Introduction to Multithreading

Godmar Back

Virginia Tech

September 28, 2023

The Case for Application-Level Concurrency

General purpose OS already provide the ability to execute processes concurrently. In many applications, we would like to pursue multiple, concurrent computations simultaneously *within* a process, e.g.

- Parallel Computing: perform multiple tasks or work on shares of data simultaneously
- Overlap I/O & Computation: checksum and repair while downloading in a file sharing program
- Serve a UI while performing background activity (spell check, contact server or backend for autosuggestions)
- Handling multiple clients simultaneously in a network server

Such *application-level* concurrency is supported by having multiple threads of execution.

Threads vs Processes

- Processes provide concurrent, separate logical flows of control within a system/machine
- Threads provide separate logical flows of control within a process

	share	do not share
Processes	machine resources, files on disk, inherited file descriptors, terminals	address space
Threads	address space[1], open file descriptors	stack[2] & registers

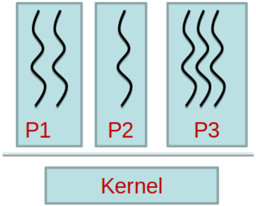
- Think of threads as multiple programs executing concurrently within a shared process, sharing all data and resources, but maintaining separate stacks and execution state.

[1] heap objects, all global variables

[2] local variables, function arguments, thread-local variables

VIRGINIA TECH

Implementing Threads



Question

Does the ability to maintain multiple flows of control require support from the underlying OS kernel?

Or...

Can it be implemented purely using libraries, etc. using the non-privileged instructions and other facilities available at user level?

Cooperative Multi-Threading

- It's possible to maintain multiple control flows entirely without kernel level support
- Exists in multiple variants in different languages, known as coroutines or user-level threads depending on variant
- Requires a primitive that saves & restores execution state
- Non-preemptive model: threads' access to the CPU is not preempted (taken away) unless the thread yields access to the CPU voluntarily
- Yield may be directed (saying which coroutine should run next) or undirected (run something else next), e.g. uthreads example
- In some higher-level languages, functions can "yield" temporary results as their execution state is saved and restored (e.g., Python or ES6 `yield`)
- Can be combined with asynchronous I/O: yield a promise object that represents an in-progress operation: `async`/`await`

Cooperative Multi-Threading

Advantages

- Requires no OS support
- Very lightweight and fast context switching
- Absence of certain data races, e.g. x++ is atomic
- Can yield scalable designs when combined with asynchronous I/O

Disadvantages

- Cannot make use of multiple CPUs
- Cannot preempt long-running or uncooperative threads easily
- Blocking I/O system calls will block all threads/entire process

Kernel-supported Threads

If the OS kernel supports threads directly, the above-mentioned problems can be solved

- Parallelism (using multiple CPUs/cores simultaneously) is possible because OS can assign threads to different CPUs, which enables speedup
- When performing I/O, the OS will move only the calling thread into the BLOCKED state
- The OS's preemptive scheduling model can share access to a CPU even if threads do not yield the CPU by (forcefully) interrupting threads and moving them to the READY state

Kernel-supported Threads

Dominant model today, supported by all major OS. Aka as kernel-level (as opposed to user-level) threading, but not to be confused with pure (inside the) kernel threads. Sometimes also called lightweight processes (LWP).

Hybrid Models

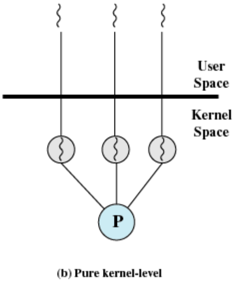


Figure 1: 1:1 model

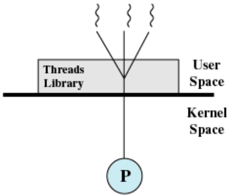


Figure 2: 1:N model

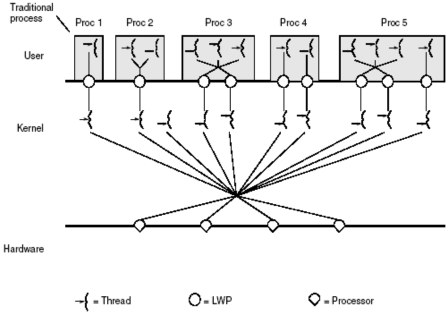


Figure 3: M:N model

Hybrid Models (cont'd)

- Pure user-level threading uses a 1:N model (N user-level threads share 1 OS-level thread)
- Pure kernel-level threading uses a 1:1 model (1 OS thread for each user thread)
- Hybrids (M:N) models try to obtain the best of user-level and kernel-supported threads.
- Examples: Windows Fibers, (now defunct) Solaris M:N model
- Increase in complexity (and lack of payoff) led to the M:N model being largely abandoned.
- Heavy investment/optimization in reducing the costs of the 1:1 model, e.g. fast user-level synchronization facilities

VIRGINIA TECH

POSIX Threads “pthreads”

- Execution model and API
- de facto standard for Unix-like OS, specified in IEEE Std.1003.10-2017
- retrofitted into overall POSIX standard as an extension, defining interaction between traditional process-based facilities and threads, e.g. signals
- many languages provide direct bindings for it - e.g., Java threads, C++ async, etc.

POSIX Threads Example

Create and Join

```c
struct thread_info {
  const char * msg;
};

static void *
thread_function(void *_arg)
{
  struct thread_info *info = _arg;
  printf("Thread 1 runs, "
         "msg was `%s'\n",
         info->msg);
  return (void *) 42;
}
```

```c
int
main()
{
  struct thread_info info = {
         .msg = "Hello, Thread" };

  pthread_t t;
  pthread_create(&t, NULL,
                 thread_function,
                 &info);

  uintptr_t status;
  pthread_join(t, (void **) &status);
  printf("Status %lu\n", status);
  return 0;
}
```

VIRGINIA TECH.

Java Threads Example

Create and Join

```java
public class JavaThread
{
  static class Example
    implements Runnable
  {
    String msg;
    int result;
    Example(String msg) {
      this.msg = msg;
    }

    @Override
    public void run() {
      System.out.println(msg);
      result = 42;
    }
  }
```

```java
  public static
  void main(String []av)
          throws Exception
  {
    var ex =
      new Example("Hello Thread");
    Thread t = new Thread(ex);
    t.start();
    t.join();
    System.out.println(ex.result);
  }
}
```

Concurrency Management

- Applications rarely create separate, new threads for individual tasks, particularly if small
- Instead, they manage the number of threads needed to perform work and distribute work to threads
- Trade-off:
  - Too many threads: leads to increased contention for resources and resulting overhead from managing that
  - Too few threads: risks underutilization of CPUs/cores
- Target: number of READY + RUNNING threads around equal to number of cores
- Solution: thread pools [1]

VIRGINIA TECH

Java's ExecutorService Example

```
import java.util.concurrent.*;
public class FixedThreadPool
{  static final int N = 8;
   public static void main(String []av) throws Exception {
      ExecutorService ex = Executors.newFixedThreadPool(3);
      Future<?> f[] = new Future<?>[N];
      for (int i = 0; i < N; i++) {
         final int j = i;
         f[i] = ex.submit(new Callable<String>() {
            public String call() {
               return "Future #" + j + " brought to you by "
                                 + Thread.currentThread();
            }
         });
      }
      for (int i = 0; i < N; i++)
         System.out.println(f[i].get());
      ex.shutdown();
   }
}
```

Parallel Divide-and-Conquer Example

Pseudocode Source: Lea [1]

```
Result solve(Param problem) {
  if (problem.size <= GRANULARITY_THRESHOLD) {
    return directlySolve(problem);
  } else {
    in-parallel {
      Result l = solve(lefthalf(problem));
      Result r = solve(rightHalf(problem);
    }
    return combine(l, r);
  }
}
```

Challenge

An execution framework must map the tasks created in `in-parallel` to threads.

VIRGINIA TECH.

References

[1] Doug Lea.
A java fork/join framework.
In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.