Processes and Threads

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

- **Threads**
  - Key idea: decouple 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 segment:
  \[
  \text{for } (k = 0; k < n; k++) \\
  a[k] = b[k] \times c[k] + d[k] \times e[k]
  \]

- Is there a missed opportunity here?
The Case for Threads

Consider a Web server:
- get network message from client
- get URL data from disk
- compose response
- send response

A Third Way

Run the server as a single finite state machine
- a large event queue
- a single thread
- no blocking system calls: on I/O, save current state in a table and fetch next event
  - if new request, then start on it
  - if I/O completion, then fetch state from table and continue
- Harder to program against

Multithreaded Processing Paradigms

Run the server as a single finite state machine:
- a large event queue
- a single thread
- no blocking system calls: on I/O, save current state in a table and fetch next event
  - if new request, then start on it
  - if I/O completion, then fetch state from table and continue
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
- Much faster context switching!
- Examples:
  - OS-supported: Sun's LWP, POSIX's threads
  - Language-supported: Modula-3, Java

<table>
<thead>
<tr>
<th>Per Process</th>
<th>Per Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open Files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and thread handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting info</td>
<td></td>
</tr>
</tbody>
</table>

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.
Implementing Threads

CreateThread(pointer_to_procedure, arg0, ...) {
  // allocate a new TCB and stack
  TCB tcb = ... created thread
  ReadyQ.add(tcb);
}

Stub(proc, arg0, arg1, ...) {
  (*proc)(arg0, arg1, ...);
  DeleteCurrentThread();
}

Threads Life Cycle

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

How Can it Help?

Consider again the following code fragment
for(k = 0; k < n; k++)
  a[k] = b[k] \times c[k] + d[k] \times e[k];
CreateThread(fn, n/2, n);
fn(l, m)
  for(k = l; k < m; k++)
    a[k] = b[k] \times c[k] + d[k] \times e[k];

Threads vs. Processes

<table>
<thead>
<tr>
<th>Threads</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>No data segment or heap</td>
<td>Have data/code/heap and other segments</td>
</tr>
<tr>
<td>Multiple can coexist in a process</td>
<td>Include at least one thread</td>
</tr>
<tr>
<td>Have own stack and registers, but no isolation from other threads in the same process</td>
<td>Have own address space, isolated from other processes</td>
</tr>
<tr>
<td>Inexpensive to create</td>
<td>Expensive to create</td>
</tr>
<tr>
<td>Inexpensive context switching</td>
<td>Expensive context switching</td>
</tr>
</tbody>
</table>
**User-level Threads**

- No need of OS support!
- Threads execute on top of a run time system
  - User-level library implementations for
    - `CreateThread()`, `DestroyThread()`, `Yield()`, ...
- If thread may be suspended (sys call, synch operation, etc)
  - call runtime system
  - check if thread must be suspended; if so
    - save current context
    - find unblocked thread
    - load that thread’s context

**Pros and Cons of User-level Threads**

**Pros**

- Very low context switch cost
  - ~2 orders of magnitude
- Flexible scheduling
  - Application specific
  - Process specific
- Threads voluntarily give up CPU—easy to reason about!

**Cons**

- OS is unaware of user-level threads
  - Blocking sys calls invoked by a thread block entire process
  - Page fault: same thing!
  - Round-robin, anyone?
  - OS schedules processes independent of number of threads within a process

**What to do???

- Change all blocking calls to non-blocking calls
- Rewrite sys call library to detect, if possible, whether a call will block
  - Use `SELECT` before `READ`
- Have the runtime system request periodic interrupts (say, every second)
- Implement threads in the kernel!

**Kernel-level Threads**

- All calls to run time system are now sys calls!
  - higher cost in managing thread operations
    - 1/10 of process; 10x of user-level threads
  - the "cost of generality": a service in the kernel should fit all applications!
Scheduler Activations – I

- **Goal:** combine functionality and ease of use of kernel threads with performance of user-level threads (ult)
- **Abstraction:** virtual multiprocessor for each ult system
  - kernel allocates processors to address spaces
  - each address space has complete control on which threads run on allocated processor
  - Kernel notifies thread package on:
    - change in # of processors
    - thread blocking or waking up on I/O events
- Thread package notifies kernel on:
  - adding or returning processors—other thread management operations are performed in user space

Scheduler Activations – II

- **Scheduler activations** let kernel processor allocator communicate with user-level threads
  - data structure similar to a kernel thread
  - multiple execution stacks (one mapped to kernel, one for each thread in user space)
  - kernel keeps activation control block
  - when kernel knows that the state of thread has changed, it activates ult package with an upcall

On interrupt:
- switch to kernel
- if interrupt does not apply to virtual processor, continue
- otherwise, suspend interrupted thread and give control to run time system of user level thread package

Any negatives?
- What happens if interrupt occurs when a thread holds a lock?

Concurrency is great ...

```java
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 value of `a` and `b` at the end of execution?

...but can be problematic

```java
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?
Some More Examples

What are the possible values of $x$ in these cases?

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

<table>
<thead>
<tr>
<th>Initially $x = 0$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread1: $x = x + 1$;  Thread2: $x = x + 2$;</td>
</tr>
</tbody>
</table>

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"
    - $x := x+1$ is not atomic
      - read $x$ from memory into a register
      - increment register
      - store register back into memory
- Goal: Ensure correctness under **ALL** possible interleaving

We have a problem...

- Enumerating all cases is impractical
- We need to
  - define constructs to help with synchronization and coordination
  - develop a programming style that eases the construction of concurrent programs
  - more fundamentally, we need to know what we are talking about---we we mention "synchronization" or "coordination"...