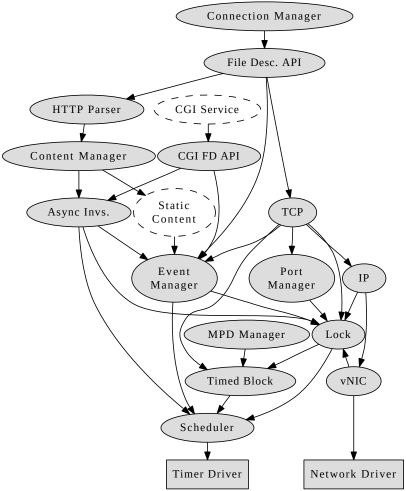
The Case for Thread Migration: Predictable IPC in a Customizable and Reliable OS

Gabriel Parmer

The George Washington University
Computer Science Department
gparmer@gwu.edu

Component-Based OSs and μ-kernels



Configurability in RT/Embedded systems

- scheduling policies
- resource sharing protocols
- interrupt scheduling

Reliability in RT/Embedded Systems

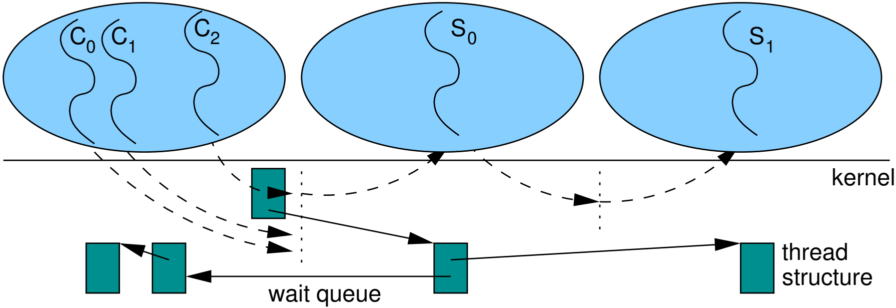
- limit scope of failures

System policies/abstractions are *components*

- User-level, separate protection domains

IPC implementation key for system performance and **predictability**

Synchronous IPC Between Threads



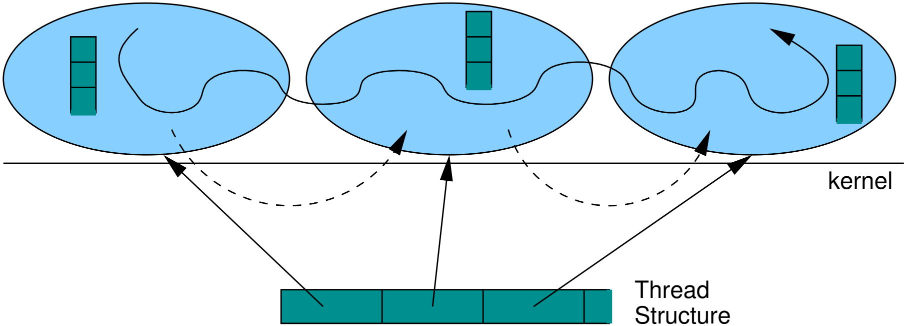
Communication operations:

- Clients: send + wait_reply = *call*
- Servers: reply + wait_msg = *reply_wait*

Threads

- bound to a component
- separate scheduling parameters
- communication end-points

Thread Migration



Invocation/procedure call semantics for communication

- *Scheduling context* migrates between components
- Separate *execution context* in each component
- Kernel thread structure tracks invocations

Synchronous IPC and Thread Migration

Functionally identical

- interface definition language (IDL)

L4 and Composite performance

- Similar published overheads
- $> 2/3$ invocation cost is hardware overhead in Composite

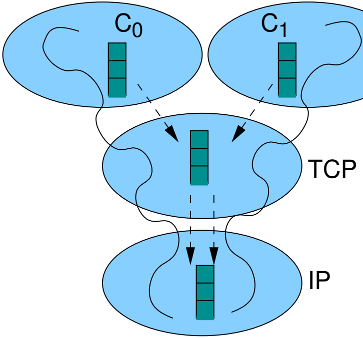
Thread Migration vs. Synchronous IPC

Qualitative comparison wrt **predictability**

- worst-case overheads, relevant parameters
- implications for system design

1. CPU Accounting and Scheduling
2. Communication End-point Contention
3. System Configurability

CPU Accounting, Scheduling: Thread Migration



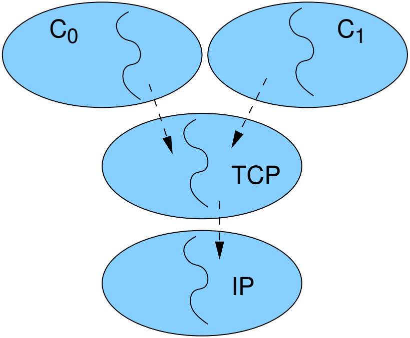
Accounting and Scheduling

- execution charged to, and scheduled wrt the client thread

Resources within each component must avoid *priority inversion*

- component-based locking policies

CPU Accounting, Scheduling: Sync. IPC



1. Accounting
   - execution time not accounted to clients
   - bandwidth servers?
2. Scheduling
   - threads have independent priorities
   - possible priority inversion

CPU Accounting, Scheduling: L4 Implementations

Lazy scheduling, direct switch

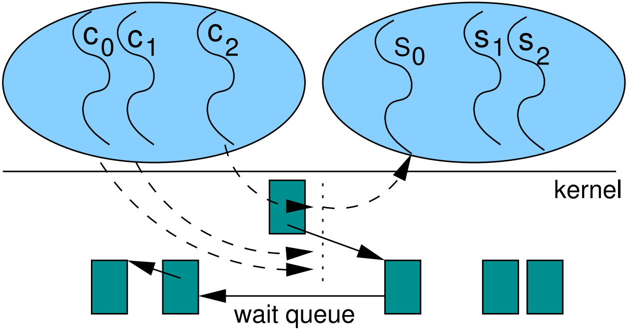
- don't call scheduler, and directly switch to server
- optimizations for performance
- unpredictable scheduling and accounting

Credo

- decouple scheduling context from execution context
- server runs with scheduling ctxt of client thread
- tracking correct scheduling context
  - O(n) in depth of invocations

Sync IPC $\rightarrow$ thread migration

Communication End Points: Sync. IPC

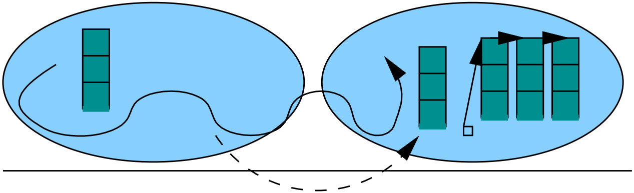


IPC worst-case execution-time?

- O(length of wait queue)

Client decides thread to invoke, server/kernel know which are available

Communication End Points: Thread Migration



The target of invocation is the component

- server contains code to locate execution context
- specialized policies
  - service differentiation
  - allocate new execution contexts

Configurability for Real-Time/Embedded Systems

- temporal policies
  - scheduling
  - synchronization
- reliability vs. predictability

User-Level Component-Based Scheduling

COMPOSITE supports component-based scheduling

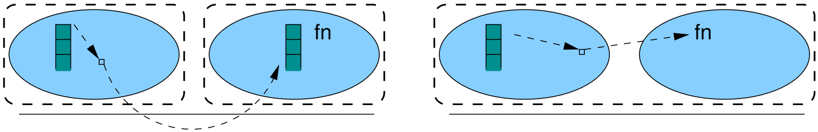
- no kernel scheduler, only dispatching

Synchronous IPC requires scheduler interaction

- switch between threads
- activation of multiple threads

Can double (or triple) invocation cost

Mutable Protection Domains



Dynamically trade-off fault-isolation for performance

Raise and lower protection domain boundaries between components

- remove boundaries where communication overheads are significant
- raise boundaries when “hot-paths” change

Overhead of thread dispatching >> overhead of function call

Conclusions

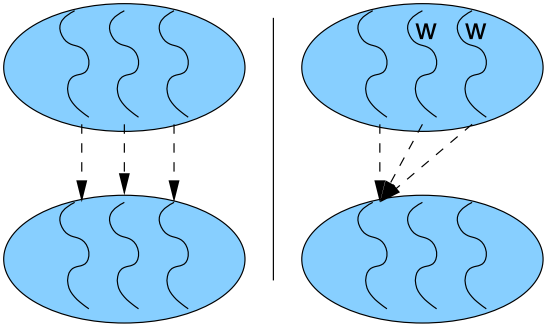
Summary Chart:

	Accounting/ Scheduling	Communication End Point Contention	Customiz - ability
Sync. IPC:			
pure	×	×	✓
ls/ds	××	×	✓
credo	✓*	×	✓
Thd Migration	✓	✓*	✓✓

Qualitative assessment of the predictability of the two models

- Moving sync. IPC implementations toward migrating thread increases predictability
- Migrating thread model good fit for predictable, reliable, configurable systems

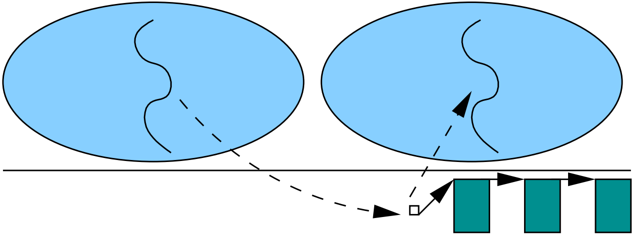
? || /* */



Execution contexts are the target of invocation

- client decides which to invoke
- kernel/server know which are busy

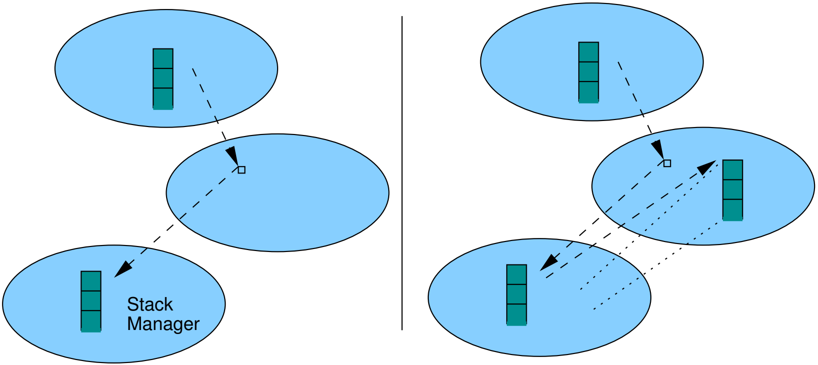
Communication End Points: Sync IPC III



Synchronous IPC addressing the component

- kernel manages server threads

Synchronous IPC → Thread Migration



Upon "stack miss", invoke stack manager component

- allocate new stack (shown)
- priority inheritance
  - when a thread using a stack is done, it returns it to be used by the requesting thread
- QoS-aware stack allocation