Operating System Concepts

Lecture 15: POSIX Thread Library

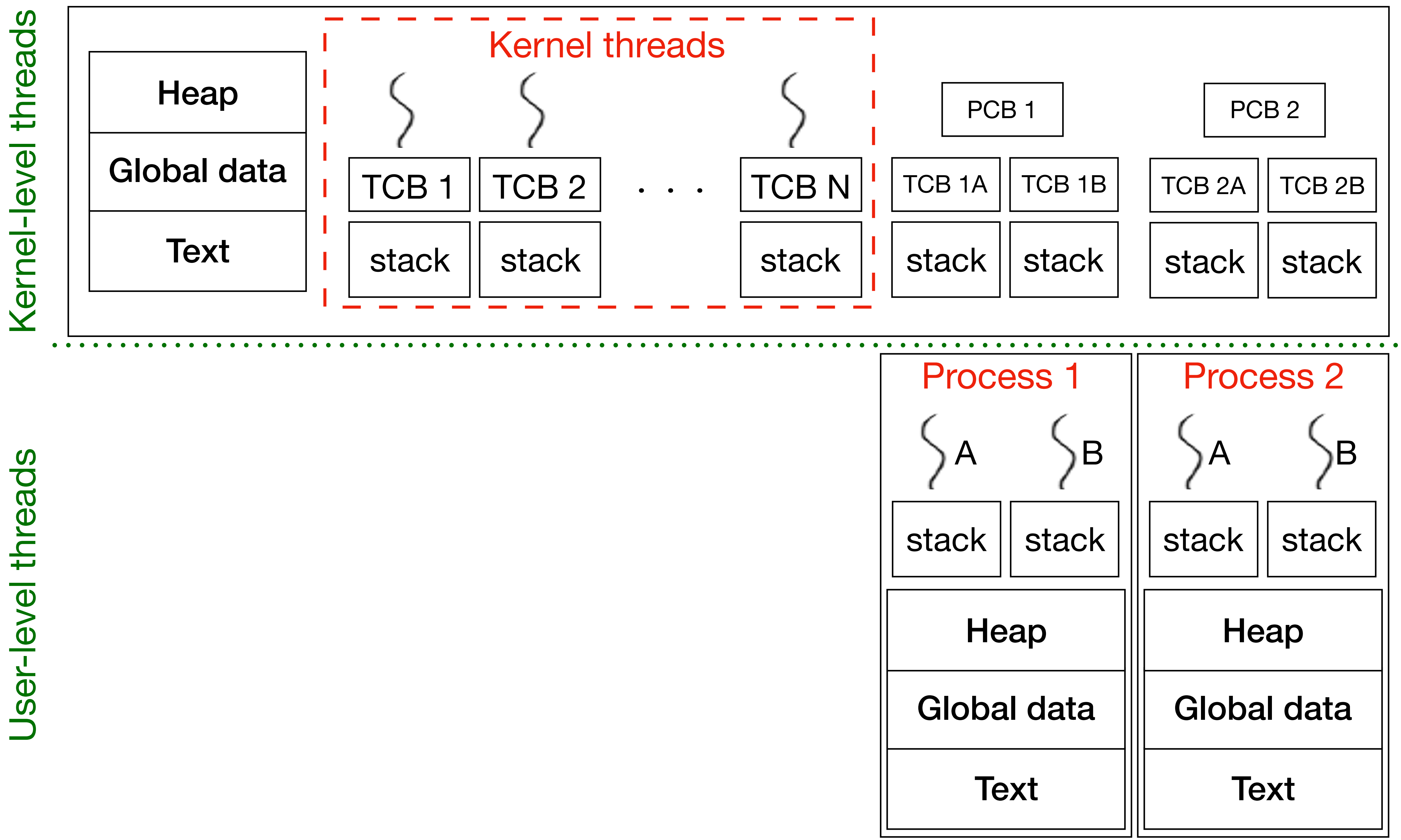
Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today’s class

- Implicit versus explicit threading
- POSIX threads
- Thread Pool

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



What happens on a fork?

- the `fork( )` system call has two versions
  - one that duplicates all threads of the parent process for the child process
  - one that creates a single-threaded child process (usually the thread that called fork)
- if you call `exec( )` right after `fork( )`
  - duplicating the threads is unnecessary as `exec( )` will replace the whole memory image of the process

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.

places a significant burden on the developers to ensure that the implementation avoids race conditions and other bugs

Implicit threading

- how to make writing multithreaded applications easier?
  - transfer thread creation and management to compilers and run-time libraries (implicit threading)
  - programmers just need to identify tasks that can run in parallel

Implicit threading

- how to make writing multithreaded applications easier?
  - transfer thread creation and management to compilers and run-time libraries (implicit threading)
  - programmers just need to identify tasks that can run in parallel
- OpenMP is a set of compiler directives and library routines available for FORTRAN/C/C++ that instruct the compiler to automatically generate a certain 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 that can be executed in parallel `#pragma omp parallel`

Explicit threading libraries

- Definition: API for creating and managing threads
- two primary ways of implementing
  - library entirely in user space
  - kernel-level library supported by the OS
- POSIX Pthreads, Windows thread library, and Java are three examples thread libraries

POSIX Pthreads Library

- IEEE 1003.1c is the POSIX standard for thread creation and synchronization
  - can be provided as user-level or kernel-level library
- API specifies behaviour of the thread library, not its implementation
- common in UNIX operating systems (Solaris, Linux, Mac OS X)
- WIN32 Threads: Similar to POSIX, but for Windows
  - In POSIX Pthreads and Windows thread library, global variables are shared among all threads of a given process

Pthreads:

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

Win32 threads:

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

POSIX Pthreads Library

- Pthreads has a thread container data type of `pthread_t` which is the handle of the 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 the specified thread to terminate
    - the calling thread is blocked until that currently executing thread has completed
  - `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( )`
- see `man pthread`

Example

Compile and link with `-pthread` flag

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

void *athread (void *arg) {
     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);
}

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

How to handle signals in a multithreaded program?

- in a single-threaded program signals are always delivered to the corresponding process

How to handle signals in a multithreaded program?

- in a single-threaded program signals are always delivered to the corresponding process
- how about multithreaded programs?
  - synchronous signals must be delivered to the thread that caused it
  - 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

How to handle signals in a multithreaded program?

- in a single-threaded program signals are always delivered to the corresponding process
- how about multithreaded programs?
  - synchronous signals must be delivered to the thread that caused it
  - 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 (a flag) if it should terminate; 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( )`

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 target 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

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 target 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( )`

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

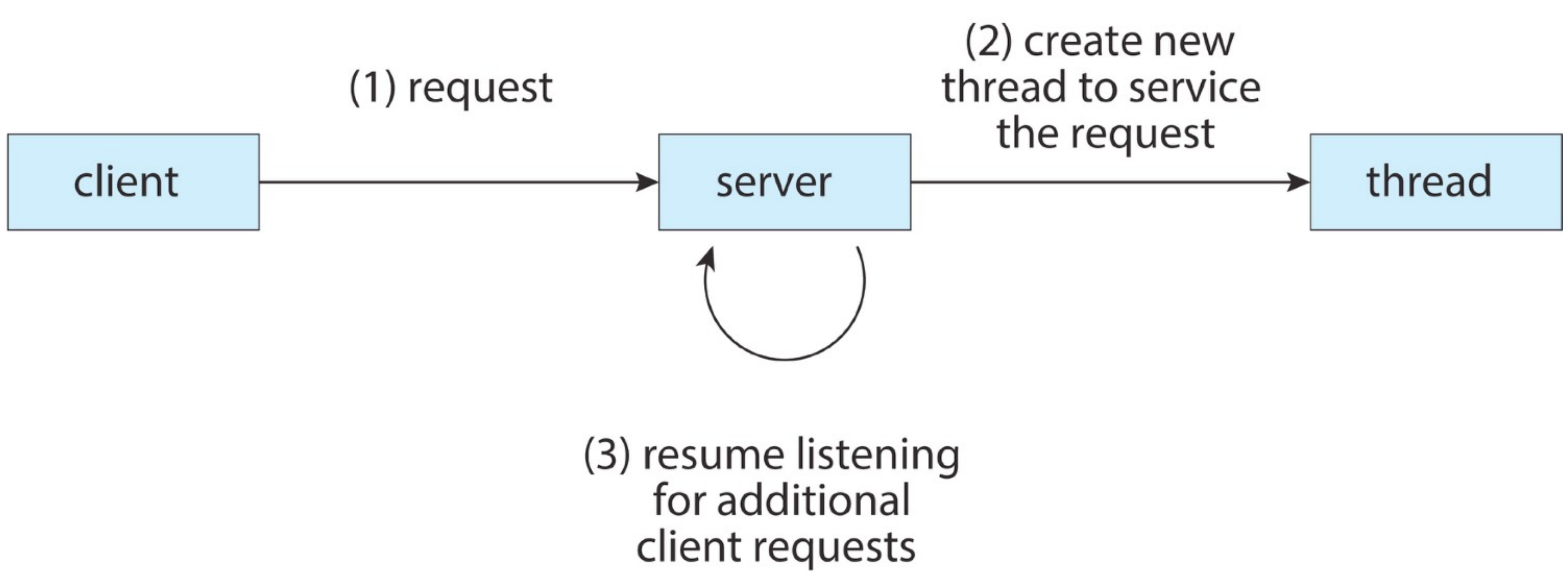
- a thread may set its cancellation state and type using the API
  - three options: disable cancellation; allow deferred cancellation; allow asynchronous cancellation
  - default is deferred cancellation; can be changed by `pthread_setcanceltype( )`
- many blocking system calls in the POSIX are cancellation points where a target 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 the thread's resources

Example: multithreaded server (loose syntax)

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

Problem: what if we get a lot of requests?

- might run out of memory
- schedulers usually have trouble with too many threads



Thread pool

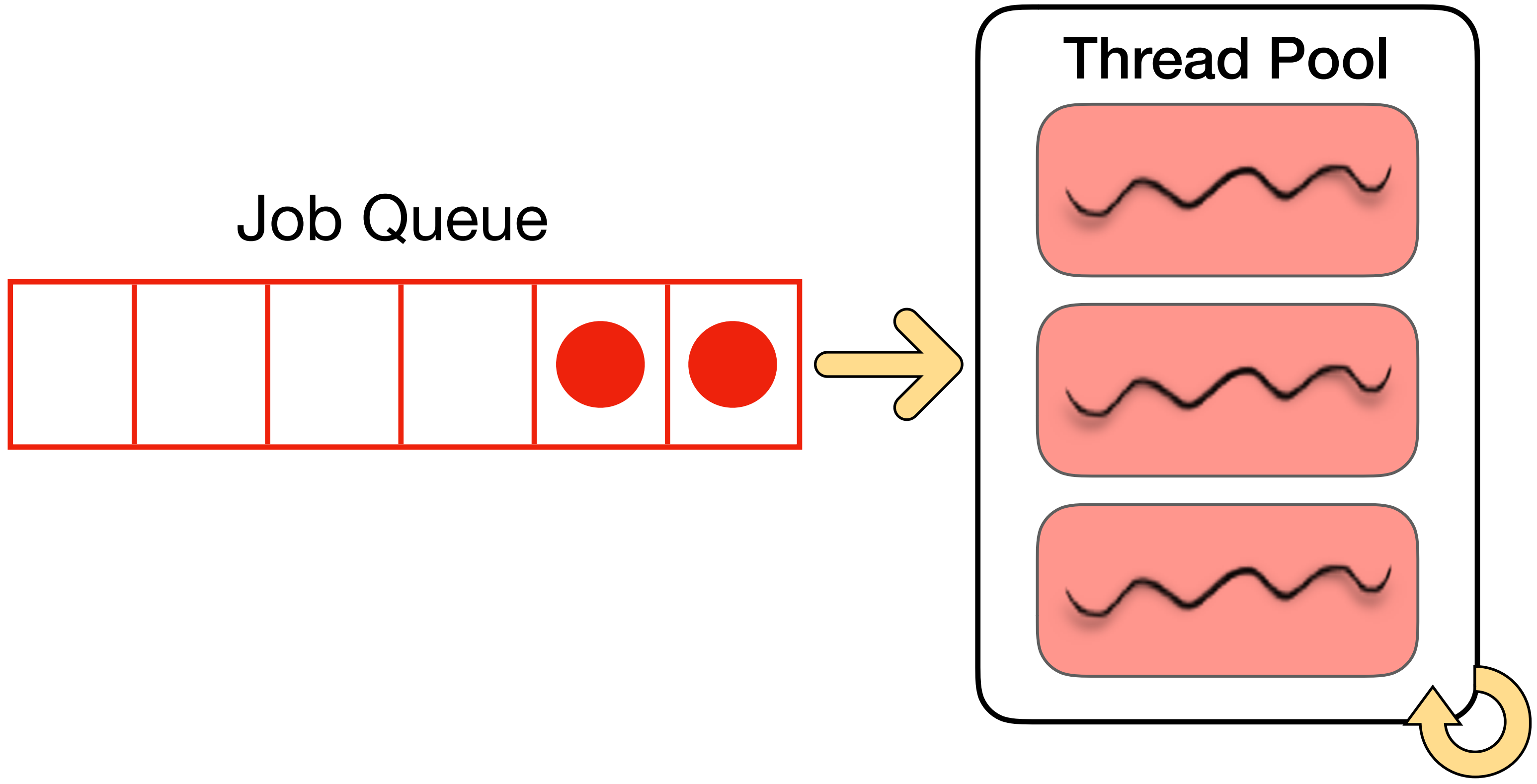
- Definition: a collection of worker threads that execute callback functions on behalf of the application
    - a callback function is a function passed into another function as an argument, which is then invoked inside that function

Thread pool

- Definition: a collection of worker threads that execute callback functions on behalf of the application
  - a callback function is a function passed into 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
    - more important when threads do a small amount of work
  - unlimited threads could exhaust system resources
    - they consume a significant amount of memory and contend for resources

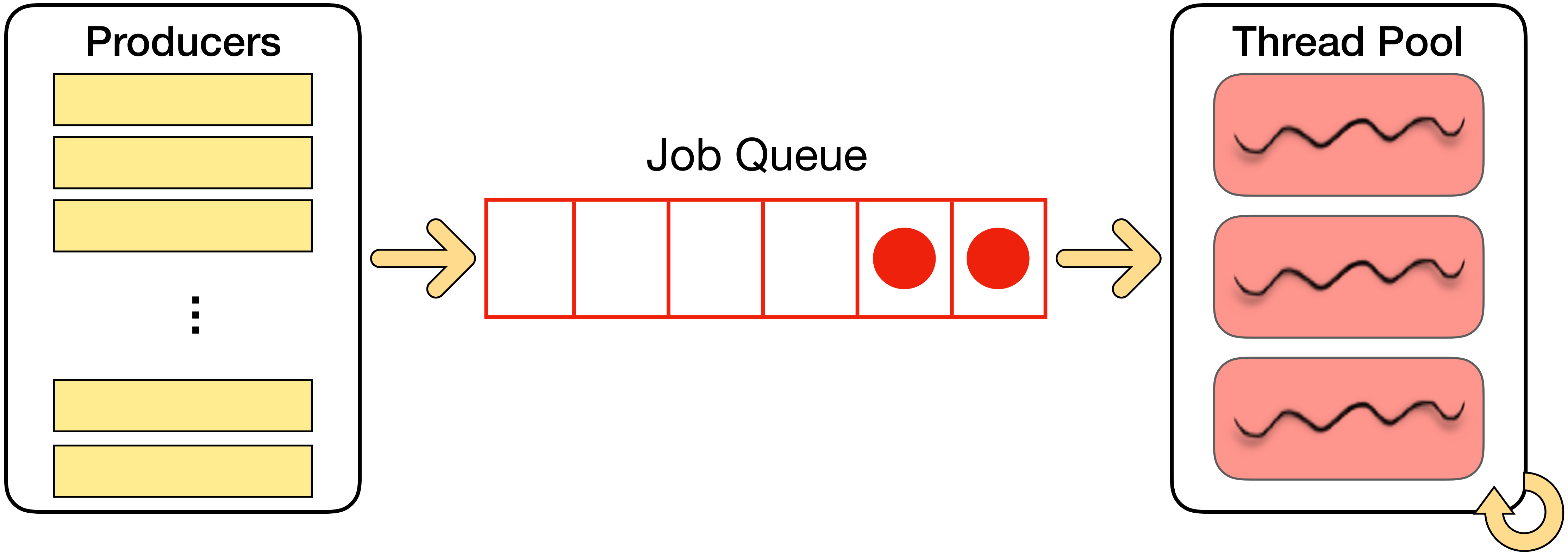
Thread pool — Producer/Consumer

- when a new request is received, a server submits it to the thread pool and resumes waiting for additional requests
  - if there is an available thread in the pool, it is awakened to immediately serve the request
  - otherwise the request is queued until a thread becomes available
    - requires having a queue of pending requests
- once a thread completes its service it returns to the pool (i.e., it becomes idle and ready to be dispatched to another task)



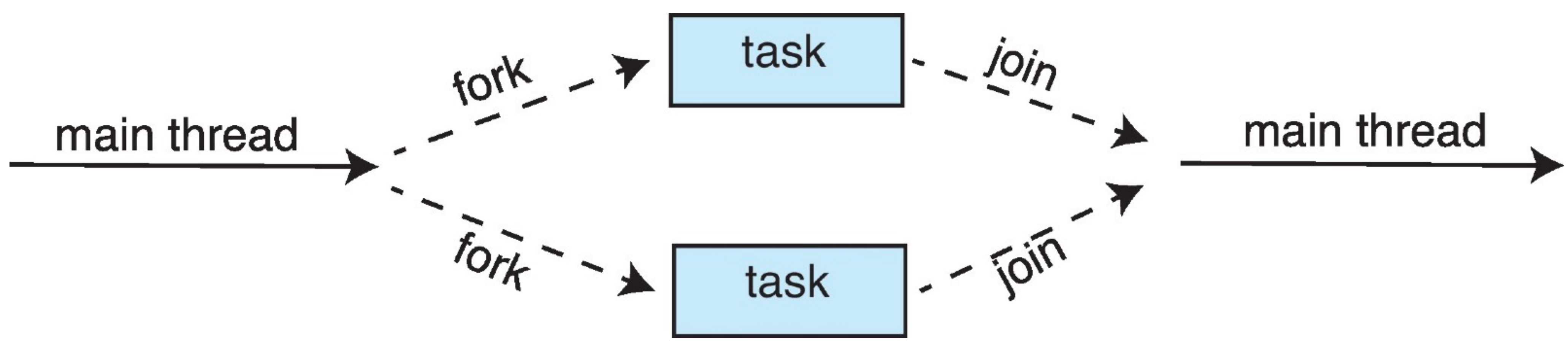
Thread pool — Producer/Consumer

- when a new request is received, a server submits it to the thread pool and resumes waiting for additional requests
  - if there is an available thread in the pool, it is awakened to immediately serve the request
  - otherwise the request is queued until a thread becomes available
    - requires having a queue of pending requests
- once a thread completes its 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 sequential execution after joining the spawned threads
- 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