Concurrency and Threads

Thread: an abstraction for concurrency

- A single-execution stream of instructions that represents a separately schedulable task
  - OS can run, suspend resume thread at any time
  - Finite Progress Axiom: execution proceeds at some unspecified, non-zero speed
- Virtualizes the processor
  - programs run on machine with an infinite number of processors
- Allows to specify tasks that should be run concurrently...
  - ...and lets us code each task sequentially

Where threads are useful

- To express a natural program structure
  - updating the screen, fetching new data, receiving user input
- Exploiting multiple processors
  - different threads may be mapped to distinct processors
- Masking long latency of I/O devices
  - do useful work while waiting

A simple API

```c
void sthread_create(thread, func, arg)
  \tcreates a new thread in thread, which will execute function func with arguments arg

void sthread_yield()
  \tcalling thread gives up the processor

sthread_join(thread)
  \twait for thread to finish, then return the value thread passed to sthread_exit.

sthread_exit(ret)
  \tfinish caller; store ret in caller's TCB and wake up any thread that invoked sthread_join(caller)
```
Implementing the thread abstraction: the state

One abstraction, many flavors

Threads Life Cycle

Threads Life Cycle
Threads Life Cycle

 Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states.

- Thread creation (e.g. \texttt{sthread_create()})
- Scheduler resumes thread
- Scheduler suspends thread (e.g. \texttt{sthread_yield()})
- Thread yields

TCB: Ready List
Registers: in TCB
Waiting

TCB: Running List
Registers: Processor
Waiting
Threads Life Cycle

Threads (just like processes) go through a sequence of Init, Ready, Running, Waiting, and Finished states.

- Thread creation (e.g., `sthread_create()`)
- Scheduler resumes thread
- Ready
- Running
- Waiting
- Finished
- Scheduler suspends thread (e.g., `sthread_yield()`)
- Thread waits for event (e.g., `sthread_join()`)
- Event occurs (e.g., other thread calls `sthread_yield()`)
- Scheduler suspends thread (e.g., `sthread_exit()`)
- Thread waits for event (e.g., `sthread_join()`)
- Thread exit (e.g., `sthread_exit()`)
Context switching in-kernel threads

You know the drill:
- Thread is running
- Switch to kernel
- Save thread state (to TCB)
- Choose new thread to run
- Load its state (from TCB)
- Thread is running

Policy decision left to the scheduler

What triggers a context switch?

- Internal events
  - system call
    - thread blocks for I/O
    - synchronization: thread wait for another thread to do something
    - thread explicitly gives up CPU (thread_yield())
  - exception
- External events
  - interrupt
    - I/O (type character, disk request finishes,...)
    - timer interrupt

One story, two perspectives
System calls: one story, two perspectives

In-kernel thread's viewpoint

Thread 1
```java
while (true) {
    sthread_yield()
}
```

Thread 2
```java
while (true) {
    sthread_yield()
}
```

In-kernel thread's viewpoint

System calls: one story, two perspectives

Thread 1
```java
while (true) {
    sthread_yield()
    call sthread_yield()
    save state to stack
    save state to TCB
    choose to run T2
    load T2 state
    1. change SP to T2
    2. pop T2's general purpose registers
    3. pop SP and execution flags
}
```

Thread 2
```java
while (true) {
    sthread_yield()
    call sthread_yield()
    save state to stack
    save state to TCB
    choose to run T1
    load T1 state
    call sthread_yield()
    save state to stack
    save state to TCB
    choose to run T2
    load T2 state
```
Thread 1
while (true) {
    sthread_yield()
}

In-kernel thread's viewpoint
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T1
- load T1's state
- return sthread_yield()
- call sthread_yield()
- save state to stack
- save state to TCB
- choose to run T2
- load T2's state
- return sthread_yield()
Multi-threaded kernel, single-threaded processes

<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 their handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting info</td>
<td></td>
</tr>
</tbody>
</table>

In-kernel ready list includes both TCBs and PCBs

Interrupts/exceptions:
- hw & sw cooperate
- but no need to save SP when within kernel

Library calls vs System calls:
- in kernel, use simple procedure call
- in user mode, needs system call to access PCB in kernel

Process creates threads via system call:
- threads PCB in kernel
- stack in user space

User-level Threads

- No OS support
  - TCBs, ready list, finished list, waiting list — in user space
  - thread library calls are just procedure calls!
- Use upcalls to virtualize interrupts and exceptions
  - use system call to register a signal handler
  - on interrupt, save state of process P and run kernel handler:
    - copy P's saved state in signal stack in P's address space
    - load state with PC = &signal_handler; SP -> state on stack
    - signal handler moves state from stack to TCB
    - restores state of some other TCB on ready list

Pros and Cons of User-level Threads

**Pros**

- Better than nothing!
  - use to be only game in town
- More portable
  - Java’s green threads
- Low context switch cost

**Cons**

- OS is unaware of user-level threads
  - can’t use for parallel processing
- can’t use to mask I/O latency
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

---

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>Share code, data, heap and I/O</td>
<td>Have own address space, isolated from other processes</td>
</tr>
<tr>
<td>Have own stack and registers, but no isolation from other threads in the same process</td>
<td>Expensive to create</td>
</tr>
<tr>
<td>Inexpensive to create</td>
<td>Expensive context switching</td>
</tr>
<tr>
<td>Inexpensive context switching</td>
<td></td>
</tr>
</tbody>
</table>

---

Concurrency is great …

```c
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` and `b` at the end of execution?

---

…but can be problematic

```c
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>Thread1: $x = 1$;</th>
<th>Thread2: $x = 2$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially $y = 10$;</td>
<td>Thread1: $x = y + 1$;</td>
</tr>
<tr>
<td>Initially $x = 0$;</td>
<td>Thread1: $x = x + 1$;</td>
</tr>
</tbody>
</table>

Everyone’s a winner (?)

Who wins?
Is a winner guaranteed?
What if they proceed in lockstep?

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
  - even loads and stores on 64 bit machines are not atomic
- 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
    - restore modularity
  - more fundamentally, we need to know what we are talking about when we mention “synchronization” or “coordination”...