CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

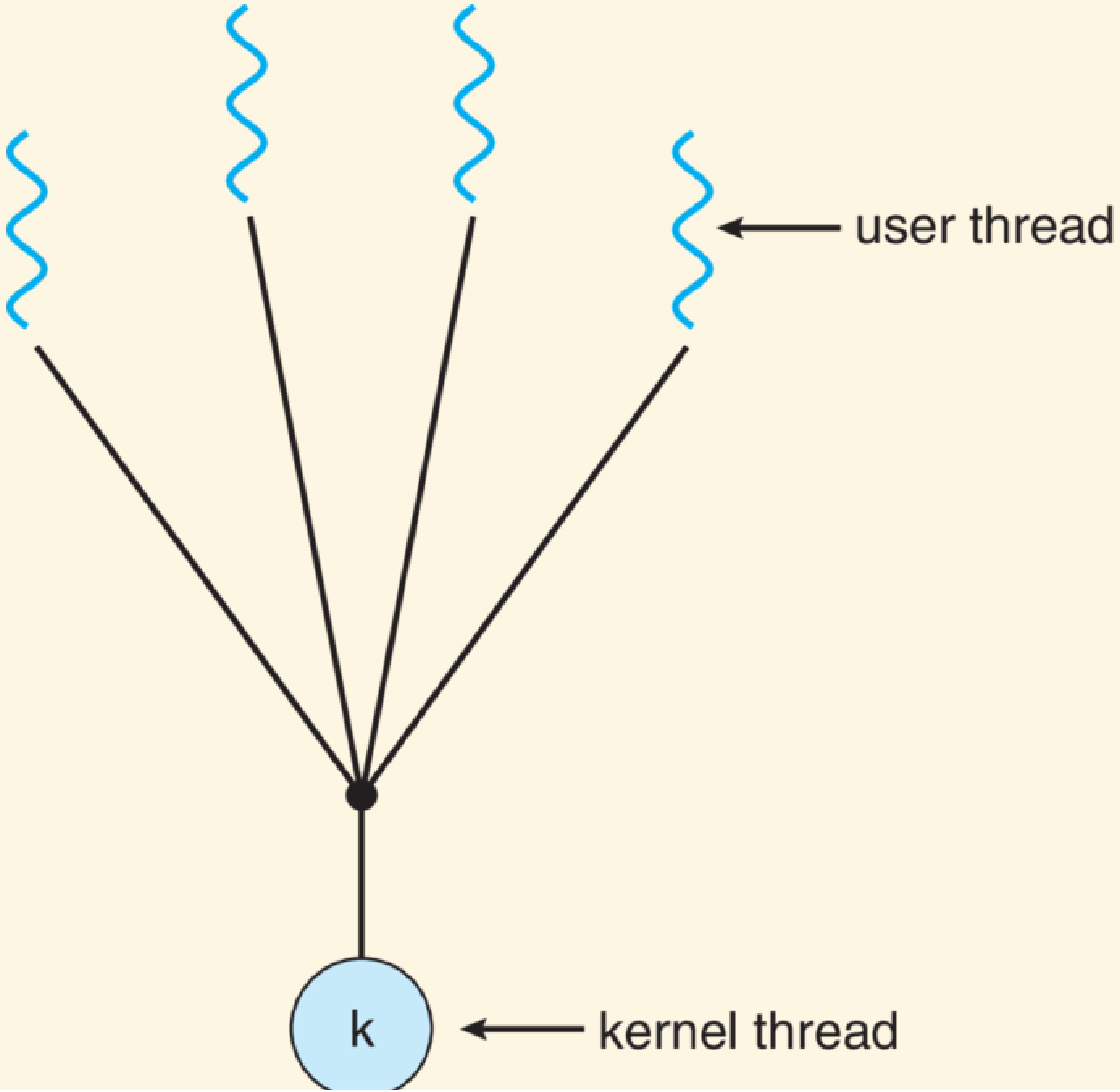
Thread Models

Threads can exist at two levels: user threads and kernel threads

User threads are implemented in user space

User-level library/runtime-environment responsible for creating and scheduling threads

Kernel does not know user-level threads exist -- can be all one thread to the kernel


user thread
k
kernel thread

User threads can be implemented without OS support

Context switching between user threads is very fast because it does not require a system call

Because all user threads in same process are seen as the same by the kernel, most benefits of threading are gone

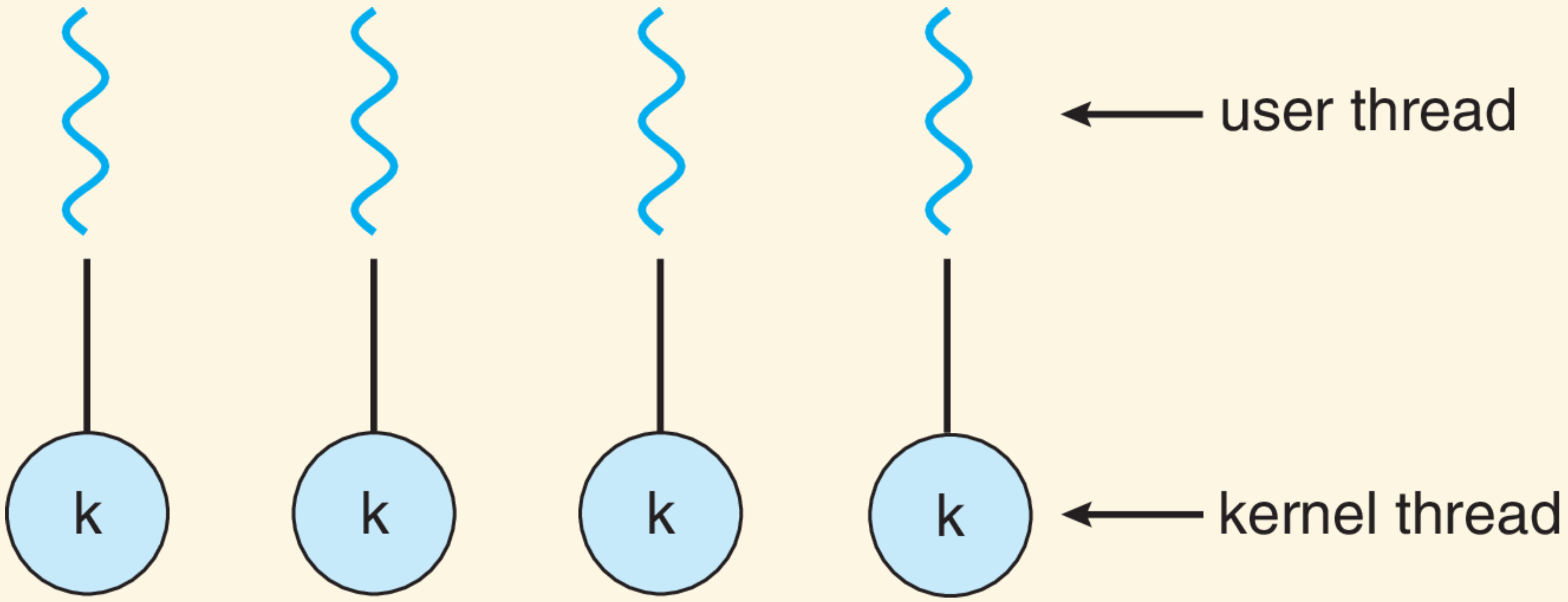
If any thread makes a system call, all threads are blocked

Threads cannot truly run in parallel because kernel will not schedule them to different cores

Book refers to this as "many-to-one" model: many user threads map to one kernel thread

If kernel supports more than one thread per process, each user thread can be tied to a single kernel thread

This is "one-to-one" model: one user thread maps to one kernel thread

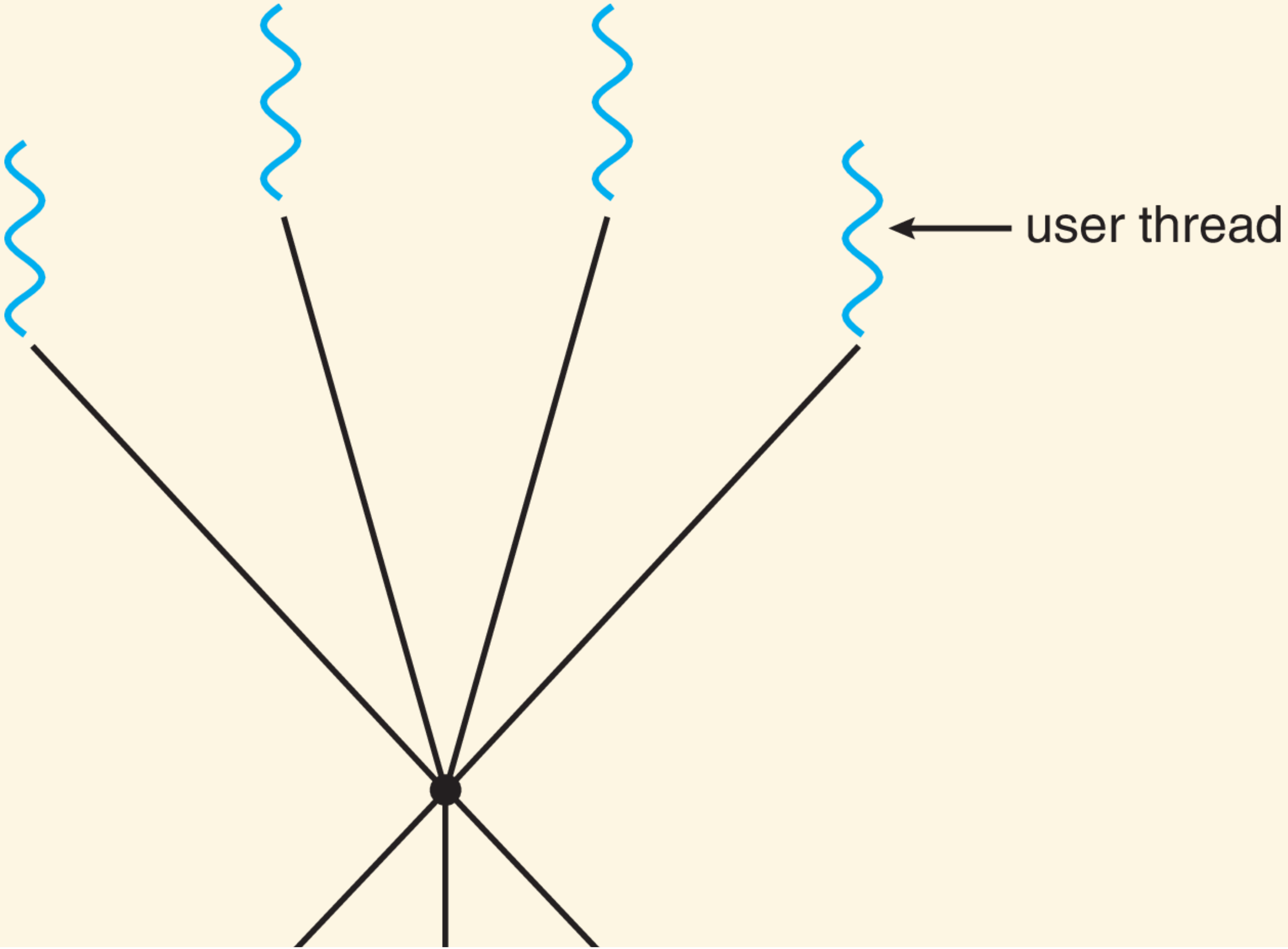

user thread
k
k
k
k
kernel thread

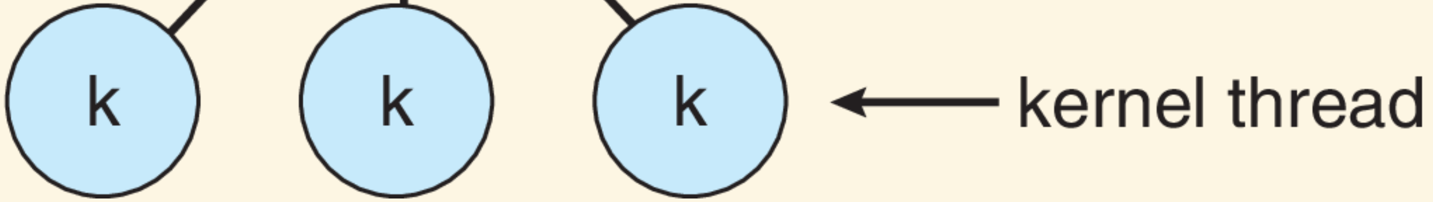
Because kernel knows about threads, true parallelism is possible

Blocking call by one thread need not block every thread in a process

However, thread management requires system calls, and scheduling is handled by OS

Both Linux and Windows support one-to-one model


user thread


k
k
k
kernel thread

Many-to-many model combines best features of other models

However, not implemented in most OSes

Users create and manage threads through calls to a *thread library*

Like with system calls and `libc`, user need not know underlying details of threading when using a library

Three main libraries in use are POSIX threads (Pthreads), Windows, and Java

Java threads run on JVM, so are generally implemented on one of the other two underneath

Let's look at a simple Pthreads program for summing the integers from 1 to N

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

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

int main(int argc, char *argv[])
{
```

Global variables (`sum` here) are shared by all threads

Threads, unlike forked processes, do not pick up execution from same spot as parent. Instead, they begin in a designated function (`runner` in this case)

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

if (argc != 2) {
 fprintf(stderr,"usage: a.out <integer value>\n");
 /*exit(1);*/
 return -1;
}

if (atoi(argv[1]) < 0) {
 fprintf(stderr,"Argument %d must be non-negative\n",atoi(argv[1]));
 /*exit(1);*/
 return -1;
}

/* get the default attributes */
pthread_attr_init(&attr);

/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);

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

Key routine is

```
pthread_create(&tid, &attr, runner, argv[1]);
```

Create a Pthread with given attributes, "return" thread ID of new Pthread into `tid`, and start new Pthread in function `runner` with argument `argv[1]`

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

 if (upper > 0) {
  for (i = 1; i <= upper; i++)
   sum += i;
 }

 pthread_exit(0);
}
```

Notice that `runner` doesn't return the sum -- it doesn't need to!

`sum` is a shared variable

Calling `pthread_exit` allows other thread to escape from `pthread_join` (similar to `wait` for processes)

Thread cannot return to main function. Calling `return 0` is the same as `pthread_exit(0)` in this context

Program follows a common pattern known as "fork-join" or *synchronous* threading

Worker thread is spawned and main thread waits until worker is done before proceeding

In *asynchronous* threading, worker threads are spawned and main thread goes back to its work without waiting for worker thread

Web server from earlier was example of asynchronous threading