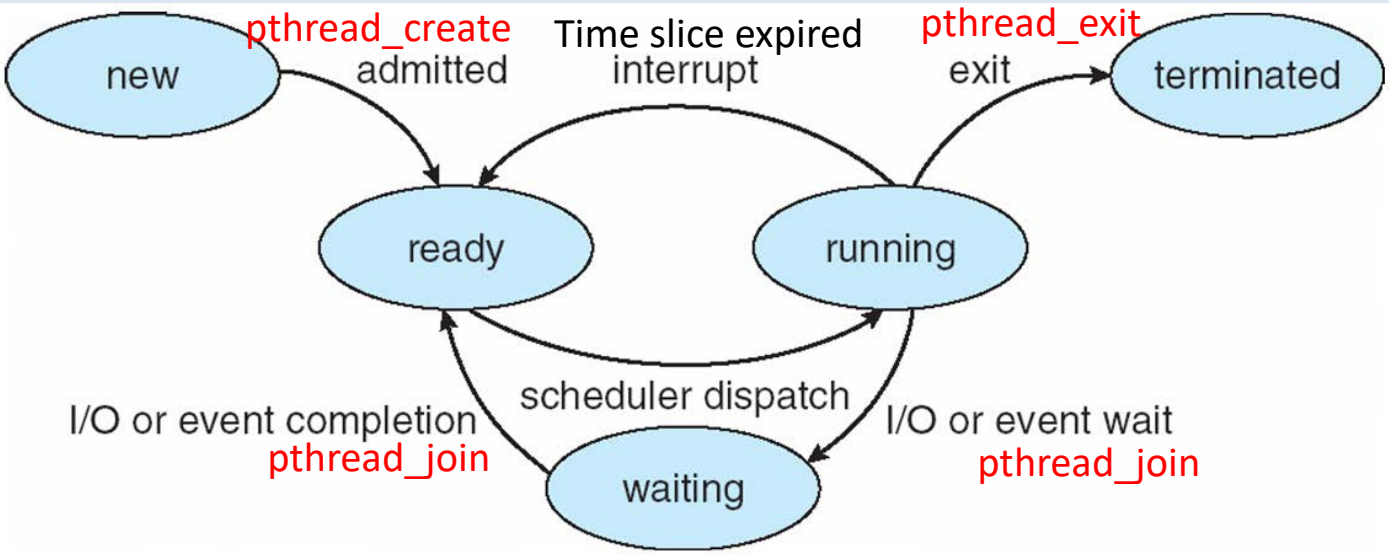
- Threads in one proc
  - Share: heap, data, code (text), global variables, file descriptors, proc id, etc.
  - Individual (TCB): stack, registers, program counter, thread id, etc.
- Some system calls affect only thread (e.g., _r), some affect the entire proc (e.g., exit)
- Linux API _r: thread safe. E.g., `random` vs `random_r`
- Linux: errno – thread-local (one instance per thread)

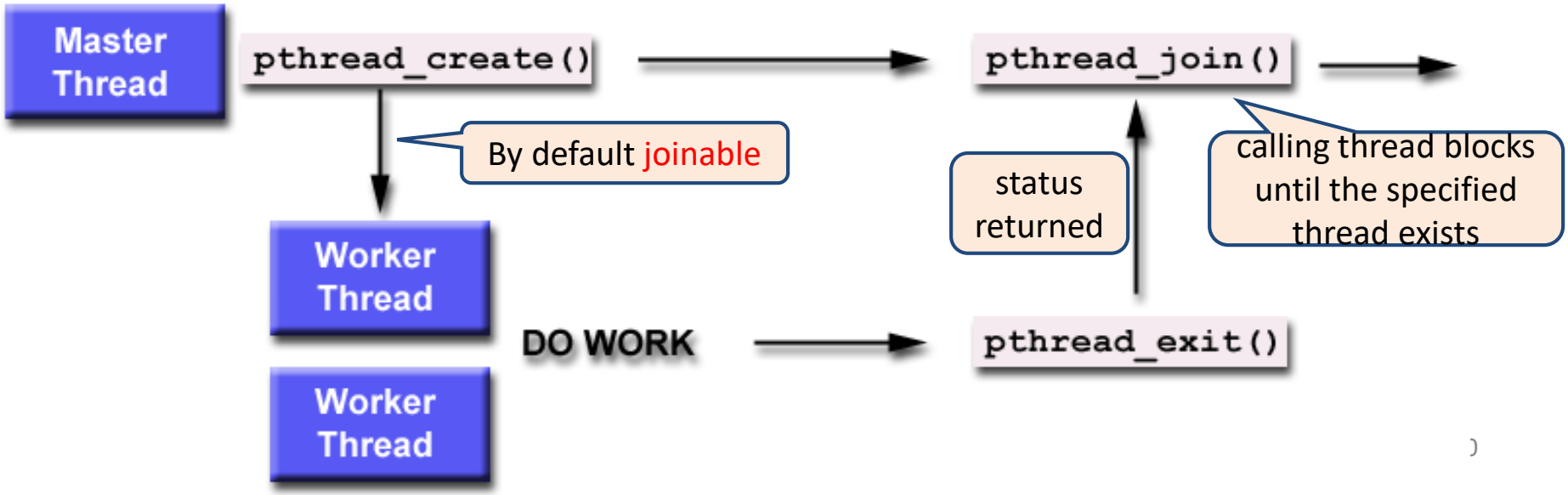
Thread library: Pthreads

- POSIX standard (IEEE 1003.1c) API
  - Specification, not implementation
  - API specifies behavior of thread library
  - implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

- man pthreads
- Compilation on Linux: gcc test.c -pthread
  - Not recommended: gcc test.c -lpthread
- Eclipse IDE: by default does not include "-pthread" in project settings
  - Project Explorer pane: project -> Properties -> C/C++ build -> Settings -> Tool Settings ->
    - GCC C Compiler -> Miscellaneous, Add "-pthread" into the beginning of "Other Flags"
    - GCC C Linker -> Miscellaneous, Add "-pthread" into the beginning of "Linker Flags"
- https://computing.llnl.gov/tutorials/pthreads/

Thread State and Lifecycle





Thread Creation

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void*(*start_routine) (void *), void *arg);
```

- Create a new thread which starts executing `start_routine`, with the parameter `arg`
- init its attrs using `attr`
- thread attributes:
  - priority, stack size, name, detach-state, etc.
- by default: Joinable thread

Linux thread attributes:

Detach state = PTHREAD_CREATE_JOINABLE

Scope = PTHREAD_SCOPE_SYSTEM

Inherit scheduler = PTHREAD_INHERIT_SCHED

Scheduling policy = SCHED_OTHER

Scheduling priority = 0

Guard size = 4096 bytes

Stack address = 0x40196000

Stack size = 0x201000 bytes

- Q: why detached thread?

Thread Termination & Join

```
#include <pthread.h>

void pthread_exit(void *value_ptr);
int pthread_join(pthread_t thread, void **value_ptr);
```

should not be located on the calling thread's stack (why?)

- `pthread_exit()`
  - Calling thread exits & returns status
- `pthread_join()`
  - Calling thread blocks until the specified thread exits
  - Get status back
- Thread termination
  - returns from its starting routing
  - call `pthread_exit()`
  - cancelled by `pthread_cancel()`
  - process terminated

Process	Thread
fork()	pthread_create()
exit()	pthread_exit()
exec()	N/A
waitpid(pid, …)	pthread_join()
wait()	N/A
zombie process	zombie thread
kill()	pthread_cancel()
getpid()	pthread_self()

Pthreads Example

gcc … -pthread

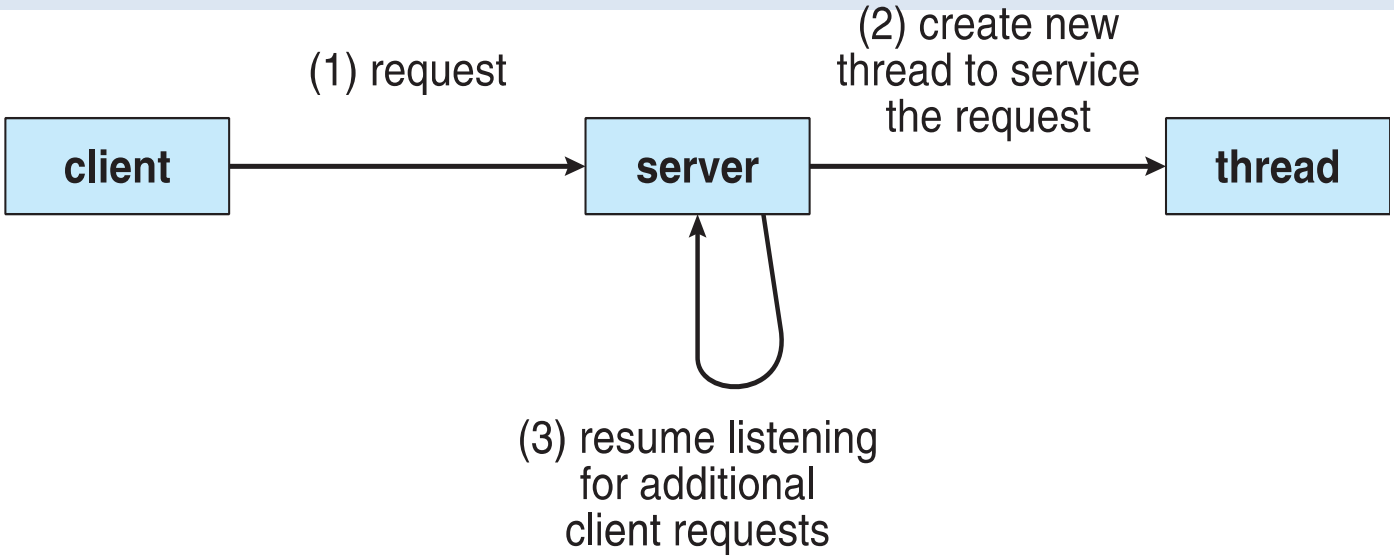
```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

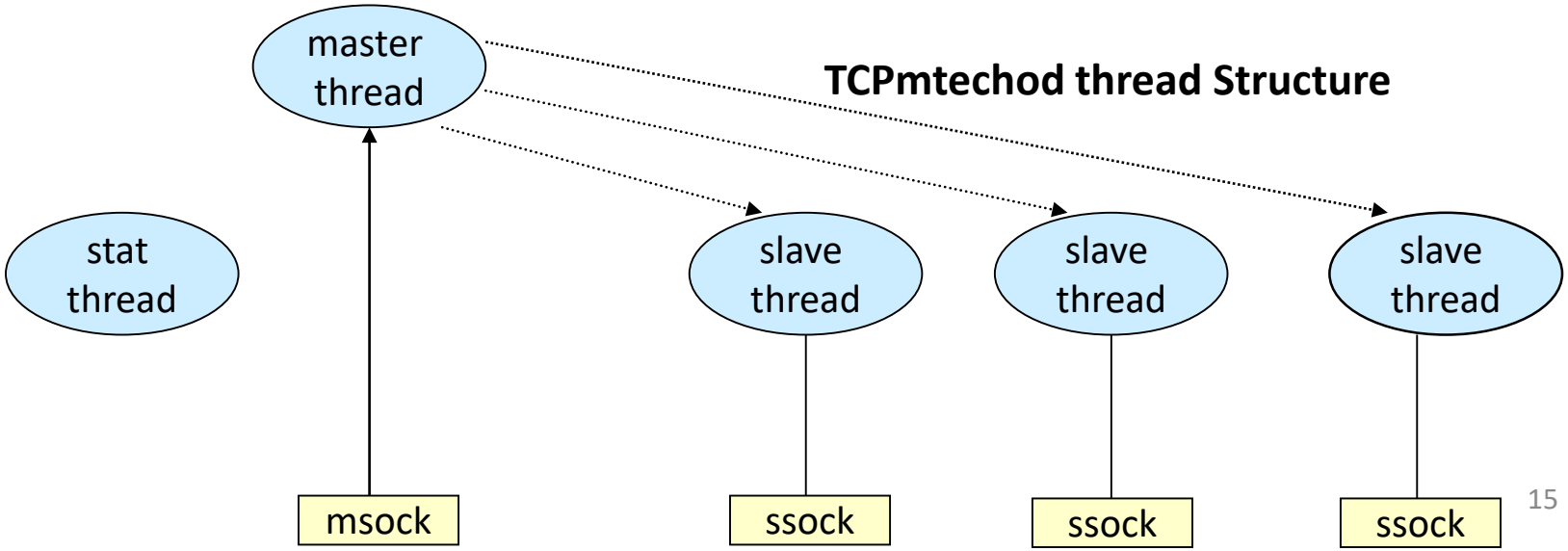
Demo!

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Multithreaded Server Architecture





Pthreads: TCPmtechod.c (Master Thread)

```
pthread_attr_t ta;
msock = passiveTCP(service, QLEN);
(void) pthread_attr_init(&ta);
(void) pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
(void) pthread_mutex_init(&stats.st_mutex, 0);
if (pthread_create(&tid, &ta, (void * (*)(void *))prstats, 0) < 0)
   errexit("pthread_create(prstats): %s\n", strerror(errno));
while (1) {
   alen = sizeof(fsin);
   ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
   if (ssock < 0) {
      if (errno == EINTR)
         continue;
      errexit("accept: %s\n", strerror(errno));
   }
   if (pthread_create(&tid, &ta, (void * (*)(void *))TCPechod, (void *)ssock) < 0)
      errexit("pthread_create: %s\n", strerror(errno));
}
```

thread attributes obj

detached thread
- not joinable
- exit status discarded
- default is joinable

one stat thread

Error code in errno

starting routine

parameter

many echo threads

Thread Cancellation

- Request to cancel thread: terminating a thread before it has finished
  `int pthread_cancel(thread_t tid);`
- Actual cancellation depends on thread state `pthread_setcancelstate`

	Mode	State	Type
	Off	Disabled	–
default →	Deferred	Enabled	Deferred
	Asynchronous	Enabled	Asynchronous

Why?

- If state == disabled, cancellation remains pending until thread enables it
- If state == enabled, determined by type `pthread_setcanceltype`
  - If type == async, terminates the target thread immediately
  - If type == deferred, cancellation occurs when thread reaches **cancellation point** (certain functions)
- Request delivery of pending cancel req: `pthread_testcancel();`
- Cleanup: invoke **cleanup handler**(s) async before terminating thread
- On Linux systems, thread cancellation is handled through signals

Java Threads

- managed by the JVM
- using the threads model provided by underlying OS
- Creation:
  - Define a new class that
    - Derived from the Thread class
    - Override run() method
  - Define a new class that implements the `Runnable` interface

```
public interface Runnable
{
    public abstract void run();
}
```

    - `run()`: thread starting routine
- Thread mgmt: `java.util.concurrent` package

Java Multithreaded Program

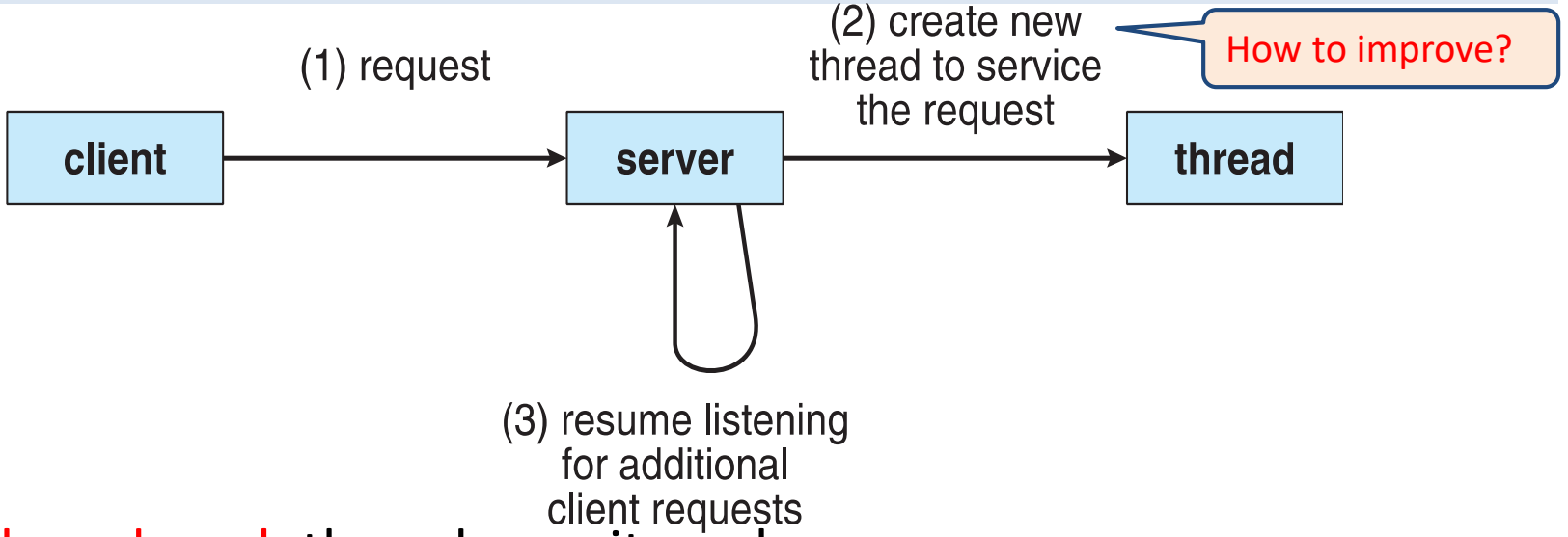
```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

thread starting routine

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
         } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Demo!

Multithreaded Server Architecture



- thread pool: threads await work
- Advantages:
  - (for server) faster to service request than creating a new thread
  - Allows # of threads in app(s) to be bound to the size of the pool
  - Separating task (to be performed) from mechanics (of creating task): different strategies for running task
    - Tasks could be scheduled to run periodically

Semantics of fork() and exec()

- `fork()`: duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
  - Linux: dup only the calling thread in the new proc; no other threads
    - Cleanup other threads? `pthread_atfork()`

- `exec()`: replace the running proc (including all threads)

Semantics of Signal Handling

- Signals: UNIX way to notify a proc that an event has occurred
  - Signal generated by particular event (inside or outside of a proc)
  - Signal delivered to a proc
  - Signal handling (per-signal basis): default action, ignore, catch
- man 7 signal

- Single-threaded: signal delivered to process
  - man 2 kill
  - API: `kill(pid_t pid, int signal);`
- Multi-threaded: signal delivered to which thread?
  - Thread specific: to thread causing the signal, e.g., SIGSEGV, SIGFPE
  - Proc specific: to any thread not blocking the signal, e.g., Control-C
  - API: `pthread_cancel(pthread_t tid, int signal);`

Thread-Local Storage

- **Thread-local storage (TLS)**
  - Each thread to have its own copy of data; unique to each thread
  - Similar to static data but per thread basis. e.g., `errno`
  - When is it useful?
    - when one does not control thread creation (i.e., thread pool)

- local variables vs TLS
  - Local variables: visible only during single function invocation
  - TLS: visible across function invocations

Summary

- Thread vs Process
  - Share what?
  - Which info is private to each thread (not shared with process)?
  - What is the content in thread control block (TCB)?
- Thread life cycle
  - Which state transition is triggered by which pthread API?
  - Pthread: create, termination, join, cancel
  - Process APIs vs Pthread APIs
  - Java thread
  - Multi-threaded server & Thread pool
- Fork vs exec
- Thread signal handling
- Thread local storage

Self Exercises

- 4.7 ~ 4.9, 4.13, 4.15, 4.17
- 4.20, 4.22, 4.25