Threads

Bojan Nokovic

Based on: "Operating Systems Concepts", 10th Edition Silberschatz Et al.

Jan. 2021

What is a Thread?

A sequential execution stream (thread) within a process (also called lightweight process).

A process has at least one thread of control.

A process has two parts: threads (concurrency) and address spaces (protection).

Motivation

Most modern applications are multithreaded.

A word processor may have a thread for

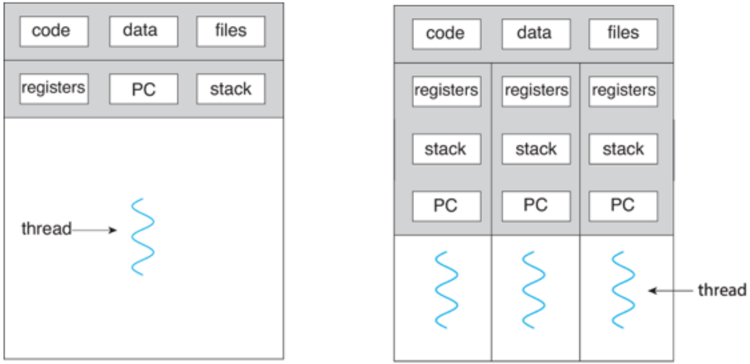
- Displaying graphics
- Responding to keystrokes from the user
- Spell and grammar checking

Multithreading can simplify code, increase efficiency.

Multiprocessing (multiprocessor systems): Split program into multiple threads to make it run faster by running on multiple processors.

Parallel programming.

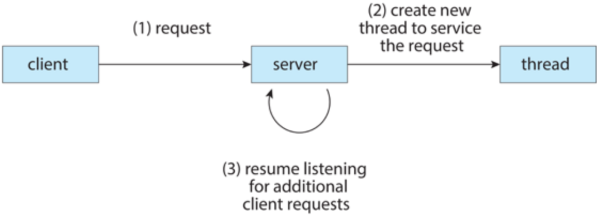
Single and Multithreaded Processes



Single-threaded process and a multithreaded process.

Multithreaded Server Architecture

Web server accepts client requests for web pages, images, sound, and so forth.



Old approach: new process-creation to serve request.

- Time consuming and resource intensive.

New approach: create a new thread to service the request and resumes listening for additional requests.

Thread States

States shared by all threads in the same process/address space:

- global variables
- file system
- code

States private to each thread

- PC, registers
- execution stack contains parameters, temporary variables, return addresses

Benefits

Responsiveness

- May allow continued execution if part of process is blocked, especially important for user interfaces.
- The time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.

Resource Sharing

- Processes - shared memory and message passing. Must be explicitly arranged by the programmer.
- Threads share resources of process by default.

Benefits

Economy

- Thread switching lower overhead than context switching.
- Thread creation consumes less time and memory than process creation.

Scalability

- Multithreading can take advantage of multicore architectures, where threads may be running in parallel on different processing cores.

Multicore Programming

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

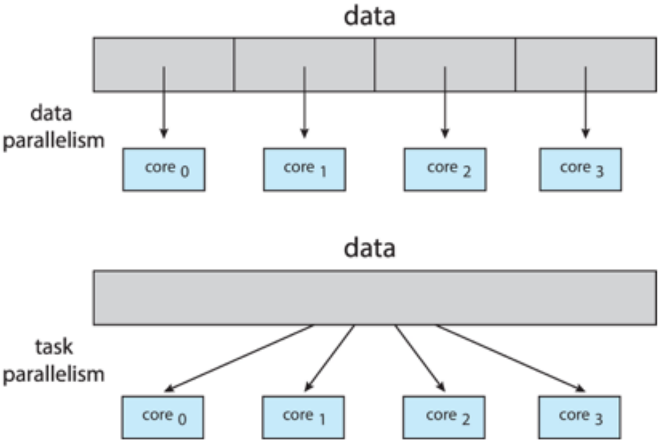
Types of parallelism

- 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

Example: Summing an array of size N

- Single core: Sum the elements [0] ...[N - 1].
- Dual core: Thread A sum elements [0] ...[N/2 - 1], while thread B sum elements [N/2] ...[N - 1]

Data and Task Parallelism



Amdahl's Law

Identifies performance gains from adding additional cores to an application that has both serial and parallel components.

S is serial portion %, N number of processing cores

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

What is *speedup* if application is 75% parallel and 25% serial when moving from 1 to 2 cores?

What happens when $N \rightarrow \infty$?

User Threads and Kernel Threads

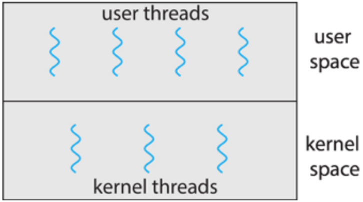
User threads - management done by user-level threads library

Three primary thread libraries:

- POSIX Pthreads
- Windows threads
- Java threads

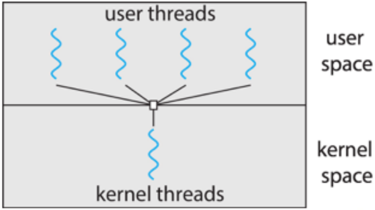
Kernel threads - Supported by the Kernel

- Windows, Linux, Mac OS X, iOS, Android



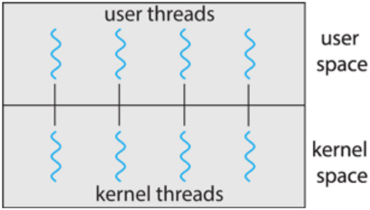
Many User Threads Mapped to One Kernel Thread

- 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: Solaris Green Threads, GNU Portable Threads



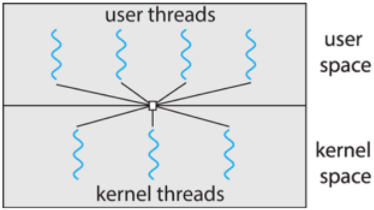
One to One

- Each user-level thread maps to kernel thread
- More concurrency than many-to-one ☺
- Number of threads per process sometimes restricted due to overhead
- Windows, Linux



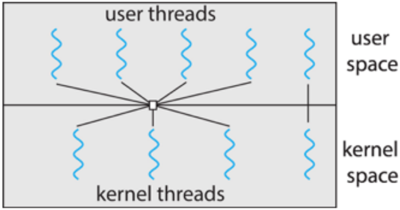
Many-to-Many

- Allows the operating system to create a sufficient number of kernel threads - a smaller or equal number of user threads.
- Windows with the *ThreadFiber* package
- Not very common



Two-level Model

- Similar to Many-to-Many, except that it allows a user thread to be bound to kernel thread



Although the many-to-many (M:M) model appears to be the most flexible of the models discussed, in practice it is difficult to implement.

Most operating systems now use the one-to-one model.

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

Pthreads

May be provided either as user-level or kernel-level

A POSIX specification standard (IEEE 1003.1c) API for thread creation and synchronization
`https://standards.ieee.org/standard/1003_1-2017.html`

API specifies behavior of the thread library, implementation is up to development of the library

Common in UNIX, Linux & Mac OS X

Pthreads Example sum.c

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

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

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

pthread_attr_init(&attr); /* get the default attributes */
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
pthread_join(tid,NULL); /* now wait for the thread to exit */

printf("sum = %d\n",sum);
}
```

Pthreads Example Continue

```
/**
 * 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);
}
```

On Linux compile with:

```
gcc -g -Wall -pthread sum.c -lpthread -o sum
```

Thread Programming Challenges

Write programs with multiple simultaneous points of execution, synchronizing through shared memory.

Global variables are shared among all the threads of the same process.

Threads can read and write the same memory locations.

The programmer is responsible for using the synchronization mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer!

Implicit Threading

Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads.

Creation and management of threads done by compilers and run-time libraries rather than programmers.

Five methods explored:

1. Thread Pools
2. Fork-Join
3. OpenMP
4. Grand Central Dispatch
5. Intel Threading Building Blocks

Thread Pools

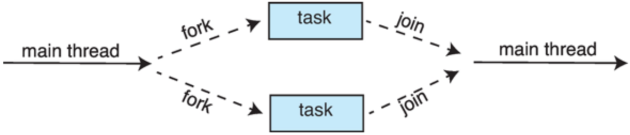
Create a number of threads in a pool where they await work.

Advantages

- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool
- Separating task to be performed from mechanics of creating task allows different strategies for running task
  - Tasks could be scheduled to run periodically

Fork-Join Parallelism

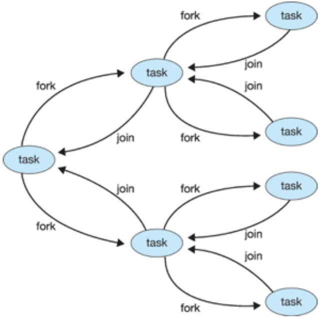
Multiple threads (tasks) are forked, and then joined.




```
Task(problem)
  if problem is small enough
    solve the problem directly
  else
    s1 = fork(new Task(subset of problem))
    s2 = fork(new Task(subset of problem))

    result1 = join(s1)
    result2 = join(s2)

    return combined results
}
```



Open Multi-Processing (OpenMP)

- Set of compiler directives and an API for C, C++, FORTRAN
- Support for parallel programming in shared-memory environments
- Identifies parallel regions - blocks of code that can run in parallel

```
#include <omp.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  /* sequential code */

  #pragma omp parallel {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

https://www.openmp.org/

Grand Central Dispatch

Apple technology for macOS and iOS operating systems

Extensions to C, C++ and Objective-C languages, API, and run-time library

```
^{ printf("I am a block");}
```

Blocks placed in dispatch queue - assigned to available thread in thread pool when removed from queue

Intel Threading Building Blocks (TBB)

Template library for designing parallel C++ programs

```
for (int i = 0; i < n; i++) {
        apply(v[i]);
}
```

```
parallel for (size_t(0), n,
   [=](size_t i) {apply(v[i]);}
 );
```

A serial version, and TBB version of a simple `for` loop.

Issues in Designing Multithreaded Programs

1. Semantics of `fork()` and `exec()` system calls
2. Signal handling - synchronous and asynchronous
3. Thread cancellation of target thread - asynchronous or deferred?
4. Thread-local storage
5. Scheduler Activations

Semantics of `fork()` and `exec()`

The semantics of the `fork()` and `exec()` system calls change in a multithreaded program.

If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded?

If a thread invokes the `exec()` system call, the program specified in the parameter to `exec()` will replace the entire process-including all threads.

Which of the two versions of `fork()` to use depends on the application.

Signal Handling

A signal is used in UNIX systems to notify a process that a particular event has occurred.

A signal handler is used to process signals

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled either by default or user-defined signal handler.

Every signal has default handler that kernel runs when handling signal

- User-defined signal handler can override default

Signal Handling

Synchronous signal example

- Illegal memory access
- Division by 0

Delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously.

- Terminating a process with specific keystrokes (such as `<control><C>`)
- Having a timer expire

Signal Handling

For single-threaded, signal is delivered to process.

Where should a signal be delivered for multi-threaded?

- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
- Assign a specific thread to receive all signals for the process

The method for delivering a signal depends on the type of signal generated - i.e. (`<control><C>`, for example) should be sent to all threads.

Thread Cancellation

Terminating a thread before it has finished.

Thread to be canceled is target thread

- Asynchronous cancellation terminates the target thread immediately
- Deferred cancellation allows the target thread to periodically check if it should be cancelled (default type)

```
pthread_t tid;
/* create the thread */
pthread_create(&tid, 0, worker, NULL);
...
/* cancel the thread */
pthread_cancel(tid);

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

Thread Cancellation

Invoking `pthread_cancel()` indicates only a request to cancel the target thread. However actual cancellation depends on how the target thread is set up to handle the request.

When the target thread is finally canceled, the call to `pthread_join()` in the cancelling thread returns.

A thread may set its cancellation state and type using an API.

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

The cancellation only occurs when thread reaches cancellation point i.e. `pthread_testcancel()` then cleanup handler is invoked.

Thread-Local Storage (TLS)

Allows each thread to have its own copy of data.

Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Different from local variables

- Local variables visible only during single function invocation
- TLS visible across function invocations

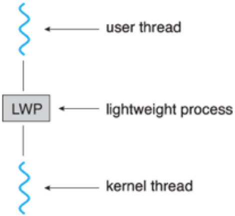
Similar to `static` data - unique to each thread

Scheduler Activations

Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.

Typically use an intermediate data structure between user and kernel threads - lightweight process (LWP)

- Appears to be a virtual processor on which process can schedule user thread to run
- Each LWP attached to kernel thread - how many to create?



Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library.

Thank you !

Operating Systems are among the most complex pieces of software ever developed !