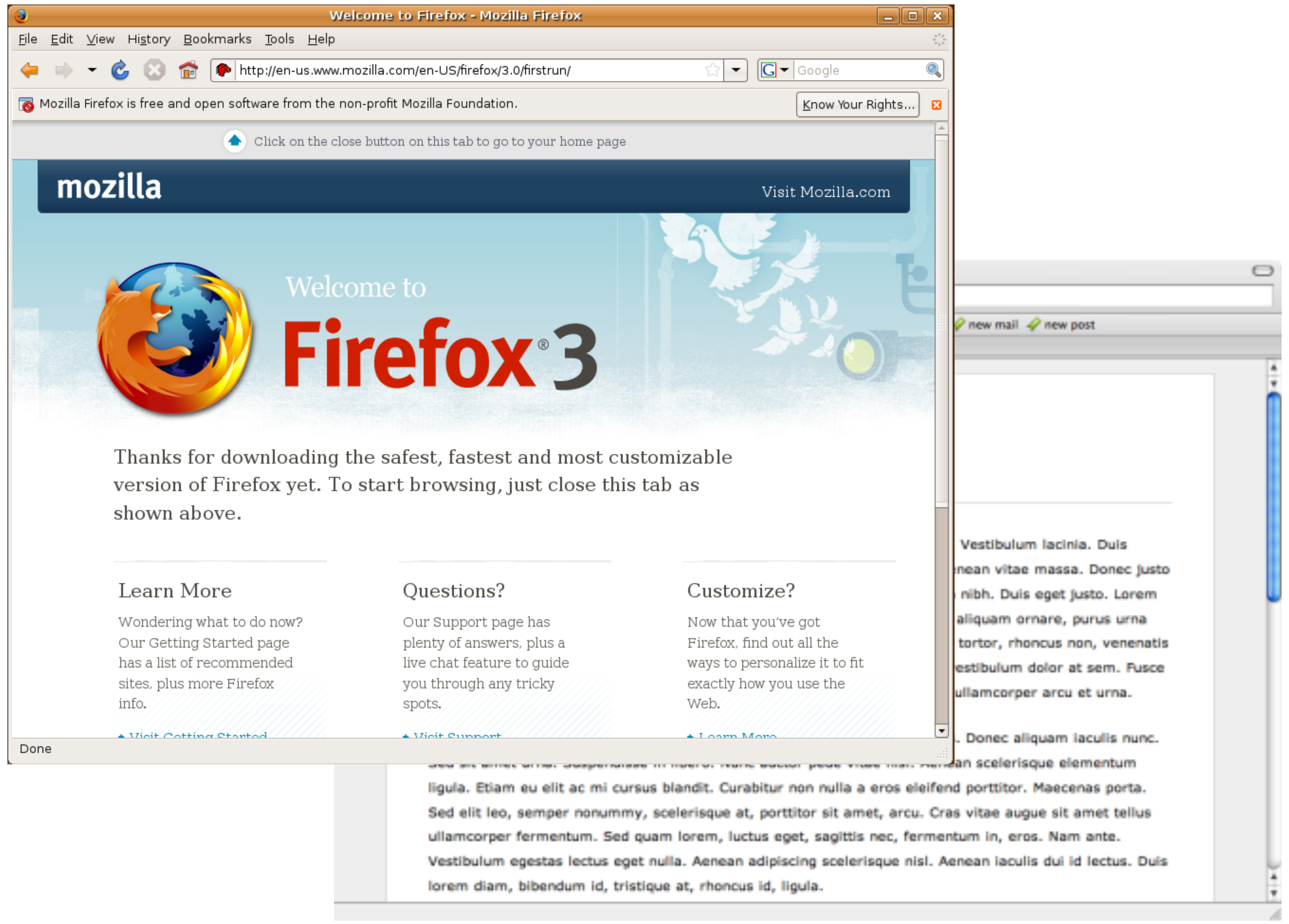
# Multi-Threading

## Lightweight Multi-Processing

# Motivation for Multi-Threading

- Most modern applications are multi-threaded
- Threads run within a process
- Multiple tasks within the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request



# Motivation for Lightweight Processes

- Process creation is heavy-weight — application can be slowed down by process creation, termination, and management overhead
- We can use process pools to reduce the overhead of creation on demand
- Thread creation is much more lightweight compared to process creation
- Threads also simplify code (modularization) and increase efficiency
- Kernels are generally multithreaded

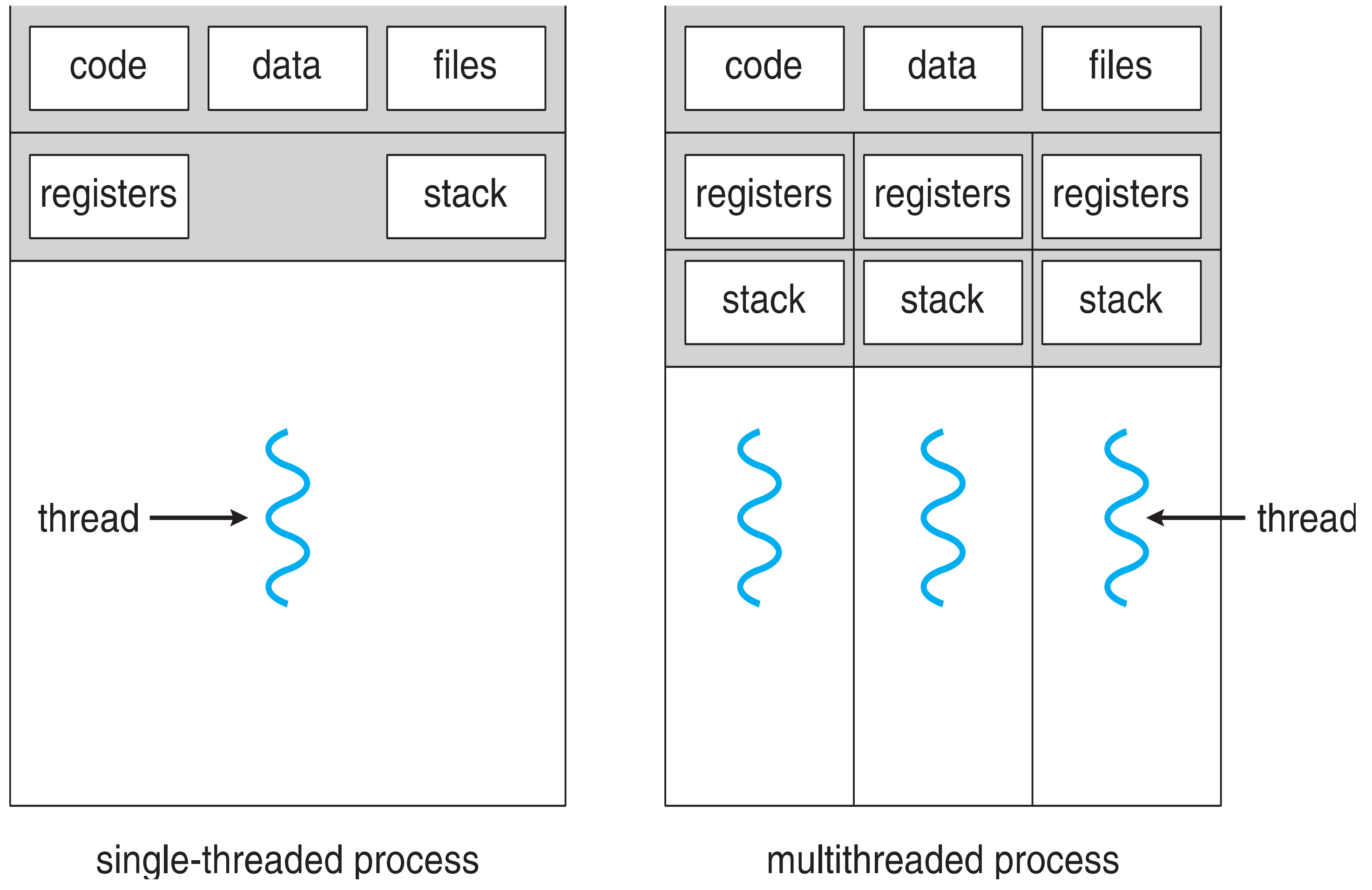
# Process Vs. Threads

**Processes**

- Create a new address space at process creation
- Allocate resources at creation
- Need IPC to share data
- Deeper isolation for security and fault tolerance

**Threads**

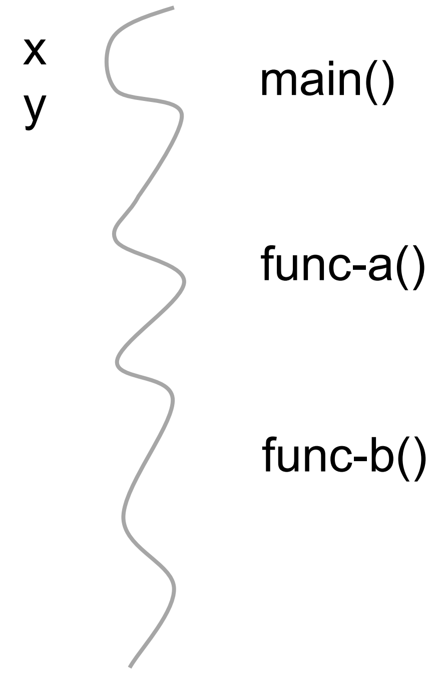
- Same address space
- Quicker creation times — actual times depend on kernel versus user threads
- Sharing through shared memory
- Fault sharing between all threads within a process



# Single Process Vs. Processes Vs. Threads

## Comparing all three

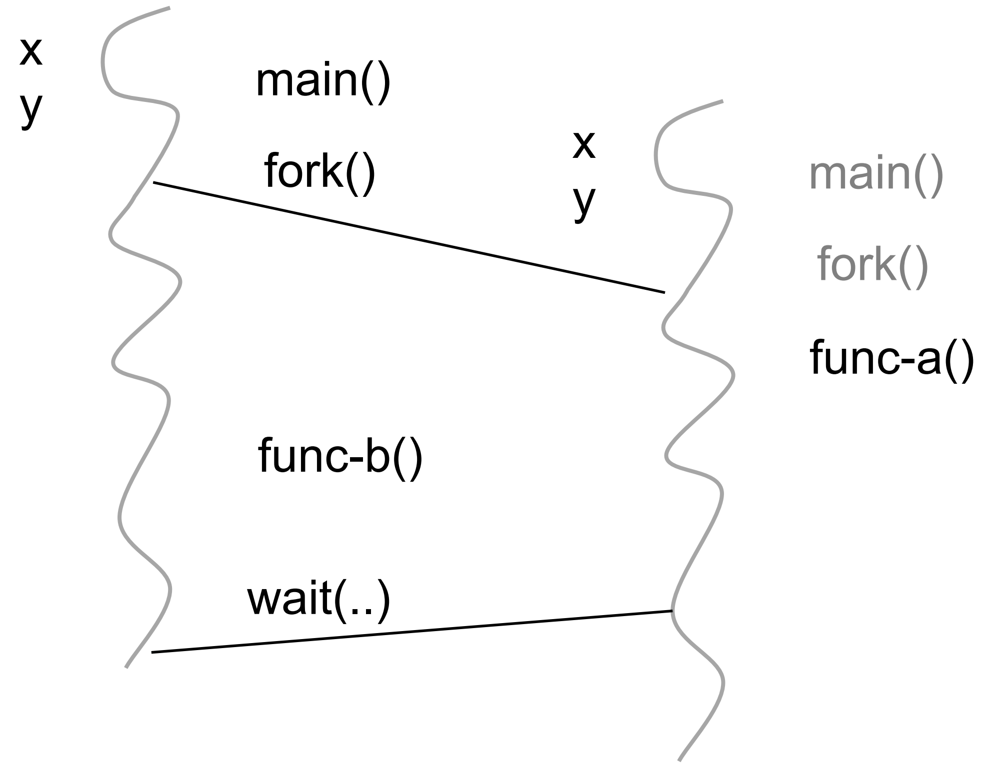
```
int x = 10;
int y = 20;

void func-a() {
        x = x +10;
        y = y – 10;
}

func-b() {
        printf(“x = %d”, x);
        printf(“y = %d\n”, y);
}

main() {
        func-a();
        func-b();
}
```



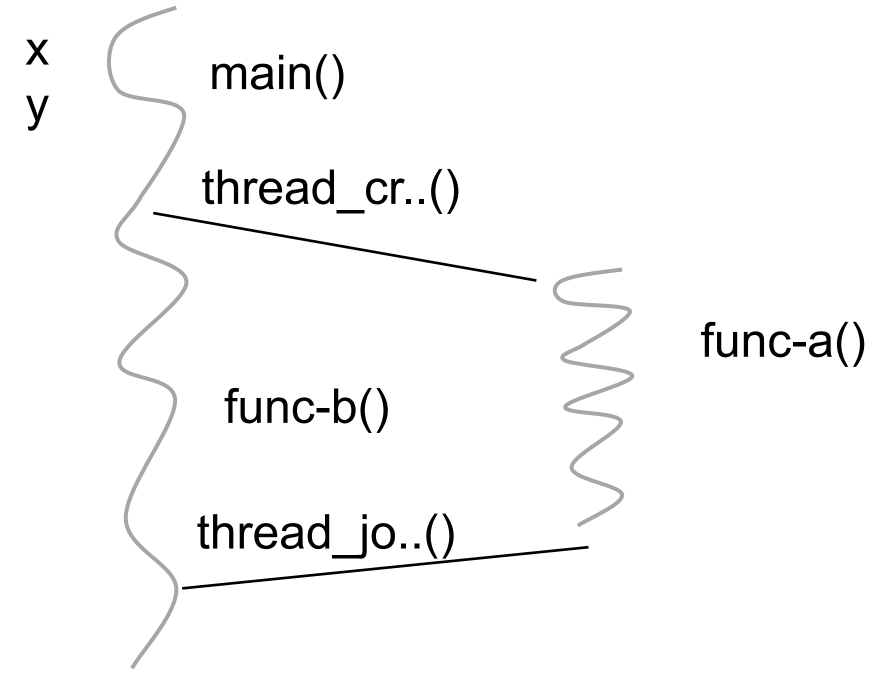

**Everything in a single process (default)**

**Using multiple processes**

```
int x = 10;
int y = 20;

void func-a() {
        x = x +10;
        y = y – 10;
}

func-b() {
        printf(“x = %d”, x);
        printf(“y = %d\n”, y);
}

main() {
        if (fork() == 0) func-a();
        func-b();
        wait(..)
}
```



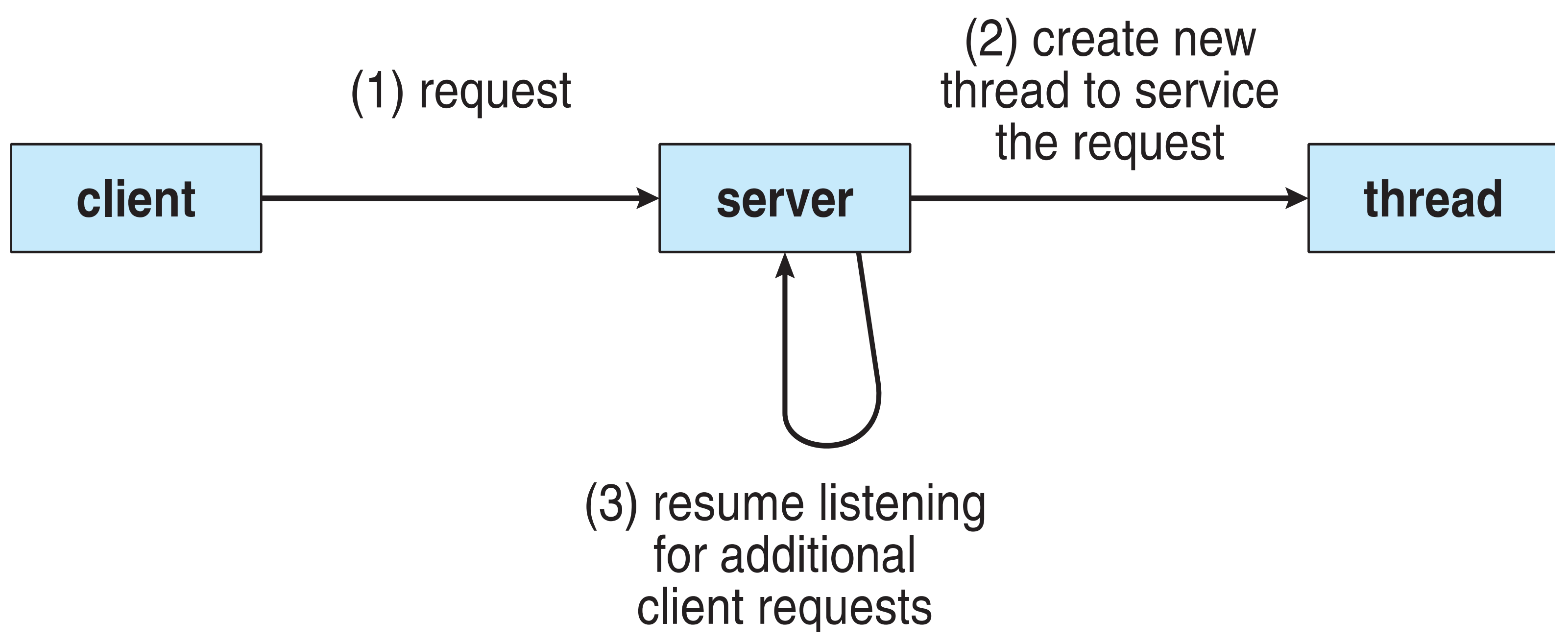

```
int x = 10;
int y = 20;

void func-a() {
        x = x +10;
        y = y – 10;
}

func-b() {
        printf(“x = %d”, x);
        printf(“y = %d\n”, y);
}

main() {
        thread_create_and_start(func-a());
        func-b();
        thread_join()
}
```

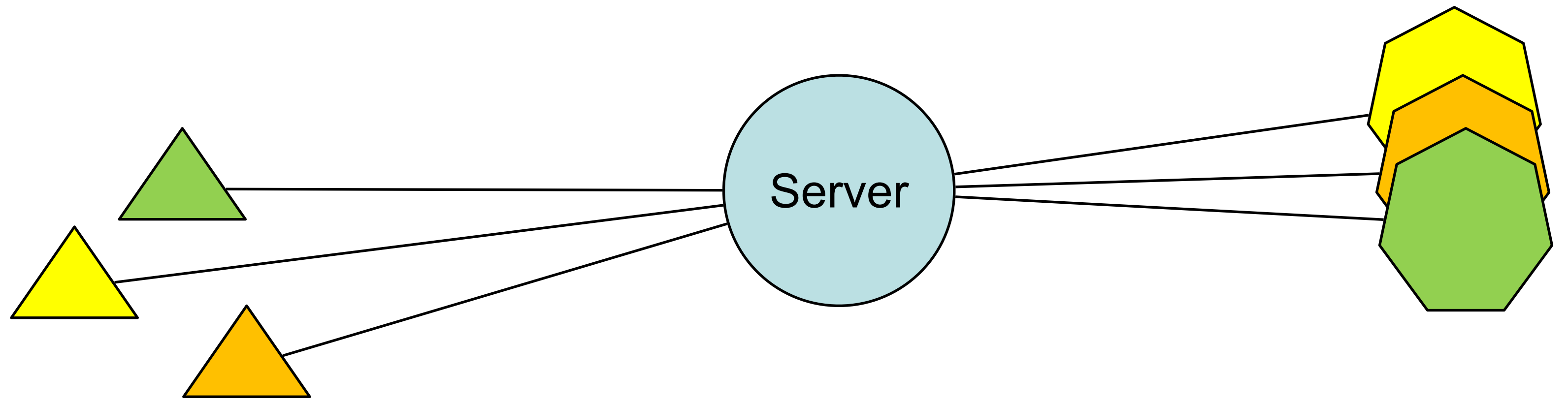



**Using multiple threads**

# Multi-Threaded Server Architecture

- Servers need to maximize throughput - number of connections handled per second
- Threads are lightweight so they minimize the overhead associated with request processing - accessing shared data, creating threads (on demand if needed), terminating old handlers, etc



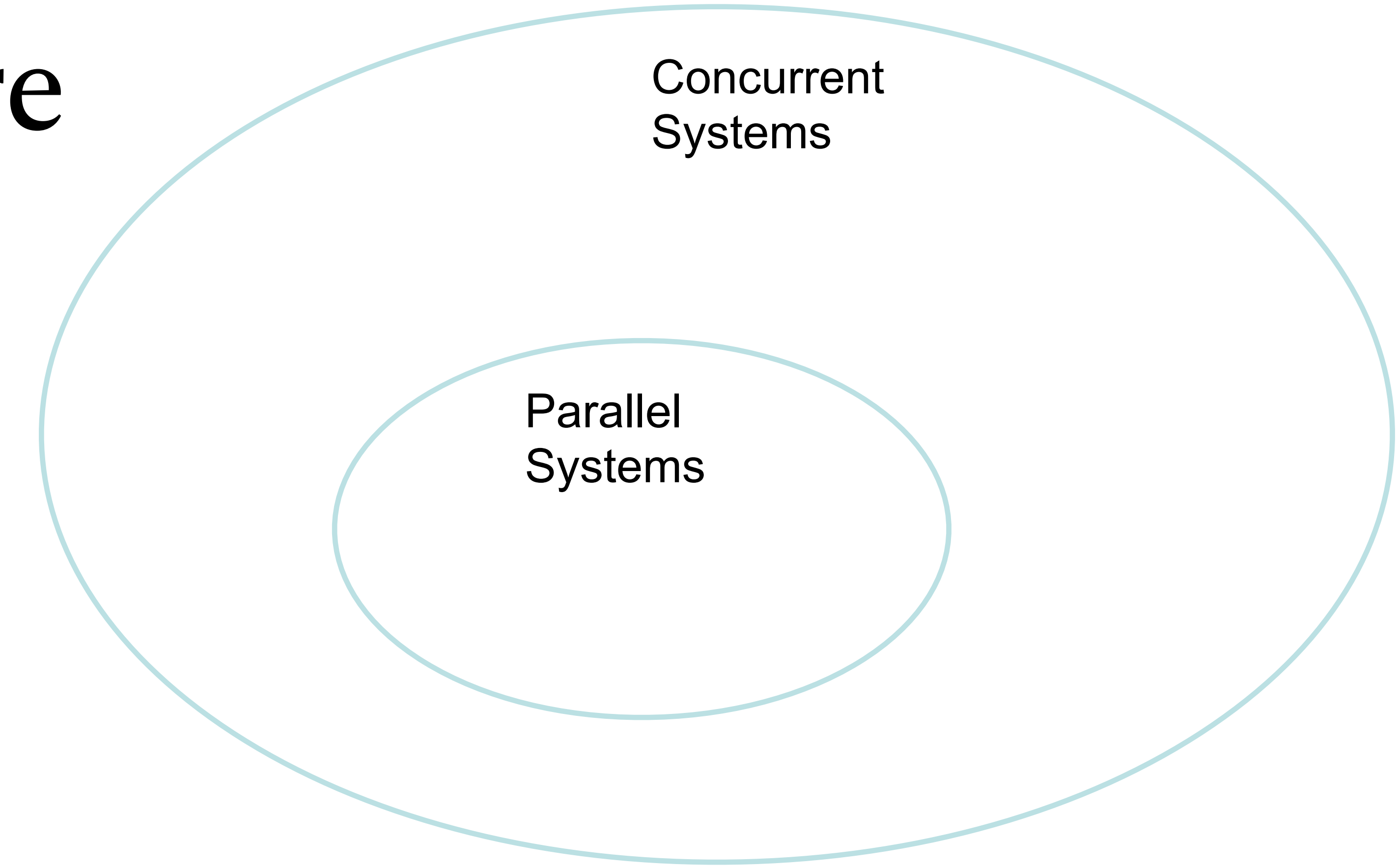



# Benefits of Multi-Threading

- **Responsiveness** - may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource sharing** - threads share resources of process, easier than shared memory or message passing
- **Economy** - cheaper than process creation, thread switching has lower overhead than context switching at the process level
- **Scalability** - process can take advantage of multiprocess architectures

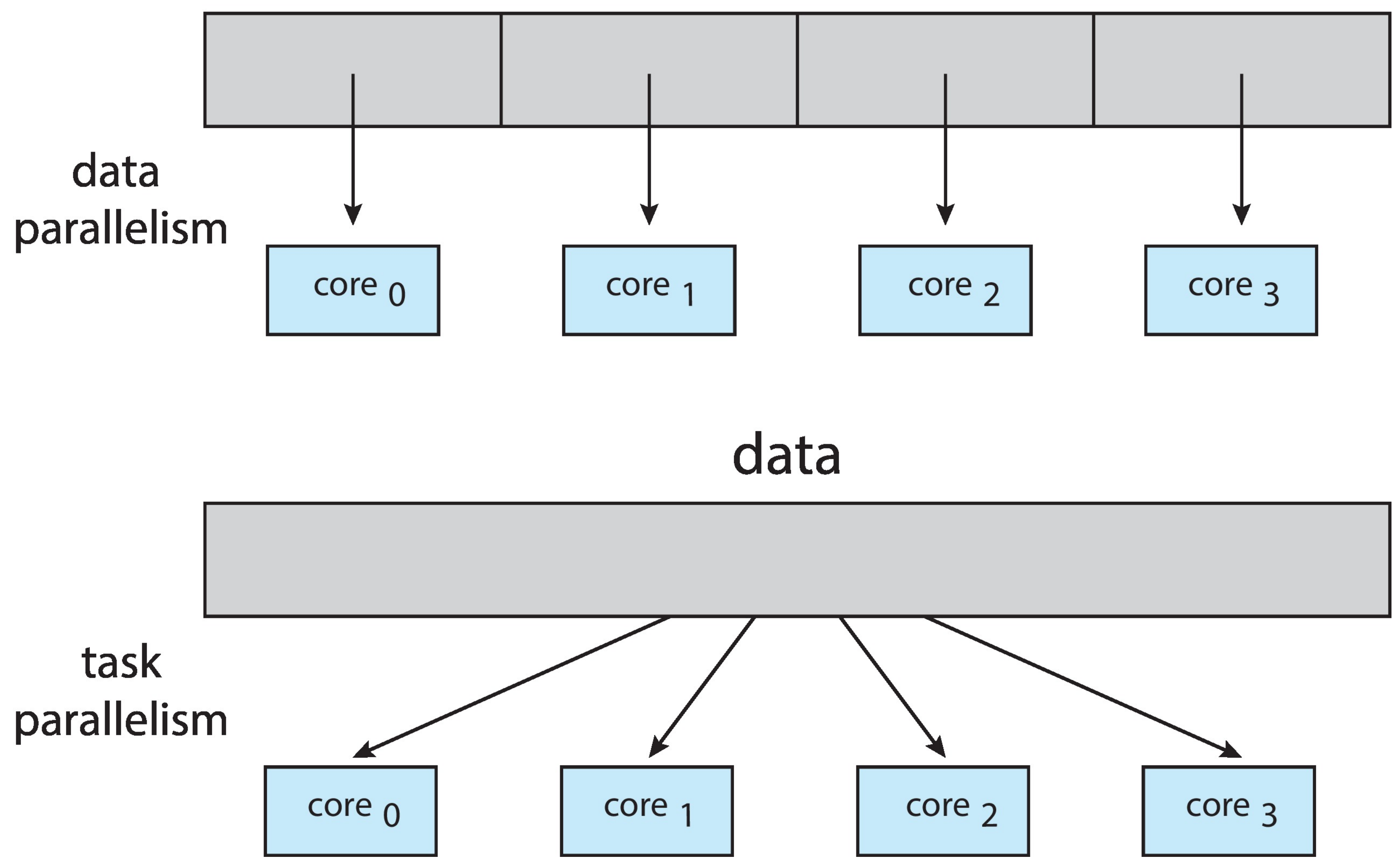
# Multi-Core Programming

- Multi-core or multi-processor programming is putting pressure on programmers
  - Dividing the applications
  - Balancing processing workload
  - Data splitting
  - Data dependency
  - Testing and debugging
- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress
  - Single process core, scheduler providing concurrency

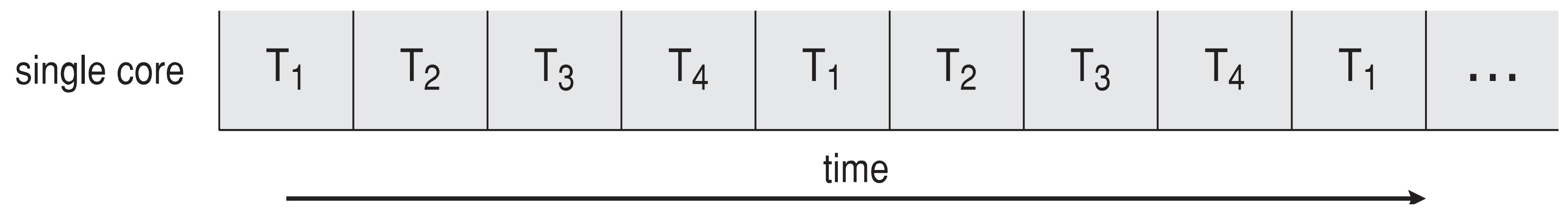


# Multi-core Programming

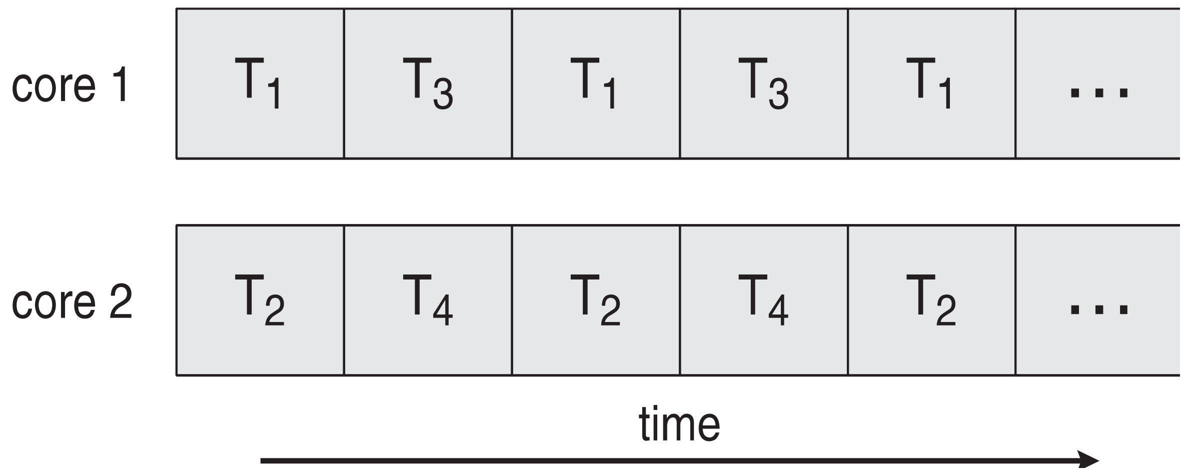
- Types of parallelism
  - Data parallelism - distributes subsets of the data across multiple cores, same operation on each data
  - Task parallelism - distributing threads across cores, each thread performing unique operation




- Concurrent execution on single-core system




- Parallelism on a multi-core system

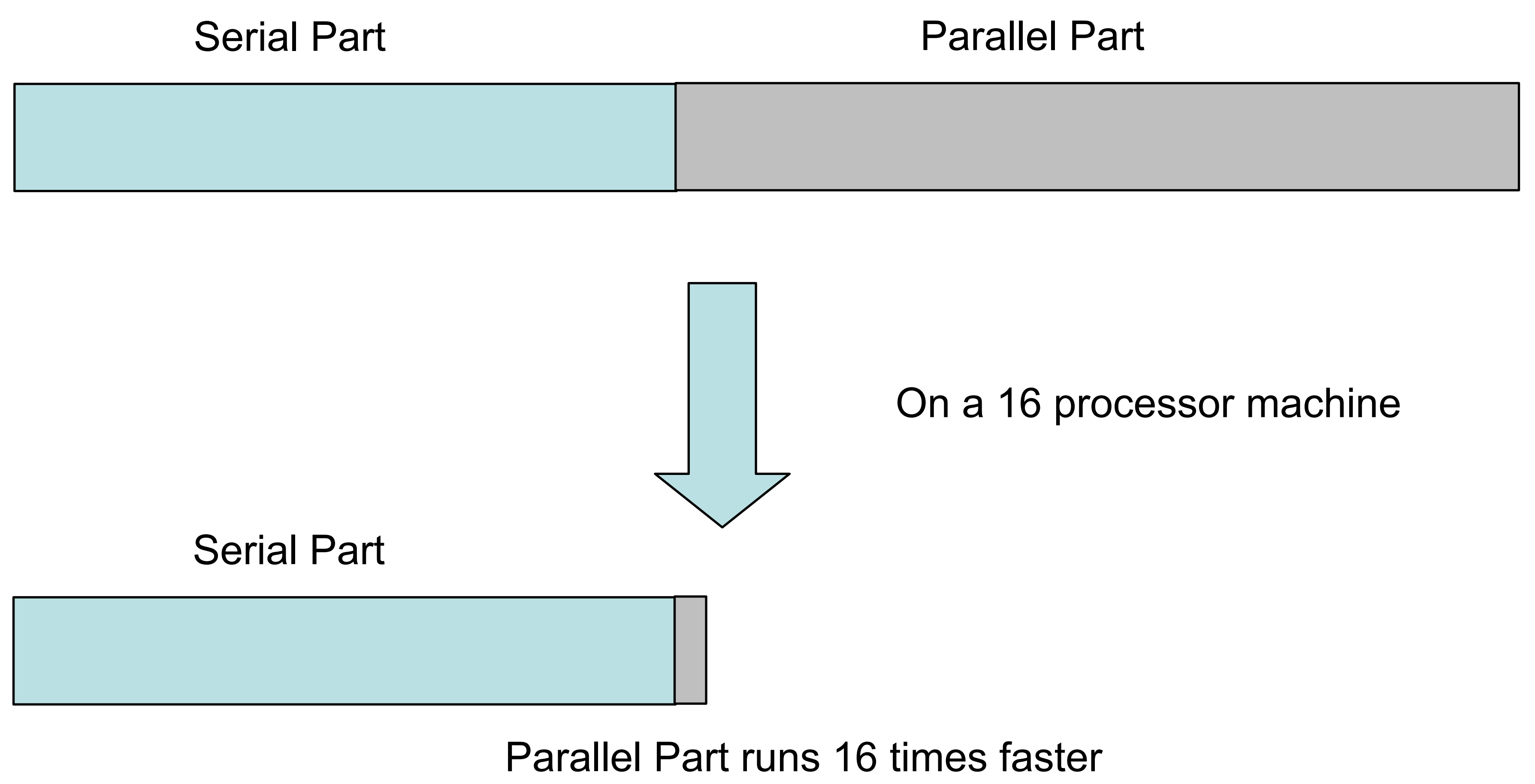


# Amdhal's Law

- Identifies performance gains from adding additional core to an application that has both serial and parallel components
- S is the serial portion
- N is the number of processing cores

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

Speedup = Serial Time/Parallel Time




**Problem**: An application is 75% parallel / 25% serial, moving from 1 to 2 cores results what will be the speedup?

Resulting improvement from an enhancement is "limited" by the fraction of the task that can be improved

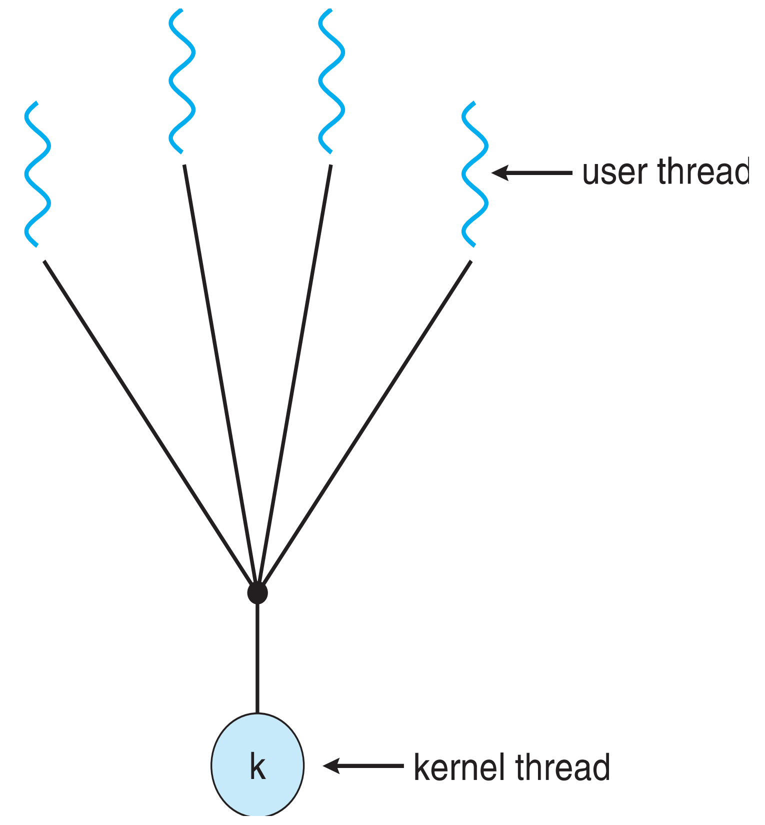
# User Vs. Kernel Threads

- User threads - management done by user-level thread library (GNU Pth)
- Kernel threads - supported by the kernel
- Examples - virtually all general purpose OSes provide kernel level threads including: Windows, Linux, MacOS
- POSIX Threads - is a thread programming standard - most of the time implemented as kernel-level threads

# Different Threading Models

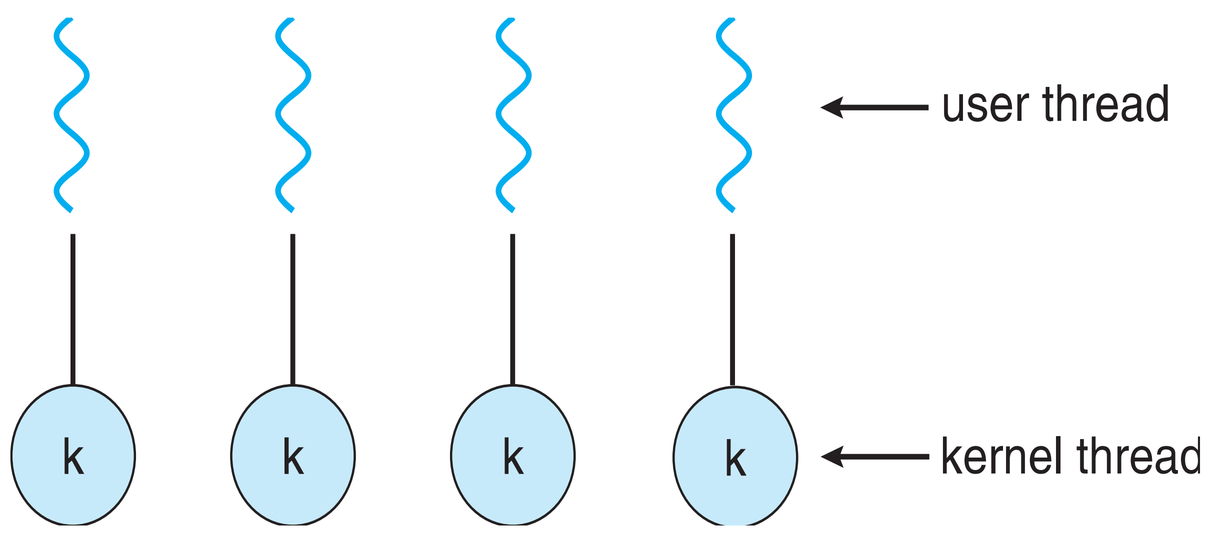
**<u>Many-to-One</u>**

- Many user-level threads mapped to single kernel-level thread
- One thread blocking causes all to block
- Multiple threads cannot run in parallel in multi-core system because kernel level has one thread
- GNU Pth is one example



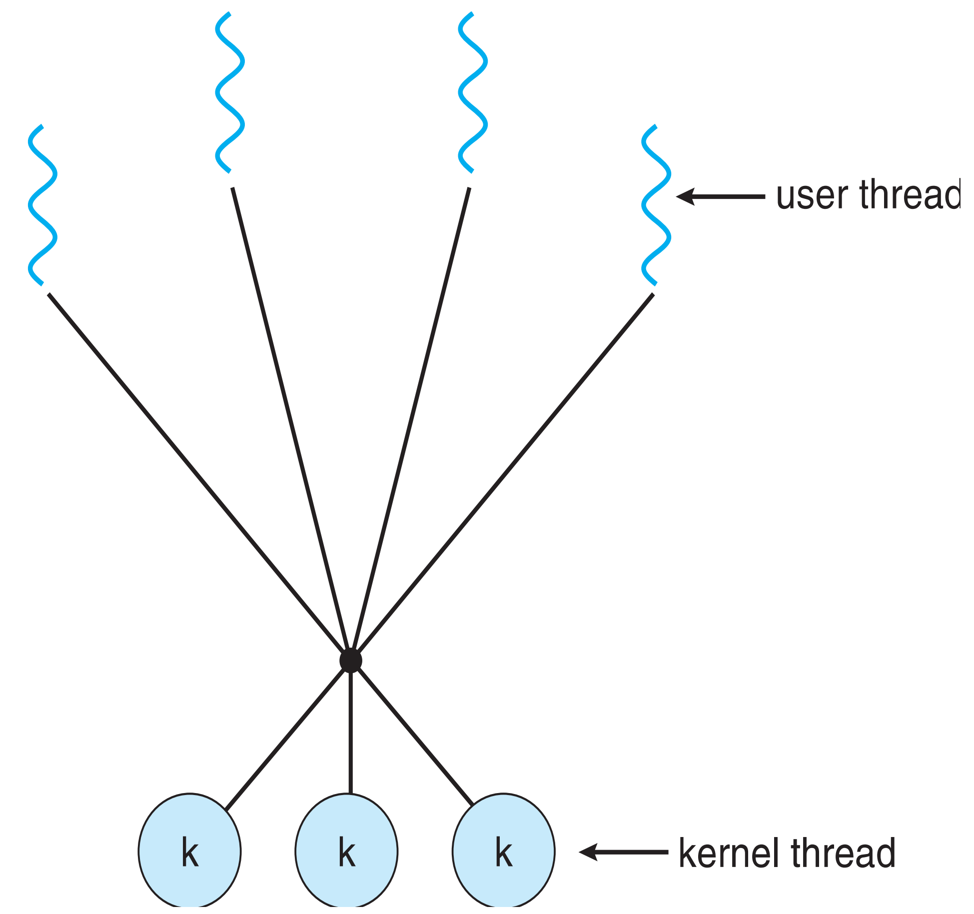

**<u>One-to-One</u>**

- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead



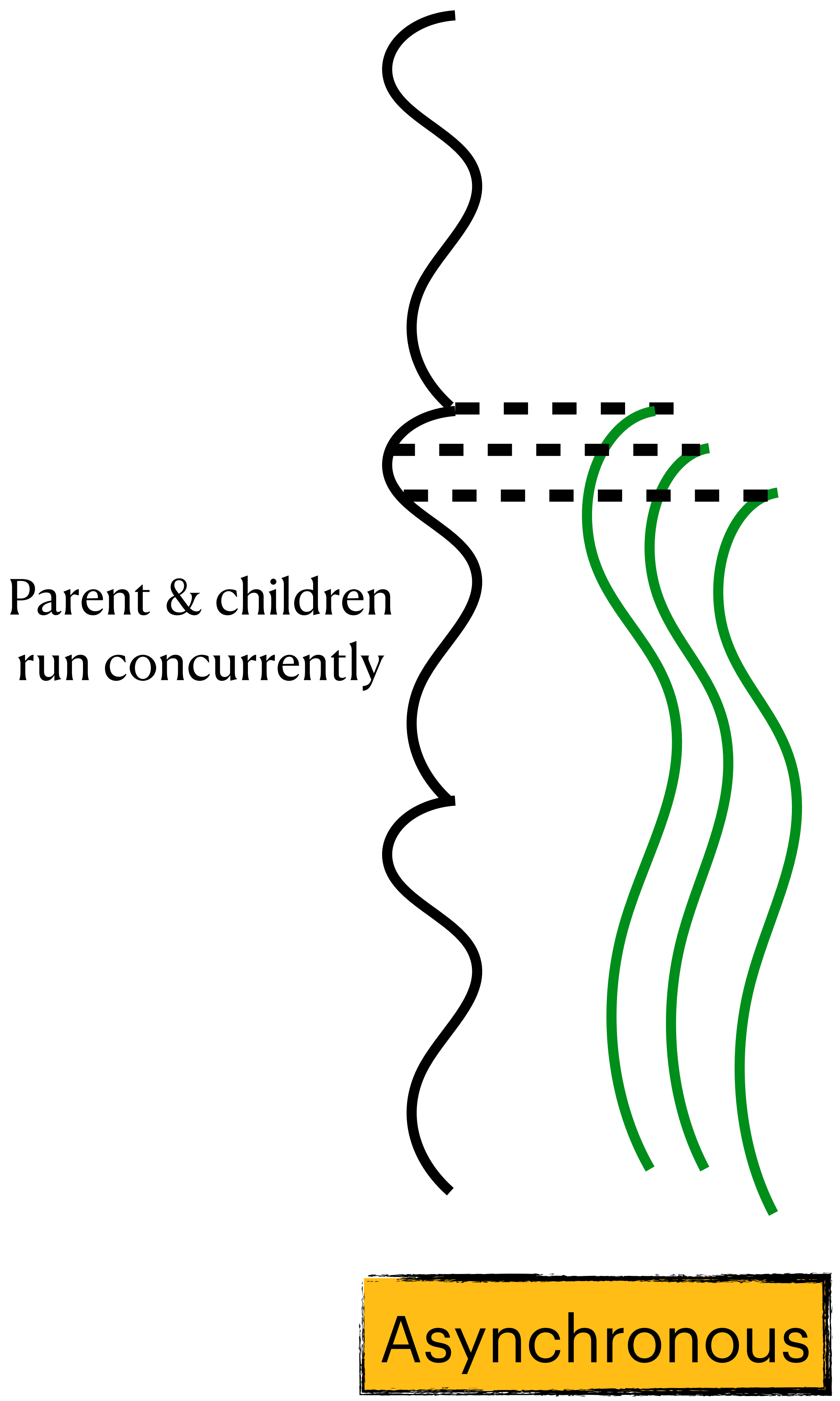

**<u>Many-to-Many</u>**

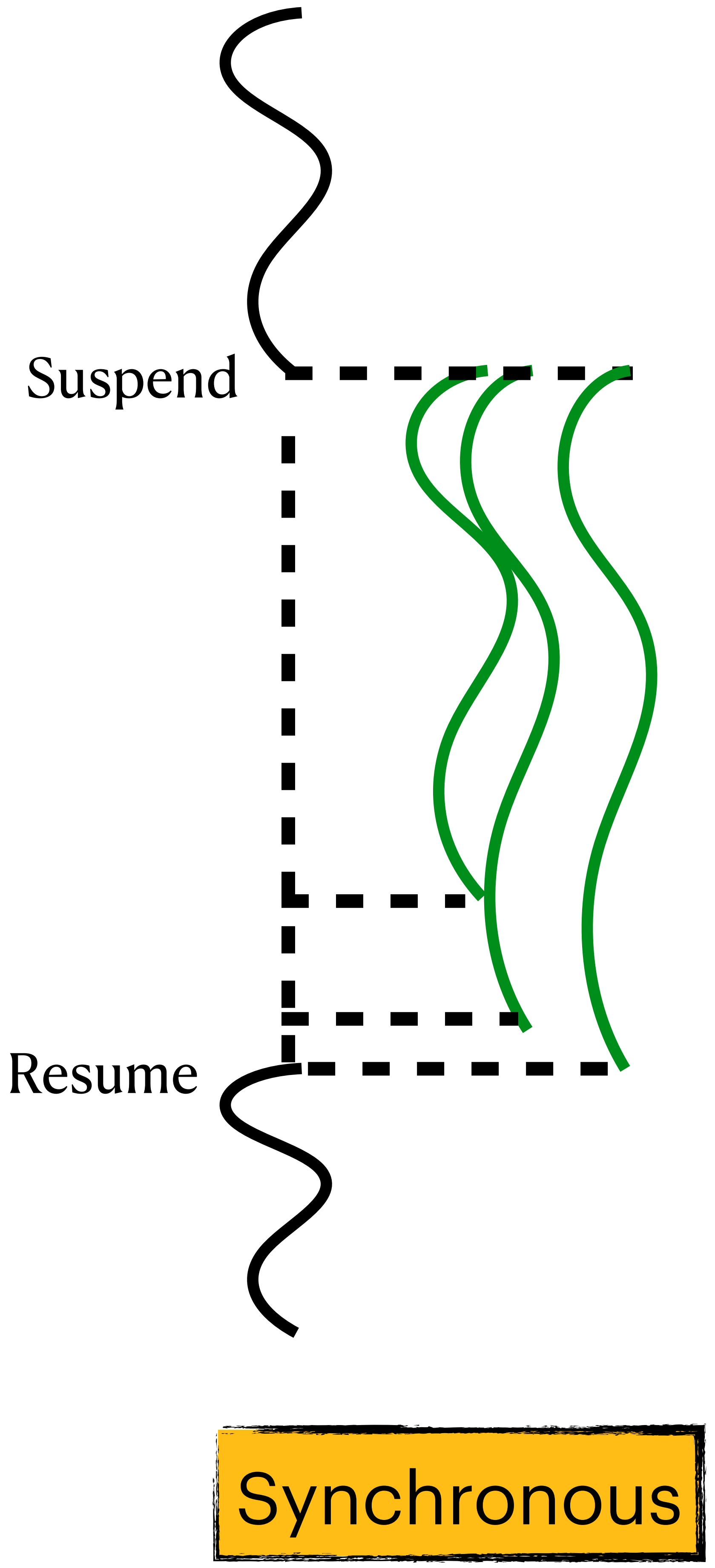
- Allows many user level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel thread
- At any given time a user thread is mapped onto a single kernel level thread
- Locality of mapping is a concern



# Asynchronous Vs. Synchronous Creation

- Asynchronous threading
  - Parent spawns a child thread, parent resumes execution
  - Parent and child continue concurrently and independently of one another
  - In a typical deployment, parent and child share little data
- Synchronous threading
  - Parent spawns a child thread and waits for child to terminate
  - Children exchange lot of data among themselves - no data with the parent



# Thread Libraries

- Thread library provides the API for creating and managing threads
- Thread libraries
  - Entirely in **user space** with no kernel support
  - Invoking a thread function is an entirely user-level affair - no system call
  - **Kernel-level thread library** supported by the Operating System - library exists is kernel space
- Kernel-level thread library: POSIX Pthreads (could have user-level versions), threads in Windows, threads in Java

# Implicit Threading

- We have CPUs with many cores - how to parallelize code to exploit the hardware parallelism (we are ignoring the GPU side of things)
- **Option A**:
  - Ask the programmer to explicitly organize the code using threads
  - Ask programmer to split the data
- **Option B**:
  - Let programmer identify tasks - use automatic ways of running tasks concurrently by mapping them to available threads - task is a function

# Implicit Threading - Thread Pools

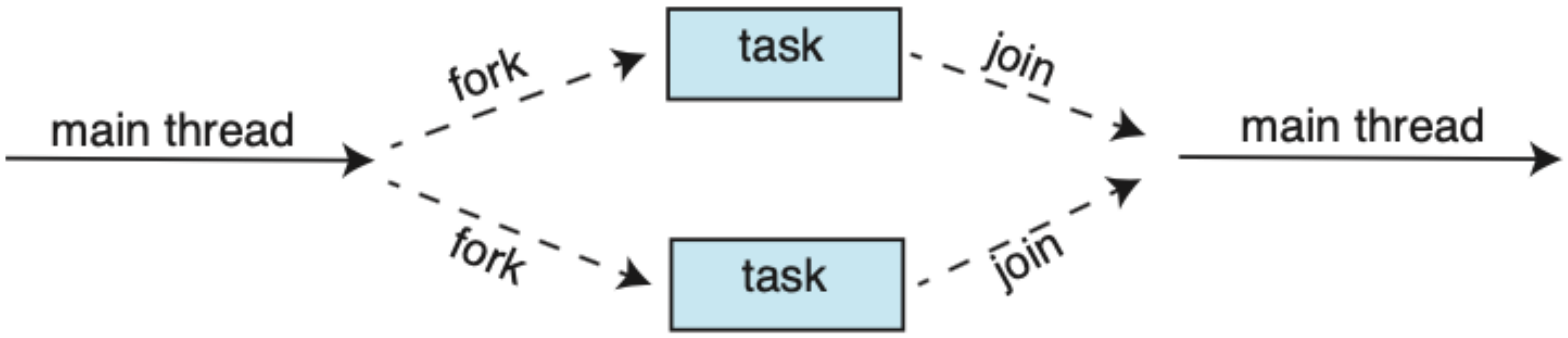
- Thread pools solve two problems
- On demand threads have a thread creation latency - incoming request (task) have to wait for the thread to be created
- Also, we cannot control the maximum number of threads created - too many threads can create resource exhaustion

**<u>Thread pools</u>**

- Create a fixed set of threads ahead of time
- Select available thread within the pool to serve the next request
- Queue the request if there are free threads in the pool

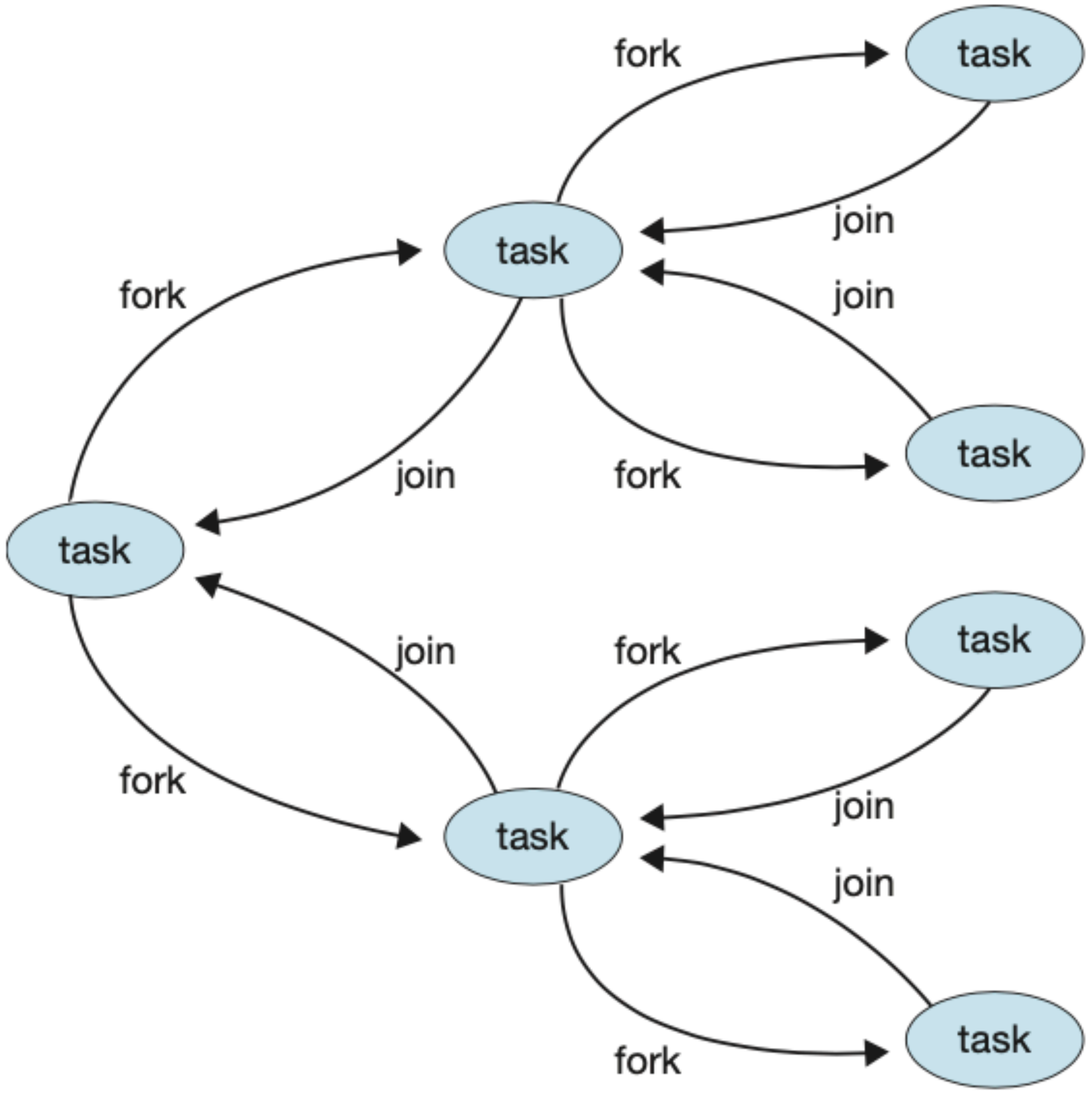
# Fork-Join Parallelism

- A model for managing large scale parallelism
- Fork (spawn) threads - join (wait on a child) thread
- Fork could be for the main task to create sub tasks
- Join is to gather all the results from the subtasks



# Another Fork-Join Example

```
Task(problem)
  if problem is small enough
    solve the problem directly
  else
    subtask1 = fork(new Task(subset of problem)
    subtask2 = fork(new Task(subset of problem)

    result1 = join(subtask1)
    result2 = join(subtask2)

    return combined results
```



# Open Multi-Processing (OpenMP)

- OpenMP is a set of compiler directives to implicitly parallelize C, C++, FORTRAN code
- Programmers insert compiler directives (special comments) to indicate code that should be parallelized

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

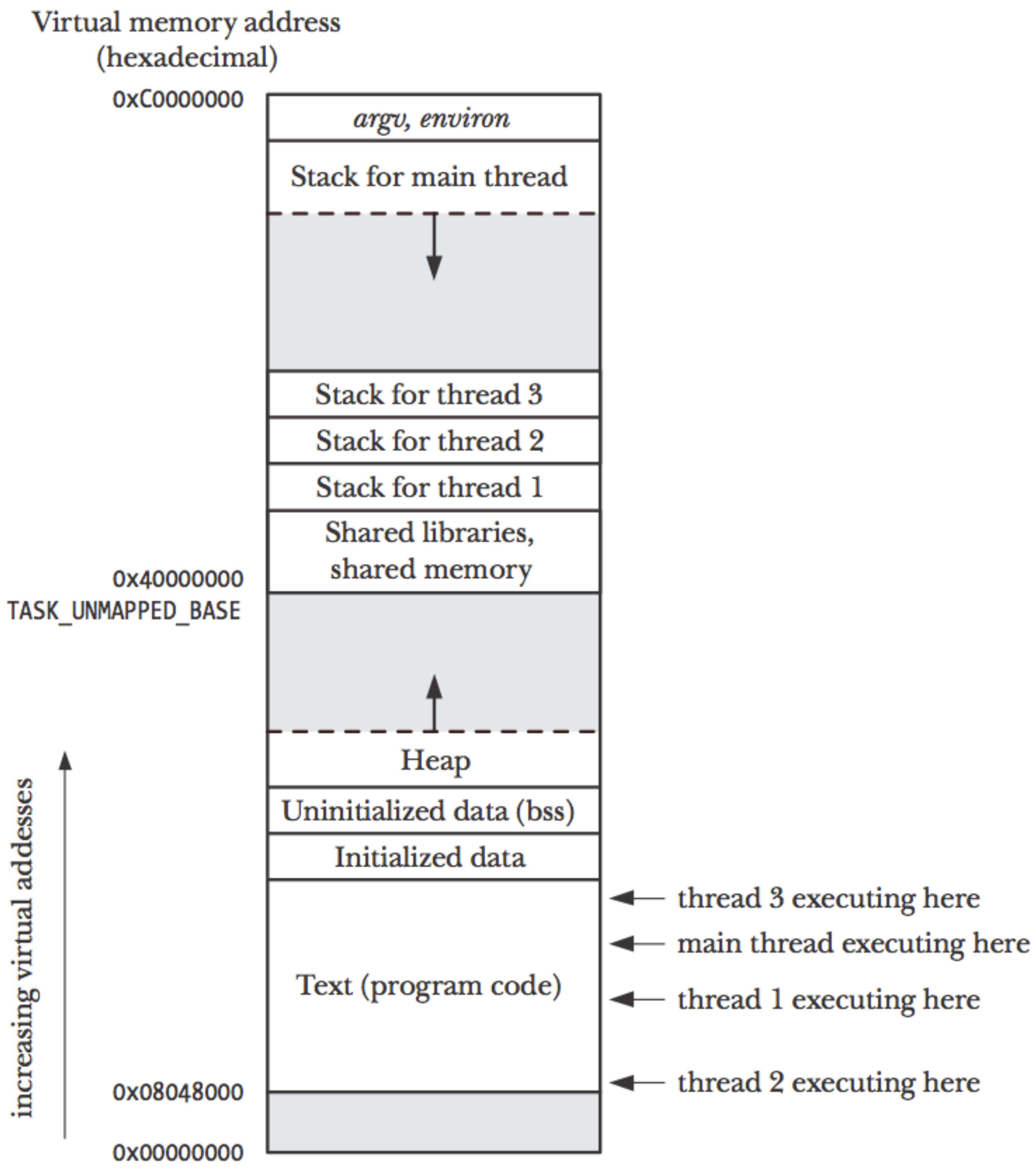
# POSIX Pthreads

- Pthreads is a specification for thread behaviour - not an implementation
- OS designers provide the Pthreads implementation

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

#include <stdlib.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 */

  /* set the default attributes of the thread */
  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 execute 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);
}
```

# Threads in Linux

- Four threads executing in Linux
- Kernel level threads
- Threads have specific stacks



# Pthread Primer

- Creating a thread - new thread continues with *start()* and main continues with the statement after

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void *), void *arg);
```

Returns 0 on success, or a positive error number on error

- A thread can terminate for the following reasons
  - start() function returned
  - thread calls pthread_exit()
  - thread is cancelled using pthread_cancel()
  - any thread call exit() or main thread returned

```
include <pthread.h>

void pthread_exit(void *retval);
```

- Each thread is uniquely identified by an ID
  - returned to the caller of pthread_create()
  - can be obtained using pthread_self()

```
include <pthread.h>

pthread_t pthread_self(void);
```

Returns the thread ID of the calling thread

- IDs allow checking if two threads are the same

```
include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns nonzero value if *t1* and *t2* are equal, otherwise 0

- A thread can wait for another thread using the pthread_join()

```
include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Returns 0 on success, or a positive error number

# Simple Pthread Example

- Parent creates child
- Child prints a message
- Parent & child join (child terminates at that point)

```c
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}

int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

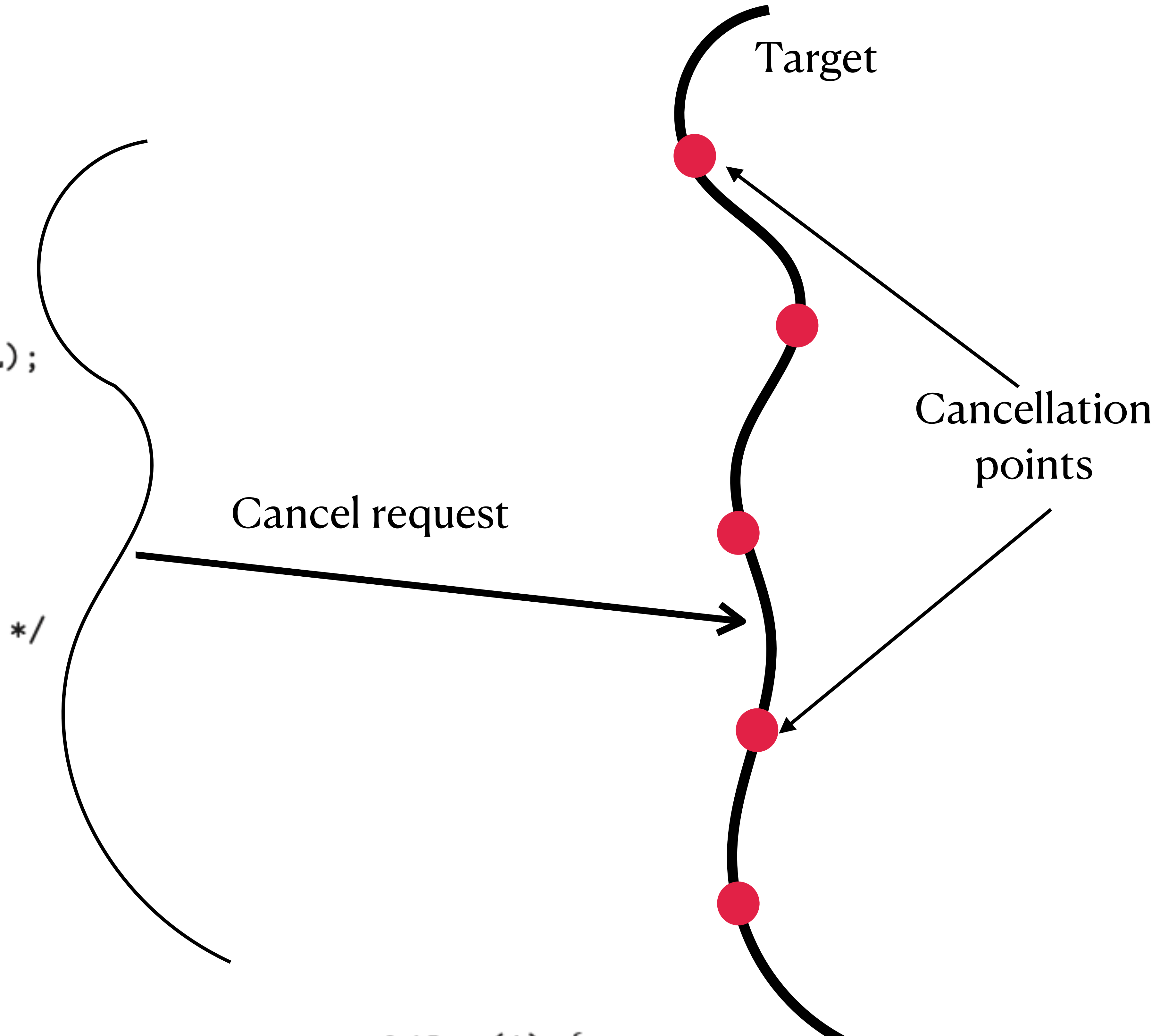
# Thread Cancellation

- User launches a request for downloading a webpage
- Hits cancel to stop the request
- We need to cancel the thread that is launching the network request
- Cancelled thread is known as the **target** thread
- Target can be cancelled in two different scenarios
  - **Asynchronous cancellation** — immediately terminate the target thread
  - **Deferred cancellation** — target thread periodically checks whether it should terminate in an orderly manner

# Deferred Thread Cancellation

- Cancellation is initiated using **pthread_cancel()**
- Target is checking for the reception of cancel at pre-defined points
- If a cancel is there, **pthread_testcancel()** will not return - thread will terminate

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```




```
while (1) {
  /* do some work for awhile */

   . . .

  /* check if there is a cancellation request
  pthread_testcancel();
}
```