CS 3214
Computer Systems
Multi-threading and Concurrency
MULTI-THREADING
Processes vs Threads

• Processes execute concurrently and share resources:
  – Files on disk; Open files (via inherited file descriptors); Terminal, etc.
  – Kernel-level concurrency
• but do not (usually) share any of their memory
  – cannot share data easily by exchanging pointers to it
• Threads are separate logical flows of control (with separate stacks!) that share memory and can refer to same data
  – Different models and variations exist
  – Application-level concurrency
Cooperative Multi-threading

- Special case of user-level threading
  - Is easy to implement using ‘swapcontext’ – see next slide
- Support multiple logical execution flows
  - Each needs own stack ⇒ so has its own (procedure-) local (automatic) variables
  - But share address space ⇒ so shares heap, global vars (all kinds: global, global static, local static)
- In cooperative multi-threading, a context switch can occur only if a thread voluntarily offers the CPU to (any) other thread (“yield”); later resumes and returns
  - Can build resource abstractions on top where threads yield if they cannot obtain the abstracted resource
- This is called a “non-preemptive” model
- If yield is directed (“yield to x”) this model is called “co-routines”
Cooperative Multithreading via ‘swapcontext’

```c
static char stack[2][65536]; // a stack for each coroutine
static ucontext_t coroutine_state[2]; // container to remember context

// switch current coroutine (0 -> 1 -> 0 -> 1 ...
static inline void
yield_to_next(void)
{
    static int current = 0;

    int prev = current;
    int next = 1 - current;

    current = next;
    swapcontext(&coroutine_state[prev],
                &coroutine_state[next]);
}

static void
coroutine(int coroutine_number)
{
    int i;
    for (i = 0; i < 5; i++) {
        printf("Coroutine %d counts i=%d (&i=%p)\n",
               coroutine_number, i, &i);
        yield_to_next();
    }
}
```
Cooperative Multi-threading (cont’d)

• Advantages:
  – Requires no OS support
  – Context switch very fast (usually involves only saving callee-saved regs + stack pointer)
  – Reduced potential for certain types of race conditions
    • E.g., i++ will never be interrupted
  – Used in very high-performance server designs & discrete event simulation

• Disadvantages
  – OS sees only one thread in process, system calls block entire process (if they block)
  – Cannot make use of multiple CPUs/cores
  – Cannot preempt infinitely-looping or uncooperative threads
    • (though can fake preemption in just-in-time compiled languages by letting compiler insert periodic checks)
Use Cases for App-level Concurrency

• Overlap I/O with computation
  – E.g. file sharing program downloads, checksums, and saves/repairs files simultaneously

• Parallel computation
  – Use multiple CPUs

• Retain interactivity while background activity is performed
  – E.g., still serve UI events while printing

• Handling multiple clients in network server apps

• By and large, these are best handled with a preemptive, and (typically) kernel-level multi-threading model
Preemptive Multi-Threading

• Don’t require the explicit, programmer-inserted “yield” call to switch between threads
• “Switch” mechanism can be implemented at user-level or kernel-level
  – User-level threads: can be built using signal handlers (e.g. SIGVTALRM)
    • Requires advanced file descriptor manipulation techniques to avoid blocking entire process in system call
  – Kernel-level threads: natural extension for what OS already does when switching between processes
    • Integrated with system calls – only current thread blocks
  – Hybrids
• Kernel-level threads is the dominant model today
Threading Models

Hybrid, so-called M:N model:

Source: Solaris documentation (left), Stallings (right)
# Threading Implementations Overview

<table>
<thead>
<tr>
<th>Does OS know about threads within a process?</th>
<th>Do blocking syscalls (e.g., read()) block entire process?</th>
<th>Can threads be preempted even if they don’t call yield()?</th>
<th>Can multiple cores/CPUs be utilized?</th>
<th>* Most common model today</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Preemptive Kernel-level Threads *</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Preemptive user-level threads</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Cooperative user-level threads</td>
</tr>
</tbody>
</table>
Threading Models

- Linux, Windows, Solaris 10 or later, OSX: use 1:1 model with kernel-level threads. OS manages threads in each process
  - threads in Java/C#, etc. are typically mapped to kernel-level threads
- Solaris (pre-10), Windows “fibers”: provide M:N model
  - Attempted to obtain “best of both worlds” – turned out to be difficult in practice
- User-level Threads
  - used mainly in special/niche applications today
Threading APIs

• How are threads embedded in a language/environment?
• POSIX Threads Standard (in C)
  – pthread_create(), pthread_join()
  – Uses function pointer to denote start of new control flow
  – Largely retrofitted in Unix world
  – Needed to define interaction of signals and threads
• Java/C#
  – Thread.start(), Thread.join()
  – Java: Using “Runnable” instance
  – C#: Uses “ThreadStart” delegate
Example pthread_create/join

```c
static void * test_single(void *arg)
{
    // this function is executed by each thread, in parallel
}

pthread_t threads[NTHREADS];
int i;
for (i = 0; i < NTHREADS; i++)
    if (pthread_create(threads + i, (const pthread_attr_t*)NULL,
                       test_single, (void*)i) == -1)
        { printf("error creating pthread\n"); exit(-1); }

/* Wait for threads to finish. */
for (i = 0; i < NTHREADS; i++)
    pthread_join(threads[i], NULL);
```

Use Default Attributes – could set stack addr/size here

2\textsuperscript{nd} arg could receive exit status of thread
Java Threads Example

```java
public class JavaThreads {
    public static void main(String[] args) throws Exception {
        Thread[] t = new Thread[5];
        for (int i = 0; i < t.length; i++) {
            final int tnum = i;
            Runnable runnable = new Runnable() {
                public void run() {
                    System.out.println("Thread "+tnum);
                }
            };
            t[i] = new Thread(runnable);
            t[i].start();
        }
        for (int i = 0; i < t.length; i++)
            t[i].join();
        System.out.println("all done");
    }
}
```

Threads implements Runnable – could also have subclassed Thread & overridden run()

Thread.join() can throw InterruptedException – can be used to interrupt thread waiting to join via Thread.interrupt
import java.util.concurrent.*;
public class FixedThreadPool {
    
    public static void main(String[] av) throws Exception {
        ExecutorService ex = Executors.newFixedThreadPool(4);
        final int N = 4;
        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();
                }
            });
        }
        System.out.println("Main thread: " + Thread.currentThread());
        for (int i = 0; i < N; i++)
            System.out.println(f[i].get());
        ex.shutdown();
    }
}
Explicit Threads vs. Pools

- Overhead:
  - Startup overhead per thread relatively high (between $10^4$ & $10^5$ cycles); pools amortize

- There is no point in having more threads than there are physical cores
  - Compete for available CPUs
  - Unless some subset is blocked on I/O or other conditions

- Still, sizing of pools that maximizes throughput can be challenging
  - "cachedThreadPool" creates thread whenever needed, but reuses existing ones that are idle
  - "fixedThreadPool" - # of threads fixed
  - Can implement custom policies