PES UNIVERSITY ONLINE

OPERATING SYSTEM

Threads & Concurrency

Dr Rahul Nagpal
Computer Science

OPERATING SYSTEM

Threads & Concurrency

Dr. Rahul Nagpal
Computer Science

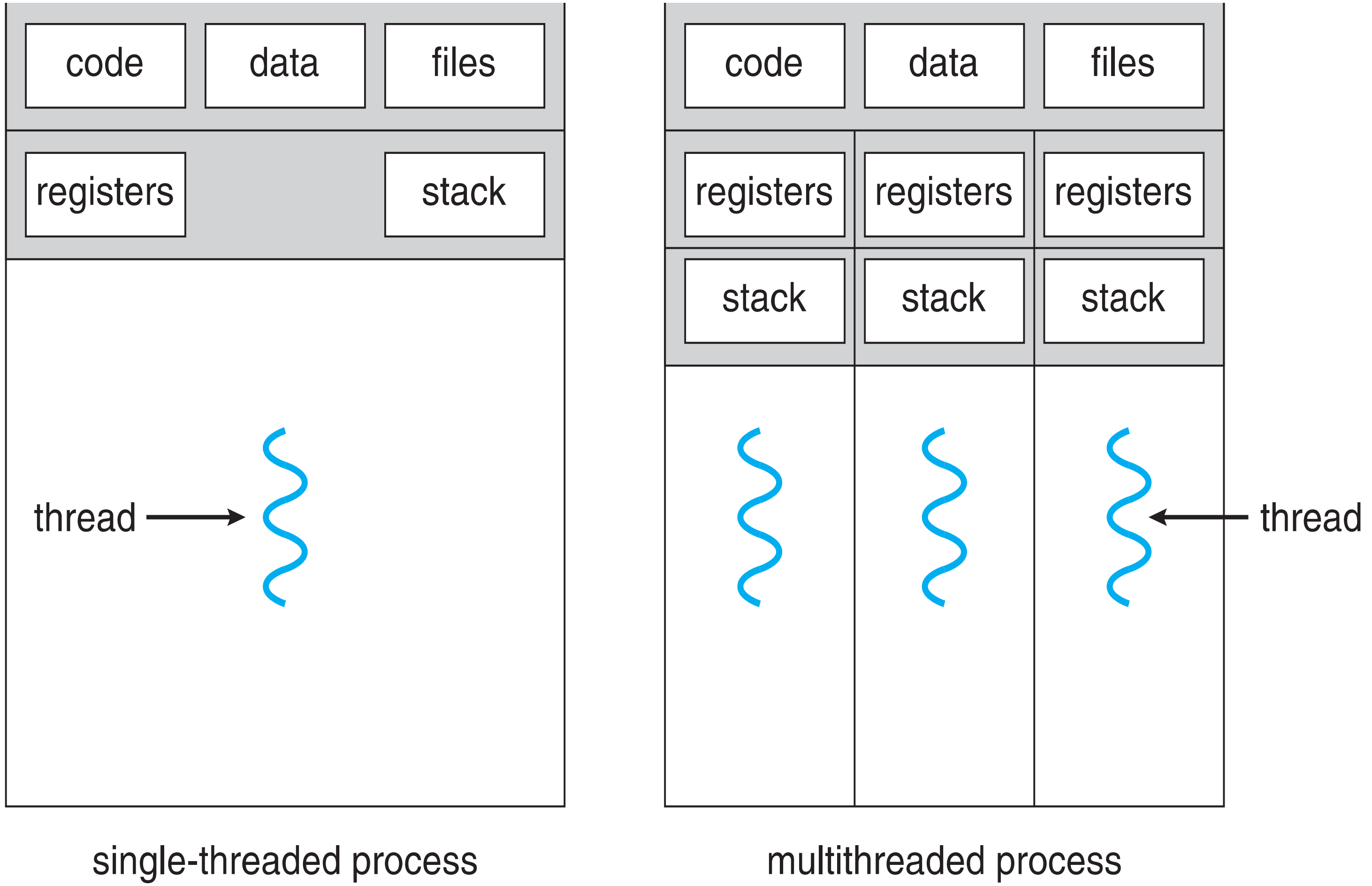
Outline

- Threads and Concurrency
- Multicore Programming
- Multithreading Models
- Thread Scheduling

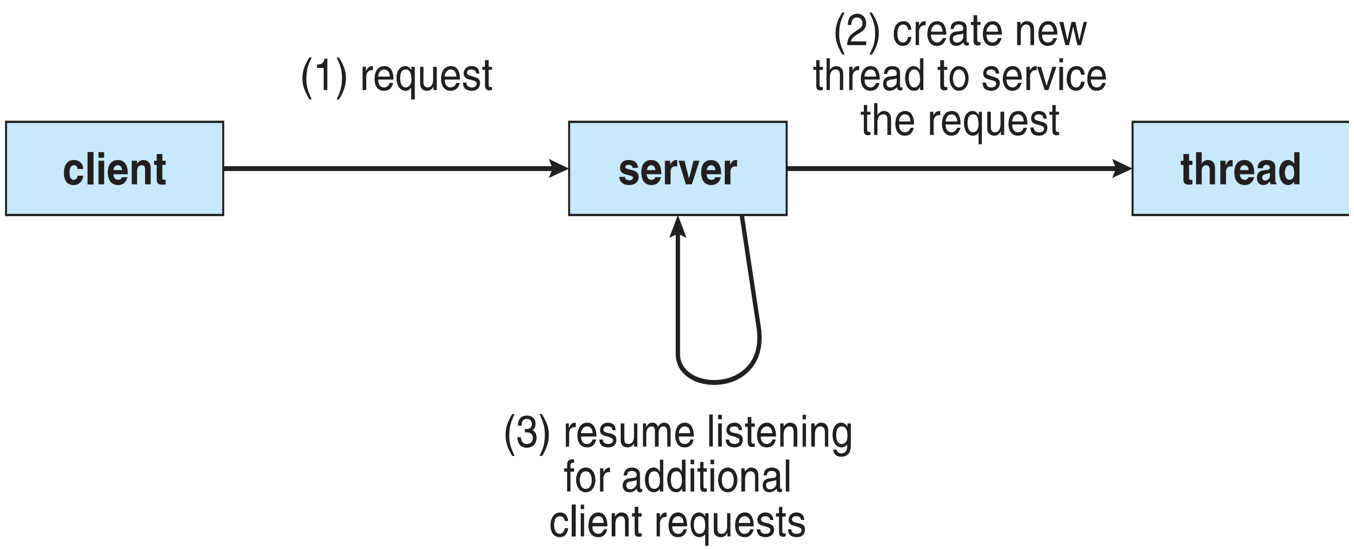
Motivation

- Process creation is heavy-weight while thread creation is light-weight
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks in application can be implemented by threads
  - Update display
  - Fetch data
  - Spell checking
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

OPERATING SYSTEMS
Single and Multithreaded Processes



Multithreaded Server Architecture



Benefits

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 lower overhead than context switching

Scalability – process can take advantage of multiprocessor architectures

Multicore or **multiprocessor** systems putting pressure on programmers, challenges include:

- **Dividing activities**
- **Balance**
- **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 processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

Types of parallelism

- **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each

- **Task parallelism** – distributing threads across cores, each thread performing unique operation

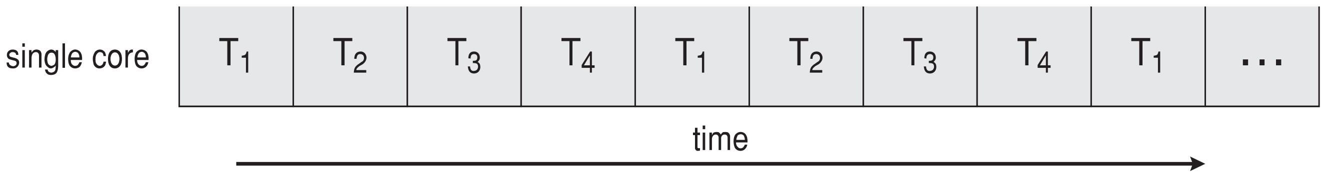
As # of threads grows, so does architectural support for threading

- CPUs have cores as well as ***hardware threads***

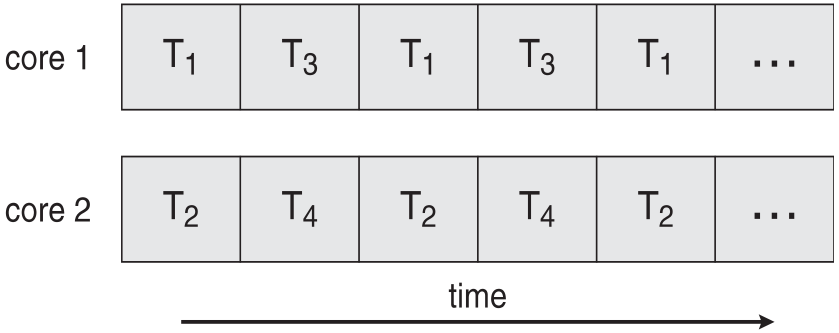
- Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Concurrency vs. Parallelism

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



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



User Threads and Kernel Threads

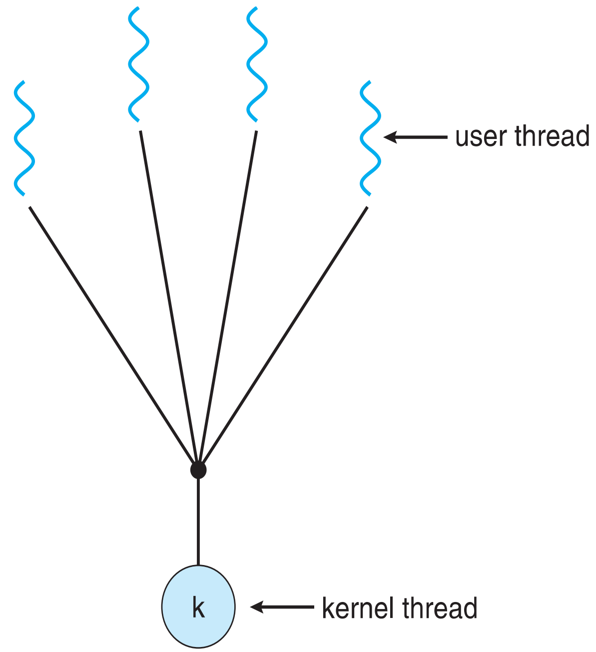
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

User Threads and Kernel Threads

- Many-to-One
- One-to-One
- Many-to-Many

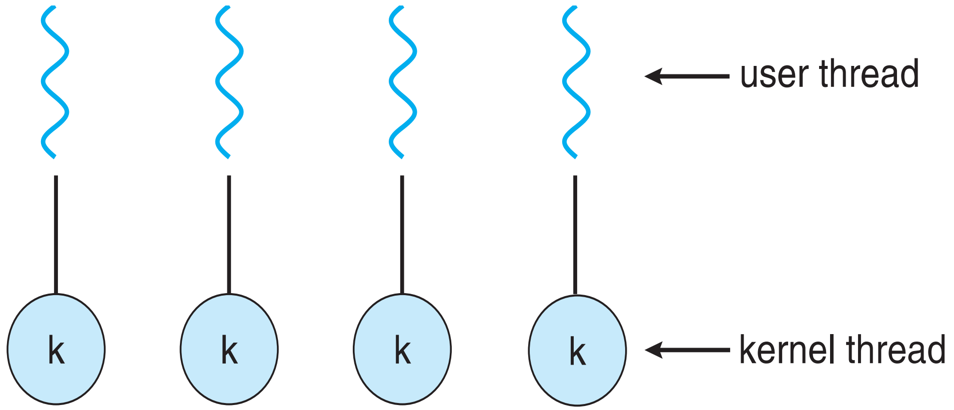
Many-to-One

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



One-to-One

- 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



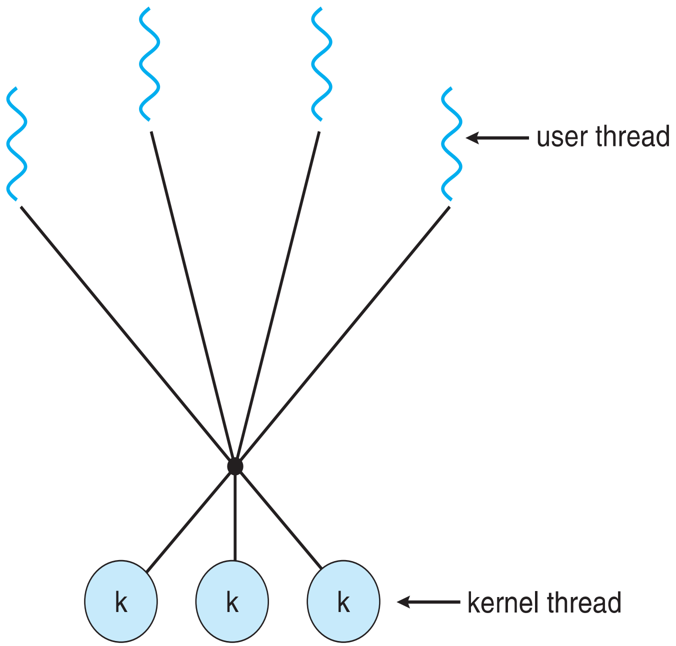
Many-to-Many

Allows many user level threads to be mapped to many kernel threads

Allows the operating system to create a sufficient number of kernel threads

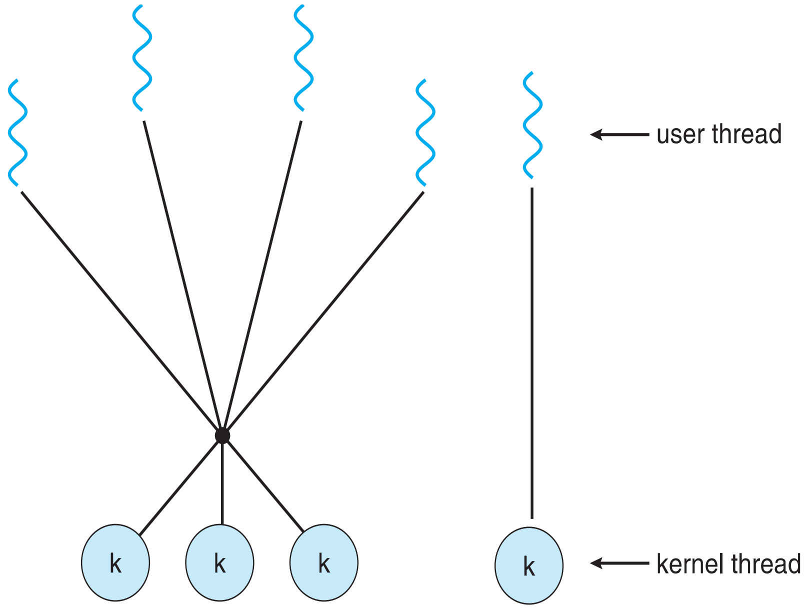
Solaris prior to version 9

Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

OPERATING SYSTEMS

Pthreads Example

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

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  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 begin control 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);
}
```

Pthread Scheduling API

```
#include <pthread.h>

#include <stdio.h>

#define NUM_THREADS 5

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

   int i, scope;
   pthread_t tid[NUM THREADS];

   pthread_attr_t attr;

   /* get the default attributes */

   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)

      fprintf(stderr, "Unable to get scheduling
scope\n");

   else {

      if (scope == PTHREAD_SCOPE_PROCESS)

         printf("PTHREAD_SCOPE_PROCESS");

      else if (scope == PTHREAD_SCOPE_SYSTEM)

         printf("PTHREAD_SCOPE_SYSTEM");

      else
         fprintf(stderr, "Illegal scope value.\n");

   }
```

```
/* set the scheduling algorithm
to PCS or SCS */

   pthread_attr_setscope(&attr,
PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS;
i++)

pthread_create(&tid[i],&attr,run
ner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS;
i++)

      pthread_join(tid[i],
NULL);

}

/* Each thread will begin
control in this function */

void *runner(void *param)
{

   /* do some work ... */

   pthread_exit(0);

}
```

PES UNIVERSITY ONLINE

THANK YOU

Dr Rahul Nagpal

Computer Science

rahulnagpal@pes.edu