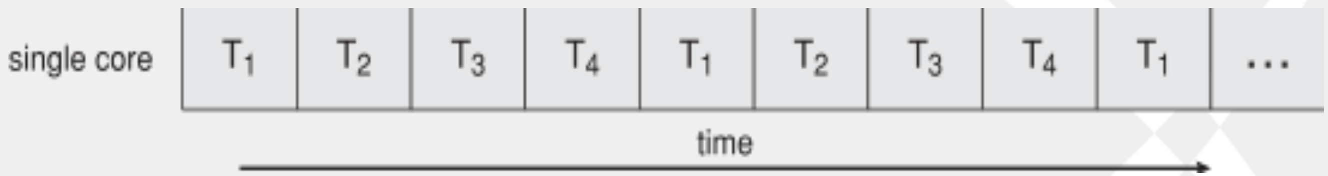
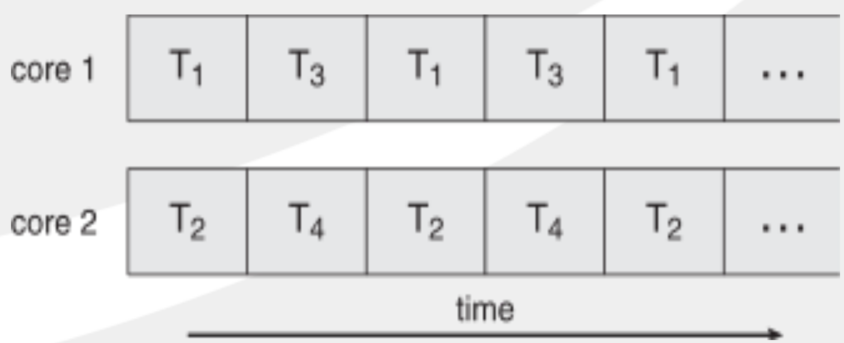
Processes and Threads (3)

Concurrency vs. Parallelism

- Concurrent execution on single-core system:

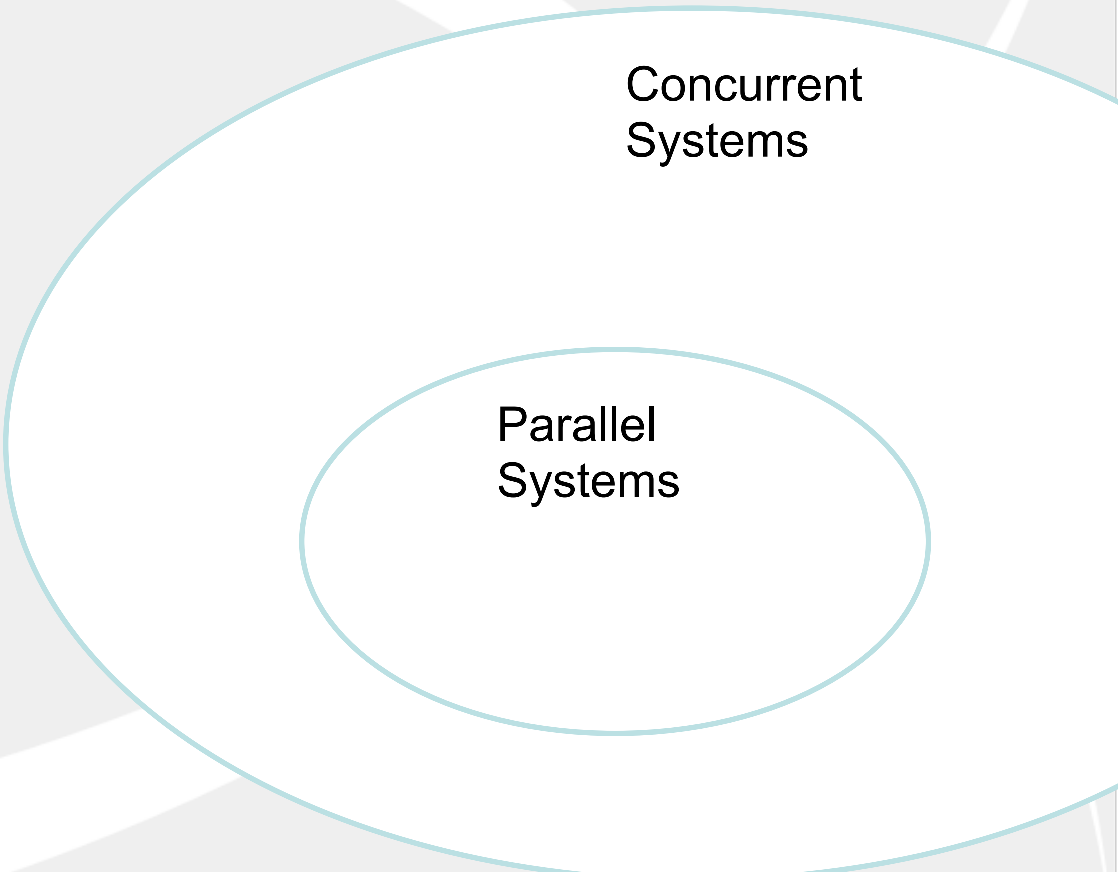


- Parallelism on a multi-core system:



Concurrency vs. Parallelism

- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency



Purpose of Parallelism

- Why do we want to use parallelism?
  - We want to run some computation faster
  - How fast can we go is an interesting question
- Can we keep on reducing the computation time

Application Structure

- Application are not completely parallel
  - Serial portion
  - Parallel portion

Example

Amdhal's Law

- Performance improvement obtained by an applying an enhancement on an application execution is limited by the fraction of the time the enhancement can be applied.

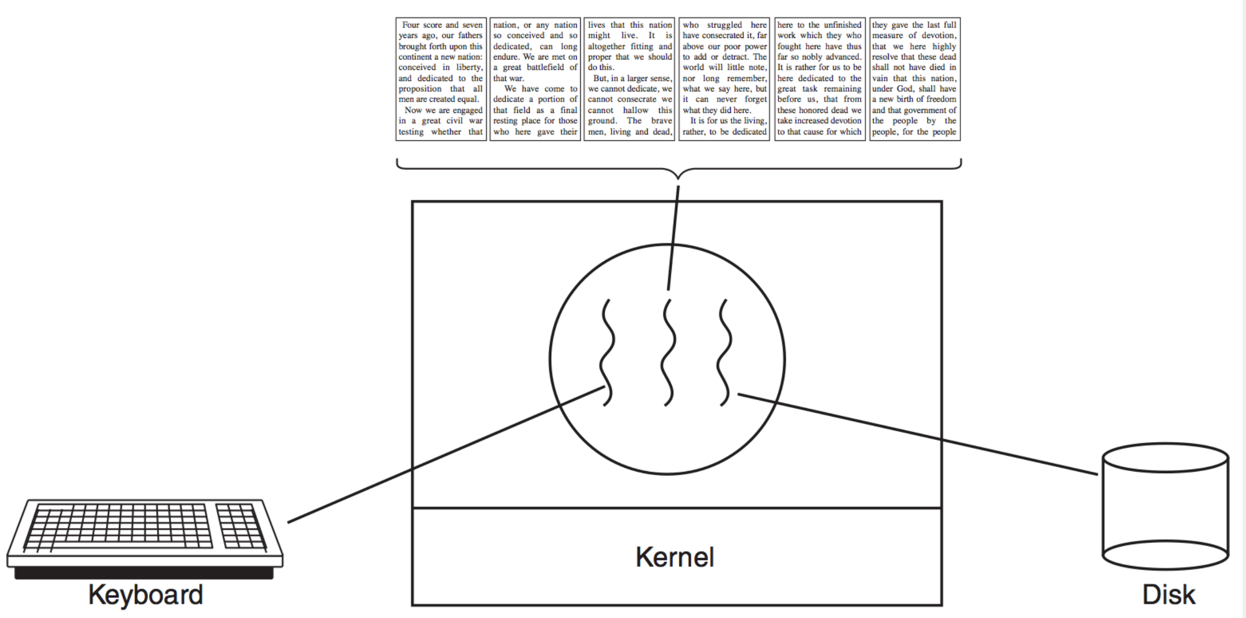
Speedup for an Application

- Compute the speedup obtainable by adding N cores to an application
- *S* is serial portion
- *1-S* is the parallel portion

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

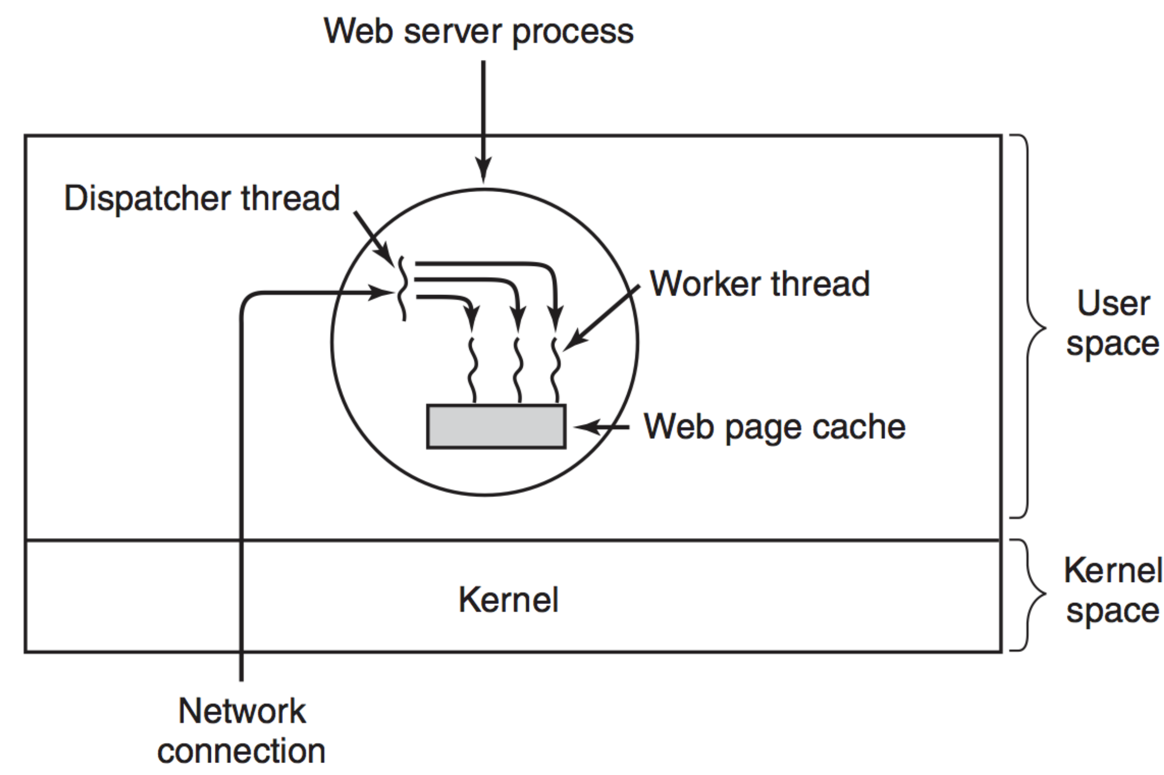
Motivation for Threads

- Most modern applications are multithreaded
- Why?
  - Applications want to do many different **not so tightly coupled** activities
  - Threads can make such activities happen at the ”same” time



Another Application: Multi-threaded Web Server

- Web servers are multi-threaded to handle many requests in a given amount of time



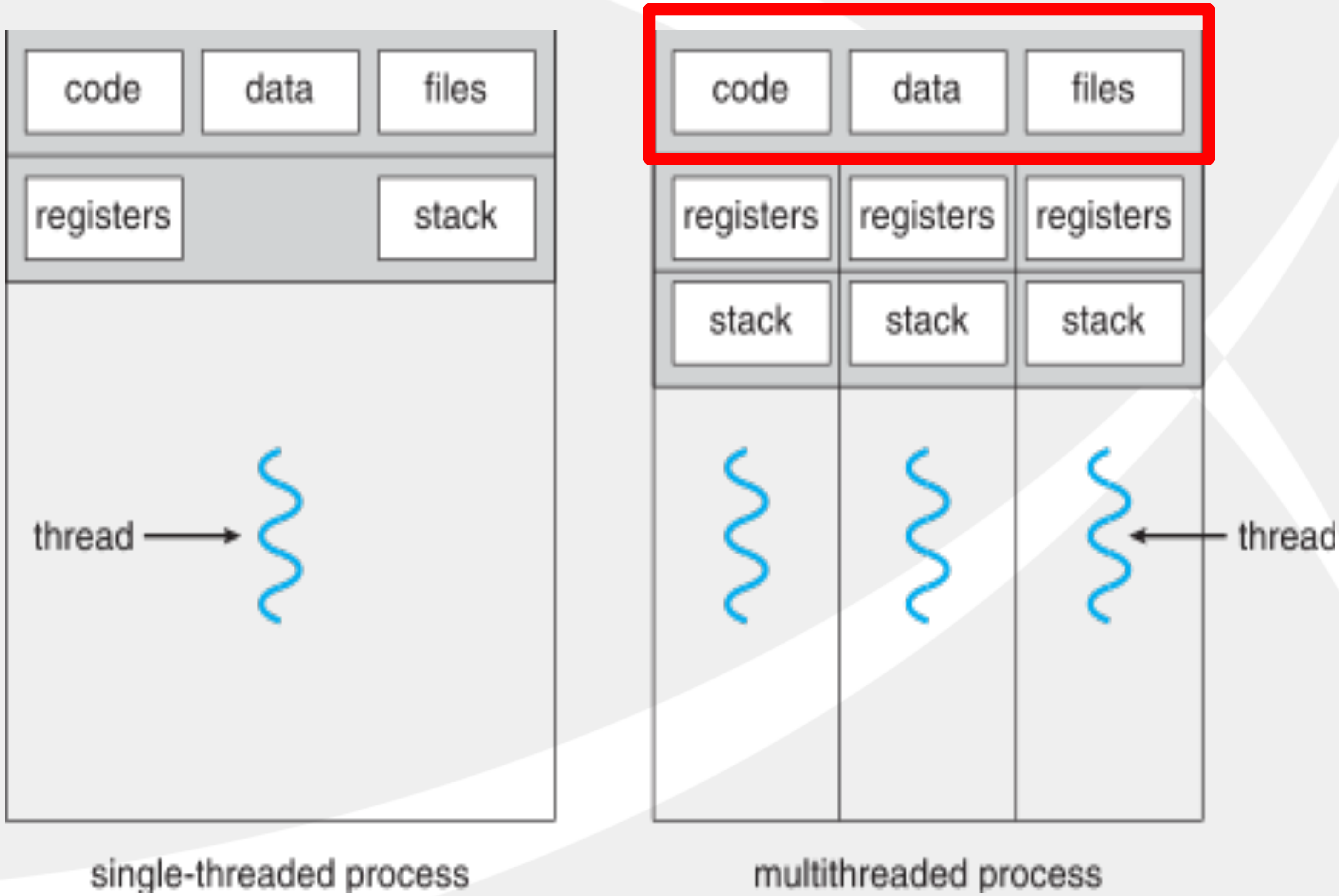
Why Not Use Processes?

- Processes are heavy-weight
- Threads are light-weight
- Threads are also sharing memory – easier programming compared to multiple processes
- Threads are less fault tolerant
- Threads can introduce lot of synchronization issues (e.g., race conditions) if not done correctly

Why Process Creation Heavy?

- Process creation:
  - Needs to setup a new address space, allocate resource
  - Kernel per-process data structures need to be allocated and initialized
- Why threads lightweight?
  - Threads live within processes
  - All threads share resources with other threads within the process (minus the stack)

Single and Multithreaded Processes

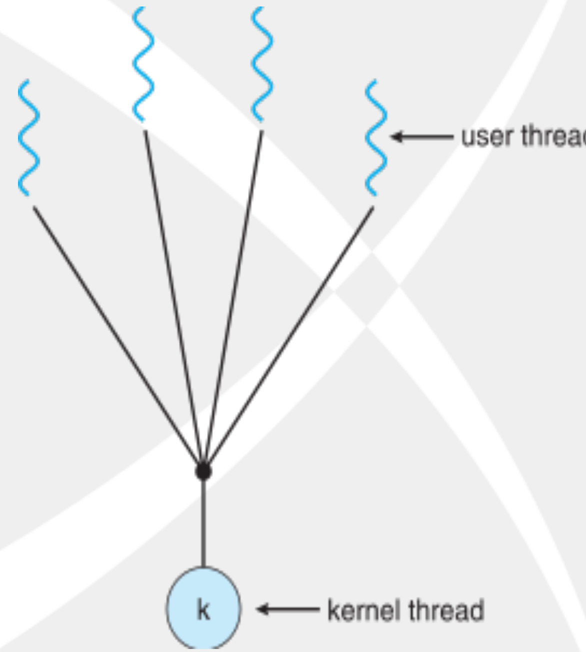


User vs. Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including: Windows, Solaris, Linux

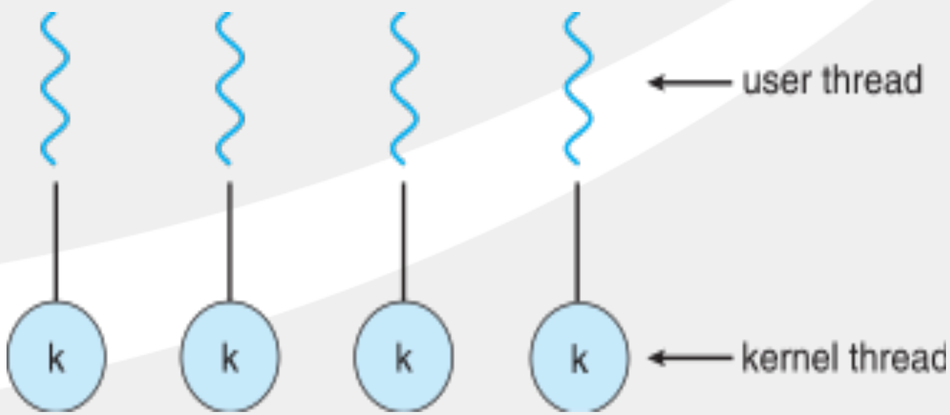
User Level Threads

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples: **Solaris Green Threads, GNU Portable Threads**



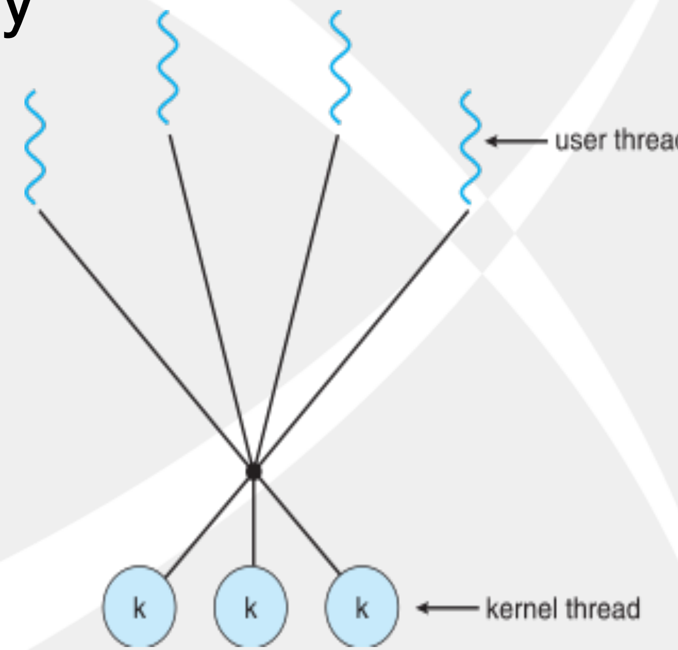
Kernel-Level Threads

- Each user-level thread maps to 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
- Examples: **Windows, Linux, Solaris 9 and later**



Hybrid Threads

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

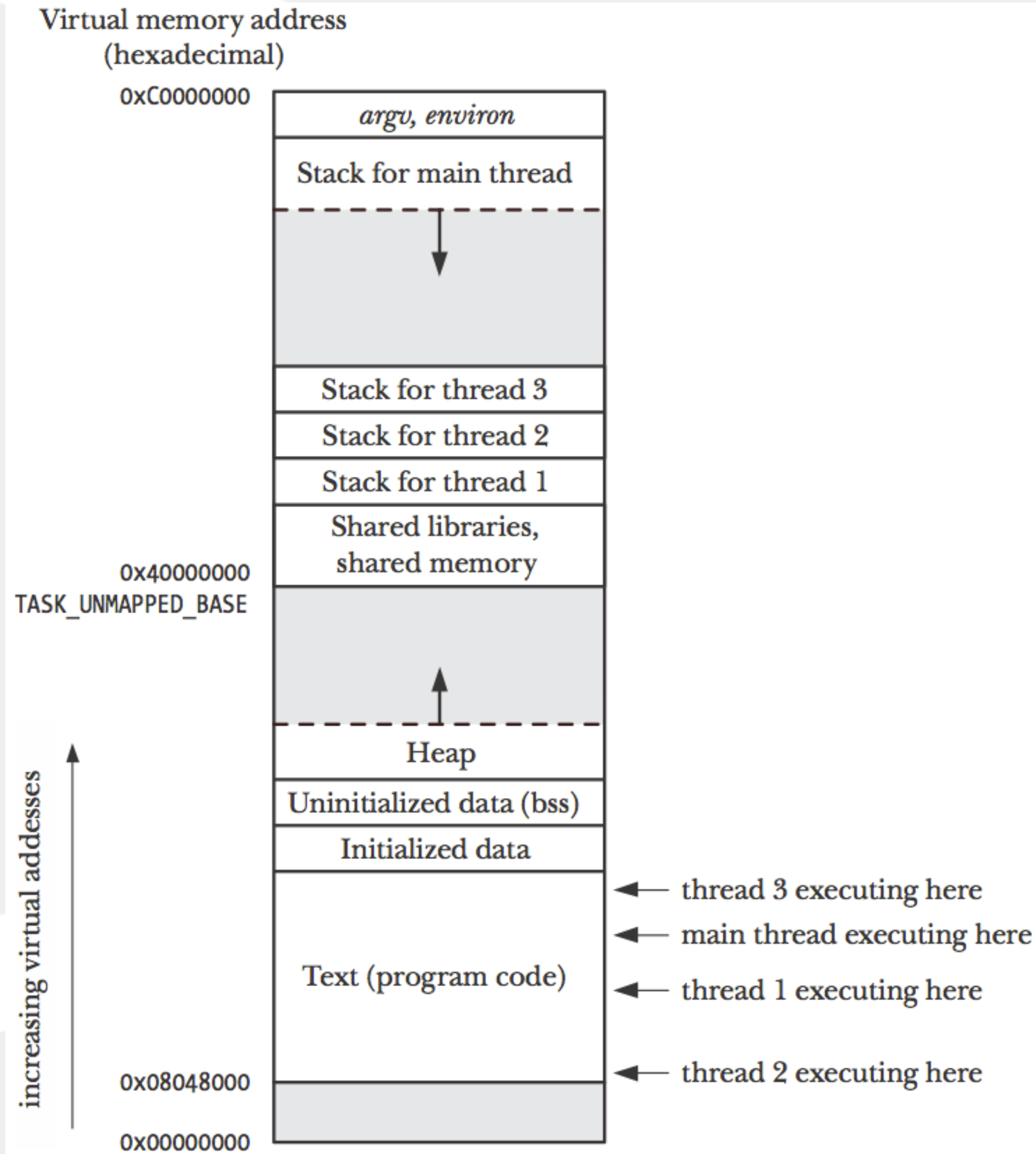


Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Threads in Linux

- Four threads executing in Linux
  - Kernel level threads
  - Threads have specific stacks – thread local storage



Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthread Creation

- Process has the *main* thread at the beginning

```
#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

- New thread continues with start() and main continues with the statement after

Pthread Termination

- A thread terminates for the following:
  - The start() function performs a return
  - Thread calls a pthread_exit() function
  - Thread is cancelled using pthread_cancel()
  - Any thread calls exit() or main thread returns

```
include <pthread.h>

void pthread_exit(void *retval);
```

Identities of Threads

- Each thread is uniquely identified by an ID
  - returned to the caller of pthread_create()
  - thread can obtain own ID 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 same

```
include <pthread.h>

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

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

Joining a Terminated Thread

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

```
include <pthread.h>

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

Returns 0 on success, or a positive error number

- If a created thread is not *detached*, we ***must join*** with it, otherwise "zombie" thread will be created

Pthread Example

```
#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);
}
```

Detaching a Thread

- Default – a thread is joinable – another thread is going to retrieve the return state
- If no thread is interested in joining we need to detach the thread

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Returns 0 on success, or a positive error number

thread

Thread Attributes

- Attributes can be used to set properties of threads – such as detached

```
pthread_t thr;
pthread_attr_t attr;
int s;

s = pthread_attr_init(&attr);                 /* Assigns default values */
if (s != 0)
    errExitEN(s, "pthread_attr_init");

s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
if (s != 0)
    errExitEN(s, "pthread_attr_setdetachstate");

s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
if (s != 0)
    errExitEN(s, "pthread_create");
```

Protecting Shared Variables

- Advantage of threads – can share via global variables
- Must ensure multiple threads are not modifying the variables at the same time
- Use a pthread_mutex variable

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Both return 0 on success, or a positive error number

Example Program

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

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *                    /* Loop 'arg' times increr
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;

    for (j = 0; j < loops; j++) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");
```

```
        loc = glob;
        loc++;
        glob = loc;

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops")

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

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

. . .

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

Thread Cancellation…

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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

- If thread has cancellation disabled, cancellation remains pending until thread enables it

Thread Cancellation...

- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals