CSCE 313-200
Introduction to Computer Systems
Spring 2022

Threads
Dmitri Loguinov
Texas A&M University

February 1, 2022
Updates

- Quiz on Thursday
  - System Programming Tutorial (pay attention to exercises)
  - Pointers, VS debugging tools/strategies, APIs
  - Common Microsoft data types
  - The last two lectures (OS concepts, processes)
- Common issues in hw1p1
  - Not waiting for CC.exe to exit
  - Printing room with %X instead of %l1x
  - Not handling CC errors in ResponseCC::status
- Make sure to check for API errors
  - Catches bugs sooner, simplifies debugging
Chapter 4: Roadmap

4.1 Processes and threads
4.2 SMP
4.3 Micro-kernels
4.4 Windows threads
4.5 Solaris threads
4.6 Linux threads

Part II

Chapter 3: Processes
Chapter 4: Threads
Chapter 5: Concurrency
Chapter 6: Deadlocks
Motivation

• Why parallelize a single program?
• Two main reasons
  – Take advantage of multi-core CPU capacity
  – Perform many concurrent blocking operations quickly
• While non-blocking I/O helps with the second issue, it doesn’t solve the first one
  – Also makes code more complex
**Taxonomy**

- Why not fork a new process then?
- Two main issues:
  - Frequent process context switch is expensive
  - Data sharing may be inefficient (i.e., through kernel) and possibly tedious to program
- Thus, there is a need for a simpler/faster concurrency model that uses threads
  - **Thread** is a dispatchable unit of work within a process
How to Implement Threads

• Historically, threads didn’t exist in multi-tasked OSes
  - Users wrote special libraries (e.g., pthreads) to emulate threads
  - OS scheduled the process, then library scheduled threads

• Benefits of User-Level Threads (ULT):
  - Thread switch completely in user mode (i.e., fast)
  - Control over scheduler and its policy
  - Portability of code (no dependency on OS APIs)

• Problems:
  - When kernel APIs block, the entire process is blocked
  - No ability to run concurrently on multiple CPUs
How to Implement Threads

- Later, OSes became thread-aware and offered Kernel-Level Threads (KLT)
  - Another term is Light Weight Processes (LWT)

- Benefits of KLT:
  - Multi-CPU usage by the same program, non-blocking I/O

- Drawbacks compared to ULT:
  - Requires kernel mode switch after each slice (lower performance)
  - Less flexibility with scheduling
**Performance**

- How expensive is context switch?
  - Traditional numbers suggest ULT switch is 10x faster than KLT, which is 4-5x faster than process switch
- Windows benchmark agrees with the last ratio
  - ULT rarely used on Windows, no performance results readily available

<table>
<thead>
<tr>
<th>Operation</th>
<th>ULT</th>
<th>KLT</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>34</td>
<td>948</td>
<td>11,300</td>
</tr>
<tr>
<td>Event wait + switch</td>
<td>37</td>
<td>441</td>
<td>1,840</td>
</tr>
</tbody>
</table>

old VAX Unix

<table>
<thead>
<tr>
<th>Operation</th>
<th>ULT</th>
<th>KLT</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event wait + switch</td>
<td>0.44</td>
<td>2.2</td>
<td></td>
</tr>
</tbody>
</table>

AMD Phenom II X6 2.8 GHz

- While these latencies are small, they do increase as the # of threads/processes in the ready state rises
Kernel Threads

- Difference from the single-threaded model
  - Threads have separate stacks and execution context called Thread Control Block (TCB), but share all virtual memory
Kernel Threads

• OS still enforces separation between processes
  – However, threads are not protected from each other
  – Buffer overflow in one thread may wipe out data of other threads in the same process

• Process owns
  – Virtual address space and shared memory
  – Security attributes of all objects (e.g., open files)

• Threads own
  – TCB that includes thread state (e.g., blocked, running, ready), thread context (registers), scheduler priorities and its auxiliary info, pending wait events
  – Execution stack (user and kernel)
Using Threads

- In Windows:
  - Security = NULL, stacksize = 0 (default), flags = 0
  - Must provide the address of start function
    - Thread executes from that address
    - Current thread continues as normal
  - Definition of a thread function:

```c
typedef DWORD (__stdcall *LPTHREAD_START_ROUTINE)( [in] LPVOID lpThreadParameter );
```

```c
HANDLE WINAPI CreateThread (  
    __in_opt  LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    __in     SIZE_T dwStackSize,  
    __in     LPTHREAD_START_ROUTINE lpStartAddress,  
    __in_opt LPVOID lpParameter,  
    __in     DWORD dwCreationFlags,  
    __out_opt LPDWORD lpThreadId );
```

```c
DWORD __stdcall MyThread (LPVOID lpThreadParameter);
```
# Using Threads

```c
#define THREADS_TO_RUN 100

void main (void) {
    HANDLE thread [THREADS_TO_RUN]; // stores thread handles
    ThreadParams t [THREADS_TO_RUN]; // parameters passed to threads
    MyExample me;   me.count = 0;

    for (int i = 0; i < THREADS_TO_RUN; i++) { // start a bunch of threads
        t[i].threadID = i; // assign seq # to this thread
        t[i].me = &me; // must pass a pointer to shared variables/classes
        // run thread with default stack size
        if ((thread[i] = CreateThread (NULL, 0, ThreadStarter, t + i, 0, NULL)) == NULL) {
            printf (“failed to create thread %d, error %d\n”, i, GetLastError());
            exit (-1);
        }
    }

    for (int i = 0; i < THREADS_TO_RUN; i++) { // now hang here waiting for threads to quit
        WaitForSingleObject (thread[i], INFINITE);
        CloseHandle (thread[i]);
    }
    printf (“result = %d\n”, me.count);
}
```

```c
DWORD __stdcall ThreadStarter (LPVOID p) {
    ThreadParams *t = (ThreadParams*) p;
    t->me->Run (t->threadID);
    return 0;
}
```

```c
class MyExample {
public:
    int count;
    void Run (int threadID);
};
```

```c
class ThreadParams {
public:
    MyExample* me;
    int threadID;
};
```
Using Threads

- Try to encapsulate all functionality inside your class member functions
- Local variables are never shared (they stay in thread stack)
- Global and static variables
  - Shared between threads, but they are considered bad style and thus not recommended
- Heap-allocated blocks
  - Normally not shared unless you provide a common pointer to multiple threads and they dereference it
Using Threads

- Thread execution is **non-deterministic**
  - Threads can be interrupted at any time
  - Speed of execution may differ by any factor
- Make sure each thread gets its own copy of ThreadParams to avoid problems like this:

```c
void MyExample::Run (int threadID) {
    Sleep (100);
    count ++;
    printf ("Thread %d finished\n", threadID);
}
```

```c
ThreadParams t;
t.me = &me;

for (int i = 0; i < THREADS_TO_RUN; i++) {          // start a bunch of threads
    t.threadID = i;                                // assign # to this thread
    if ((thread [i] = CreateThread (NULL, 0, ThreadStarter, &t, 0, NULL)) == NULL) {
        printf ("failed to create thread %d, error %d\n", i, GetLastError());
        exit (-1);
    }
}
```

all threads may get their threadID = THREADS_TO_RUN-1
Chapter 4: Roadmap

4.1 Processes and threads

4.2 SMP

4.3 Micro-kernels

4.4 Windows threads

4.5 Solaris threads

4.6 Linux threads
• **SMP (Symmetric Multi-Processing)**
  - Consists of multiple CPUs connected by bus (e.g., HyperTransport in AMD)
  - Each CPU contains multiple cores and dedicated memory controller

• SMP benefits:
  - Performance
  - Availability (e.g., failure of some CPUs does not have to crash the system)
  - Scalability (e.g., more CPUs can be added to an existing motherboard if it supports them)
SMP

- CPU clock speed no longer scales due to insurmountable heat problems
  - Scaling cores is much easier at this stage
- Consumer-grade computers today
  - Intel Xeon w/56-cores, 8-CPU configurations (448 cores per motherboard), Intel Phi expansion card w/60 cores
  - CUDA (nVidia Titan) video cards with 5000+ cores
- Evolution of computer architecture:
  - Sequential computers had a single CPU
  - Traditional 1940s-1950s mainframes
SMP

- Notation:
  - S = single, M = multiple
  - I = instruction, D = data
- Level 1
  - SISD: single core, no internal parallelism
  - SIMD: single core, can run the same instruction on multiple RAM locations in parallel (e.g., video cards, SSE, MMX, AVX)
  - MIMD: different instructions on different data (i.e., multiple cores)
  - MISD: rarely implemented
• **Level 2:**
  - **Shared memory:** single motherboard
  - **Distributed memory:** multiple computers

• **Level 3:**
  - **Asymmetric:** OS run on dedicated core, programs run everywhere else
  - **SMP:** OS and programs share all cores (modern computers and kernels) ← this course
  - **Clusters:** racks of servers, possibly geographically distributed in datacenters
Wrap-up

- Cache coherence issues drastically affect consistency and performance when multiple threads modify the same RAM location.