CSE 410: Systems Programming
Concurrency

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo
Logical Control Flows

The text defines a logical control flow as:

[A] series of program counter values that [correspond] exclusively to instructions contained in [a program’s] executable object file or in shared objects linked to [it] dynamically at run time.

The system provides each program with the illusion that its logical control flow runs on a dedicated computer.
Concurrency

Concurrency is when more than one logical control flow is present in the system at the same time.

Concurrent flows are logical control flows whose execution overlap in time.

Concurrent flows can be present even with only one processor.

Multiple flows can coexist on one processor via multitasking.

Multitasking time slices between multiple logical control flows.

- Each flow runs for a brief period of time, then is interrupted
- A context switch changes control to another flow
- The new flow runs for a brief period of time (repeat)
Multitasking and Multiprocessing

Multitasking

A

B

C

Time

Multiprocessing

X

Y

Z
Multitasking

Concurrent flows in a *multitasking* environment do not execute simultaneously.

However, from the point of view of any given flow, other flows are making progress while it executes.

Consider:

- Process A is executing at PC location L
- A **context switch** occurs, removing A from the CPU and switching to Process B
- Process B does something
- A context switch occurs, switching to Process A at location L

Process A will observe progress in Process B before and after L.
Concurrent flows may be related or unrelated in:

- Design
- Implementation
- Memory space
- Resource requirements
- Timing requirements
- ...

When concurrent flows are completely unrelated, the dedicated computer abstraction provided by modern systems is both mostly complete and very appropriate.

When concurrent flows are more related, it gets more complicated.
Motivation for Concurrency

There are many reasons to use concurrent flows:

- Making computational progress while \textit{blocked} on a slow device
- Achieving \textit{rapid response} to a particular condition (e.g., human input, external event)
- Utilizing \textit{multiple physical processors}
- …

In addition, \textit{simply taking advantage of the dedicated computer model} to simplify design and implementation.
Processes

We have already seen process-level concurrency.

Multiple processes may:

- Proceed independently on unrelated tasks
- Proceed independently on related tasks
- Cooperate on tasks
Independent, unrelated tasks are things like:

- Your windowing environment versus a terminal session
- A code editor and a music player

These tasks need not be aware of each other, and fit the dedicated computer model very nicely.
Independent, Related Tasks

Independent, related tasks might be:
- A shell pipeline
- Make and the compiler

These are programs that may or may not have been designed together, but are doing related work within the dedicated computer model.
Cooperating Tasks

Cooperating tasks could be:

- The child of your shell before `exec()`
- Forked processes handling individual clients or blocking tasks

These processes work closely together and may use the dedicated computer model for isolation, but are closely aware of each other.
Designing for Multiple Processes

A multi-process design can be robust and reliable.

The isolation in memory and resources provided by the system protects processes from certain faults in their neighbors.

Communication and cooperation can be expensive, though:

- Separate memory spaces protect, but also divide
- Many IPC mechanisms require system calls
Interprocess Communication

We have seen some IPC methods:

- Process exit status
- Pipes
- Signals
- Files

Others are available:

- Sockets (network communication)
- Shared memory
- Message queues
- Semaphores
- Condition variables
- …
Threads

Threads provide a conceptually similar abstraction to processes. Threads also represent a logical control flow.

However:

- One process may have multiple threads
- Two threads within one process are much less isolated than two processes, or threads in different processes

In particular, threads within a process share a memory map.
# Threads vs. Processes

<table>
<thead>
<tr>
<th>Processes</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>P3</td>
</tr>
<tr>
<td>Kernel</td>
<td>Kernel</td>
</tr>
<tr>
<td>Stack</td>
<td>T1 Stack</td>
</tr>
<tr>
<td>Heap</td>
<td>Heap</td>
</tr>
<tr>
<td>BSS</td>
<td>BSS</td>
</tr>
<tr>
<td>Data</td>
<td>Data</td>
</tr>
<tr>
<td>Text</td>
<td>Text</td>
</tr>
</tbody>
</table>

© 2018 Ethan Blanton / CSE 410: Systems Programming
Threading Advantages

Threads are **much cheaper** than processes:

- They share memory maps
- They share permissions and kernel resources
- Context switches between two threads in the same process are much less involved than between processes

Inter-thread communication is **trivial**, due to shared memory.
Concurrency access to shared resources is very tricky.

Many established APIs are not thread-safe.

Breaking down the dedicated computer model makes reasoning about process behavior harder.
Threading Use Cases

Threading is often appropriate for tasks which require:

- Very rapid change of control between parallel tasks
- Lots of large, shared data structures
- Blocking operations that do not inhibit other progress
- More rapid computation than can be performed on a single CPU

Multiple processes may also solve some of these problems.

The costs of threading must be weighed against its advantages on a case-by-case basis.
Inter-Thread Communication

Because threads share memory, they can easily share state.

Nonetheless, there are inter-thread communication mechanisms:
- Pipes
- Message queues
- Signals
- Semaphores

Where these overlap with IPC mechanisms, they:
- Sometimes have special concerns
- May require separate APIs

E.g., kill() versus pthread_kill() for inter-thread signals.
Summary

- **Concurrent flows** appear to run simultaneously.
- **Multitasking** is switching between concurrent flows on one processor.
- **Concurrency** makes some things simpler, but has special concerns.
- Both **processes** and **threads** provide abstractions for concurrent flows.
- **Threads** are cheaper but less isolated than processes.
References I

Required Readings

License

Copyright 2018 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.