From Processes to Threads

Processes and Threads

- **Process** abstraction combines two concepts
  - **Concurrency**: Each process is a sequential execution stream of instructions
  - **Protection**: Each process defines an address space
    - Address space identifies all addresses that can be touched by the program

- **Threads**
  - Key idea: separate the concepts of concurrency from protection
  - A thread represents a sequential execution stream of instructions
  - A process defines the address space that may be shared by multiple threads

The Case for Threads

Consider the following code fragment

```c
for(k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];
```

Is there a missed opportunity here? On a Uni-processor?
On a Multi-processor?

The Case for Threads

Consider a Web server

- get network message from client
- get URL data from disk
- compose response
- send response

How well does this web server perform?
Introducing Threads

- A thread represents an abstract entity that executes a sequence of instructions
  - It has its own set of CPU registers
  - It has its own stack
- Threads are lightweight
  - There is no thread-specific heap or data segment (unlike process)
  - Therefore, context switching is much cheaper than for a process
- Examples:
  - OS-supported: Sun's LWP, POSIX's threads
  - Language-supported: Modula-3, Java

Programmer's View

```c
main()
{
    some code
    tid = CreateThread(fn1, arg0, arg1, ...);
    some more code
    fn1(int arg0, int arg1, ...)
    some code
}
```

After `CreateThread` is called, execution in parent thread continues in main function, and, in parallel, execution in child thread starts at `fn1`

How Can it Help?

- Consider the following code fragment
  ```c
  for(k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];
  ```
- Rewrite this code fragment as:
  ```c
  CreateThread(fn, 0, n/2);
  CreateThread(fn, n/2, n);
  fn(l, m)
  for(k = l; k < m; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];
  ```
- What did we gain?

How Can it Help?

- Consider a Web server
  Create a number of threads, and for each thread do
  - get network message from client
  - get URL data from disk
  - compose response
  - send response
- What did we gain?
**Implementing Threads**

CreateThread(pointer_to_procedure, arg0, ...) {
  // allocate a new TCB and stack
  TCB tcb = new TCB();
  Stack stack = new Stack();
  // initialize TCB and stack with initial register values and address of
  first instruction
  tcb.pc = Stub;
  tcb.stack = stack;
  tcb.arg0Reg = pointer_to_procedure;
  tcb.arg1Reg = arg0;
  ...
  // Tell the dispatcher about the newly created thread
  ReadyQ.add(tcb);
} Stub(proc, arg0, arg1, ...) {
  (*proc)(arg0, arg1, ...);
  DeleteCurrentThread();
}

**Threads vs. Processes**

**Threads**
- No data segment or heap
- Multiple can coexist in a process
- Share code, data, heap and I/O
- Have own stack and registers, but no isolation from other threads in the same process
- Inexpensive to create
- Inexpensive context switching

**Processes**
- Have data/code/heap and other segments
- Include at least one thread
- Have own address space, isolated from other processes’
- Expensive to create
- Expensive context switching

**Threads Life Cycle**

- Processes define address space; threads share address space
- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information
- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, ...), register values, a pointer to PCB, ...

**Threads (just like processes) go through a sequence of start, ready, running, waiting, and done states**
### Dispatching Kernel Threads

**Basic operation**

- Thread is running
- Switch to kernel
- Save thread state (TCB)
- Choose new thread to run
- Load state of the chosen thread (from TCB)

**Switch to kernel initiated by**

- System call (e.g., IO, yield CPU, etc.)
- Exception
- Interrupt (e.g., timer interrupt)

### User-level Threads

**Motivation**

- Threads are a useful programming abstraction
- Implement thread creation/manipulation using procedure calls to a user-level library rather than system calls

**User-level threads**

- User-level library implementations for
  - `CreateThread()`, `DestroyThread()`, `Yield()`, ...
- User-level library performs same actions of corresponding system calls
- **Main difference with kernel threads**: thread management is under the control of user-level library

### Pros and Cons of User-level Threads

**Pros**

- No context for switching between threads
- Flexible scheduling
  - Application specific
  - Process specific
  - Threads voluntarily give up CPU—easy to reason about!

**Cons**

- OS is unaware of user-level threads
  - Thread blocked for I/O blocks entire process
  - OS schedules processes independent of number of threads within a process

### Kernel Threads vs. User-level Threads

**Kernel threads:**

- Known to OS
- Switching between them within same process is inexpensive (though not quite as for user-level threads)
  - The values of registers, PC, and stack pointers are changed
  - Memory management information does not change
- Kernel uses process scheduling algorithms to manage threads

**User-level threads:**

- OS does not know about user-level threads
- OS is only aware of the process containing threads
- OS schedules processes, not threads
- Programmer uses a threads library to manage threads (create, delete, synchronize and schedule)
Concurrency is great ...

```
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++; 
}
fn2(int arg1) {
    a = arg1;
}
```

What are the values of a & b at the end of execution?

...but can be problematic

```
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++; 
}
fn2(int arg1) {
    a = 0;
}
```

What are the values of a & b at the end of execution?

Some More Examples

What are the possible values of x in these cases?

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread1: x = 1;</td>
<td>Thread2: x = 2;</td>
</tr>
<tr>
<td>Initially y = 10;</td>
<td></td>
</tr>
<tr>
<td>Thread1: x = y + 1;</td>
<td>Thread2: y = y * 2;</td>
</tr>
</tbody>
</table>

Initially x = 0;
| Thread1: x = x + 1; | Thread2: x = x + 2; |

This is because ...

- Order of process/thread execution is **non-deterministic**
  - A system may contain multiple processors and cooperating threads/processes can execute simultaneously
  - Thread/process execution can be interleaved because of time-slicing

- Operations are often not **atomic**
  - An atomic operation is one that executes to completion without any interruption or failure—-it is "all or nothing"

- Goal: Ensure correctness under ALL possible interleaving

- **Challenge**
  - Enumerating all cases is not possible!
  - Need to define synchronization constructs and programming style for developing concurrent programs