ADMINISTRIVIA

Problem Set 1, due 11/9 in class

Reading for Wednesday:

- FLASH: K+94
- Alewife: KA93
- Tempest & Typhoon: R+94

OUTLINE

Review

Deadlock

Protocol Layering

APPLICATION-LEVEL FLOW CONTROL

If possible, use a sequence of permutations

- Use barriers to keep permutations separate (3x)

For unknown patterns, use packet interleaving:

- Makes even use of the network resources (2x)
- Avoids head-of-line blocking

Bandwidth matching pays off for bulk data movement:

- Minimizes congestion for a given throughput (25-200%)
- Claim: can handle bisection limitations cleanly

Keep phases separate with barriers:

- Can improve performance and rarely hurts
- Improves predictability and robustness

HETEROGENEOUS NODES

Problem: reformat data during transfers

- Ex 1: big vs. little endian (MIPS vs. SPARC)
- Ex 2: IEEE floating point vs. Cray floating point
- Ex 3: ASCII data: newlines, special characters

Presentation layer converts to/from canonical form

- 30-97% of the overhead for TCP
- Requires non-trivial transformation of *all* data

Right solution for MPP (TCP?):

- Avoid canonical form
- Establish end-to-end transformation at setup
- *Sender* should transform data

Application-Level Deadlock

Network is deadlock free *provided that nodes accept packets*

Scenario 1: Barriers between permutations

- A waits at a barrier for B
- B waits trying to inject a message
- Network waits for A to remove a packet

This actually happens with CMMD

Solution:
A must accept packets while waiting for the barrier

Same for all control-network operations

Fetch Deadlock

Scenario 2: *Fetch Deadlock*
All nodes performing RPC

- A waits trying to inject
  ⇒ A must accept packets to avoid deadlock
- B waits trying to inject
- A and B eventually run out of memory
  ⇒ can't accept packets ⇒ deadlock

Solution 1: can't send unless receiver has space

- Requires window flow control for all pairs
- Expensive software overhead: counting + acks

Fetch Deadlock

Solution 2: *request* & *reply* networks (CM-5)

- Key: replies can always be handled (don't use memory)
- Use two *independent* networks: requests and replies
- If you can't inject a reply, then stop accepting *requests*
  ⇒ can't run out of memory
- Still accepting replies ⇒ reply network empties
  ⇒ current request completes
  ⇒ all requests eventually accepted
  ⇒ no deadlock

Problems:

- Must poll two locations (one for each network)
- How do you stop accepting packets with interrupts
- Network complexity... networks must be independent!

Problems...

Requests can't generate requests, only replies...

- A waits trying to inject a request
- A handles next packet, which is also a request
- Request generates a new request ⇒ must wait
  ⇒ ties up memory ⇒ eventual deadlock

Key difference:

- Requests tie up new memory (stack space)
- Replies use existing memory

Three classes:

- Replies (r)
- Requests that only generate replies (Rr)
- Requests that generate requests (RR)

Request-Request Solution

Key idea: requests can never block on requests

1. When injecting a new RR, reserve space for the *generated* request
2. When handling an RR and the request injection stalls, send the request over the *reply* network back to the sender
3. Queue up requests that arrive on the reply network using the *preallocated* storage
4. When the request network becomes free, send the queued-up requests to their real destination

Optimizations

Can always send replies on both sides

When can you accept requests?

- Anytime the reply network is free
- Anytime you provably have sufficient space
  - Tricky with implicit stack allocation
  - Easy with heap allocation (count free frames)

One-way traffic equivalent to replies:

- All block transfer packets ⇒ use both sides
- Any non-blocking active message (to any node)

Application-Level Framing

Idea: application-level unit is the basis for all steps

- Unit of error detection, encryption
- Unit of retransmission
- Unit of out-of-order delivery
- Unit of format transformations

Key aspect: ADUs can be handled out of order

- Example: individual block-transfer packets

Application handles errors:

- video/voice packets ⇒ just ignore
- AAL5: lose one cell ⇒ retransmit entire packet
- If transport layer detects partial drop ⇒ drop rest of ADU
- Avoids unnecessary retransmission in lower layers

Application-Level Framing

Format transformations:

- Sender/Receiver share common naming scheme
  - Example: offsets into larger block transfer
- Receiver can transform out-of-order (since destination is known)
- Can move transformation to sender side (sometimes)

Flow control:

- ADUs are also unit of flow control
- Sender-side transformation can exploit delay cycles
- Claim: better performance
  - Perform entire ADU handling together
  - *Then* delay for flow control
  - Fewer context switches, better cache behavior

Integrated Layer Processing

Goal: [CT90]
perform all data manipulation in 1 or 2 tight loops

Traditional structure: separate loop for each layer
⇒ read/write data for every layer

Loop 1: demultiplex/reorder & error detection

- This is done in hardware for user-level segments
- Done by handlers for block-transfer packets
- Requires application-level framing

Loop 2: decryption/decompression & format conversion

- Optional and application specific (MPEG)
- Decryption in the first loop requires trust (Clipper)

Micro-Protocols

Build protocols out of many little blocks (and layers):

- Allows optimized protocols (even dynamically)
- Simplifies debugging and reuse of blocks
- Very flexible

Example block: “BLAST”

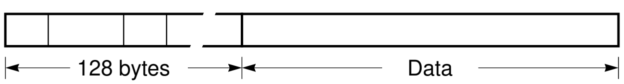
- Fragments blocks into packets at sender
- Rebuilds block at receiver

Problems:

- Too many layers ⇒ hefty performance hit (?)

Micro-Protocols

Message format: header-stack plus data:



- Separation supports application-level framing
- Fixed offset for data speeds manipulation, simplifies blocks

Headers form a stack:

- First header is “address” of handler
- Sequence of headers specifies the micro protocols
- Headers may have variable size:
  Checksum header: address + header CRC + data CRC

Micro-Protocol Performance

Metric	Sprite	x-Kernel		
Layers	2	2	7	10
Protocol blocks	2	2	14	39
Latency (msec)	2.6	1.8	1.8	2.4
KB/sec	700	860	840	820

⇒ Layering itself must not be a problem

Implementation

1. Process per message, not per protocol
   - ◆ A process for each layer requires expensive context switches
   - ◆ Can completely process message without a context switch (assuming no blocking for resources)
2. Process can migrate from kernel to user space
   - ◆ Process starts in kernel, then shepherds message into the application space
   - ◆ Process returns to pool after handling the message
3. Fast dispatch: exploits temporal locality
4. Lazy Messages...

Lazy Messages

Instead of manipulating data, just track list of functions

Example:

- ◆ $f(x)$ decrypts one word at a time
- ◆ $g(x)$ copies one word to application buffer
- ◆ $\Rightarrow g(f(x))$ copies and decrypts
- ◆ Use 1 loop instead of 2

Manipulation is lazy: postpone as long as possible

- ◆ Integrated layer processing
- ◆ Allows compiler optimizations: (?)
  - Some protocols cancel (endian conversion)
  - Move copies to end and eliminate extra copies

Extending the Kernel

Q: How can we add protocols to the kernel without recompiling and without forfeiting safety?

A: software fault isolation

Idea: convert arbitrary code to safe code

- ◆ Transform object code into safe object code
- ◆ Kernel can call safe object code
- ◆ Can be done dynamically

Implementation:

- ◆ *sandbox* all reads and writes: add explicit range checks to every read and write (can only play in the sandbox)
- ◆ Dynamically check all indirect jumps