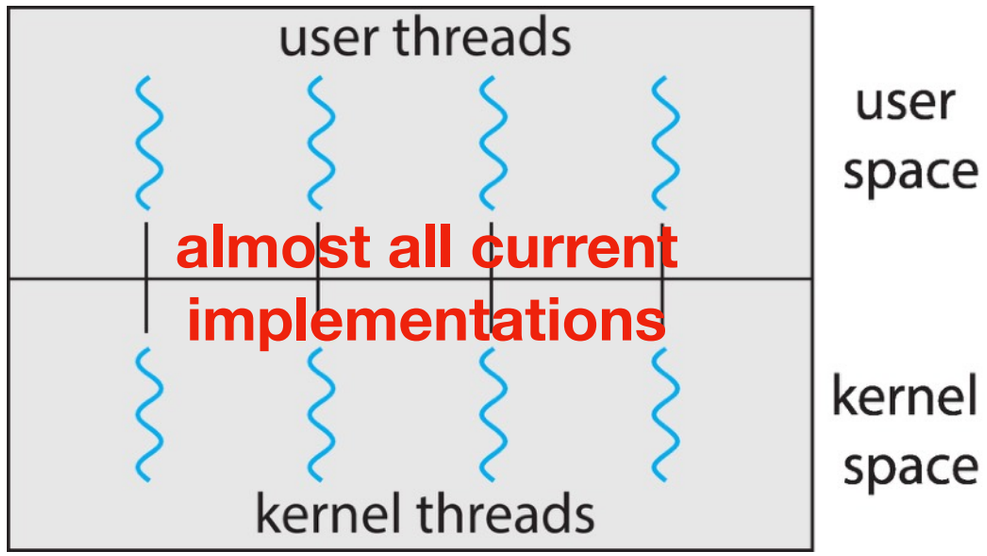
Operating System Concepts

Lecture 15: POSIX Thread Library

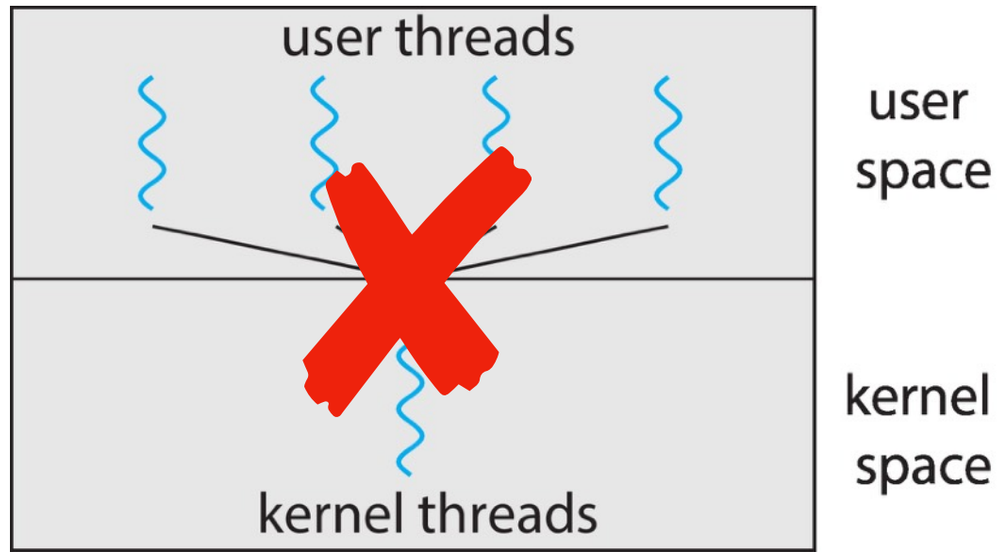
Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Threading models



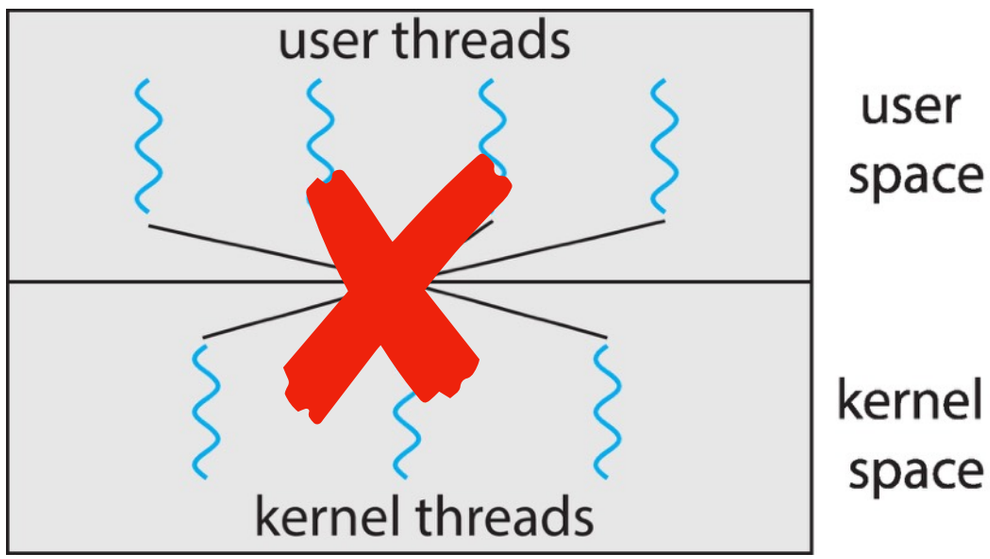
one-to-one

higher parallelism/concurrency



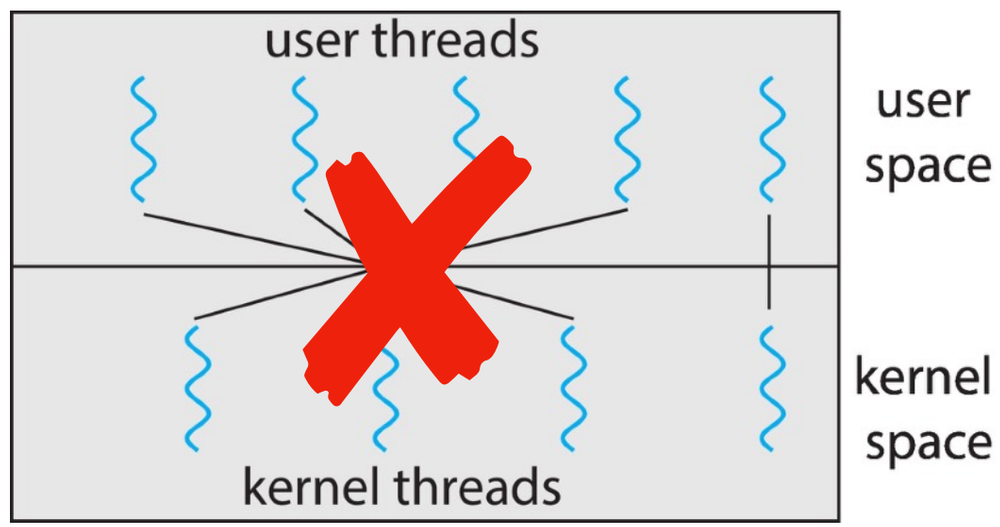
many-to-one

not suitable for multicore systems as threads of the same process can't run in parallel



many-to-many

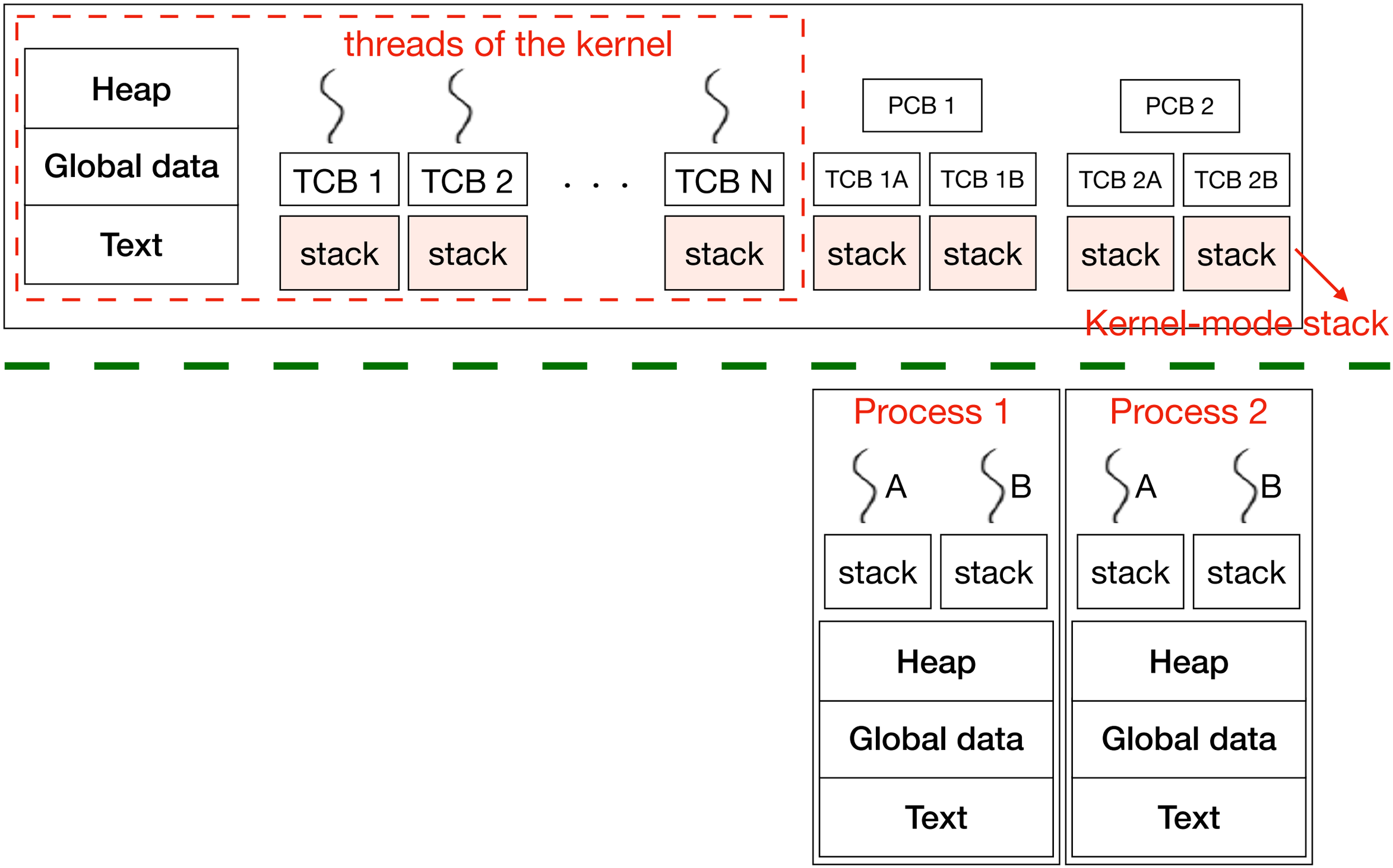
a smaller or equal number of kernel threads



two-level

One-to-one mapping (Linux, macOS, Windows)

Kernel-level threads (known by scheduler)



User-level threads

What happens on a fork?

- The `fork( )` system call in a multithreaded program creates a single-threaded child process (the thread that called fork)

Today’s class

- Implicit vs. explicit threading
- POSIX threads
- Thread Pool

Developing concurrent applications and verifying their correctness are difficult

Google Is Uncovering Hundreds Of Race Conditions Within The Linux Kernel

Written by Michael Larabel in Google on 3 October 2019 at 02:06 AM EDT. 43 Comments

One of the contributions Google is working on for the upstream Linux kernel is a new "sanitizer". Over the years Google has worked on AddressSanitizer for finding memory corruption bugs, UndefinedBehaviorSanitizer for undefined behavior within code, and other sanitizers. The Linux kernel has been exposed to this as well as other open-source projects while their newest sanitizer is KCSAN and focused as a Kernel Concurrency Sanitizer.

The Kernel Concurrency Sanitizer (KCSAN) is focused on discovering data-race issues within the kernel code. This dynamic data-race detector is an alternative to the Kernel Thread Sanitizer.

In their testing just last month, in two days they found over 300 unique data race conditions within the mainline kernel.

There was a recent discussion about the Kernel Concurrency Sanitizer on the LKML. For those wanting to learn more, the code at least for now is being hosted on GitHub.

placing a significant burden on the developers to ensure that implementation is free of race conditions and other bugs

Implicit threading

- How to make it easier to write multithreaded applications?
  - transfer thread creation and management to compilers and run-time libraries (**implicit threading**); programmers just have to identify tasks that run in parallel
- OpenMP is a set of compiler directives and library routines that are available for C/C++/Fortran instructing the compiler to automatically generate a number of threads to run a parallel block of code
  - OpenMP is built on top of the pthread library in C
  - OpenMP directives demarcate code region that can be executed in parallel `#pragma omp parallel`

    see https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf

Explicit threading libraries

- API for creating and managing threads
- Two primary approaches:
  - library is in the user space entirely; thread management and scheduling is done without involvement of kernel; kernel sees one process only
  - kernel-level library supported by the OS
- POSIX Pthreads, Windows thread library, and Java are three examples thread libraries
  - Pthreads can be implemented as a user-level library or a kernel-level library
  - almost all modern UNIX systems implement it as a kernel-level library using 1:1 mapping between user and kernel threads

POSIX Thread Library

- POSIX.1c (IEEE 1003.1c) is the standard for thread creation and synchronization
- The API specifies behaviour of the thread library, not its implementation
- Pthreads is commonly used in UNIX systems (Solaris, Linux, macOS)
- WIN32 Threads: Similar to POSIX, but for Windows
  - In Pthreads and Windows thread library, global variables are shared among all threads of a process

Pthreads:

```
pthread_attr_init(&attr);  /* set default attributes */
pthread_create(&tid, &attr, thread_function, (void*)param);
```

Win32 threads:

```
ThreadHandle = CreateThread(NULL, 0, Sum, &Param, 0, &ThreadID);
```

POSIX Thread Library

- Pthreads has a thread container data type of `pthread_t` which is the **handle** of a thread
  - `pthread_create( )` creates a separate thread and returns its handle
    - takes a **start_routine** which is invoked with the specified arguments
  - `pthread_attr_init( )` sets the initial attributes of a thread
    - scheduling information, stack size, stack address, etc.
  - `pthread_join( )` allows the calling thread to wait for another thread to terminate
    - the calling thread is blocked until the target thread terminates
  - `pthread_detach( )` marks a thread detached (non-joinable) causing its resources to be released upon termination without the need for another thread to join with it
  - `pthread_cancel( )` sends a cancellation request (implemented using signals) to the target thread
  - `pthread_yield( )` causes the calling thread to relinquish the CPU voluntarily
  - `pthread_exit( )` terminates the calling thread and executes clean-up handlers defined by `pthread_cleanup_push( )`
  - `pthread_kill( )` sends the specified signal to the target thread
- See `man pthread`

POSIX Thread Implementation in Linux

- Thread creation (`pthread_create`) uses the `clone` system call
  - unlike `fork()`, `clone()` allows the newly created process/thread to share only parts of its execution context with the calling process/thread
- Thread synchronization primitives (discussed later) use the `futex` system call

Example

Compile and link with `-pthread` flag

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

void *thread_function (void *arg) {  // thread start routine
      int i; pid_t pid; pthread_t tid;
      pid = getpid();
      tid = pthread_self();  // obtain the handle (ID) of the calling thread
      printf("Process ID: %d, thread ID: %u, arg: %s\n", (unsigned int) pid,
            (unsigned int) tid, (char *) arg);
     return; // equivalent to calling pthread_exit
}

int main (int argc, char *argv[]) {
      int i, rval;
      pthread_t tid;
      for (i= 0; i < argc; ++i) {
            rval= pthread_create(&tid, NULL, thread_function, (void *) argv[i]);
            if (rval) perror("thread creation failed!”);
      }
      pthread_exit(0);
}
```

How to handle signals in a multithreaded program?

- In single-threaded programs, signals are always delivered to the corresponding process
- How about multithreaded programs?
  - synchronous signals must be delivered to the thread that caused the event
  - asynchronous signals (indicating external events) should be sent either to all threads that have not blocked that kind of signal or to the first thread that has not blocked it
    - since signals must be handled only once, they are typically sent to the first thread that is not blocking it
- To deliver a signal to a specific thread, POSIX Pthreads has
  - `pthread_kill(pthread_t pid, int signal)`

How to cancel a thread (terminate it before completion)?

- Asynchronous cancellation:
  - one thread immediately cancels the target thread; this may not free a necessary system-wide resource allocated to the target thread because OS may not be able to reclaim it in a proper manner
- Deferred cancellation:
  - the target thread periodically checks, **at cancellation points**, if it should terminate (i.e. there's a pending cancellation request); this way termination is done by the same thread in an orderly and safely fashion
- In Pthreads, thread cancellation is initiated by `pthread_cancel(pthread_t pid)`

How to cancel a thread (terminate it before completion)?

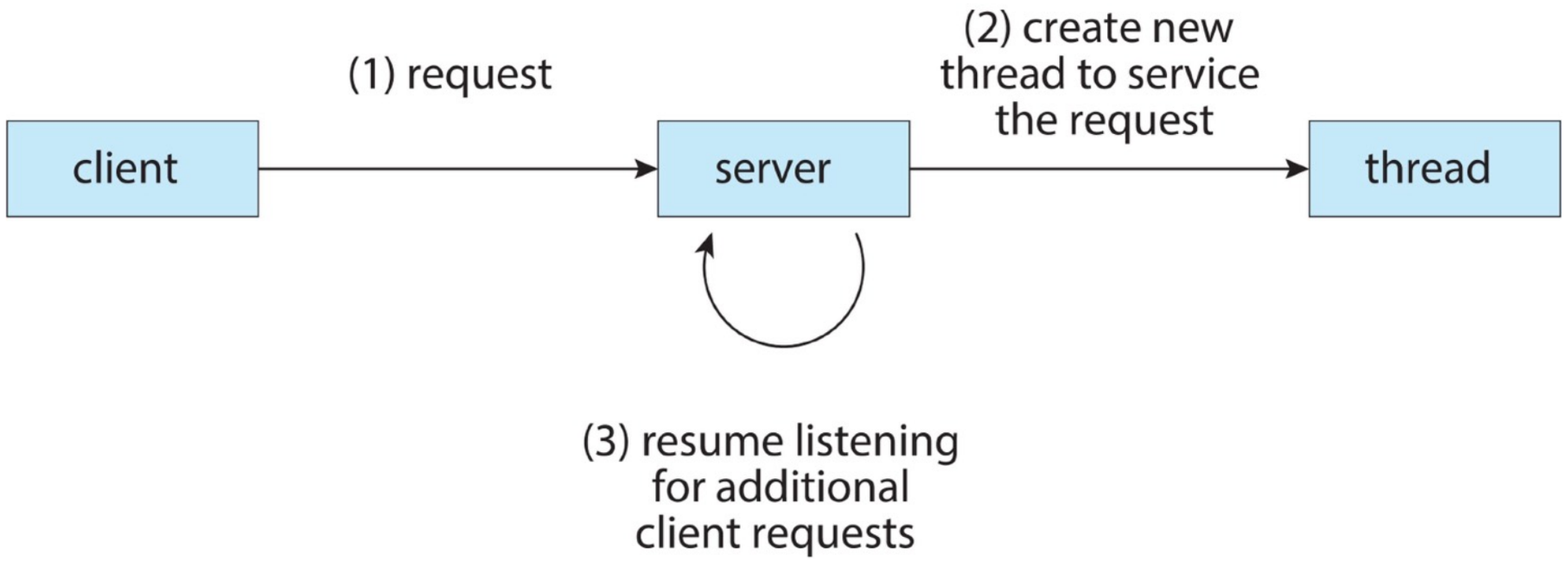
- A thread may set its cancellation state and type using the API
  - three options: disable cancellation; allow deferred cancellation; allow asynchronous cancellation
  - default is deferred cancellation; can be changed by `pthread_setcanceltype( )`
- Many blocking system calls in the POSIX are **cancellation points** where a thread can cancel itself when reaching those points
  - for example the `read( )` system call is a cancellation point; this allows for cancelling a thread that is blocked while waiting for input
- A thread can explicitly check for a cancellation request before reaching a cancellation point by calling `pthread_testcancel( )`
- When a thread is cancelled, a cleanup handler is invoked to release its resources

Example: multithreaded server (loose syntax)

```
serverLoop() {
    connection = AcceptNewConnection();
    thread_fork(ServiceRequest, connection);
}
```

What if we get an arbitrarily large number of connection requests?

- might run out of memory
- must create and destroy many threads (high overhead)
- schedulers usually have trouble with too many threads

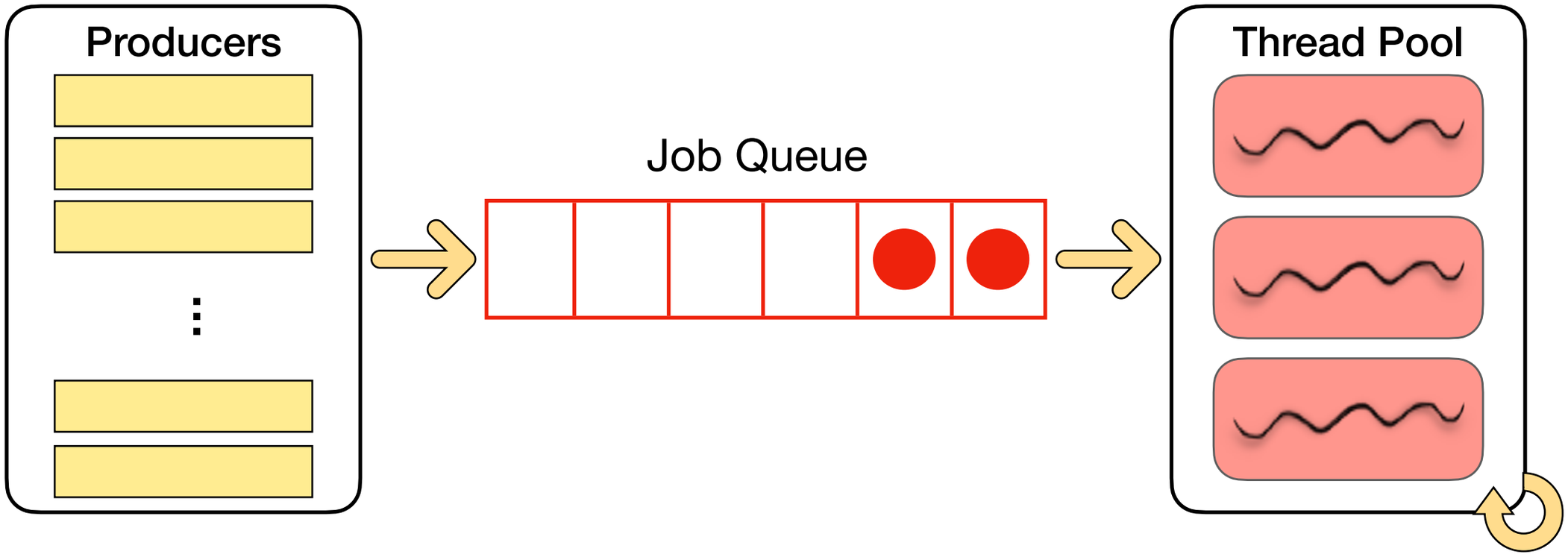


Thread pool

- Definition: a collection of worker threads that execute **callback functions** on behalf of the application
  - a callback function is a function passed to another function as an argument, which is then invoked inside that function
- Basic idea: create **a fixed number of** threads at startup and place them into a pool where they sit and wait for work
  - resources are allocated in advance: so no thread creation and destruction **overhead**
    - serving a request with an existing thread (i.e. reusing it) is often faster than waiting for creation of a new thread on demand
    - more important when threads do a small amount of work
  - creating unlimited threads could exhaust system resources
    - they consume a significant amount of memory and contend for resources
  - controlling load by setting the number of threads in the thread pool
    - e.g. balancing the number of threads with the number of processing cores

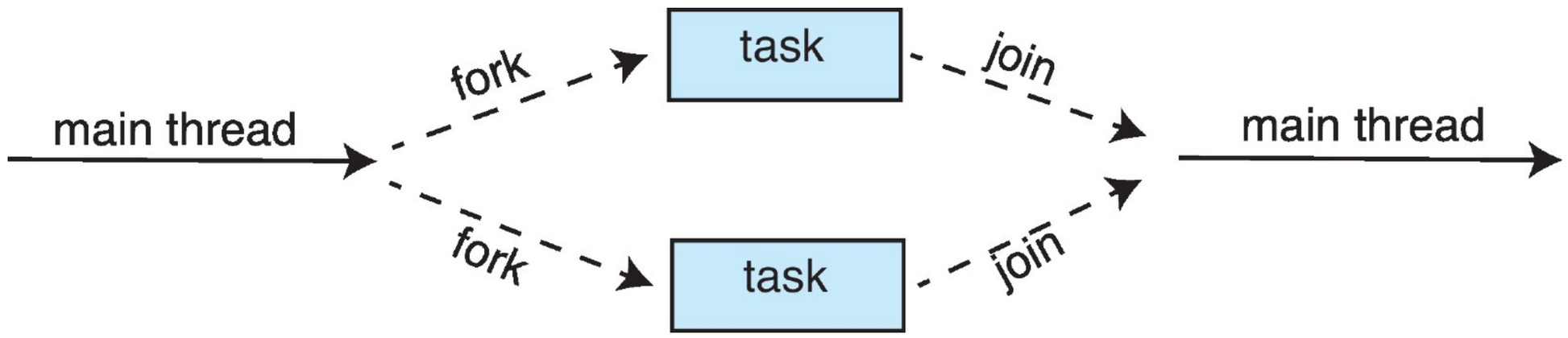
What if we get more requests than the number threads in the thread pool?

- When a new request arrives, the main thread submits it to the thread pool and resumes waiting for additional requests; it doesn't join worker threads so they can be detached
  - if there is an available thread in the pool, it is awakened to immediately service this request
  - otherwise the request is queued until a thread becomes available
    - so it requires having a queue of pending requests
- Once a thread completes service, it returns to the pool (i.e., it becomes idle and ready to be dispatched to another task)



Fork-join model

- The main thread forks a number of subthreads, passes them arguments to work on, joins them, and collects results
  - the number of forked threads depends on the number of tasks
- The main thread resumes execution (e.g. to combine results) after joining with the spawned threads
- It can be thought of as the **synchronous** version of thread pools in which a library determines the actual number of threads to create



Homework

- implement the multithreaded version of merge sort using pthreads