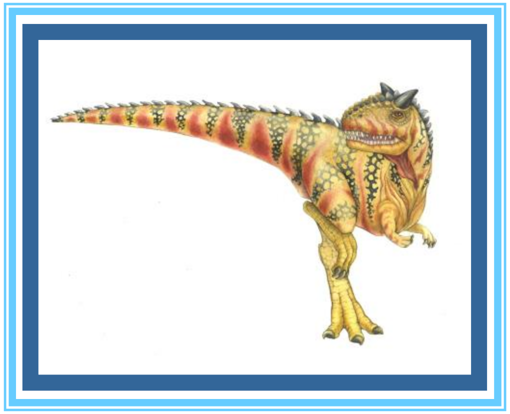
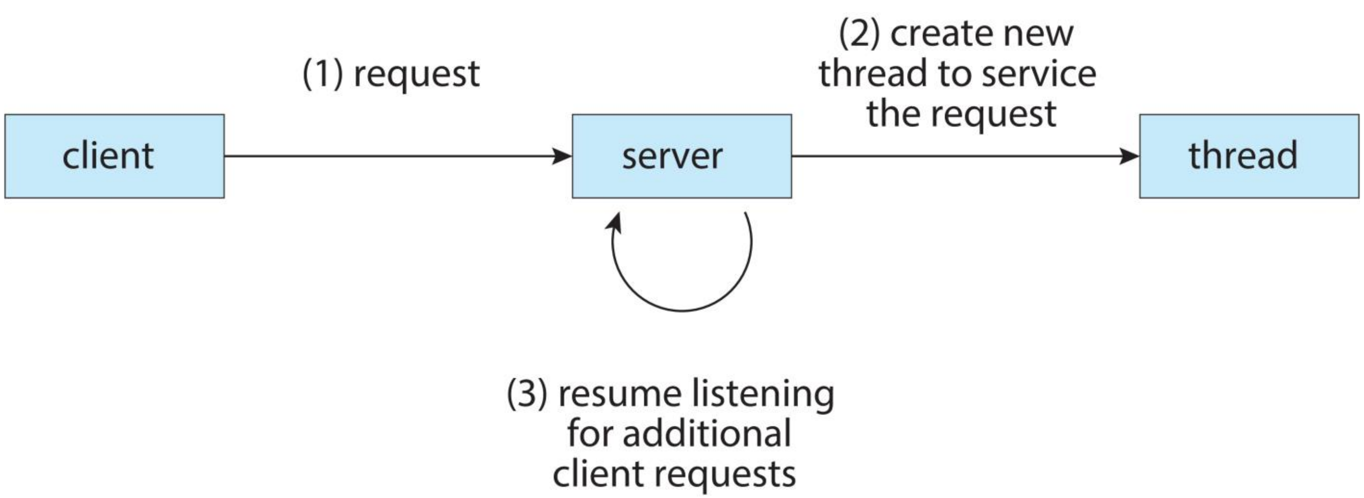
# Chapter 4: Threads & Concurrency



# Motivation

- Most modern applications are multithreaded
  - E.g., web browser, word processor, web server
- Multiple tasks within the application can be implemented by separate threads
  - E.g., update display, fetch data, spell checking, answer a network request



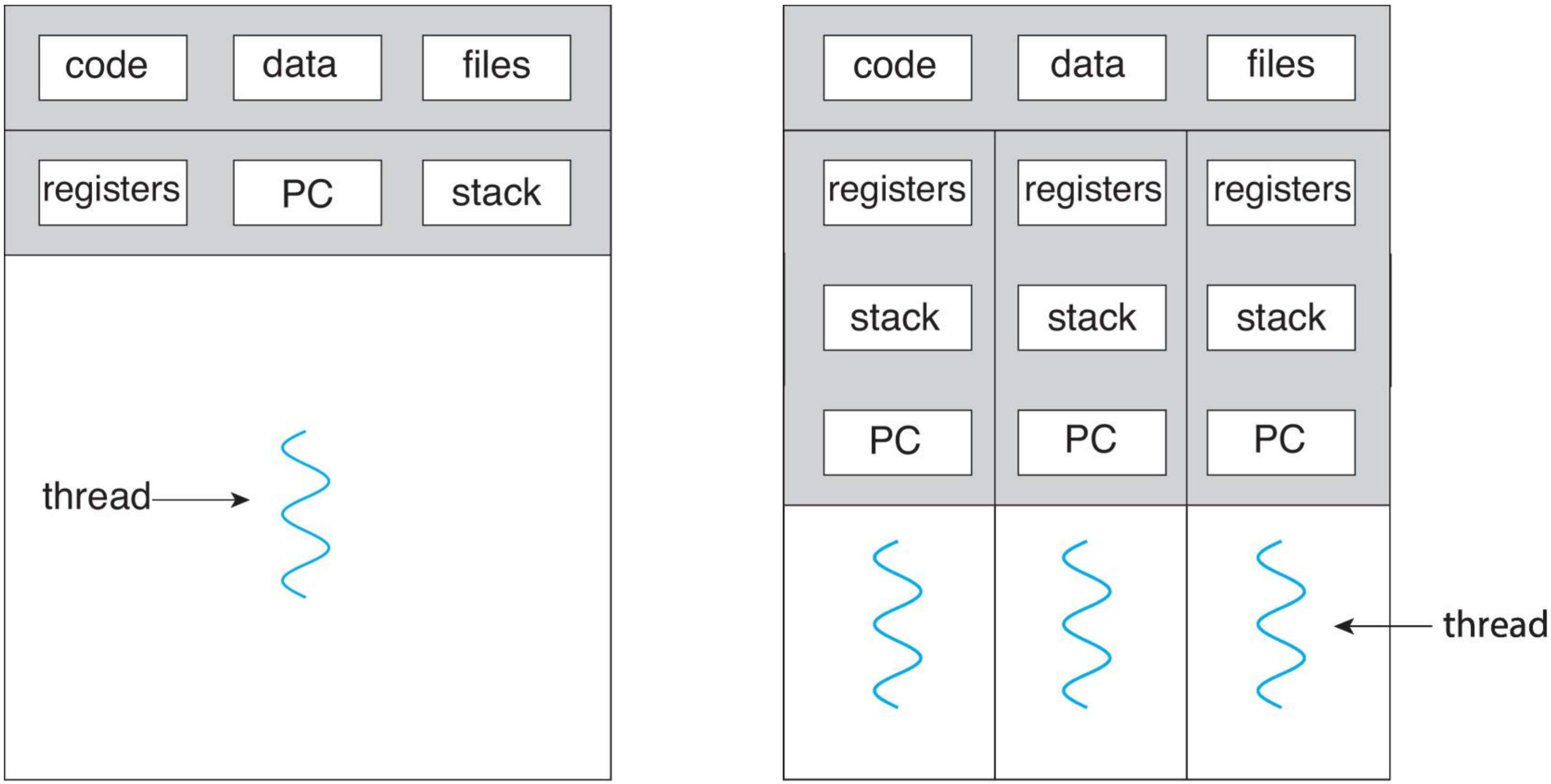

Multithreaded server architecture

# What is a Thread?

- A **thread** is a basic unit of CPU utilization
- A process can contain multiple threads of control
- A thread consists of
  - A thread ID
  - A program counter
  - A register set
  - A stack
- Threads belonging to the same process share
  - code section, data section, heap
  - open files

Linux command to display threads in a process
$ top -H -p <pid>
$ ps –T –p <pid>

# Single and Multithreaded Processes




single-threaded process

multithreaded process

# Thread Control Block (TCB)

- OS maintains a TCB for each thread
- TCB contains
  - Thread ID
  - Stack pointer
  - Program counter
  - Register values
  - Pointer to PCB of the process that the thread belongs to

# Benefits of Multithreading

- **Responsiveness –** may allow continued execution of a program if part of it is blocked or is performing a lengthy operation; especially important for user interfaces
- **Resource Sharing –** threads share resources of process, sharing data easier than shared memory or message passing
- **Economy –** thread creation is cheaper (i.e., consumes less time & memory) than process creation, thread context-switching has lower overhead than process-context switching
  - Thread context-switching involves saving/loading registers and program counter to/from TCB; no need to save/load memory management information
- **Scalability –** multithreaded process can take advantage of multicore architectures
  - Threads may be running in parallel on different processing cores

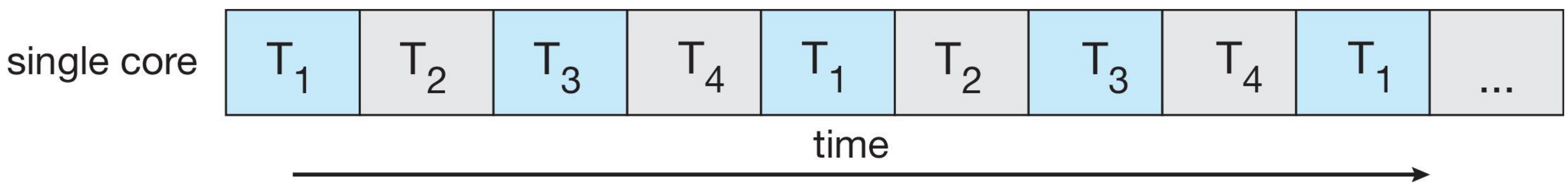
# Multicore Programming

- Recent trend in computer design is to place multiple computing cores on a single processing chip – **multicore** systems
- Multithreaded programing provides more efficient use of multiple computing cores and improved concurrency
- ***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
  - It is possible to have concurrency without parallelism

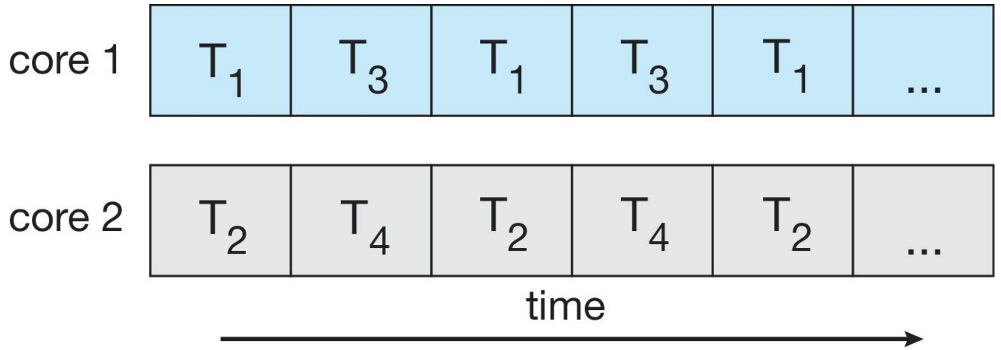
# Concurrency vs. Parallelism

Consider an application with 4 threads:

- **Concurrent execution on single-core system:**




- **Parallelism on a multi-core system:**



# Amdahl's Law

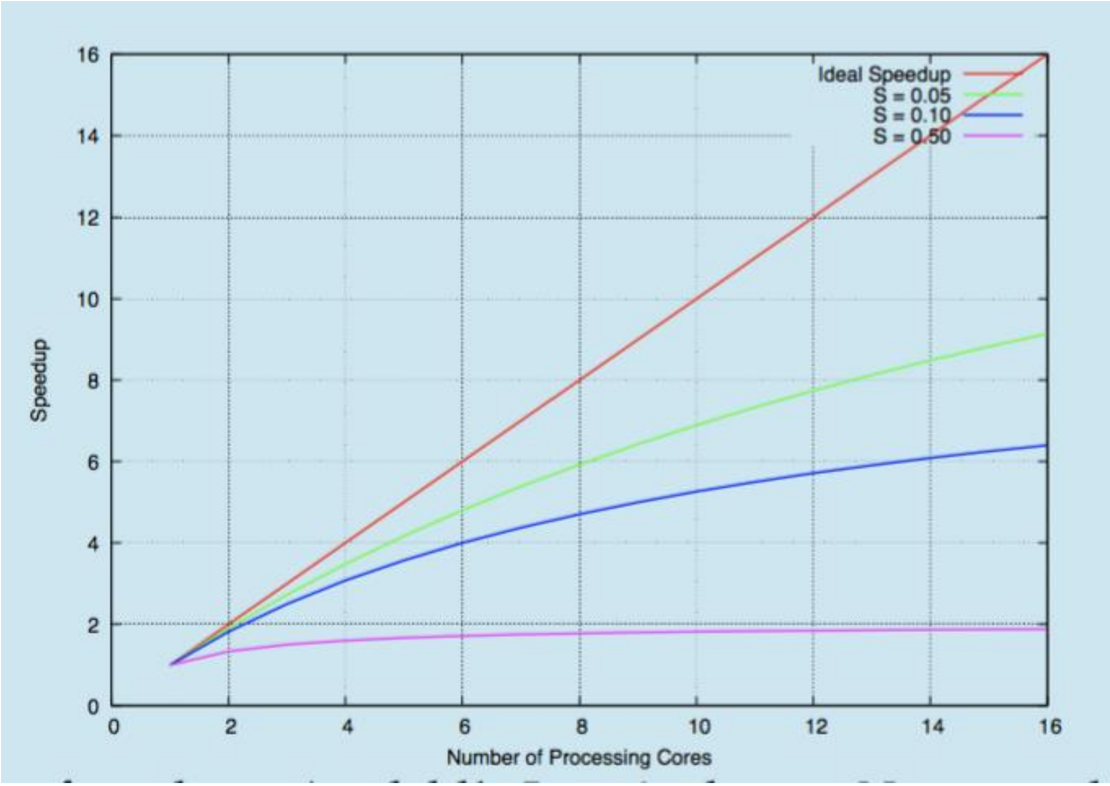
- Identifies performance gains from adding additional computing cores to an application that has both serial and parallel components
- $S$ is serial portion of application, $N$ processing cores

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

- E.g., if application is 75% parallel & 25% serial, moving from 1 to 3 cores results in speedup of 2 times
- As $N$ approaches infinity, speedup approaches $1/S$

**Serial portion of an application has disproportionate effect on performance gained by adding additional cores**

# Amdahl's Law



Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
0
2
4
6
8
10
12
14
16
Number of Processing Cores

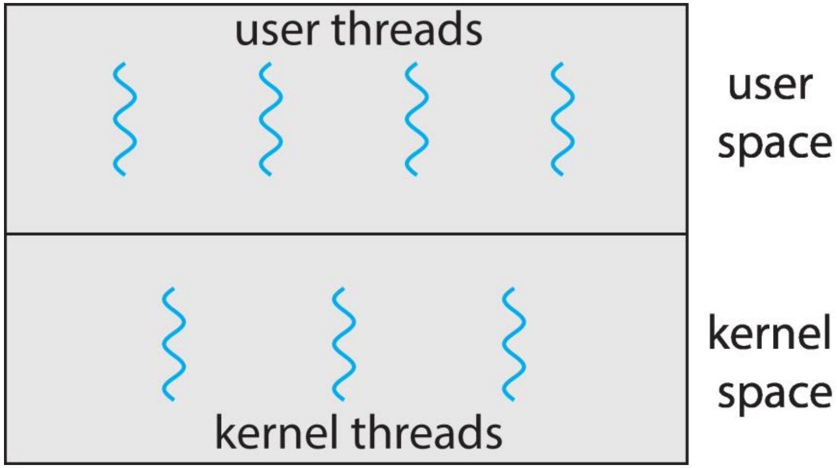
# User Threads and Kernel Threads

- **User threads** – supported at the user level, management done by user-level threads library
- **Kernel threads** – supported by the kernel
- Virtually all comptemporary operating systems support kernel threads:
  - Windows
  - Linux
  - macOS
  - iOS
  - Android

Linux command to see Kernel threads
$ ps -ef

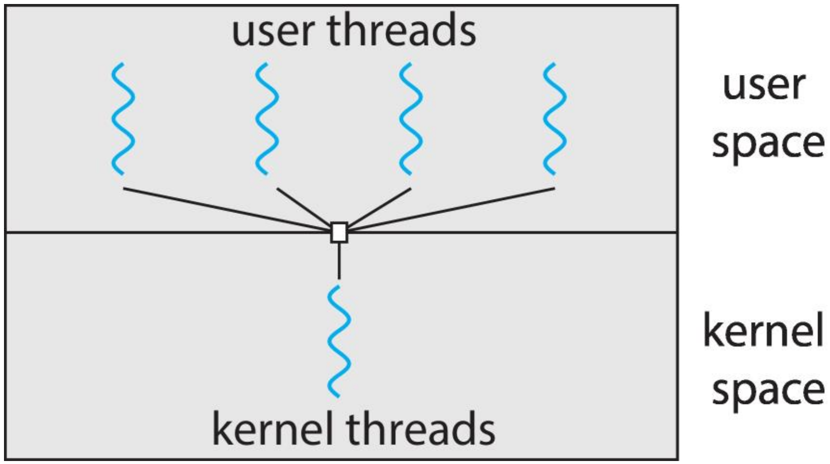
# Multithreading Models

- Relationship between user threads and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many



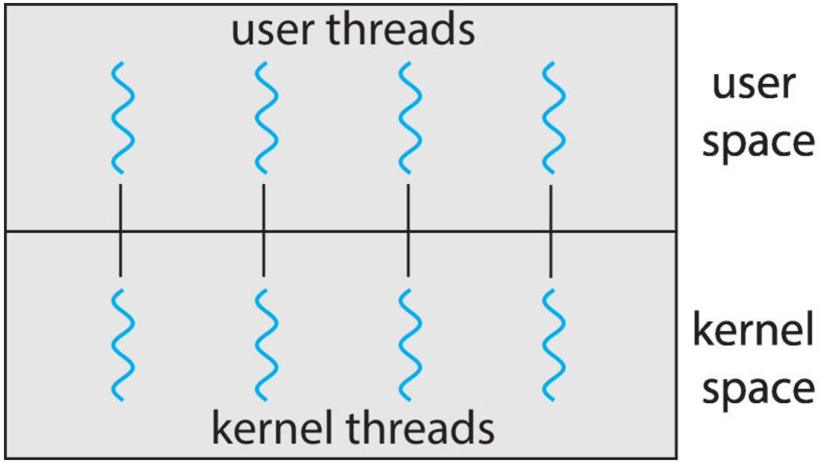
# Many-to-One

- Many user threads mapped to one kernel thread
- Thread management done by thread library in user space → efficient
- Entire process will block if a thread makes a blocking system call
- Multiple threads cannot run in parallel on multicore system
- Few systems currently use this model



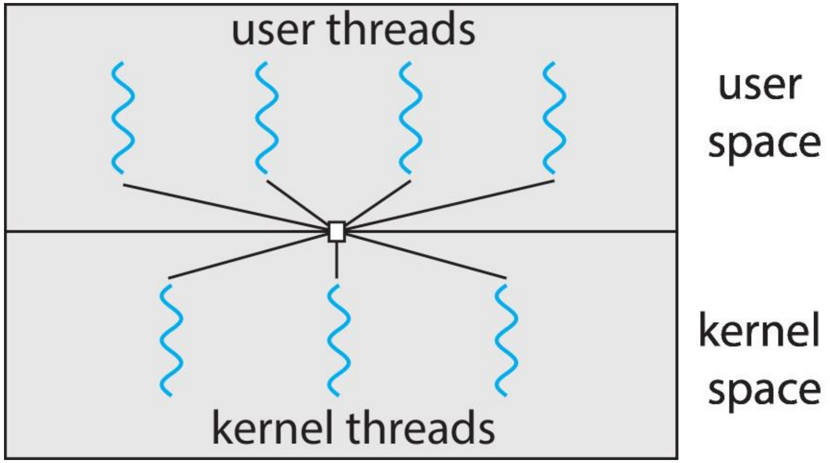
# One-to-One

- Each user thread is mapped to a kernel thread
- Greater concurrency than many-to-one: another thread can run when a thread makes a blocking system call
- Multiple threads can run in parallel on multicore system
- Creating a user thread requires creating a kernel thread → expensive
  - Number of threads per process sometimes restricted due to overhead
- Most operating systems now use one-to-one model

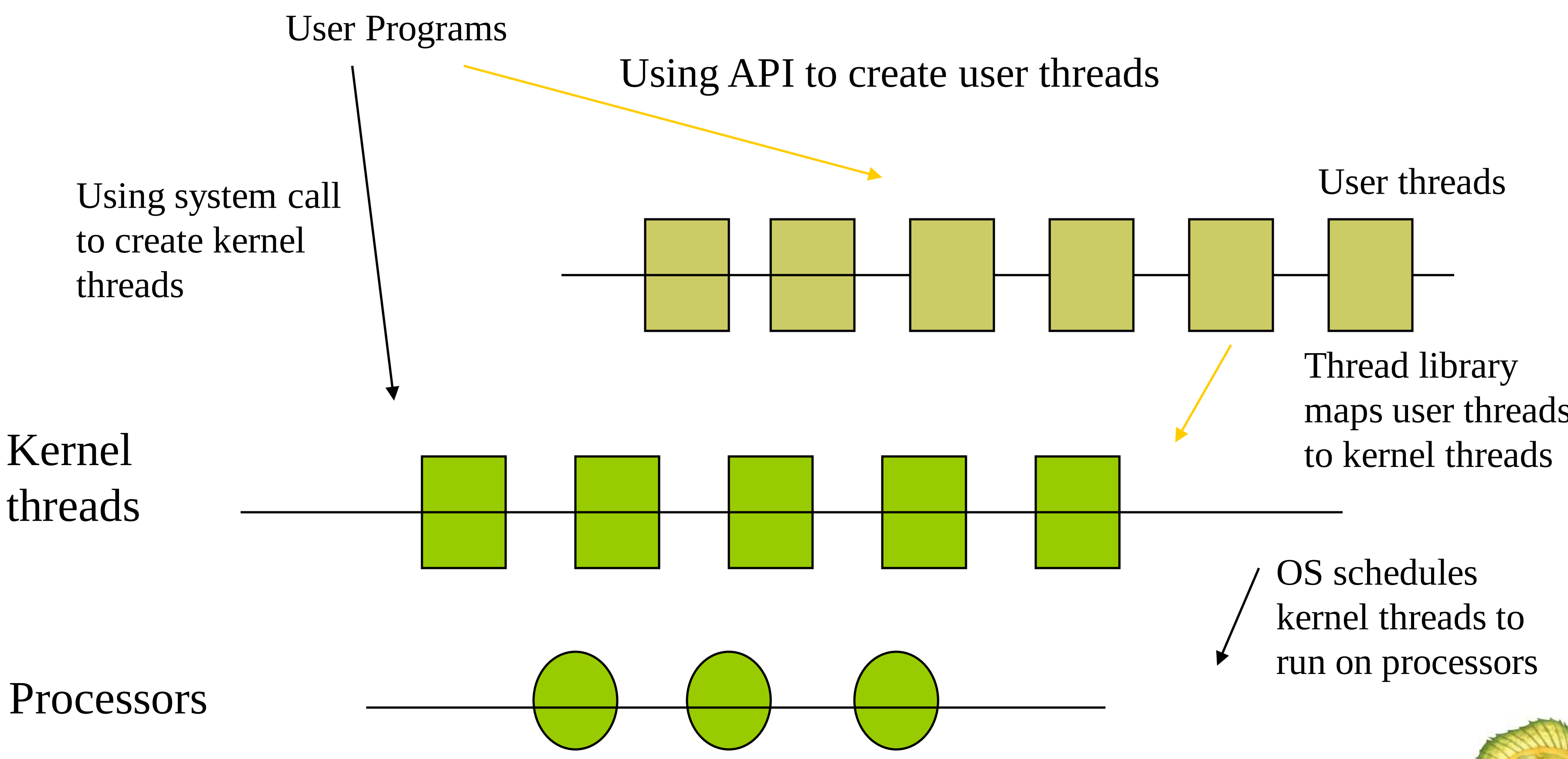


# Many-to-Many Model

- Many user-level threads are multiplexed to a smaller or equal number of kernel threads
- Application developer can create as many user threads as necessary, corresponding kernel threads can run in parallel on a multicore system
- When a user thread performs a blocking system call, the kernel can schedule another thread for execution
- Not very common



# Threads in a Computer System



# Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing thread library
  - Library entirely in user space: invoking a library function results in a local function call in user space
  - Kernel-level library supported by the OS: invoking a library function results in a system call to kernel
- Three main thread libraries:
  - POSIX pthreads: can be user level or kernel level
  - Windows threads: kernel level
  - Java threads: implemented using a thread library available on the host system

# Pthreads

- pthreads is a Portable Operating System Interface (POSIX) standard API for thread creation and synchronization
  - API specifies behavior of the thread library, implementation is up to development of the library
  - Available on many UNIX-like operating systems (Solaris, Linux, macOS)

- Using pthreads on pyrite:

  #include <pthread.h>

  Compile and link with the pthread library:  gcc pthreads.c -lpthread

# pthread_create (1)

int  pthread_create(pthread_t *thread, pthread_attr_t *attr,
 void * (* start_routine) (void *), void *arg) – create a new thread

- thread points to a buffer that stores the ID of the new thread
- attr points to a structure containing the attributes of the new thread
  - If attr is NULL, the thread is created with default attributes
- The new thread starts execution by invoking start_routine()
- arg is a pointer to the argument of start_routine()
  - If multiple arguments are needed, arg points to a data structure that contains all arguments
- Returns 0 on success, returns an error number on error

# pthread_create (2)

- The new thread executes concurrently with the parent thread
- The new thread runs until one of the following happens
  - It returns from start_routine
  - It calls **pthread_exit()**
  - Any of the threads in the process calls **exit()** or the main thread performs a return from **main()**. This causes the termination of all threads in the process.

# pthread_create() Example

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

void *my_thread (void *arg)
{
      char *msg = (char *) arg;
      printf("Thread says \%s\n", msg);
}

int main (int argc, char *argv[])
{    pthread_t t;
     char msg[20] = "Hello World";
     pthread_create(&t, NULL, my_thread, msg);
     sleep(3); //what happens if this statement is removed?
     return 0;
}
```

https://onlinegdb.com/HyTorJxED

# pthread_exit()

#include <pthread.h>

void pthread_exit(void *retval) – terminate calling thread

- The function returns a value via <u>retval</u> that is available to another thread in the same process that calls pthread_join()
- The function does not return to the caller

# pthread_join()

#include <pthread.h>

int pthread_join(pthread_t th, void **retval) - wait for a thread to terminate

- th: the thread to wait for
- If retval is not NULL, then **pthread_join**() copies the exit status of the target thread into the location pointed to by retval
- Returns 0 on success, returns an error number on error
- When a thread terminates, its TCB is not deallocated until another thread performs **pthread_join()** on it

# Pthread_join() Example

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

int N;

void * thread(void *x)
{
  int *id = (int *)x;
  while (N != *id);
  printf("Thread %d \n", *id);
  N--;
  pthread_exit(NULL);
}
```

```
int main()
{
  N = 0;
  int id1=1;
  int id2=2;
  int id3=3;

  pthread_t t1, t2, t3;
  printf("Parent creating threads\n");
  pthread_create(&t1, NULL, thread, &id1);
  pthread_create(&t2, NULL, thread, &id2);
  pthread_create(&t3, NULL, thread, &id3);
  printf("Threads created\n");
  N = 3;
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_join(t3, NULL);
  printf("Threads are done\n");
  return 0;
}
```

[https://onlinegdb.com/r1vdKyeVw](https://onlinegdb.com/r1vdKyeVw)