Threads
An abstraction for concurrency

Rethinking the process abstraction

The Process, as we know it, serves two key purposes in the OS:
- It defines the granularity at which the OS offers isolation
  - each process defines an address space that identifies what can be touched by the program
- It defines the granularity at which the OS offers scheduling and can express concurrency
  - each process defines a stream of instructions executed sequentially

Thread: a new abstraction for concurrency

- A single-execution stream of instructions that represents a separately schedulable task
  - OS can run, suspend, resume a thread at any time
  - bound to a process (lives in an address space)
  - Finite Progress Axiom: execution proceeds at some unspecified, non-zero speed
- Virtualizes the processor
  - programs run on machine with an infinite number of processors (hint: not true)
- Allows to specify tasks that should be run concurrently...
  - ...and lets us code each task sequentially

Why threads?

- To express a natural program structure
  - updating the screen, fetching new data, receiving user input — different tasks within the same address space
- To exploit multiple processors
  - different threads may be mapped to distinct processors
- To maintain responsiveness
  - splitting commands, spawn threads to do work in the background
- Masking long latency of I/O devices
  - do useful work while waiting
How can they help?

- Consider the following code segment:
  
  ```c
  for (k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k]
  ```

- Is there a missed opportunity here?
  ```c
  thread_create(T1, fn, 0, n/2)
  thread_create(T2, fn, n/2, n)

  fn(l,m) {
    for (k = l; k < m; k++)
      a[k] = b[k] * c[k] + d[k] * e[k]
  }
  ```

How can they help?

- Consider a Web server

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

Create a number of threads, and for each do

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

What did we gain?

Overlapping I/O & Computation

<table>
<thead>
<tr>
<th>Request 1</th>
<th>Request 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1</td>
<td>Thread 2</td>
</tr>
<tr>
<td>get network message (URL) from client</td>
<td>get network message (URL) from client</td>
</tr>
<tr>
<td>get URL from disk</td>
<td>get URL from disk</td>
</tr>
<tr>
<td>(disk access latency)</td>
<td>(disk access latency)</td>
</tr>
<tr>
<td>send data over network</td>
<td>send data over network</td>
</tr>
</tbody>
</table>

Total time is less than Request 1 + Request 2
All you need is Love
(and a stack)

- All threads within a process share
  - heap
  - global/static data
  - libraries
- Each thread has separate
  - program counter
  - registers
  - stack

All threads within a process share
- heap
- global/static data
- libraries
- Each thread has separate
  - program counter
  - registers
  - stack

Implementing the thread abstraction: the state

Shared State
- Heap
- Global Variables
- Code

Per-Thread State
- Stack Control Block (TCB)
  - Stack pointer
  - Other Registers (PC, etc.)
  - Thread metadata (ID, priority, etc.)

Per-Thread State
- Stack Control Block (TCB)
  - Stack pointer
  - Other Registers (PC, etc.)
  - Thread metadata (ID, priority, etc.)

Note: No protection enforced at the thread level!

Processes vs. Threads: Parallel lives
Processes vs. Threads: Parallel lives

Processes
- Have data/code/heap and other segments
- Include at least one thread
- If a process dies, its resources are reclaimed and its threads die
- Interprocess communication via OS and data copying
- Have own address space, isolated from other processes’
- Each process can run on a different processor
- Expensive creation and context switch

Threads
- No data segment or heap
- Needs to live in a process
- More than one can be in a process. First calls main.
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory
- Have own stack and registers, but no isolation from other threads in the same process
- Each thread can run on a different processor
- Inexpensive creation and context switch

A simple API

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void thread_create (thread, func, arg)</td>
<td>Creates a new thread in thread, which will execute function func with arguments arg.</td>
</tr>
<tr>
<td>void thread_yield()</td>
<td>Calling thread gives up processor. Scheduler can resume running this thread at any time</td>
</tr>
<tr>
<td>int thread_join (thread)</td>
<td>Wait for thread to finish, then return the value thread passed to thread_exit. May be called only once for each thread.</td>
</tr>
<tr>
<td>void thread_exit (ret)</td>
<td>Finish caller; store ret in caller’s TCB and wake up any thread that invoked thread_join(caller).</td>
</tr>
</tbody>
</table>

Multithreaded Processing Paradigms

User Space
- Dispatcher/Workers
- Web page cache
- Web requests

Kernel
- Dispatcher/Workers
- Specialists
- Request queues
Multithreaded Processing Paradigms

Threads considered harmful

- Creating a thread or process for each unit of work (e.g., user request) is dangerous
  - High overhead to create & delete thread/process
  - Can exhaust CPU & memory resource
- Thread/process pool controls resource use
  - Allows service to be well conditioned
    - Output rate scales to input rate
    - Excessive demand does not degrade pipeline throughput

Threads Life Cycle

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

Threads Life Cycle

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

TCB: being created
Registers: in TCB
Threads Life Cycle

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

1. **Init**
2. **Ready**
3. **Running**
4. **Waiting**
5. **Finished**

**Thread creation** (e.g., `thread_create()`) is the initial state where a thread is created but not yet ready to run. The thread is then placed in the Ready list.

- **Scheduler resume thread**
- **Thread yields**
- **Scheduler suspends thread** (e.g., `thread_yield()`) moves the thread to the Waiting state.

**Registers**:
- In TCB (or pushed on thread's stack)
- Processor

**TCB**:
- Ready list
- Running list

**Notes**:
- 21
- 22
- 23
- 24
Threads Life Cycle

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

- Thread creation (e.g. `thread_create()`) leads to the Ready state.
- Scheduler resumes a Ready thread.
- Running threads yield to the scheduler, which suspends them (e.g. `thread_yield()`).
- Waiting threads wait for an event (e.g. `thread_join()`).
- Event occurs (e.g. another thread calls `thread_exit()`).
- Finished threads exit (e.g. `thread_exit()`).

TCB: Synchronization variable's waiting list
Registers: TCB

TCB: Running list
Registers: Processor

TCB: Ready list
Registers: in TCB (or pushed on thread's stack)

TCB: Finished list (to pass exit value), then deleted
Registers: TCB
Kernel thread context switches

- Voluntary event
  - via a call to the thread library: thread_yield(), thread_wait(), thread_exit()
- Involuntary event
  - e.g., timer or I/O interrupt; processor exception

Voluntary Kernel thread context switch

- Defer interrupts
- Choose next thread to run from ready list
- Switch!
  - save register and stack of current thread in TCB
  - add current thread to ready list
  - switch to new thread's stack
  - slurp in new thread's state from its TCB
  - change state of new thread to RUNNING
- Enable interrupts

Involuntary Kernel thread context switch

- Save the thread's state in the TCB
  - through a combination of hardware and software
- Run kernel handler
  - can use stack of kernel thread to push variables used by handler
- Restore next ready thread

Single-threaded processes + kernel threads

- Each kernel thread has its own TCB and its own stack.

  Each user process has a stack at user-level for executing user code and a kernel interrupt stack for executing interrupts and system calls.
Multi-threaded processes: kernel threads

Each user-level thread has a user-level stack and an interrupt stack in the kernel for executing interrupts and system calls.

User-level Threads

Motivation
- Threads are a useful programming abstraction
- Calling OS to manage threads is expensive.
- Implement thread creation/scheduling using procedure calls to a user-level library rather than system calls

User-level threads
- User-level library implementations of thread_create(), thread_yield(), etc.
- UL library performs same set of actions as corresponding system calls, but thread management is controlled by user-level library
- What happens if a user-level thread makes a system call?

User-level Threads: Pros and Cons

Benefits:
- Small context for switching between threads of a process
- Thread scheduling is more flexible
  - Can use application-specific scheduling policy
  - Each process can use a different scheduling algorithm
  - Threads voluntarily give up CPU

Drawbacks:
- OS is unaware of the existence of user-level threads
  - Poor scheduling decisions
  - If a user-level thread waits for I/O - entire process waits
- OS schedules processes independent of number of threads within a process

Can we do better?

Why not a user level thread scheduler that spawns a kernel thread for blocking operations?

Forget spawning, use a pool of kernel threads!

But how do we know if an operation will block?

read might block, or data might be in page cache.

Any memory reference might cause a page fault to disk!
Scheduler Activations
(best of both worlds)

- Kernel assigns to process k “virtual processors” (initially, k=1), implemented as kernel threads.

- Kernel notifies (activates) via an upcall the user-level thread scheduler for any kernel event that might affect user-level threads.
  - e.g., if a thread calls a blocking system call, kernel notifies user-level scheduler to schedule a different thread.

- Kernel notifies user-level scheduler whenever it adds or reclaims a virtual processor assigned to the process.