THREADS

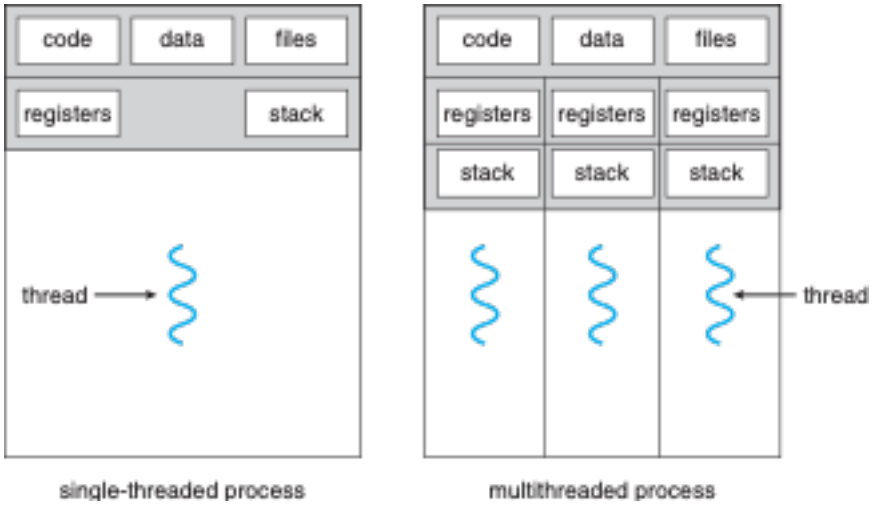
Motivation

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

Concept of Thread

- A thread is a basic unit of CPU utilization.
  - It comprises a thread ID, a program counter (PC), a register set, and a stack.
  - It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
  - A traditional process has a single thread of control.
  - If a process has multiple threads of control, it can perform more than one task at a time.

Single and Multithreaded Processes



Relationship Between Threads and Processes

Characteristics	Threads	Processes
Definition	A thread is a lightweight segment that is a part of a process.	A process is any program or software that is executing.
Creation Time	Threads usually take very little time to create since they are lightweight.	A process requires more time to create because the process is heavier than a thread.
Termination Time	A thread requires little time to terminate because of its simple nature.	A process requires more time to terminate because of its complex structure.

Relationship Between Threads and Processes

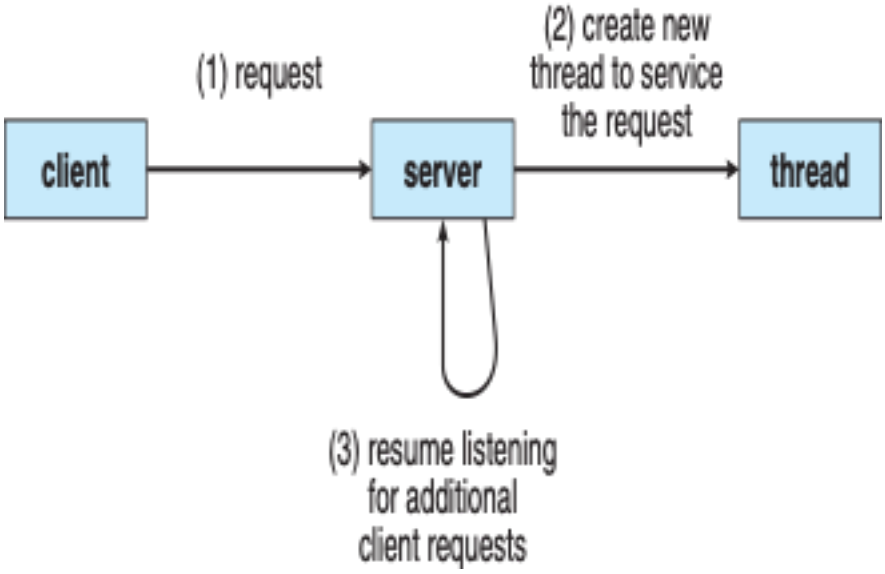
Resource Usage	A thread needs a minimal number of resources to do its task.	A process uses more resources than threads.
Memory Sharing	Threads share memory with other threads based on the task to perform.	All the processes created in an operating system are isolated. They don't communicate with any other processes.
Communication	Threads can communicate with other threads within the same process more effectively than a process.	Processes are less efficient than threads.
Context Switching	A thread requires less time for context switching in an OS. It is less expensive.	A process requires more time for context switching because of its heaviness. It is more expensive in processes.
Management	Threads do not depend on any OS system calls.	A process depends on system calls.

Examples of Multi-threaded applications

- An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
- A web browser might have one thread display images or text while another retrieves network data.
- A word processor may have a thread for displaying graphics, another for responding to keystrokes from the user, and a third for performing spelling and grammar checking in the background.
- A web server accepts client requests for web pages, images, sound, etc. If the web server runs as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

Multithreaded Server Architecture

- One solution is to have the server run as a single process that accepts requests
- When the server receives a request, it creates a separate process to service that request.
- But the process creation is time-consuming and resource intensive, therefore, a multithreaded server can create a thread to service a client.



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 Programming

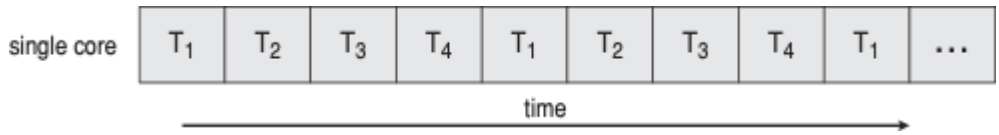
- Multicore or multiprocessor systems have many challenges
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
  - Parallelism implies a system can perform more than one task simultaneously
  - Concurrency supports more than one task making progress
  - Single processor/core, scheduler providing concurrency

Multicore Programming (Cont.)

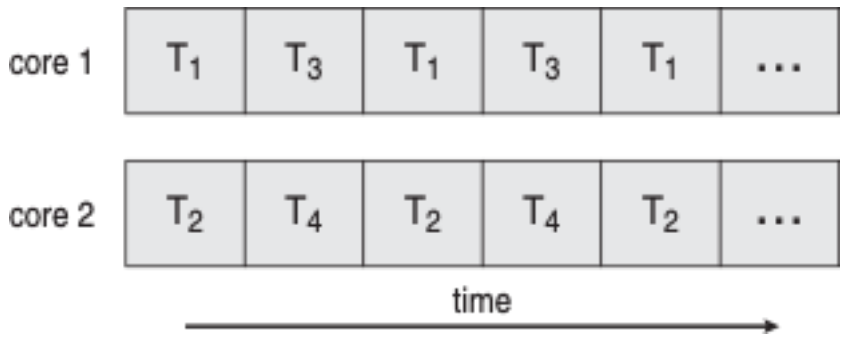
- Types of parallelism
  - Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs 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:



Amdahl's Law

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

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components:
  - S is a serial portion
  - N processing cores
- That is, if the application is 75% parallel / 25% serial, moving from 1 to 2 cores results in a speedup of 1.6 times.
- As N approaches infinity, speedup approaches $\frac{1}{S}$

Amdahl's Law

- A serial portion of an application disproportionately affects performance gained by adding additional cores.

$$Speedup_{symmetric}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f \cdot r}{perf(r) \cdot n}}$$

- But does the law take into account contemporary multicore systems?
- Here (f) is the parallelizable fraction, (n) is total chip resources, and ® is the resource devoted to increasing the performance of each core.

User Threads and Kernel Threads

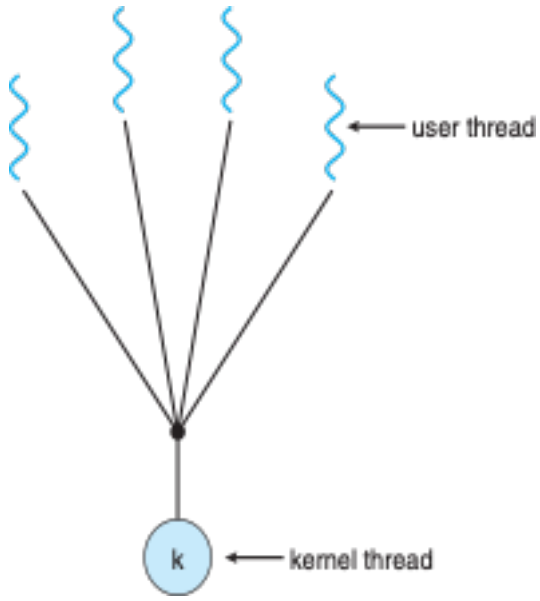
- User threads - management is done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows threads
  - Java threads
- Kernel threads - Supported by the Kernel

Multithreading Models

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

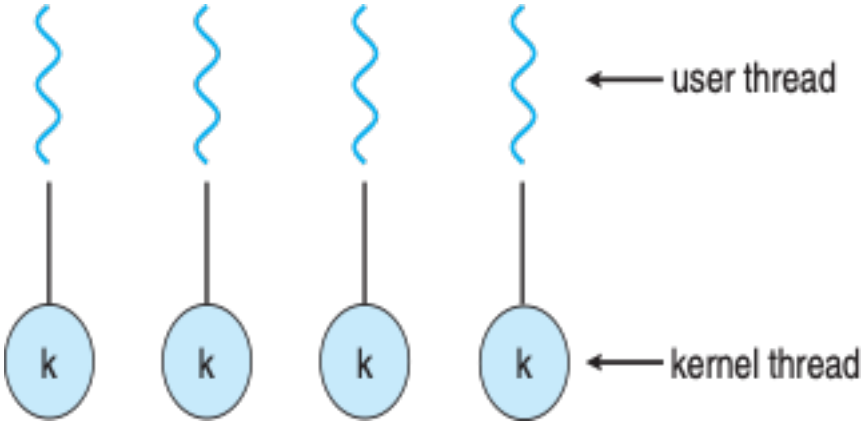
Many-to-One

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



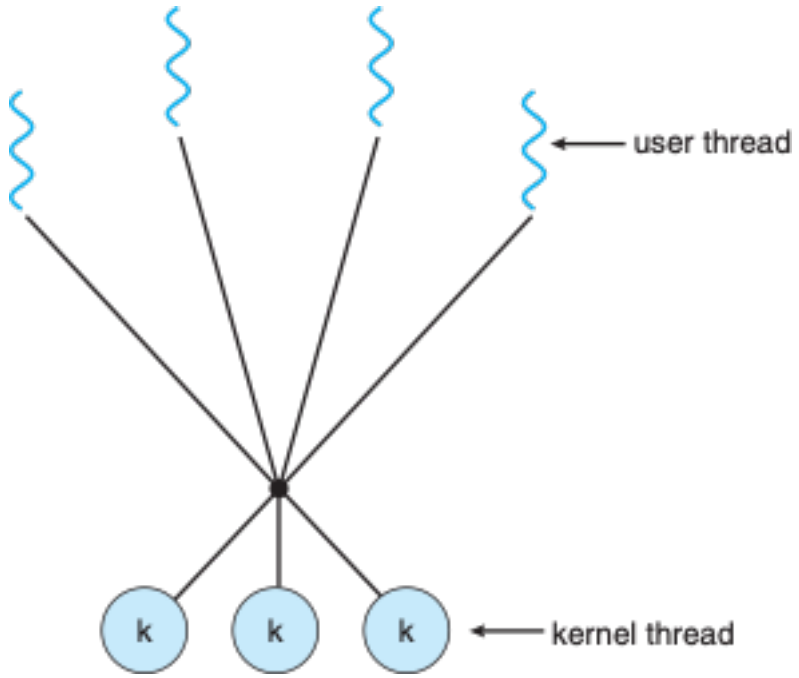
One-to-One

- 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



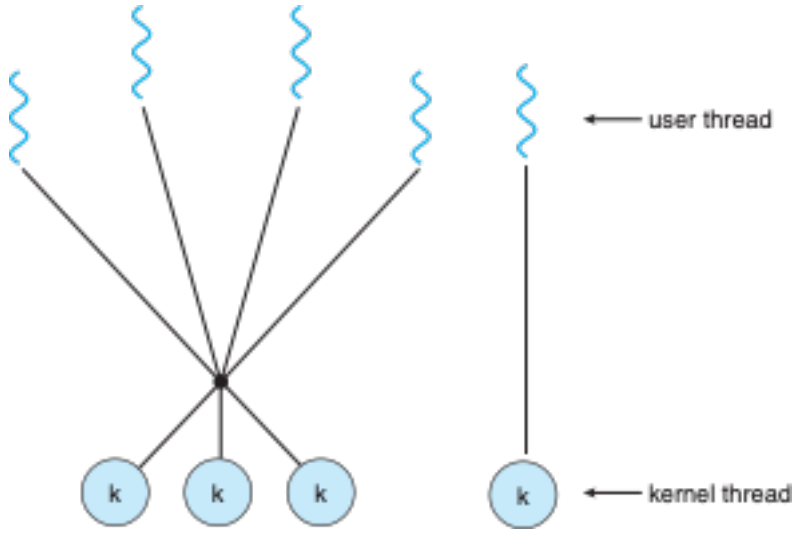
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Examples:
  - Solaris
  - Windows with 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



Thread Libraries

- Thread library provides the 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
    - Three main thread libraries are in use today:
        * POSIX Pthreads , Windows, and Java.
        * Pthreads , the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.
        * The Windows thread library is a kernel-level library available on Windows systems.
        * The Java thread API allows threads to be created and managed directly in Java programs.

Strategies for creating multiple threads

- Asynchronous threading
    - With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another.
        * Because the threads are independent, there is typically little data sharing between them.

Strategies for creating multiple threads

- Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes.
    - Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution.
    - Typically, synchronous threading involves significant data sharing among threads. For example, the parent thread may combine the results calculated by its various children. All of the following examples use synchronous threading.

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)
- The C programming language does not have built-in library support for multithreaded programming.
- The library to develop portable multithreaded applications is `pthread.h`; that is, the POSIX thread library. POSIX stands for portable operating system interface.
- POSIX threads are lightweight and designed to be very easy to implement. The pthread.h library is an external third-party library that helps you effectively do tasks.

Thread operations

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management, and process interaction.
- A thread does not maintain a list of created threads or know the thread that created it.
- All threads within a process share the same address space.

Thread operations

- Threads in the same process share:
  - Process instructions
  - Most data
  - open files (descriptors)
  - signals and signal handlers
  - current working directory
  - User and group id

Thread operations

- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno
  - pthread functions return “0” if OK.

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;
   }
```

Pthreads Example (Cont.)

```
   /* 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);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

The following are the functions in the `pthread.h` library that create, manipulate, and exit the threads.

- pthread_create
- pthread_join
- pthread_self
- pthread_equal
- pthread_exit
- pthread_cancel
- pthread_detach

pthread_create

- pthread_create creates a new thread with a thread descriptor.
- A descriptor is an information container of the thread state, execution status, the process that it belongs to, related threads, stack reference information, and thread-specific resource information allocated by the process.
- This function takes four arguments as parameters. The return type of this function is an integer.
- The following shows the syntax.

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

- The following describes the parameters.
  - `pthread_t` is a thread descriptor variable that takes the thread descriptor, has an argument, and returns the thread ID, which is an unsigned long integer.
  - `pthread_attr_t` is an argument that determines all the properties assigned to a thread. If it is a normal default thread, then you set the attribute value to NULL; otherwise, the argument is changed based on the programmer's requirements.
  - `start_routine` is an argument that points to the subroutines that execute by thread. The return type for this parameter is an void type because it typecasts return types explicitly. This argument takes a single value as a parameter. If you want to pass multiple arguments, a heterogeneous datatype should be passed that might be a struct.
  - `args` is a parameter that depends on the previous parameter; it takes multiple parameters as an argument.

pthread_join

- This function waits for the termination of another thread. It takes two parameters as arguments and returns the integer type. It returns 0 on successful termination and –1 if any failure occurs.
- The following shows the syntax.

  ```
  int pthread_join(pthread_t *thread, void thread_return)
  ```

- The following describes the parameters.
  - `thread` takes the ID of the thread that is currently waiting for termination
  - `thread_return` is an argument that points to the exit status of the termination thread, which is a NULL value.

pthread_self

- This function returns the thread ID of the currently running thread. The return type of this thread is an integer or the thread_t descriptor.
- It takes zero parameters as arguments. The following shows the syntax.

  ```
  pthread_t pthread_self()
  ```

  or

  ```
  int pthread_self()
  ```

pthread_equal

- This function checks whether two threads are equal or not.
- If the two threads are equal, then the function returns a nonzero value.
- If the threads are not equal, then it is zero.
- It takes two parameters as arguments and returns the integer as output. The following shows the syntax.

  ```
  int pthread_equal(pthread_t thread1, pthread_t thread2);
  ```

- `thread1` and `thread2` are the IDs for the first and second thread, respectively.

pthread_exit

- This function terminates a calling thread.
- It takes one argument as a parameter and returns nothing. The following shows the syntax.

  ```
  void pthread_exit(void *retval);
  ```

- `retval` is the return value of a thread that you want to detach it.

pthread_cancel

- This function is used for thread cancellation. It takes one parameter as an argument and returns an integer value. The following shows the syntax.

  ```
  int pthread_cancel(pthread_t thread);
  ```

- `pthread` is the thread ID of the thread that you want to cancel.

pthread_detach

- This function `detaches` a thread in a detached state.
- It takes a thread descriptor as an argument and returns the integer value as output. The following shows the syntax.

  ```
  int pthread_detach(pthread_t thread);
  ```

- `thread` is a descriptor variable that is passed as an ID, which you want to detach it.

References

- A. Silberschatz, P.B. Galvin, and G. Gagne. Operating System Concepts , 10th Edition; 2018; John Wiley and Sons.

Related Topics

- Topics from text: 4.1, 4.2, 4.3,4.4.1