CSE221 Lecture 7

Aronya Baksy

October 2024

1 The Distributed V Kernel and its performance for diskless workstations

1.1 Claims

- Diskless workstations perform similarly to local disks
- General purpose IPC in V is sufficient against any specialized remote file access protocol

1.2 IPC in V

- **Pros** of diskless workstations:
  - Lower h/w and maintenance costs
  - Lower mem overhead for file processing per workstation
  - Fewer problems with replication/consistency of data
- **Con**: doing file access over network
- file access is, built on top of a general-purpose IPC facility
- Use synchronous message passing for all data transfer instead of streaming for file access (easy to implement but performance?)
- basic model of V kernel: multiple small processes communication via message passing:
  - **Synchronous** msg passing (slow/inefficient but easier buffer mgmt, less concurrency and familiar programming model to local calls)
  - Different facilities for data transfer and IPC (32-byte control messages)

1.3 Kernel Implementation

- RPCs are implemented directly in the kernel (i.e. there is no separate server that sends/receives all kernel calls)
- IPC uses ethernet (link) layer directly (using network layer protocols like IP adds overhead for checksum calc etc.)
- Implemented reliable transmission on an unreliable n/w without an extra layer (using a Send $\longrightarrow$ Receive $\longrightarrow$ Reply protocol)
- Process identifiers contain host information to quickly distinguish local/remote proc calls
- `MoveTo`/`MoveFrom` calls for data transfer do not need per-packet acknowledgement
- **getPid** uses network broadcast (this is a new idea)

1.4 Remote Message Implementation

- `Send` wraps around `NonLocalSend` which writes to network for remote processes
- Message receivers create *alien* process descriptors as placeholders for the sender
- The alien process saves all replies for some time (in addition to the actual original sender)
- There are timeouts and retry mechanisms to handle dropped packets
- remote kernel can respond with packet, or a reply-pending packet in the case no descriptors are available.

1.5 Remote Data Transfer

- `MoveTo`: transmits data to destination as a sequence of max-sized datagrams and waits for a single ack
- Retransmission only from the last received data packet (instead of full retransmission)
- No queues or buffers, direct from dest addr space to n/w to receiver address space
- V uses **pre-copying** for process migration, where the process continues executing while its memory is transferred

1.6 Remote Segment Access

- read page with Send–Receive–`ReplyWithSegment`
- write page with Send–`ReceiveWithSegment`–Reply
- The additional `ReplyWithSegment` and `ReceiveWithSegment` primitives cut down the number of messages compared to using `MoveTo/MoveFrom` (from 4 to 2)
- Goes against second claim because file access requires specialized primitives instead of general IPC

1.7 Evaluation: Performance Measurements

1.7.1 Network Time

- Using raw ethernet instead of using TCP/IP on top of ethernet
- Network penalty: minimum time to transfer datagram from one workstation to another on an idle network assuming no errors
  - Does not account for retransmissions and error handling
- Includes processor time to transfer the datagram, time to transmit, does not include time spent in user space

1.7.2 Kernel Performance

- Main goal is to contrast local IPC and network IPC
- Measure 1000 times, subtract loop overhead and average measurements
- Remote adds around 2 ms of overhead

1.7.3 File Access

- Streaming behaviour:
  - prefetching file pages (read-ahead) and asynchronously storing modified pages (write-behind) uses streaming
  - **no streaming** in the network IPC to deal with network latency
- Arguments for no network IPC:
  - If I/O bound program: main bottleneck is disk
  - If not I/O bound: not much improvement in optimizing disk access if disk access are less

1.7.4 Program Loading

- Using `MoveTo` and `MoveFrom`
- These use **streaming** to transfer pages

2 The Sprite Network Operating System

2.1 Technology Trends

- Moving away from large time-sharing systems to personalized workstations over a network (using RPC)
- Larger physical memory allows for more computation and less time spent on I/O (caching on server and client)
- Multiprocessor workstations with OS that can take advantage of multiple processors

2.2 Design Goals

- Similar syscalls to Berkeley UNIX 4.3
- Additional facilities for resource sharing:
  - transparent network FS
  - writable shared memory b/w processes on a single workstation
  - mechanism to migrate processes between workstations (take advantage of idle workstations)
- `Proc_Fork` allows forked process to share the parent's data segment (not stack) optionally

2.2.1 Kernel Implementation Features

- Facility for RPCs
- user views the FS as a single tree even though it is a collection of domains (prefix table is used to manage the domain name space)
- Use of caching on server and client machines, take advantage of large physical memories, with cache coherency
- Virtual memory uses ordinary files as a backing store for swapping instead of dedicated swap memory
- The kernel is **multi-threaded** (> efficiency on multiprocessors)
  - Contains multiple locks for protecting individual data/code (instead of a global lock)

2.3 Physical memory allocation

- Two main uses: virtual address space, and file buffer caches
- Dynamic tradeoff between file buffer cache and virtmem (based on usage)

2.4 Process Migration

- Make efficient use of idle machines
- Using the `Proc_Migrate` syscall
- Processes sharing heap memory must migrate together
- Either manual (shell commands) or automated (tools like PMake for compiling in a distributed way)
- **Transparency** to the user and the process being migrated is important
- Sprite using backing files to transfer memory state instead of transferring memory state by freezing and unfreezing
- Each process has a *home machine* which responds to location-dependent kernel calls, maintains transparency even after migration

2.5 RPC Mechanism

- Client stub copies args to a request and captures return values in the response message
- Server stub moves args from request message to the requested procedure, and packages return values in the response
- Stubs are hand-generated instead of generated by compiler
- Each request/reponse is an *implicit acknowledgement* of the previous message

2.6 Namespace Management using Prefix Tables

- the namespace as a set of prefix tables, and each entry in the table corresponds to a domain
- manage data in a way to provide high performance while not compromising on simplicity of sharing files
- imagine fs as a single tree, but there can be subtrees in different domains
- domain represents another server
- To locate a file
  - Match the name against all entries of the prefix table, choose the entry with the longest prefix
- open directory and store a token and server address with the process to make lookups on relative entries quicker
- there is a "root" server for the fs
- prefix approach bypasses looking up on root server, so it reduces the maximal load
- prefix table entries are like hints - can be updated and adapted on server crashes or errors.

2.7 File Caching

- every server has a large cache of recently accessed file blocks
- caches organized on block (block size of 4Kb) rather than whole file to store in main memory.
- Reads always check the local cache first, then try to read the block
- Caches use LRU replacement strategy
- writes are handled via **delayed writes**: write block to cache first, then return to the application and a block is not written to disk or server until ejected from cache or 30 seconds elapsed.
- Sprite guarantees full consistency with version numbers:
  - when opening a file, server returns current version for the file
  - Keep track of last writer to identify location of an updated file
  - concurrent writes disable caching
- Pattern 1: **sequential write-sharing**: handled through version numbers
- Server forces last writer to flush to disk everytime the file is opened by a client different from the last writer
- Pattern 2: **concurrent write-sharing**: server flushes dirty blocks back from the current writer and notifies all of the clients with the file open that they should not cache the file anymore
- very slow file access when caching is disabled

2.8 Summary: main controbitions

- File caching protocols
- **dynamic balance between VM and file buffer cache**