##############  
##############  
##############  
01\_07\_01\_the\_multiprogramming

The Structure of the "THE"- Multiprogramming System

========================================================================

Early systems

OPERATOR TAKES 1 PROGRAM

Problems:

CPU time is expensive. CPU is forced to run at speed of peripheral

Only one program can run at a time. Long wait for users to get program output

FEEDS IT TO THE COMPUTER

WAITING FOR THEIR TURN

Computer processes the input

OUTPUT

Operator collects Output

The structure of the the multiprogramming system

========================================================================

Goal:

Design a system to process continuous flow of user programs

Design objectives:

Reduce turnaround-time for short programs

Economic use of peripherals

Automatic control of backing store combined with economic use of CPU

Support applications that only require the flexibility of general purpose processor

========================================================================

Challenges & Solutions:

Ease of System design and validation

Layered System design

Resource management among programs

Abstraction of HW at each layer

Race conditions

Semaphores

Short Response-time

Priority Scheduling

Fairness of Scheduling

Real-time Clock interrupts

========================================================================

Tool:

EL X8 (N.V. Electrologica, Rijswijk)

Core Memory 32K (=> Present day Main memory)

Drum 512K words (=> Present day Hard Disk Drive)

Indirect addressing for stack implementation

Sound mechanism to handle I/O and interrupts

Peripherals: Tape readers, punches, Teleprinter …

System Software:

OS Programmed in Assembly instructions

User programs written in ALGOL 60

========================================================================

Processor Allocation:

In a single sequential process, correctness of program is dependent on sequence of event execution

Speed of execution doesn't affect correctness

In THE, whole system is designed as a harmonious society of cooperating sequential processes progressing at undefined speeds

What constitutes a process in ‘THE'?

Each user program

Each input peripheral & Each output peripheral

Segment controller & message interpreter

Virtualization of CPU is achieved

Processes cooperate with each other through mutual synchronization

Semaphores

Number of cooperating processes is independent of number of actual physical processors in the system

========================================================================

Storage Allocation:

‘THE' implements automatic control of secondary storage

Memory Units: Smallest unit is called Page

Core Pages: Pages in main memory

Drum Pages: Pages in drum (hard disk)

Information unit: Segments (> Level 1 perspective)

One segment fits in one page (Virtual Page)

Number of Virtual Pages is much greater than physical pages (core and drum)

Segment identifier contains a segment variable in core

Segment variable provides the page number where the segment can be found

Consequences:

When core page is dumped to drum, the page need not be returned to the same drum page from which it came

Free drum page with least latency time is selected

Program pages need not be stored in consecutive drum pages

Total abstraction of memory locations from programs

Thus, primitive version of Virtual memory management was achieved

Layer 4

User process requests Memory access;

Provides Segment identifier

Layer 3

Control transferred to Layer 1

Layer 2

Layer 1

Process reads SegVar to identify page by reading Core Table Entry.

Layer 0

CORE MEMORY (512 WORDS)

If Page present in core, segment provided to user program;

Else, "Segment Controller" initiates transfer of page from drum to core.

Drum issues interrupt when transfer is complete. Segment Controller returns data to program.

DRUM MEMORY (512K WORDS)

========================================================================

System Hierarchy:

‘THE' system follows strict hierarchy

Broken into levels from 0 to 5

Higher level can talk only to level below

Communication happens in one direction only

Advantages:

Each layer can be validated individually for correctness before implementing next layer

Abstraction of hardware information from layer above

========================================================================

‘THE' System Structure

LEVEL 0

Tasks:

CPU allocation to processes

Real-time clock interrupt scheduling (Timer ticks)

Priority scheduling for quick response to the system

Synchronization among processes

Provides CPU Abstraction to higher layers

LEVEL 1

Tasks:

Perform memory accesses

Synchronizes drum interrupt and processes on higher levels

Manage segment information

Provides Abstraction of memory to higher layers

LEVEL 2

Tasks:

Message interpreter

Allocation of console Keyboard to processes

Processes above Level 2 think they have exclusive access to console

Provides Abstraction of console to higher layers

LEVEL 3

Tasks:

Buffering input streams

Unbuffering output streams

Manages all "logical communication units"

Provides Abstraction of peripherals to higher layers

LEVEL 4

Independent User programs run at this layer

Programs written in ALGOL 60

LEVEL 5

End- User

========================================================================

Synchronization:

Processes are synchronized via "Semaphores"

Semaphore initialized as special integer variables before they can be used by processes

2 Operations on semaphores: P (wait) and V (signal)

It was observed that semaphore was used in 2 ways:

Mutual exclusion (Semaphore initialized with value 1)

Private Semaphore (Semaphore initialized with value 0)

========================================================================

Validation:

Two kinds of validation:

Logical soundness before implementation

Harmonious Cooperation of processes

Thorough verification of each layer

Layer by Layer testing of software

========================================================================

Harmonious co-operation:

Sequential process in the system can be regarded as "cyclic process"

All processes when at rest are at a neutral point – "Homing position"

Harmonious cooperation is proved in 3 stages:

Process performing task can only generate finite tasks for other processes

In ‘THE', higher layer can only generate task for lower layers

All processes cant be in "homing position" if there is a pending task in system

After acceptance of initial task, all processes eventually return to homing position.

"Circular waits" has to be prevented from occurring ;

Else "Deadly Embrace" can result

========================================================================

Software Testing:

Validation of system was done in layers

Starting at Level 0 , each layer was thoroughly tested

Next layer is added after previous layer's been validated completely

Test structure forces the system into all different relevant states and verified if the system performs according to specification

Hardware failures couldn't restrict system validation due to hierarchical design of system

========================================================================

Summary:

Layered OS design helped in abstraction of hardware resources to programs

Level 0: Abstracted processor

Level 1: Abstracted Memory through segments

Level 2: Abstracted console

Level 3: Abstracted peripherals

Sequential processes collaborated with each other "harmoniously" through Synchronization

Mutual Exclusions and Private Semaphores

SW memory segmentation achieved optimal use of core and drum memory##############  
##############  
##############  
01\_07\_02\_nucleus

The Nucleus of a Multiprogramming System

Introduction:

The main problem in the design of a multiprogramming system is to supply a system nucleus that can be extended with new operating systems in an orderly manner.

The purpose of the system nucleus is to implement these fundamental concepts: simulation of processes; communication among processes; creation, control, and removal of proesses.

Processes:

internal process: the execution of one or more interruptable programs in a given storage area. An internal process is identified by a unique process name.

program: a collection of instructions describing a computational process, whereas an internal process is the execution of these instructions in a given storage area.

peripheral device: an item of hardware connected to the data channel and identified by a device number.

document: a collection of data stored on a physical medium.

external process: the input/output of a given document identified by a unique process name.

multiprogramming and communication between internal and external processes are coordinated by the system nucleus -- an interrupt response program with complete control of input/output, storage protection, and the interrupt system.

Process Communication:

the semaphore concept alone does not fultill our requirements of safety and efficiency in a dynamic environment in which some processes may turn out to be black sheep and break the rules of the game. Instead, message buffering within the system nucleus as the basic means of process communication. The system nucleus administers a common pool of message buffers and a message queue for each process.

Advantages:

The multiprogramming system is dynamic in the sense that processes can appeaar and disappear at any time. Therefore a process does not in general have a complete knowledge of the existence of other processes.

Once a communication has been established between two processes, they need a common identification of it in order to agree on when it is terminated. It enables two processes to exchange more than one message at a time (regard the selection of a buffer as the creation of an identification of a conversation).

Prepared for the occurrence of erroneous or malicious processes in the system. The system nucleus ensures that no process can interfere with a conversation between two other processes by storing the identity of the sender and receiver in each buffer and checking it whenever a process attepts to send or wait for an answer in a given buffer.

Efficiency is obtained by the queueing of buffers.

Drawback of message buffering:

resource problem, since the common pool contains a finite number of buffers.

External Processes:

For each kind of external process, the system nucleus contains a piece of code that interprets a message from an internal process and initiates input/output using a storage area specified in the message. When input/output is terminated by an interrupt, the nucleus generates an answer to the internal process with information about actual block size and possible error conditions.

External processes are created on request from internal processes.

Internal Processes:

Internal processes are created on request from other internal processes. The storage area must be within the parent's own area.

The difference between them is merely a matter of processing capability.

Process Hierarchy:

parent processes have complete control over child processes.

The family tree of processes can be extended to any level, subject only to a limitation of the total number of processes.

Strategy:

each process has the power to control the scheduling and resource allocation of its children.

Only rule: a process can only allocate a subset of its own resources (including storage and message buffers) to its children; a process can only start, stop, and remove its own children (including their descendants).

After removal of a process, its resources are returned to the parent process.

Characteristics of the system:

1. New OS can be implemented as other programs without modification of the system nucleus. It is possible to write OS in high-level languages.

2. OS can be replaced dynamically.

3. Standard programs and user programs can be executed under different OS without modification. ##############  
##############  
##############  
01\_12\_01\_tenex

TENEX, a Paged Time Sharing System for PDP-10

time sharing system

It was developed because no existing system of the appropriate size and cost could meet the requirements of the research projects at BBN.

Design Goal:

1. State of the art virtual machine.

2. Good human engineering throughout system.

3. The system must be implementable, maintainable, and modifiable.

The BBN Pager:

An interface between the PDP-10 processor and the memory bus. It provides individual mapping (relocation) of each page of both user and monitor address spaces using separate maps for each. The pager uses "associative registers" and core memory tables to store the mapping information.

On each memory request from the processor, the 9 high-order bits of the address and the request-type level are compared in parallel with the contents of each associative register.

"copy-on-write" facility allows users to share large portions of an address space containing procedures and data, and to obtain private copies of only those pages which are changed. (implemented through an independent per-page status bit)

The pager maintains a record of the activity of the pages in core memory in a "core status" table.

Processor Modifications:

JSYS: a new system call instruction, provides an independent transfer mechanism into the monitor.

A context, either user or monitor, for each instruction execution.

The TENEX Virtual Machine:

The contents of the virtual memory are specified by the virtual memory map which the user may read or write via monitor calls.

A private page is automatically created whenever a process makes a reference to a page for which the map slot is empty.

Job: a set of one or more hierarchically related processes.

Heirarchy:

A process always has exactly one immediately superior process and may have one or more inferior processes.

Two processes are said to be parallel if they have the same immediate superior.

A process can communicate with other members of the structure by (a) sharing memory (b) direct control or (c) pseudo interrupts.

Pseudo Interrupt:

an asynchronous signal

User Interaction with TENEX:

Terminal service module designed to:

provide any type of interactive behavior a program might find useful;

benefit by having many short interactions with the user;

the user can in fact anticipate the machine's responses and begin typing input before output is completed.

TENEX File System:

provide symbolic file name management:

translation of a symbolic name into an internal "file descriptor block" pointer associated with that name;

checking information.

file descriptor: device name, directory name, file name, extension, version number.

Any of the fields of a file description may be abbreviated except for device and version.

Access to a file depends on two things:

the kind of access desired;

the relation of the program making the access to the owner of the file.

Five kind of access: directory listing, read, write, execute, append.

The Monitor:

TENEX scheduler is designed to meet a set of potentially conflicting requirements:

provide an equitable distribution of CPU service;

identify and give prompt service to jobs making interactive requests;

make efficient use of core memory to maximize CPU usage;

have provision for administratively controlling the allocation of resources.

1. Balance Set Scheduling: maximize system efficiency.

2. Setting Process Priorities: priority is based on a long term average ratio of CPU use to real time.

("escape clause": after a block wait of greater than minimum time, a process is given a short quantum at maximum priority.)

3. Resource Guarantees and Limitations

Debugging Aids: DDT##############  
##############  
##############  
01\_12\_02\_hydra

Protection in the HYDRA Operating System

========================================================================

What is HYDRA:

An OS kernel for multiprocessing environments which provides fine-grained protection

HYDRA believes that protection must be an integral part of the operating system

It offers a capability-based protection mechanism which supports user-defined protected sub-systems (Ex: file and directory sub-system)

Protection mechanism is flexible enough to provide a wide range of security policies

Note that protection is a mechanism, security is a policy! Policy is what you want restricted, mechanism is how you are going to achieve that

========================================================================

5 principles of HYDRA's Protection Philosophy

1. Information can be divided into distinct objects for the purpose of protection

2. Objects are distinguished by type

Built-in types: procedure, process, semaphore

User-defined: file, directory

Operations on objects: Type Specific: CALL, START, P; Type Independent: READ, WRITE

3. Access to objects is controlled by capabilities

Capability contains a large no. of access rights, can be transferred from one user to another, can be type specific/independent

The more rights you have, the more control you have

So object has no "owner" as such

4. Each program should execute with the smallest set of access rights necessary

5. Representation and implementation of operations for each type of object should be hidden in a subsystem

Subsystem ~ type of object + associated procedures

Procedures are used to manipulate objects using the mechanism of rights amplification

Given a capability for an object of a particular type, the subsystem wishes to gain the rights necessary to manipulate that objects representation.

========================================================================

Objects, Capabilities

Object:

Unique-name, type, representation (Data-part, C-list —> Numbered list of capabilities —>

Name of particular object

Access rights — > Bit vector of size 24 —>

ex: read, write

16 generic rights

8 auxiliary rights (type specific)

Cannot get access to an object w/o having capability to do so

New objects can be created in terms of existing ones

Operation on an object is a simple manipulation of Data-part or C-list

========================================================================

Manipulation of Objects

HYDRA provides operations to manipulate objects: generic and non-generic

Generic operations are:

Type-independent and generic

Implemented as calls to the kernel

Used to manipulate the Data-part and the C-list

getData, putData, addData (Data-part)

Load, Store, Append, Delete, Copy (C-list part)

C-list operations allow collections of objects to be passed around

Non-generic operations are:

Based on generic operations generic operations

Implemented as procedures

========================================================================

How Objects share Data/Rights

Object with capability structure allows sharing

Sharing of information. Example:

Three objects: Comm1, user U1 and U2

U1 and U2 have capability for Comm1

U1 stores info in the data-part of Comm1 and U2 retrieves it

Sharing of rights. Example:

U1 has read-right and write-right some file F1 and wants to grant U2 read access to F1

U1 stores file capability of read-only in Comm1 allow

U2 accesses the file to read using capability stored in Comm1

========================================================================

Procedures and Local Name Space

Procedure:

Special object that serves as an abstraction of ordinary procedure

Call to a procedure causes a change in the protection domain (recall that Protection domain: set of capabilities exercised by executing procedure)

A c-list can be passed as an argument to a procedure

A procedure may return a capability

Local Name space:

Special type of object that represents executing programs

Defines the instantaneous protection domain of an executing program

Can be thought of as a graph of objects accessible to a program

========================================================================

Protection Problems (PP)

HYDRA solves some well-known protection problems by extending the interpretation of rights

Mutual Suspicion

Modification

Limitation of Propagation

Conservation

Confinement

========================================================================

PP1: Mutual Suspicion

Problem:

Caller needs a guarantee that callee is not granted access to any of its objects except those for which it passes capabilities as parameters

Callee needs a guarantee that the caller cannot gain access to its private data unless if it explicitly allows it

Solution:

HYDRA's Protection Principle 4 states: each program should execute with the smallest set of access rights necessary. This guarantees solution to Mutual Suspicion

Protection for Callee

A caller can only operate on objects whose capabilities are present in the C-list, no access to callee's capabilities

So, callee should be protected if it keeps sensitive data in its "own" (i.e. private) capabilities.

Protection for Caller

LNSes are stacked. A procedure can't access capabilities in LNSes deeper in the process stack.

========================================================================

PP2: Modification

"User often wants to guarantee that an object passed as an argument to a procedure will not be modified as a result of the call."

Solution: MDFYRTS

To store a capability in an object, one must have a capability for the object with both STORTS and MDFYRTS.

User passes a capability to callee procedure restricting MDFYRTS

HYDRA enforces that MDFYRTS can never be gained through amplification.

========================================================================

PP3: Limiting Propagation of Capabilities

"A user wishes to allow another user to access an object but wants to guarantee that the other user can't share access with a third user."

Solution: ENVRTS

A capability may only be stored in an object if the capability contains ENVRTS

In this way a capability does not escape outside of the executing ENVironment (LNS)

========================================================================

PP4: Conservation

"A user wishes to pass a capability for an object to a procedure. Though he expects the procedure to modify the object, he wants to guarantee that on one will continue to modify the object after the procedure returns."

Solution: ENVRTS

If the capability for an object is passed to a procedure with ENVRTS restricted, the LNS of the procedure cannot store this capability in any object that another user can access

========================================================================

PP5: Confinement – Limiting propagation of information

WHAT IF???

A procedure when called, creates a new object, gives it the capability allowing ENVRTS and MDFYRTS, copies data from old object to new object and shares the capability for the newly created object with other objects.

"A user needs a guarantee that no information may escape from the called procedure except to objects specified by caller."

Solution: MDFYRTS

Restrict MDFYRTS to guarantee against leakage of information

========================================================================

Conclusions

Separation of Policy and Mechanism

Protection mechanism is minimal yet efficient.

Using this mechanism, user can develop a complex policy to guarantee certain kinds of behaviour

Very flexible, fine-grained protection

Has solved protection problems like Mutual Suspicion, Modification, Limitation of Propagation

Looks performance-intensive though…

A bit complicated to create objects and define operations on them, don't you think?##############  
##############  
##############  
01\_14\_01\_protection

Protection

At that point in time, they had very ad hoc implementations of protection/security, so he made a general paper re protection

Object - anything that you want to protect. Resources, segments, files, etc

Name must be unforgeable (OS has to control them)

Domain - the context in which an object has rights to other objects

Access matrix - designate what permissions each domain has over each object

Control vs owner - control deals with the domain's point of view, whereas owner deals with the object.

Looking at individual rows of Access Control Matrix looks like capability, looking at columns is an ACL

Message System:

consists of processes which share nothing and communicate with each other only by means of messages.

Message: consists of an identification of the sending process followed by an arbitrary amount of data.

Messages are received one at a time in the order in which they were sent.

two major flaws:

1. impossible to retain control over a runaway process

2. an elaborate system of conventions is required to get processes to cooperate

Object System:

a set of objects, a set of domains, and an access matrix or access function

##############  
##############  
##############  
01\_14\_02\_multics

# Multics

## Important points: compare ACL with capability

1. Capability method - Attach the d'th row of access matrix A to the domain d.

Domain is defined by capability list, C-list.

[Example]

Give a domain a write permission (capability) to a file (obj).

2. Access Control List method - Attach the protection information to the object

rather than the domain. Easier to revoke access. Likely to be more expensive.

[Example]

Multics needs to check ACL to map a segments from disk to a process's address

space.

3. A hybrid version, an object can be accessed by access key to obtain a

capability, which is then used for subsequent accesses.

[Example]

Check ACL in disk to gain file access (ACL)

Then use file descriptor in memory (Capability)

## Main points

### Access Control List (storage system)

Each process is associated with a principal identifier. Whenever the process attempts to access a segment, the principal identifier of the process is compared with those appearing in the access control list of the object.

ADV: easier used by users

DIS: overhead in checking, walk through the list

### Hierarchical Access Specifications

Hierarchical file structure implies hierarchical access rights

### User Login

### Memory Protection

In Multics, the function of descriptor includes modes of access (read, write, and execute) and to provide for protected subsystems with share object names with their users.

The virtual address space of a Multics process is implemented with an array of descriptors, called a descriptor segment.

Protection information is associated with the addressing descriptor rather than with the data itself. Thus, a single physical segment may appear in different address spaces with different access privileges for different users, even though they are referring to the same physical data.

The protection information found in a segment's descriptor is derived from the access control list for the segment. Any change to an access list is immediately propagated to all descriptors which have been derived from it.

+---------+----------+-------+---------+-----------------+

|SEGMENT | ADDR | RWX | CALL | PROTECTED SUBSYS|

+---------+----------+-------+---------+-----------------+

(0) (1) (2) (3)

### Protected subsystem

Any user can construct a protected subsystem, which is a collection of programs and data with the property that data may be accessed only by programs in the subsystem, and the programs maybe entered only at designated entry points, known in Multics as gates.

(2) and (3) of the addressing descriptor together allow hardware enforcement of protected subsystems.

(2) is the control of permission to enter a protected subsystem which has entry points in the segment based on this descriptor.

(3) is the controls on which protected subsystems may use this descriptor.

Protected subsystems are formed in 0-7, and the hardware permits a subsystem to use all of those descriptors containing protected subsystem numbers greater than or equal to its own.

In UNIX, User -> Kernel, System call is the transfer control

In Multics, 8 levels of privileges, generalize the system call to gate, which is the only way to transfer a control.

##############  
##############  
##############  
01\_19\_01\_unix\_timeshare

# Unix

## Basics characteristics

\* Everything look like a file

\* I/O Devices

\* Streaming model

## Mount

\* Single naming hierarchy

## Links (hard links)

\* File exists independent

\* Directory links (maps)

Names -> I-nodes

\* If delete a file, decrease reference, if zero delete

\* Cannot reference other file systems

## Permission (protected control transfer)

\* set UID

\* e.g. runs at the owner of the file

\* sudo: run as root

## Shell redirect pipes

\* fd0: stdin, fd1: stdout

\* fork

## Multics: memory mapped, Unix, streaming

Multics uses hardware supported more sophisticated virtual address spaces:

segments (which worked well with memory mapped files) and rings (generalization

of the user-kernel mode into 8 protection levels)

## What is different from today's Unix?

\* Groups

\* Filesystem

\* Name of the file, file size

\* Network

\* GUI

##############  
##############  
##############  
01\_19\_02\_plan\_9\_bell\_labs

# Plan9

## Specialize model

\* Originally: all user logged in a time shared system, now each user has a unix

\* Therefore, there is admin burden

\* Now: rejects run unix on everything

\* New model: Specialize

Different nodes have different functions.

\* Terminals

\* CPU Server

\* File Server

## Unified

\* Everything like a file (same as Unix, even more in some place, like control device is a file in Plan9)

\* Everything becomes a file server

\* 9P protocol connects all clients and servers

## Local Name Space

\* Different from Unix's global name space.

\* Customize the name space, no uniform one. Every process sees a different one.

\* Name convention /dev/con -> console windown. Change what /dev/con is mapped to, to have a local or remote console.

\* Associated with process (each process has own namespace)

## WORM

\* Three levels of storage.

\* The disk is a cache for the WORM and the memory is a cache for the disk.

\* Every day, a dump of the file system occurs automatically, written to the WORM.

## RFORK

\* rfork falls somewhere between processes and kernel threads. rfork create a new process, and specify what should be shared. BSD has rfork, and Unix has clone, which is similar.

## Union Directory

\* We can have concatenation of directories.

\* No $PATH needed.

\* /bin is a union dir of every dir we can find runnable programs. It is built as a union of$cputype/bin, /rc/bin and perhaps more defined by the user.

\* The lookup in a union directory obeys the rule that search in turn, the first one match will be taken.

## Influences

\* Unicode, UTF8

\* file caches

\* /proc is later used in Linux.

\* rfork exists in BSD, clone() in Linux.

\* NFS block access, plan9 byte access

##############  
##############  
##############  
01\_26\_01\_pilot

# Pilot: An Operating System for a Personal Computer

Q: How do the requirements of the Pilot operating system differ from the systems we have read about so far, and how does the design of Pilot reflect those differences?

## Why different?

\* Single user

\* Defensive protection

\* Resources - not on fairness

\* Network

## Feature of Mesa language

\* Object oriented (Pointer not exist)

\* Single address space (No H/W protection)

\* Strong type check

## Hints

\* Memory mapped file, pages used, can be moved

\* OS trust the app hints

\* No incentive for lying

## Virtual memory

Virtual memory -> Space

Space.Create

\* Space is servering as the unit of allocation.

\* Space can be created as nest.

Space.Map

\* Virtual memory is the only access path to the contents of files, and files are the only backing store for virtual memory.

Swapping

\* The swapping between primary memory and backing store is performed in the units of spaces.

\* The swapping stategy followed is to swap in the lowest (i.e., smallest) space containing the page.

\* Space.Activate as a hint to be swapped in asap. Space.Deactivate is the inverse operation.

## Advantage of all files memory mapped

\* Decouple read / write and paging

\* Reuse virtual memory functionality

\* Stream I/O on top

## Kernel / Manager

Kernel -> Mechanism

Manager -> Policy

Kernel: Memory page out, page in.

Manager: Decides which page to page out.

##############  
##############  
##############  
01\_26\_02\_rethinking\_software\_stack

Motivation:

Create dependability with protection and isolation

Move error detection closer to design time

Goal: The primary Singularity goal is reliability and robustness (in their terms, “dependability and trustworthiness”)

Protection and Isolation:

Faults in application/driver assemblies shouldn’t affect overall system stability or stability of other programs

Inherently more difficult to create malicious applications

Eschews HW memory protections in favor of SW approaches

Programming language mechanisms:

The vast majority of Singularity’s kernel is written in Sing#, a memory-safe programming language derived from C#. But Singularity’s language integration goes beyond memory safety. The Sing# programming language was extended in several directions to make certain programming errors simply impossible. The biggest example is inter-process communication, or contract-based channels. IPC is defined by state-machine-like contracts whose specifications are verified by the compiler. This ensures that every process has explicit code to handle every possible message

Singularity also addresses robustness by seriously limiting what processes can do. Singularity processes are sealed. They cannot load libraries dynamically, modify their own code, or share memory with other processes. Some serious limitations: just-in-time compilation is impossible in sealed processes, for example. The implicit argument is that dynamically linked libraries, self-modifying code, and shared memory are inherently dangerous and should be eliminated. But another argument is that code without these features is much easier to statically analyze. It’s not clear which of these arguments led to the development of sealed processes.

Finally, Singularity’s manifest-based programs bring type checking to process creation. A manifest defines a bunch of checkable program properties that the Singularity kernel can verify before starting a process. For example, the manifest says what ABI versions a program needs, what IPC interfaces are required, what other processes must be started, and so forth. The kernel can check “type and memory safety, absence of privileged-mode instructions, conformance to channel contracts,” and other, more specific properties, such as “that [a device] driver will not access hardware used by a previously installed device driver.”

Evaluating Software Isolated Processes (SIPs)

Singularity processes are generally isolated only by software. They are called Software Isolated Processes, in fact. Most processes run in the same address space as the kernel. Software verification and language safety ensures that SIP code can’t abuse the kernel privilege under which it runs.

Hardware isolation ain’t free. Kernel crossings, which require special instructions, are much more expensive than simple function calls. Hardware virtual memory, which is irrelevant if you trust your memory-safe language, introduces a TLB and associated costs. So Singularity systems recover some performance lost to sealing and garbage collection by collocating processes with kernel code in a single privileged address space, and then optimizing accordingly.

SIP memory management: Page-disjoint heaps

Sing# is a garbage collected language, and Singularity is a garbage collected operating system. Additionally, in the default mode, all processes cohabit the same address space. So you might expect all processes to share a single garbage collector. They don’t, and this is one of the more unusual and interesting design decisions in the Singularity system.

Each Singularity process has its own page-disjoint heap. That is, no process can ever access objects in another process’s heap, and the heaps are disjoint at the level of pages, not objects. All of process A’s objects live on process A’s pages, which are disjoint from any other process’s pages.

Reason:

Disjoint heaps mean no shared memory.

Shared memory would complicate verification.

Page-disjoint heaps simplify accounting.

A process’s memory size equals its page count.

Page-disjoint heaps simplify recovery after process exit.

No need to garbage collect its objects one at a time, just reclaim all its pages.

Page-disjoint heaps simplify experiments with address space designs (Figure 5).

Disjoint heaps allow each process to run its own garbage collector.

“The large number of existing garbage collection algorithms and experience strongly suggest that no one garbage collector is appropriate for all system or application code.” [p6, 1]

If no two processes can share objects, then each process can collect its own objects without agreeing on object layout with other processes.

In a system with a single unified GC, it would occasionally be necessary to completely stop the system for GC. (Even concurrent GCs usually have stop-the-world phaselets.) Independent GCs avoid this: GCs can run independently.

Wrinkle: In the single-address-space variant, Singularity system calls are implemented as procedure calls, and the kernel stack shares space with application stacks. So how can the kernel and application GC tell one another’s objects apart?

Solution: Special, explicit structures mark the stack boundaries between kernel and application threads. The GCs understand these structures.

Note that SIPs can choose their own GCs, but not necessarily write their own GCs. In the single-address-space variant, SIP GCs must be trusted by the kernel, so untrusted code can’t run in the GC. Most likely Singularity provides several pre-approved GC implementations; each SIP chooses one of these GCs and supplies it with optional parameters.

SIP memory management: Exchange heap:

Since normal heap data can’t be shared, a separate, explicitly-managed memory area called the exchange heap is used for message passing. Exchange heap objects must have an exchangeable type.

Exchangeable objects are thus relatively simple—think flat objects, or objects with pointers to simpler objects, such as a “packet” type that points to an array of bytes.

The kernel is ultimately responsible for managing the exchange heap’s memory; for example, it garbage collects the exchange heap to eliminate objects held by exited SIPs. But recall that for robustness, Singularity also prevents processes from simultaneously accessing objects in shared memory. Regular heaps are pagewise disjoint, but the exchange heap is explicitly designed for inter-process communication.

Singularity prevent shared memory access in the exchange heap by a fancy type system. Sing# was extended to support a linear type discipline for exchange heap objects. Linear types ensure that each process can have at most one pointer to an exchange heap object at a time. When a process sends a message, the type of the send “system call” forces the sending process to lose that sole pointer to the message. As a result, and because of memory safety, the process also loses the ability to modify the message, and each exchange heap object is accessible to at most one process at a time. The linear type discipline also facilitates explicit allocation and deallocation operations for exchange heap objects, new and delete, which quickly recycle unneeded exchange heap memory.

Linear types are cool and useful to enforce the no-shared-memory invariant. Reasons for implementing the exchange heap: performance. The exchange heap allows one Singularity process to send a message to another without copying; in the simplest case a single pointer to an exchange heap object will be transmitted. This can look great on microbenchmarks [4]. But (1)If the sending process wants to preserve a copy of the message it must make a copy explicitly. (2)The sending process must always construct the message on the exchange heap or copy it to the exchange heap from elsewhere. There is no way to reuse a message buffer, since the message buffer is explicitly lost on send. (3)As discussed by the exokernel and L3, microbenchmark performance does not always correlate with application performance.

Verification

SIP safety depends on some trusted code and some untrusted code. The trusted code includes the verifier itself, parts of the kernel, and any unsafe code that runs on behalf of the SIP, including the SIP’s garbage collector and memory allocator. The SIP’s process code is untrusted, and therefore Singularity must actively verify that it obeys Singularity’s invariants. Safety requires these checks:

1.Heap memory safety.

All local pointers point into the SIP’s heap.

Pointers cannot be fabricated from integers: new pointers only arrive from the SIP’s trusted memory allocator.

Pointers are strongly typed (i.e., can’t access memory using the wrong type, which could break memory safety later).

Enforced using known techniques (e.g., the Java bytecode verifier).

2.Exchange heap memory safety.

Exchange heap pointers obey the linear type discipline (to prevent modification of shared memory).

No double access, etc.

Exchange heap pointers are strongly typed.

3.Channel contract agreement.

Much of channel contracts are not strictly required for safety. But channel contracts affect memory safety because Singularity infers a message’s type from its name and state, as specified in a channel contract. So if Singularity didn’t verify that both ends of a channel agree on the channel contract, and that sent and received messages’ types agreed with the contract, then message passing might break exchange heap type safety, and therefore memory safety.

4.Kernel ABI agreement.

5.Instruction safety.

SIPs must not use privileged machine instructions inappropriately.

Singularity also verifies other properties that aren’t as safety sensitive.

1.Channel contracts are checked for unhandled messages.

Not very safety sensitive since unhandled messages could easily be implemented as exceptions.

2.Channel contracts are checked to verify that all cycles in contract states contain at least one receive and one send action.

Reason: If a state cycle existed involving only process A sending messages, then A could send infinitely many messages to B without waiting for a response. This might overflow any bounded queue, and Singularity wants to avoid overflow while bounding the queue size.

Why? Feels a bit random: Singularity wants message receive to involve no allocation, so that a receiver can always reliably receive a message; this requires bounded receive queues. Why not involve allocation?—perhaps a performance concern?

“Although the rule [about state cycles] seems restrictive, we have not yet seen a need to relax this rule in practice.” I.e., it’s not a safety concern.

What about a cycle like “A receives → A sends M1 → A sends M2 → B receives M1 → B sends M3 → cycle”? Won’t B’s queue grow infinitely long?

No, because contracts don’t work this way. In contracts receive events are implicit—an endpoint must receive all previous messages in a contract line before sending a message itself. The contract above would really be written as something like “state X { M1? → M2? → M3! → X; }”, which shows that B (the exporting end) must receive both M1 and M2 before sending M3.

Verification happens like this. A Sing# compiler, Bartok, compiles source code to an intermediate bytecode language, MSIL. At SIP install time (as a SIP is started), the verifier checks the bytecodes; simultaneously, a bytecode compiler generates machine code from the bytecodes (possibly interleaving that machine code with trusted machine code, such as the GC). At runtime, machine code is active.

Why not verify at compile time?

If the verification happened only at compile time, then the compiler would have to be trusted.

Why not compile to machine code?

Machine code, being lower level, is harder to verify.

MSIL bytecode is machine-agnostic, which arguably simplifies deployment in heterogenous environments (“Singularity packages manifest-based programs in the abstract MSIL format, which can be converted to any I/O processor’s instruction set.” [p10, 1])—but this is a stretch.

Nevertheless, the paper claims that in future, the compiler could generate type-safe assembly language (TAL), which is closer to machine code.

Why not JIT at run time?

The Singularity authors argue that JIT is inherently risky and unrobust.

Why not verify at run time?

Some verification does happen at run time (for example, consider the trusted Cell<T> wrapper for exchange heap pointers [4], and the bounds checks described as the “Safe Code Tax”), but in general, run-time verification is expensive. If verification can be done statically, the runtime cost is zero.

But we don’t know how fast the bytecode verifier is, and therefore how slow process startup is. There might be a tradeoff.

The paper claims future work will push Singularity verification further, with the nice goal of requiring less trust. In addition to TAL, already a type system was developed that can be used to write type-safe garbage collectors.

Contract-based channels

Channels are like type-safe pipes. “A channel is a bi-directional [lossless, in-order] message conduit with exactly two endpoints.” [p3, 1] Each endpoint is sort of like a pipe file descriptor, except that pipes handle byte streams (channels handle complex, type-safe message protocols) and pipe file descriptors can be shared by multiple processes (each channel endpoint is owned by exactly one thread at a time).

We’ve discussed channel contracts in the context of type safety, but the two Listings in Section 2.2 are worth considering. Note how new channels may be passed over old ones (see NicEvents.Exp:READY in the text and in Listing 1’s in message RegisterForEvents).

##############  
##############  
##############  
01\_28\_01\_monitors\_p549-hoare

Monitor:

A collection of data and procedures

Mutual exclusion: 1) allows controlled acquisition and release of critical resources. 2) single anonymous lock automatically acquired and released at entry and exit

Data encapsulation: monitor procedures are “entry points” for accessing data

Rules for Monitors:

1. Any process can call a monitor procedure at any time

2. But only one process can be inside a monitor at any time (mutual exclusion)

3. No process can directly access a monitor’s local variables (data encapsulation)

4. A monitor may only access its local variables

“wait” operation: current process is put to sleep

“signal” operation: wakes up a sleeping process

condition variables: 1) May have different reasons for “waiting” or “signaling”. 2) Processes waiting on a particular condition enter its queue

Scheduled Waits:

A waiting process is given a number. The process with the lowest number gets woken up first.

Proof rules:

I {b.wait} I&B

I&B {b.signal} I

I: the invariant

B: the condition that a process waiting on b wants to be true before its woken up

b: the condition variable

##############  
##############  
##############  
01\_28\_02\_mesa\_p105-lampson

Motivation for adding concurrency to Mesa:

The creation of Pilot which was the first operating system developed for personal computers that would allow for multiple processes to run concurrently where each shared a time slice of CPU. Created using concurrent programming as opposed to event based programming.

Add concurrent programming to Mesa so that side by side development of Pilot could take place.

Choices for handling concurrency in a language:

1.Voluntary yields was an option that was rejected

1)Only good for single processor

2)Must respond to time critical events so preemption needed anyways

3)Restricts programming generality, you need to know if a procedure you call will yield the processor

4)Page faults would cause involuntary yields at arbitrary points in a program

2.Message passing was an option that was rejected because it was proven to be equivalent to monitors when some mild restrictions are applied

3.Monitors was the final choice because it was equivalent to message passing and worked better with the procedural scheme of Mesa, they choose Hoare’s paper as a starting place for thinking about concurrency in their language.

Mesa Language Constructs:

1.Light weight processes

2.Monitors

3.Condition variables

Light weight processes:

Pros:

1.Easy forking and synchronization

2.Shared address space

3.Fast performance for creation, switching, and synchronization

Low storage overheads

Cons:

1.Mesa is a single user system

2.Dangling references similar to those of pointers

Creation of a process in Mesa:

1.Any non-internal monitor procedure can create a new process by the use of a new key word called FORK that is placed in front of the procedure call

2.This FORK procedure call will allow the current process to work concurrently with the new process created. It also returns a reference to the process just created so that the parent process can later joined an a result is returned or to detach from it. Different than Hoare thread that must suspend working until to the thread just created exits or yields.

P ←FORK ReadLine[Terminal];

…<concurrent computation>

buffer ← JOIN p;

3.Every thread can be joined at a later time unlike PThreads where you need to specify at time of creation. Extra overhead in Mesa thread even though in practice most Mesa programmers will immediately detach the thread.

Monitors:

1.Monitor lock for synchronization

1)Tied to module structure of the language; makes it clear what is being monitored.

2)Language automatically acquires and releases the lock.

2.Tied to a particular invariant, which helps users think about the program.

Modules and Monitors:

1.Three types of procedures in a monitor module:

1)entry (acquires and releases lock)

2)internal (no locking done): can't be called from outside the module

3)external (no locking done): externally callable

2.Allows grouping of related things into a module

3.Allows doing some of the work outside the monitor lock

4.Allows controlled release and reacquisition of monitor lock

Notification in Mesa:

Notifying process keeps the lock/control

1.Notify 2.Timeout 3.Abort 4.Broadcast (P112)

Deadlocks

A.Typical deadlock scenarios:

1.Recursion on the same module

2.Enter multiple monitors in different orders

3.Enter multiple monitors in the same order, but wait inside the second monitor does not release the lock of the first monitor

B.General problem with modular systems and synchronization

Synchronization requires global knowledge about locks, which violates the information hiding paradigm

1.Lock granularity: introduced monitored records so that the same monitor code could handle multiple instances of something in parallel

2.Interrupts: interrupt handler can’t block waiting

Introduced naked notifies: notifies done without holding the monitor lock.

Priority Inversion

“Information bus” is a shared memory region shared across the following processes:

1.Bus manager (high priority process)

2.Meteorological data gatherer (low priority)

3.Reset if Bus Manager hasn’t run for a while

4.Protected by a lock

5.If Bus Manager is scheduled by context-switching out the data gatherer, it will sleep for a bit, let the data gatherer run, which will release the lock in a short while

Another thread: communications task

Medium priority, long running task

Sometimes the communications task would get scheduled instead of the data gatherer

Neither the lower priority data gatherer nor the higher priority bus manager would run

Exceptions:

Must restore monitor invariant as you unwind the stack

The idea that you just kill a process and release the locks is naive

Entry procedures that have an exception, but no exception handler do not release the monitor lock

This ensures deadlock and a trip into the debugger, but at least it maintains the invariant

##############  
##############  
##############  
02\_02\_01\_virtual\_memory\_01653971

The VAX-11 and its VAX/VMS operating system were designed to provide extended address space and enhanced performance for Digital’s PDP-11 customers. VAX/VMS intends to support all VAX-based applications and operate on a family of processors (diff performance and physical memory capacities), the memory management system had to be capable of adjusting to the changing demands characteristic of timesharing while allowing the predictable performance required by real-time and batch processes.

Hardware

basic entity in VAX-11 system: process

each has a byte-addressable, 32-bit virtual address space (divided into 512-byte pages)

(21-bit virtual page number + 9-bit byte offset within the page)

Three region: system, P0, P1

2-bit for dividing the process address space into a number of functional regions or spaces

bit 31 = 1: system space and is shared by all processes in the system

(only half of the system space is currently used by the architecture)

bit 31 = 0: process space and is unique to each process (two segments)

bit 30 = 0: P0 (program region) grows toward higher addresses

bit 30 = 1: P1 (control region) grows toward lower addresses

each region is defined by a page table.

basic unit of mapping and protection: page

page table: contiguous array of 32-bit entries,

defined by a base address register and a length register

translation buffer: cash virtual to physical translations

divided into two sections: system translation, per-process translation

(only the process section be flushed when context switch)

PTE:

a valid bit (bit 31): whether this entry contains mapping info

a protection field (bit 30-27): what privilege is required to read or write the page

a modify bit (bit 26): whether a write access has occurred to the page

a field used by the OS (bit 25-21)

the physical page frame # (bit 20-0): locate the page in physical memory

system page table: is located by reference to the system page table base register which contains its physical address

P0/P1 page tables for each process: located in the system-space section of the address space, base registers contain virtual addresses (changed by a process context switch)

translation: (two accesses)

1. access to the system page table to calculate the physical address of the process page table

2. access to process page table to calculate the physical address of the specified element

system region:

All executive code and data, including some process-specific data structures and process page tables

Software vector region: pointer to executive service routines in system space

Memory management system: hardware access mode mechanism (executive code and data are not accessible to programs executing in the least privileged (user) mode)

Collection of procedures that exist in the address space of each process (can be called explicitly)

program region:

user’s executable program (can dynamically grow into higher-addressed sections)

the first page (0-511) is marked as no access so that any reference will cause an exception

(catching program errors involving uninitialized pointers)

control region:

store process-specific data, program image for the command interpreter (not to disturb user program images)

user’s stack in the low-address part, grows toward lower addresses

fixed-sized executive per-process stacks

——————————————————————————————————————————————

Memory management system:

the pager: an operating system procedure that executes as the result of a page fault; executes within the context of the faulting process, responsible for loading and removing process pages into and out of memory

the swapper: a process responsible for loading and removing entire processes into and out of memory

pager: resident-set list for each process

1. a process-local page replacement policy used to limit the effect of a heavily faulting process on other processes in the system. The page to be removed from memory is selected from the process requesting a new page. (FIFO within the resident-set list)

process’s resident set: the set of pages currently in memory for a process

resident-set limit is the max size of the resident set: amount of physical memory a process may occupy

2. trade-offs were made in favor of reducing processor usage at the possible cost of increased memory requirements. The amount of computation required of the pager is limited; a quick decision is made and a software caching mechanism is used to reduce the penalty for removing a page that is still in use. (no overhead is required to maintain well = behaved programs executing in sufficient physical memory)

3. modify bit for page removed = 0: add to the tail of the free page list; =1: add to the tail of the modified page list, wait for written to the paging file

If a page faults is on either list, returned to the process’s resident set.

The faulting of a page from the free list causes the page to be placed back at the top of the FIFO resident-set list.

4. attempts to reduce the # of physical disk operations by reading and writing several pages at a time (clustering) (largest gain: initial program execution)

executable program > segment > pages

A user can specify a default cluster size for each segment when a program is linked. Normally a systemwide default is used.

delay modified-page write requests through the modified page list mechanism:

1. the modified page list can be a cache of recently removed pages. when a page is referenced, it can be returned to the process at min cost.

2. when a write request must be performed, many pages can be written at once.

3. pages can be arranged on the paging file so that clustering on read requests is possible

4. it reduces the page writes since either the pages are faulted again or the program terminates

a demand-zero page: created for a program and initialized to zero on demand. invalid bit set. when the page is faulted, the pager allocates a physical memory page, fills it with zeros, and adds the page to the resident set of the process. modified bit set in PTE.

copy-on-reference pages: modify bit set for a writable page when it’s first loaded from the executable file. The paging file will be used as backup storage for the page. For sharing, each process will receive its own copy of that page.

When a process faults one of its page tables, the page table is added to the process’s private resident set. The page table will not be eligible for removal from the resident set as long as it contains any valid PTE.

swapper: a process swaps entire resident sets between memory and backing store;

also responsible for writing the modified page list

objectives: to keep the highest priority processes resident; to avoid the typically high paging rates generated by resuming a process

##############  
##############  
##############  
02\_02\_02\_machine\_independent\_vm\_p31-rashid

Mach: a portable, multiprocessor operating system

goal of Mach: explore the relationship between hardware and software memory architectures; design a memory management system that would be readily portable to multiprocessor computing engines as well as traditional uniprocessors

separation of software memory management from hardware support: few assumption about memory management hardware; primary requirement is handle and recover from page faults (arbitrary page size)

1. A task: an execution environment in which threads may run; the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory).

2. A thread: the basic unit of CPU utilization. It is roughly equivalent to an independent program counter operating within a task. All threads within a task share access to all task resources.

3. A port: a communication channel -- logically a queue for messages protected by the kernel. Ports are the reference objects of the Mach design. Send and Receive are the fundamental primitive operations on ports.

4. A message: a typed collection of data objects used in communication between threads. Messages may be of any size and may contain pointers and typed capabilities for ports.

5. A memory object is collection of data provided and managed by a server which can be mapped into the address space of a task.

Operations on objects other than messages are performed by sending messages to ports.

Copy-on-write sharing between unrelated tasks is typically the result of large message transfers. An entire address space may be sent in a single message with no actual data copy operations performed.

Read/write shared memory can be created by allocating a memory region and setting its inheritance attribute (shared, copy or none per page). (e.g. child share memory of parent by inheritance value)

Protection (read, write and execute permissions) per-page basis.

For group of pages, maximum protection (can be lowered) >= current protection

Virtual memory related functions, such as pagan and page out can be performed directly by user-state tasks for memory objects they create.

##############  
##############  
##############  
02\_04\_01\_v\_kernel\_p129-cheriton

The Distributed V Kernel and its Performance for Diskless Workstations

1. Introduction

V kernel

-message oriented kernel

-provides uniform local and network interprocess communication

-Thoth model adopted

Advantages

-Lower hardware cost per workstation

-simpler maintenance and economies of scale with shared file server.

-Little or non memory or processing overhead on the workstation for file system.

-Fewer problems with replication, consistency and distribution of files.

This paper challenges the controversial on:

-Distributed file system with diskless workstations

-The use of general purpose interprocess communication facility

-The use of Thoth-like interprocess communication model

-The use of synchronous request-response model (RPC model) of message-passing and data transfer instead of streaming protocol

2. V Kernel Interprocess Communication

IPC between processes in V kernel - RPC like communication

-Client: send a message

-Server: receive the message

-Server: optional data transfer using MoveFrom

-Server: process the request from the client

-Server: optional data transfer using MoveTo

-Receiver: send reply

All messages are 32 bytes long

V kernel directly copies data from sender's address space to receiver's address space

-No copies between user space to/from kernel space!

-Compare this to LRPC

2.1 Primitives

Send(message, pid)

-pid(32-bit): pid of receiver process

-message(32-byte): flag + access right + ... + segment addr + segment length

--flag: whether the segment is given in this message or not

--access right: access right to the segment

--segment: a part of address space of sender which sender wants receiver to access

-sender blocks until reply message is returned

-reply message overwrites the original message area

-receiver may read from or write to the segment given in the message: see MoveTo, MoveFrom, and ReplyWithSegment

pid = Receive(message)

-blocks calling process until a message is received

-pid = pid of sender

(pid, nbytes) = ReceiveWithSegment(msg, buf, len)

-same as Receive

-copy: buf[0 ~ len] <- msg's segment, if a segment is in msg

-nbytes = actual number of bytes copied

-This primitive is introduced to reduce the number of messages

Reply(message, pid)

-message: reply message

-non-blocking operation

ReplyWithSegment(message, pid, destptr, segptr, segsize)

-send message to pid

-copy: destptr <- segptr[0 ~ segsize]

-destptr: original sender's address space

MoveFrom(srcpid, dest, src, count)

-copy: dest <- src[0 ~ count]

-src: address space of srcpid

-dest: address space of calling process

-srcpid must have given read access for src to this process using 'Send'

MoveTo(destpid, dest, src, count)

-copy: dest <- src[0 ~ count]

-src: address space of calling process

-dest: address space of destpid

-destpid must have given write access for dest to this process using 'Send'

2.2 Discussion

Pros

-Synchronous request-response model makes programming easy due to its similarity to procedure call

-Distinction between small message (send, receive, reply, ...) and a separate data transfer (moveto, movefrom, ...) is good

-Synchronous communication (stop-and-wait), small fixed message: make buffering easy -> leads small kernel

-Direct copy between user spaces: no extra copies between user & kernel space

Cons

-Message is even smaller than the min packet of Ethernet -> leads padding -> inefficient use of bandwidth

-Stop-and-wait: reduce parallelism

-Separate data transfer command -> increase the number of operations going: send(msg) -> moveto -> reply

3 Implementation issues

General measures taken to achieve efficiency

-Remote operations are implemented directly in the kernel: no context switch to network process

-Raw Ethernet packets are used instead of IP-packet: similar hacking as RPC

-Stop-and-wait as reliable service: also similar to RPC connectionless reliable protocol

-pid: host id + process id

-Data transfer (MoveTo, MoveFrom): no packet-level ack, message-level ack = single ack per MoveTo or MoveFrom

-ReceiveWithSegment & ReplyWithSegment are introduced to reduce packet numbers

3.1 Process Naming

32-bit pid: 16-bit host-id + 16-bit process id: unique within the context of local network

-host-id in 3Mb Ethernet: 8 bit network address + host id

-host-id in 10Mb Ethernet: mapping table from logicalid to network address

GetPid

-look up local mapping table

-if not mapping found, broadcast a message asking network address for the logicalid

3.2 Remote Message Implementation

Send(message, pid) -> check if pid is for local process -> if fails, call NonLocalSend

=> send a packet to the receiving pid or boadcast it, if network address of the pid is unknown

-> receiving host creates an 'alien' process descriptor -> saves the message into the buffer of the 'alien'

-> virtual interaction between 'alien' process and the actual receiving process

-> receiver process replies: reply message sent to the sender + cached in 'alien' space

-> timeout, retransmission, and RPC-probe-like message are handled between 'alien' and sender

3.3 Remote Data Transfer

-Single ack per moveto or movefrom => indifinitely large packet size -> requires very reliable network

-Error -> retransmission of all packets

-Measure has been taken to cope with back-to-back failure at the same packet

-Direct copy between network interface and source or destination process address space: need to use programmed I/O

3.4 Remote Segment Access

In distributed file system, lots of page read/write are going on. In Throth model, a single page write is composed of:send -> receive -> movefrom = page transfer -> reply

-ReceiveWithSegment: receive a packet which consists of the message and the very first part of the segment

Hence, if the packet is big enough to contain the message and a page,

-page write reduces to Send -> ReceiveWithSegement -> Reply

-page read: Send -> Receive -> ReplyWithSegment

=> Original send should be changed

Look up performance in the paper

4. Network Penalty: Reasonable Lower-Bound of Communication

5. Kernel Performance

5.1 Measurement Methods

5.2 Kernel Measurements

5.3 Interpreting the Measurements

5.4 Multi-Process Traffic

6. File Access Using the V Kernel

6.1 Page-level File Access: Random Page Access

6.2 Sequential File Access

6.3 Program Loading

7. File Server Issues

8. Measurements with the 10Mb Ethernet

9 Evaluation:

Pros:-Direct data copy between sender and receiver

-Separation of short message and bulk data transfer

Cons:-No flow control: network congestion especially at the server side will aggravate the problem

-Lots of hacking similar to RPC

-Only works within local network

##############  
##############  
##############  
02\_04\_02\_sprite-ieee-computer88

The Sprite Network Operating System

1. Intro

Capitalize on three technology trends:

-Fast local networks: take advantage of communication to make a set of machines look like one (network transparency) big time-shared machine

-Large memories: do lots of file caching

-Multiprocessors: build OS to take advantage of them (authors believed every desktop would become a multiprocessor)

Approach: Keep OS interface same (BSD UNIX), but modify implementation to address above goals.Add the following to the OS interface & impl to support sharing:

-transparent network file system (all clients store files on servers; many of their workstations were diskless; all clients share server disks),

-write memory sharing for multiple processes on a single workstation (processes on a given machine share memory)

-process migration (sharing CPU across all machines)

Implementation:

-Supported kernel-to-kernel RPC (only used by kernels; OS is not message passing based, but is kernel call based; contrast with Mach, V)

-Prefix tables (supported dynamic reconfiguration of mount points on distributed FS)

-File caches on servers and clients

-VM system uses file for backing storage

-Processes behave the same whether migrated or not

2.1 File System

-Network transparent (system calls for manipulating files are same regardless of where file is on the network)

-Name transparent (name of file is same regardless of which machine on the network it is actually stored on; can move files from one machine to another w/o changing name)

-In Sprite, entire FS is shared, whereas in AFS/NFS, only parts of the FS are shared and machines can still have files on a local disk that no one else can access; not so in Sprite. (This is used to help implement process migration.)Both Sprite and LOCUS provide a single file hierarchy accessible from all workstations.

2.2 Shared Address Spaces

Implemented in OS in anticipation of multiprocessors. -Parent process could fork a child and specify if child should share data (static data + heap). -Code segment was shared read only. -Data segment was shared read-write.

Regarding memory sharing, Sprite provided calls that allowed processes to be put to sleep & awoken, but no locking primitives provided by the kernel. -Sync impl on top of test-and-set so that context switch to kernel does not need to be made for thread operations.

2.3 Process Migration

Calls forwarded to home machine when need to access I/O device on home machine? Not necessarily.Devices are just special files on the unified filesystem hierarchy... they should be accessible from any machine.

Processes that share and address space must be migrated together.

OS provides calls to migrate processes.

Migrated process still appears to be running on home machine (i.e., when user runs ps), but really is running elsewhere on an idle machine. Transparency results are same as if process was executed on home machine.

3 Kernel Structure

3.1 Multi-threading

Kernel was multi-threaded does not have one big lock that everyone blocks on like in UNIX or Linux, they can execute kernel code at the same time. Else multiple threads could only execute concurrently while is user mode.

3.2 Remote Procedure Call

efficient for request-response transaction.Consists of stubs and RPC transport. See Fig3.

It delivers message across the network and assigns incoming requests to kernel process that execute the server stub and called procedure.

used implicit acknowledgement (responses ACK requests) and fragmentation (used to ship large blocks of data; i.e. VM pages during process migration)

4 Prefix tables

Table seached for longest pathname match during name resolution. Either server returns a number corresponding to resolved file, or remaining part of path that needs to be resolved that is outside of its domain.� If given a pathname for which there is no prefix table entry, request is broadcasted so that server that manages that domain responds, and new prefix table entry is made.

5 Caches

-Advantages of caches: reduce network communication, reduce load on servers. Disadvantage: need to run consistency protocol.

Sprite gives better consistency than NFS: NFS files are opened, and client can see stale data until file is closed and reopened. Sprite always gives client most recently updated blocks.

Sprite FS- block-based like NFS; entire files not cached.

-Sequential write shared handled using version numbers.On open, client notifies server, and uses its cached version if server does not tell it about a more recent version number.

When a client that is not the last writer opens a file, the server forces the last writer to flush any modified pages.

-Concurrent write-sharing (two clients have file open at same time), client caching is disabled.

Disadvantages: bad performance for concurrent write sharing case.� clients must contact servers on opens.

This scheme only provides consistency; FS\_Lock provided for synchronization.

6 Virtual Memory

6.1 Backing storage for each machine is on network file server.

Advantages: Simplifies process migration, VM system can reuse existing FS syscalls, don't have to preallocate a static amount of disk space for swap space.

Since servers have large caches, VM pages can be read from servers faster than accessing local disk due to fast networks and slow disks (seek time).

6.2 Sticky segments don't swap out programs code when execution completes until it becomes least recently used segment; this allows new process to start up quickly if code is still in memory.

6.3 Double-Caching

Since VM uses FS, pages could get doubly cached in FS cache and VM address space. Careful implementation required. Solution: VM system bypasses local file cache when reading/writing backing files. Backing files are cached on servers.However, servers do not cache backing files for their own processes.

6.4 VM-FS Negotiation

VM / FS Physical Memory sharing: The amount of physical memory allocated to VM and FS can change dynamically in Sprite. For processes that are memory intensive (simulations), most of memory gets used for VM pages. For processes that are disk I/O intensive (databases), most of memory becomes a file cache. This is impelemented as follows: whenever a page needs to get kicked out of physical memory, the oldest VM page is compared with the oldest file cache page, and the oldest out of either of them are thrown out.

7 Process Migration

Sprite: dirty pages are sent to server, code segment is reused if it already exists on target machine, or is loaded from backing store server, and data is demand paged from backing store

advantages: requires less total data to be copied than V, does not require old machine to server page faults; disadvantages: process is frozen longer than in V or Accent.

Process migration criterion: transparency, time to kick process off old machine, time that process is frozen, time old machine is servicing pages after migration. Sprite: Machine specific calls in the migrated process are forwarded to home machine to achieve transparency. (done using kernel-to-kernel RPC)

##############  
##############  
##############  
02\_09\_02\_p201-feeley

Implementing global memory management in a workstation cluster

1 Abstract & Intro

Our objective is to use a single, unified, but distributed memory management algorithm at the lowest level of the operating system. By managing memory globally at this level, all system- and higher-level software can benefit from available cluster memory.

Important parts of GMS is it page aging procedure, which provides information for global decision making.

2 Differs from previous:

-integrated with the lowest level of the system and encompasses all memory activity

-manage memory globally, attempting to make good choices both for the faulting node and the cluster as a whole.

-gracefully handle addition and deletion of nodes in the cluster without user intervention.

3.1 Algorithm:

find in paper p202.

-Global-memory size changes dynamically

-Local pages may be replicated on multiple nodes

-Each global page is unique

-Using swap when page faults. 4 cases.

-Over time, nodes will fill their memories with local pages and will begin using remote memory in the cluster.

-Local hit faster than global hit, so that replace global page first if they are same age.

3.2 Managing Global Age Information

Guarantee the validity of the age information and deciding when it must be updated.

-time is divided into epochs (5 or 10 seconds)

-each epoch, nodes send page-age information to a coordinator

-coord. assigns weights to nodes s.t. nodes with more old pages have higher weights

-on replacement, we pick the target node randomly with probability proportional to the weights

-over the period, this approximates our (global LRU) algorithm

3.3 Node Failures and Coherency

Node failures in the cluster do not cause data loss in global memory.If a node housing a requested remote page is down, the requesting node simply fetches the data from disk.

3.4 Disscus about this idea

Minimize memory reference time by minimization of disk access. It means that they show a way to maximize memory utilization in tightly-coupled computers. Avoid impacting programs not using global memory.

-Choose those nodes most likely to have idle memory to house global pages.

-Avoid burdening nodes that are actively using their memory

-Ultimately maintain in cluster-wide primay memory the page most likely to be globally reused

-Maintain those pages in the right places.

4 Implementation

Fig.3 in paper p204 shows implementation, two key points:

-the VM system, which support anonymous pages devoted to process stacks and heaps

-the Unified Buffer Cache, which caches file pages.

4.1 Basic Data Structure

-Every page is identified by a cluster-wide UID

--UID is 128-bit ID of the file block backing a page

--IP node address, disk partition, inode number, page offset

-Page Frame Directory (PFD): per-node structure for every page (local or global) on that node

-Global Cache Directory (GCD): network-wide structure used to locate IP address for a node housing a page. Each node stores a portion of the GCD

-Page Ownership Directory (POD): maps UID to the node storing the GCD entry for the page.

Detail implementation see below sections:

4.2 Collecting Local Age Info

4.3 Inter-node Communication

4.4 Addition and Deletion of Nodes

4.5 Basic Operation of the Algorithm

6 Limitations

- the failures of initiator or master nodes is difficult to handle.

- trust. security. easily in hardware level.

- LRU may bot the best choice

- only few nodes have idle memory, CPU load will high

##############  
##############  
##############  
02\_11\_01\_micro\_kernel\_p66-hartig

The Performance of Microkernel-Based Systems

1 Intro

Goal: Show that micro-kernel based system are usable in practice with good performance

Experiments: -Implement L4-Linux, and compare it with Mk-Linux and native Linux

-Implement mapping related OS extension in user-level of L4-Linux

-Implement L4 on an Alpha 21164 machine to check whether the L4 abstractions are reasonably independent of Pentium platform.

3 L4 Essentials

Two basic concepts:

-Threads

--Activities executing inside an address space.

--Associated with its own pagers.

-Address spaces

--Recursive construction of address spaces.

--Granting, mapping, and un-mapping

--All address spaces are constructed and maintained by user-level servers, called pagers.

--Page faults are propagated via IPC to the pager associated with the faulting thread. The memory management policies can be implemented in user-level.

-I/O ports

--treated as parts of address spaces so that that can be mapped and un-mapped in the same manner as memory pages.

-Hardware interrupts

--handled as IPC.

--Implement all device drivers as user-level servers.

-Exceptions and traps

--handled inside the raising threads.

-Pentium-specific feature

--Small-address-space optimization

4 Linux on Top of L4

4.1 Linux Essentials

Two parts:

-Architecture-independent: 98% of the source codes

-Architecture-dependent: Encapsulates the underlying hardware architecture.

Memory management

-uses a three-level architecture-independent page tables

Interrupt handlers

-Top halves: Highest priority. Can interrupt each other.

-Bottom halves: Lower priority. Can be interrupted by the top halves, but cannot interrupt each other.

4.2 L4Linux

Restrict all modifications to the architecture-dependent part.

Did not tune (optimize) Linux to L4

Single-server approach: provide Linux services via a single Linux server.

The Linux Server

-Acts as a pager for the user process it creates.

-One thread to handle all activities induced by system calls and page faults.

-One thread per top-halve interrupt.

-One thread to handle too bottom-halve interrupt.

Linux User Process

-One task may have several L4 threads.

-Each task is associated with the Linux server as its pager.

System Calls

-System calls are handled by IPC

Signalling

-L4-Linux add additional signal-handler thread to each Linux user process.

Scheduling

-All L4 threads are scheduled by the L4 internal scheduler.

6 Extensibility Performance

Objective

-Show L4-kernel can perform better in some applications

Pipes and RPC

-Implementation of synchronous L4 RPC is 5 times faster than Linux pipe with larger bandwidth.

Virtual Memory Operations

-Memory management can be handled in user-level and become more intelligent.

-The resulting implementation is several times faster than the native Linux.

Cache Partitioning

-Co-existence of time-sharing and real-time memory management.

-Use cache partitioning to reduce the worst-case execution time

##############  
##############  
##############  
02\_11\_02\_exokernel\_p251-engler

# Exokernel

## Introduction

\* Exokernel offers a lower-level exposure of H/W, with no abstraction.

\* Exokernel only securely multiplexes available hardware resources.

\* Applications can benefit greatly from having more control over how machine resources are used to implement higher-level abstractions.

\* Exokernel separates protection from management.

The architecture is as below.

+-----------------+ +-----------------+

| App | | App |

| | | |

+-----------------+ +-----------------+

| Lib OS | | Lib OS |

| | | |

+-----------------+ +-----------------+

+-----------------------------------------------------+

Exokernel

+-----------------------------------------------------+

H/W

## OS Style

\* Run apps of many types

\* Run a dedicated app, e.g., web server, DB server.

\* Library OS

OS is linked into App as a Library OS. Since Lib OS is not trusted by Exokernel, they are free to trust the application.

## Exokernel

\* Exokernel manages resource allocation, multiplexes and exports physical resources securely through a set of low-level primitives. Exokernel separates protection from management.

1. Track ownership of resources

2. Ensure protection by guarding all resource usage or binding points

3. Revoke access to resources

\* Exokernel exports all privileged instructions, hardware DMA capabilities, and machine resources. Each exported operation can be encapsulated within a system call that checks the ownership of any resources involved. Exokernel should only manage resources to the extent required by protection.

## Secure Bindings

\* An exokernel allows library OS to bind to resources using secure bindings. Exokernel provides protection for mutually distrustful Apps.

\* A secure binding is a protection mechanism that decouples authorization from the actual use of a resource. One example is a TLB entry, when a TLB fault occurs the mapping of virtual to physical addresses in a lib OS page table is performed and then loaded into the kernel (bind time), and then used multiple times (access time).

1. Hardware support

2. Cached in exokernel, for instance, an exokernel can use a large software TLB to cache address translations that do not fit in the hardware TLB.

3. Download code into the kernel. The code invoked on every resource access or event to determine ownership and the actions that the kernel should perform. This can avoid expensive crossings.

\* ASH is downloaded into kernel, to do packet filter. A packet comes in, Exokernel analyzes it, and sends it out to the right App. This is more efficient than the Exokernel querying all the server processes on every incoming packet to determine who the packet was for.

##############  
##############  
##############  
02\_16\_01\_xen\_p164-barham

Xen and the Art of the Virtualization

========================================================================

Challenges to build virtual machines:

1. Performance isolation

Process Scheduling

Memory Usage

Network Traffic

Disk Access

2. Support for various OS platforms

3. Minimizing performance overhead

========================================================================

Virtualization Approch

1. Conventional -> Full Virtualization

GuestOS runs without modification

GuestOS cannot access the hardware directly

Problematic for certain privileged instructions (eg. traps)

No real-time guarantees

2. Xen -> Paravirtualization

Modifications to the GuestOS necessary, no modification are required to guest applications

GuestOS runs in parallel with other modified systems

Provides some exposures to the underlying HW

Xen: A high performance resource-managed x86 virtual machine monitor

Multiplexes resources at the granularity of an entire OS

Follows the ideology of Exokernel

As opposed to process-level multiplexing

higher overhead

Target: hosting up to 100 virtual machine instances simultaneously on a modern server

========================================================================

1. Memory Management:

Depending on the hardware supports

Software managed TLB

Associate address space IDs with TLB tages

Allow coexistence of OSes

Avoid TLB flushing across OS boundaries

X86 does not have software managed TLB

guestOS initializes a page from its own memory reservation and registers it with Xen

Xen exists in a 64MB section at the top of every address space to Avoid TLB flush when an guest OS enter/exit Xen

Writes are validated by Xen

2. CPU

X86 supports 4 levels of privileges, generally described as rings, 0 is most privileged to 3 is least

0 for OS, and 3 for applications

Xen doengrades the privileges of OSes

System-call and page-fault handlers registered to Xen

"fast hangdlers" for most exceptions, Xen isn't involved (system call), page faults must be delivered via Xen

Safety is ensured by validating exception handlers when they are presented to Xen

Separation of policy and mechanism

Domain0 hosts the application-level management software:

Creation and deletion of virtual network interfaces and block devices

3. Control Transfer

Hypercall: synchronous calls from a domain to Xen

Analogous to system calls

Allows domains to perform a synchronous software trap into the hypervisor to perform privileged operation

Events: asynchronous notifications from Xen to domains

Replace device interrupts

Lightweight notification of important system events, similar to Unix signal

Event handling can be deferred by domain

4. Data Transfer

Safe indirect way to share I/O devices among OSes

Circular queue accessible by Xen and a domain

5. CPU Scheduling and Timers

Borrowed virtual time scheduling

Allows temporary violations of fair sharing to favor recently-woken domains

Goal: reduce wake-up latency

Xen provides several different types of timers

Real Time (time that always advances regardless of the executing domain)

Virtual Time (time that only advances within the context of the domain)

Wall Clock Time (time that takes in to account local offsets for time zone and DST)

6. Virtual Memory

No shadow pages (VMWare)

Xen provides constrained but direct MMU updates

All guest OSes have read-only accesses to page tables

Updates are batched into a single hypercall

Updates must be validated by Xen

Guest OSes are responsible for allocation and managing pages within their own domain

Xen exists in a generally unused section at the top of every address space to prevent paging out

7. Physical Memory

Reserved at domain creation times

Memory statically partitioned among domains

Does not guarantee contiguous regions of memory

Supports hardware~physical mapping by providing shared translation array readable by all domains

8. Network

Virtual firewall-router attached to all domains

Round-robin packet scheduler

To send a packet, enqueue a buffer descriptor into the transmit rang

Use scatter-gather DMA (no packet copying)

A domain needs to exchange page frame to avoid copying

Page-aligned buffering

9. Disk

Only Domain0 has direct access to disks

Other domains need to use virtual block devices

Use the I/O ring

Reorder requests prior to enqueuing them on the ring

If permitted, Xen will also reorder requests to improve performance

Use DMA (zero copy)

========================================================================

Performance

Close and outperform others

Performance isolation works as expected

Memory usage enough for running 100 domains on a server

##############  
##############  
##############  
02\_16\_12\_cell\_p173-andrus

Cells: A Virtual Mobile Smartphone Architecture

========================================================================

A virtualization architecture for enabling multiple virtual smartphones to run simultaneously on the same physical cellphone in anisolated, secure manner.

Summary

Novel Architecture for a Virtual Phone

How to do away with “overhead overhead”

New way to virtualize devices

Foreground‐Background Usage Modle

How to contribute a low overhead

========================================================================

Motivation

For “everyday computing,” Mobile > Desktop (+Laptop)

Many virtual phones that are isolated on a physical phone

Existing (system) virtualizations

High overhead

Hardware devices

Cells: new, LWT virtulization

========================================================================

Overview of Cells

Virtual Phones

Multiple Multiple Android Android instead instead of multiple multiple OS instances instances

Low overhead

UnionFS, KSM, Form factor ( ll Sma d l) isp ay)

New usage model

Device support

Per‐VP Phone number

Prototype using Real commercial Android phones

Nexus 1, Nexus S with Gingerbread Gingerbread

Scalability

========================================================================

Usage Model

Only allow a single VP, the foreground VP, to be displayed at any time

Background VPs are running in the background but not displayed

Access Right

Three different access right for each device on a phone hardware

No access, shared access, or exclusive access

Assigned statically when a VP is configured

========================================================================

Architecture

A virtual phone has

ID, Kernel I/F, and virtualized virtualized hardware hardware resources resources

Private virtual namespace

Filesystem Path

Virtualized {PIDs, network names, user names}

Eg. The same PIDs can exists if VP ids are different different

Linux provides the PID virtualization

========================================================================

Virtualization

Why virtualized resources?

Guarantee Isolation among VPs

How?

1. Kernel‐level: device namespaces

2. User‐level: user-level device namespace mechanism

========================================================================

Scalability & Security

Scalability

How many VPs can be built on a physical physical phone

Removing overhead per‐VM

UnionFS UnionFS (aufs), KSM, LMK

Security

How Cells isolate a VP from others

Kernel/user‐level namespace namespace

Separate FS view for each VP

No creation creation of device nodes in a VP

UnionFS (aufs)

Stacking file systems

Join the read‐only fs with a writeable files

Read‐only: /data, /system

Device Virtualization

Graphics

Multiplexing FB device driver: mux\_fb

How FB works : mmaps, standard standard IOCTLs, custom IOCTLs

GPU

GPU is already isolated by underlying Linux

Pass‐through through access in each VP context context

Switching the foreground VP, remap memory buffers

Need to access a GPU driver at a somewhat level

Requirements for proprietary GPU driver

Remap the GPU driver’s linear addresses

Re‐initialize On/Off

Ignore power management request

For GPUs without without MMU, the backing backing memory is contiguous

Telephony

Incoming/Outgoing calls

Multiple Multiple Numbers: Numbers: Phone number assigned assigned to each VP

Challenge: One SIM allows just one phone number, Caller ID

Network

Best Practice in desktop/server virtualization

Virtual Virtual wired network network adapter adapter for each virtual virtual machine machine

Configuration in a host

Requirements

Each VP Should be able to independently select its wireless connection, and configure configure it

e.g., Security method, WLAN or 3G

Some application should be able to access WLAN directly

e.g., Location‐based services, AppStore, System Updater

Similar with telephony

Android Android uses wpa\_suppli t can for wireless wireless setup

========================================================================

Experiment

Measuring overhead/power consumption for 5‐VP

VPs config ration uration for UI test

VP1: Angry Bird

VP2: Reckless Racing (3D game)

VP3: Office Suite Pro

VP4: Android Music Player

========================================================================

Conclusion

Problems in existing device virtualization

Pass‐though : other VMs cannot access

Bypass: requires special hardware support

In particular, GPU

Vmware MVP not good for games

Xen backend‐frontend: vendor support needed

========================================================================

Pros

Lightweight, No hypervisor hypervisor

Leveraging existing device drivers

Cons

Large TCB

Different mobile OS on a phone (e.g., Android + iOS)

##############  
##############  
##############  
02\_18\_01\_scheduler\_p95-anderson

Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism

========================================================================

Managing Concurrency Using Threads

User-level library

Management in application’s address space

High performance and very flexible

Lack functionality

Operating system kernel

Poor performance (when compared to user-level threads)

Poor flexibility

High functionality

New system: kernel interface combined with user-level thread package

Same functionality as kernel threads

Performance and flexibility of user-level threads

========================================================================

User-level Threads

Thread management routines linked into application

No kernel intervention == high performance

Supports customized scheduling algorithms == flexible

(Virtual) processor blocked during system services == lack of functionality

I/O, page faults, and multiprogramming cause entire process to block

Kernel Threads

No system integration problems (system calls can be blocking calls) == high functionality

Extra kernel trap and copy and check of all parameters on all thread operations == poor performance

Kernel schedules thread from same or other address space (process)

Single, general purpose scheduling algorithm == lack of flexibility

========================================================================

Kernel Threads Supporting User-level Threads

Can we accomplish system integration by implementing user-level threads on top of kernel threads?

Typically one kernel thread per processor (virtual processor)

User-level thread blocks, so does kernel thread: processor idle

More kernel threads implicitly results in kernel scheduling of user-level threads

Increasing communication between kernel and user-level will negate performance and flexibility advantages of using user-level threads

No, Also:

No dynamic reallocation of processors among address spaces

Cannot ensure logical correctness of user-level thread system built on top of kernel threads

========================================================================

New Thread Implementation

Goal: System with the functionality of kernel threads and the performance and flexibility of user-level threads.

High performance without system calls.

Blocked thread can cause processor to be used by another thread from same or different address space.

No high priority thread waits for processor while low priority runs.

Application customizable scheduling.

No idle processor in presence of ready threads.

Challenge: control and scheduling information distributed between kernel and application’s address space.

========================================================================

Virtual Multiprocessor

Application knows how many and which processors allocated to it by kernel.

Application has complete control over which threads are running on processors.

Kernel notifies thread scheduler of events affecting address space.

Thread scheduler notifies kernel regarding processor allocation.

========================================================================

Scheduler Activations

Vessels for running user-level threads

One scheduler activation per processor assigned to address space.

Also created by kernel to perform upcall into application’s address space

“Scheduler activation has blocked”

“Scheduler activation has unblocked”

“Add this processor”

“Processor has been preempted”

Result: Scheduling decisions made at user-level and application is free to build any concurrency model on top of scheduler activations.

========================================================================

New Thread Implementation

Goal: System with the functionality of kernel threads and the performance and flexibility of user-level threads.

High performance without system calls.

No high priority thread waits for processor while low priority runs.

Blocked thread can cause processor to be used by another thread from same or different address space.

Application customizable scheduling.

No idle processor in presence of ready threads.

Challenge: control and scheduling information distributed between kernel and application’s address space.

========================================================================

Processor allocation

Goal: No idle processor in presence of runnable threads.

Processor allocation based on available parallelism in each address space.

Kernel notified when:

User-level has more runnable threads than processors

User-level has more processors than runnable threads

Kernel uses notifications as hints for its actual processor allocation.

========================================================================

Implementation

Scheduler activations added to Topaz kernel thread management.

Performs upcalls instead of own scheduling.

Explicit processor allocation to address spaces.

Modifications to FastThreads user-level thread package

Processing of upcalls.

Resume interrupted critical sections.

Pass processor allocation information to Topaz.

========================================================================

Performance

Thread performance without kernel involvement similar to FastThreads before changes.

Upcall performance significantly worse than Topaz threads.

Untuned implementation.

Topaz in assembler, this system in Modula-2+.

Application performance

Negligible I/O: As quick as original FastThreads.

With I/O: Performs better than either FastThreads or Topaz threads.

##############  
##############  
##############  
02\_18\_02\_lottery\_waldspurger

Lottery Scheduling: Flexible Proportional-Share Resource Management

========================================================================

Significance of the Paper

This paper presents a randomized scheduling algorithm that is easy to implement and facilitates modular resource manangement. It provides efficient and responsive control over the relative execution rates of computations and can be generalized to manage diverse resources.

========================================================================

Key Ideas

Instead of using absolute priority values, lottery tickets can be used to represent resource rights in an abstract, relative and uniform way.

========================================================================

Important Issues

1. Lotteries

list-based, sequential search

list-based with move-to-front heuristics

tree of partial ticket sums

Starvation free: any client with a non-zero number of tickets will eventually win a lottery.

2. Tickets Transfer

using ticket transfer to redirects the client's resource rights to the server that is computing on its behalf.

Merits: solve priority inversion in a way similar to priority inheritence.

3. Tickets Inflation

inflation and deflation can be used to provide flexible control for some applications (e.g. Monte-Carlo algorithm, graphics-intensive programs).

4. Ticket Currencies

Use currency abstraction to flexibly isolate or group tasks.

5. Compensation Tickets

Compensation tickets are used to help a client who does not consume its entire allocated quantum to get its entitled share.

========================================================================

Experiments

The experiments cited in this paper emphasize on the fact that throughput and response time of different clients closely tracked to their ticket allocation ratio:

With scheduling quantum of 100 milliseconds, fairness can be achieved over 8 seconds intervals. With scheduling quantum of 10 miliseconds, fairness can be achieved over subsecond intervals, provided that scheduler overhead can be bounded.

Their conclusion:

Conventional priority mechanisms have drawbacks (arbitrarily assigned priorities, crude control over scheduling, inadequate for insulating resource allocation policies for different modules, difficult to abstract inter-module priority relationships).

Lottery scheduling, on the other hand, efficiently implements proportional-share resource management and provides excellent support for modular resource management. It can also be generalized to manage many diverse resources, such as I/O bandwidth, memory, and mutex.

========================================================================

Questions

1. If priority themselves are rarely meaningfully assigned, how about ticket allocations?

2. For many users, it is more desirable to have predicated, regular response time.

3. It suggested a manager thread be allocated a fixed percentage of funding to cause it be invoked periodically. This cannot be assured under randomization.

4. Normally, we consider a system to be starvation free only if it has some mechanisms to guarantee that any process won't be kept waiting for an unreasonably long period of time. In this sense, lottery scheduling cannot guarantee deadlock free.

##############  
##############  
##############  
02\_23\_02\_osdi14-paper-belay

#1: Protection and direct HW access through virtualization

#2: Execution model for low latency and high throughput

Analogy with VM model

| VM | Protection domain | IX |

| :----:|:----------------------:| -----:|:--------------:|

| Apps | Ring 3 | Apps |IX Control plane|

| OS | Guest(non-ROOT) Ring 0 | IX | |

| ------| ------ | ----- | |

| VMM | Host(ROOT) 0 | NIC |Kernel (Dune) |

Important assumption:

"Small Requests": Most requests can be processed by the CPU and within memory, taking short time

Key features:

\* Separation of Control and Data Plane

\* The IX Execution Pipeline

\* Design (1): Run-to-completion

Improves data-cache locality

Remove scheduling unpredictability

\* Design (2): Adaptive Batching

Improve instruction-cache locality and prefetching

Other features:

\* Design (3): Flow consistent hashing

Synchronization & coherence free operation

\* Design (4): Native zero-copy API

Flow control exposed to application

\* Libix: Libevent-like event-based programming

\* IX prototype implementation

Dune, DPDK, LWIP, ~40K SLOC of kernel code

From previous homework:

IX has been optimized for a workload where application processing per request is relatively small, and where application processing can be done in a run-to-completion model. As a result, the thread handling a packet can execute through the networking stack (data plane), call up into the application, and then call back down into the networking stack. By optimizing the networking layer, it greatly reduces overhead and overall execution time for processing a packet in the server.

IX Conclusion

\* A protected dataplane OS for datacenter applications with an event-driven model and demanding connection scalability requirements

\* Efficient access to HW, without sacrificing security, through virtualization

\* High throughput and low latency enabled by a dataplane execution model

--- Problem: 1980s Sobware Architecture ---

\* Berkeley sockets, designed for CPU time sharing

\* Today’s large-scale datacenter workloads:

Hardware: Dense Multicore + 10 GbE (soon 40)

- API scalability critical!

- Gap between compute and RAM -> Cache behavior matters

- Packet inter-arrival times of 50 ns

Scale out access patterns

- Fan-in -> Large connection counts, high request rates

- Fan-out -> Tail latency matters!

^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

##############  
##############  
##############  
02\_25\_01\_p181-mckusick

A Fast File System for UNIX

FFS improvement is based on the assumption that disk seek is expensive.

Optimizes read and write simultaneously

# Solution: Cylinder groups and larger block size (with fragment to reduce waste)

FFS allows parameterizing a file system. Each file system is parameterized so that it can be adapted to the characteristics of the disk on which it is placed. To ease the calculation of finding rotationally optimal blocks, the superblock contains rotational layout tables.

# Other improvement: two distinct layout policies, global policy and local policy.

Global allocator allocates cylinder groups

Local allocator allocates blocks within cylinder group.

Introduced new features:

\* Long file names

\* advisory locks on files

\* Symbolic links

\* More robust system call

\* Quota mechanism (90% soft quota and hard quota)

##############  
##############  
##############  
02\_25\_02\_p1-rosenblum

The Design and Implementation of a Log Structured File System

Used in Sprite, improvement done by caching

\* Larger memory sizes mean larger caches

Caches will capture most read accesses

Disk traffic will be dominated by writes

Caches can act as write buffers replacing many small writes by fewer bigger writes

Key issue is to increase disk write performance by eliminating seeks

Instead of optimizing read (it can be optimized with cache easily), Sprite(LFS) optimizes write (Wite needs to be fast to guarantee cache not getting full.)

Segments for free space management (segment summary is used when trying to determine data validity).

Dynamically changing \_\_inode map\_\_ is used to determine current inode location. Unlike Unix FFS where inode is in fixed location.

\* Inode map maintains the location of each i-node

Blocks at various location on disk

Active blocks are cached in main memory

\* A fixed checkpoint region on each disk contains the addresses of all inode map blocks

\* Faster recovery after a crash

All blocks that were recently written are at the tail end of log

No need to check whole file system for inconsistencies

\* Log

Contains data blocks, i-node blocks, blocks of i-node map, segment summaries and directory change log

\* Checkpoint area

Contains

Address of end of log at checkpoint time

Addresses of all i-node map blocks at checkpoint time

--- Segments

Must maintain large free extents for writing new data

Disk is divided into large fixed-size extents called segments (512 kB in Sprite LFS)

Segments are always written sequentially from one end to the other

Old segments must be cleaned before they are reused

See attached ppt for more details.##############  
##############  
##############  
03\_01\_01\_softupdates

Soft Updates

============

What is the problem? Metadata consistency

Suppose you delete a file and subsequently crash

\* If file system imposes write order with synchronous writes (FFS)

- Free map, count will probably be wrong

- Inode may have link count 1 but no directory entry

- New inode may point to block of deleted file

Chunks of a deleted file show up in new unrelated files

- fsck can fix 1 and 2 but not 3

\* If file system imposed no ordering (Linux ext2fs)

- Same problems as FFS, plus:

- Inode may have been recycled before directory written

Old directory will contain link to new file

- Indirect block could get reused before old inode cleared

Random file data will get interpreted as block pointers

- fsck cannot fix these

\* Log structured file system/shadow paging (LFS)

- May end up with pointer to old file system state before the delete

But always consistent--blocks never overwritten while pointed to

- fsck/mount could roll forward log to lose less state

\* Journaling/write-ahead logging (XFS)

- Can end up with same problems as ext2fs

- But fsck replays log to bring file system into consistent state

Advantages/disadvantages of FFS:

+ create/delete operations always survive after a crash

- Chunks of deleted files resurface in unexpected places

- fsck takes a long time

What are advantages/disadvantages of LFS & Journaling

\* LFS

- Cleaner overhead: many open questions in how to clean properly

\* Journaling

- Must perform more metadata writes (once in log, once in file system)

- Chunks of deleted files can resurface? (Probably--no solution in paper)

\* Both

- Contention for lock at end of log?

- fsync must wait for other files' data

+ Most operations don't require synchronous disk writes

+ fsck/mount is fast

+ True atomic rename

- Create/delete requires writes even for short-lived files

Without logging, what is correct order in which to write info to disk?

1. Never write pointer before initializing the structure it points to

2. Never reuse a resource before nullifying all pointers to it

3. Never clear last pointer to live resource before setting new one

So what are goals of this paper?

\* Eliminate most synchronous disk writes

\* Make fsck much faster, or at least don't wait for it to restart

\* Fix resurrected data problem

What is scheduler-enforced ordering?

\* Usually, scheduler orders requests on disk queue to optimize performance

\* Enforced ordering says block A must be written before block B

So now schedule may be able to do A & B in same pass

System calls can return before any of the writes have completed

+ Gives 30% performance improvement

- But makes writes async, not delayed -- does not reduce # of writes

Create 2 files in same dir, second create will wait for locked dir block

Create then delete file, delete will wait for locked blocks

What about using NVRAM

\* Use memory that can survive a crash or power failure

- Requires special hardware

- If machine fails, can't just move disk to new machine (need NVRAM state)

Straw man: Keep inter-buffer dependencies?

Delay all writes (like ext2fs)

But keep dependency ordering in buffer cache

Order all writes to avoid bad ext2fs states

Problem: Dependency cycles and false sharing

Several inodes or directory entries in same block

Example: figure 1

- Create file A, delete file B, same dir/inode blocks

Can't write directory until inode A initialized

Can't write inode B until pointer cleared in directory

Why can't you write inode B until pointer cleared? (see footnote 2)

- Have to make sure it isn't reallocated

- Fsck would have to check every directory entry before restart (slow)

- Otherwise, might get incorrect link count

deleting file would clear inode even when another link existed!

Problem: Crash might occur between ordered but related writes

E.g., summary information wrong after block freed

Problem: Block aging

Block that always has dependency will never get written back

What are soft updates?

\* Data structure for each updated field or pointer, contains:

- old value

- new value

- list of updates on which this update depends

\* Can write blocks in any order

- But must temporarily undo updates with pending dependencies

- Must lock rolled-back version so applications don't see it

- Choose ordering based on disk arm scheduling

\* p. 134: other dependencies can "be more efficiently handled by

postponing in-memory updates until after the updates on

which they depend reach stable storage."

Example: Create A delete B revisited

\* See figure 2 - requires directory to be written twice?

\* What if inode written first?

\* How many writes required in XFS?

3 - one for log, one for inode block, one for directory block

\* How many in LFS?

All part of one big write (+ checkpoint for many updates)

Four main structural changes requiring sequenced updates:

1. Block allocation

Must write: disk block, free map, pointer

Req: Disk block & free map must be written before pointer

Use: Undo/redo on pointer (+ possibly file size)

2. Block deallocation

Must write: previous pointer, free map

Just update free map after pointer written

Or, immediately deallocate blocks if pointer was never written to disk

How do you know? (there will be a dependency structure)

3. Link addition

Must write: Directory entry, inode, and free map (if new inode)

Req: inode and free map must be written before dir entry

Use: Undo/redo on i-number in dir entry (ignore entries w. ino 0)

4. Link removal

Must write: Dir. entry, inode and free map (if nlinks==0)

Req: Decrement nlinks only after pointer cleared

Use: Clear directory entry immediately

decrement in-memory nlinks once pointer written

If directory entry was never written, decrement immediately

Issues

\* fsync

- Must ensure names for files are also stably on disk

- Must ensure names of parent directories are stably on disk!

keep data structures to track such dependencies

recurse to higher level directories

but parent directories can be written in any order, so still

good disk arm scheduling

\* unmounting a file system

- May need to flush dirty buffers multiple times

\* memory usage

- Deleting large directory trees--memory goes faster than disk

- Cap number of directory structures allocated

\* useless write-backs

- syncer wrote many blocks at once--worst case

even with circular dependencies better to write one at a time

- LRU evection scheme tweaked to know about dependencies

\* fsck - split into to parts: Foreground and Background

- Quick: What must be done before remounting FS

Need to make sure per-cylinder summary info makes sense

Recompute free block/inode counts from bitmaps -- very fast

Will leave FS consistent, but might leak disk space

- Full: Traditional fsck operations

May be done in background after mounting to recuperate free space

Must be done in forground after a media failure

Differences from traditional FFS fsck

May have many, many inodes with non-zero link counts

Don't stick them all in lost+found (unless media failure)

Performance

\* Figure 3: Is this what we expect? yes

- Why the dip after 64KB?

Write coalescing in 64K chunks

Indirect block kicks in at 104K

- How would you expect LFS and XFS to do here?

LFS - probably better (no seeks)

XFS - For small files, between conventional and soft-updates,

since more writes needed. For larger files, possibly

better that soft updates, since extent maps probably

avoid indirect blocks.

\* Figure 4: How does soft updates beat No Order?

- Soft updates defers the work from actually removing files. With

soft updates, unlink system call can actually return even if the

inode and indirect block are not in the buffer cache! No Order

must at least read inode and indirect block into memory (can see

big dip at 104K where indirect block kicks in).

\* Figure 5: How does soft-updates beat no order?

- Artifact of benchmark:

reallocation is process of coalescing writes into 64K chunks

sometimes relocate blocks to do this. May be farther from

indirect block if old space freed and made available

\* Figure 6: Pretty good, yes?

\* Figure 7:

How does concurrency hurt convenional? less locality

Helps soft-updates because more flexibility in disk scheduling

too much concurrency just adds overhead

\* Figure 8: journaling has cost

Limitations of soft updates:

\* Not as general as logging, e.g., no atomic rename

might be hard to use b-trees requiring atomic updates to several blocks

\* Metadata updates might proceed out of order

create /dir1/a than /dir2/b

after crash /dir2/b might exist but not /dir1/a

\* Suppose you rename a directory called dir/short:

% mv dir/short dir/much\_longer\_name\_that\_does\_not\_fit

FFS needs to put longer name in a different directory chunk

If crash before old name removed, might have two names for directory

Can soft updates correct this? (don't know...)

##############  
##############  
##############  
03\_01\_02\_rio\_p74-chen

The Rio File Cache: Surviving Operating System Crashes

SUMMARY

This paper describes the Rio File Cache, which uses a combination of hardware- and software-based protection techniques to make the contents of RAM persistent across warm reboots, thereby eliminating the need for the periodic writes to disk traditionally done for the sake of reliability. Their technique enables systems to have a RAM filesystem cache that is at least as fast as a write-back cache while

being at least as safe as a write-through cache.

They implemented their prototype on DEC Alphas running Digital UNIX - an important point because the buffer cache on these systems is always located in the same block of physical RAM and is never paged. So their problem essentially reduces to making sure nothing messes with this area of RAM during a crash and during warm reboot. (OK, they also maintain their own "registry" of metadata that normally wouldn't be so easy to find after reboot, so what goes for the buffer cache goes for this registry, as well.) They use a combination of hardware VM protection and program re-writing (inserting address checks before instructions that might write to the protected area) to implement protection.

----------------------------------------------------------------------

EVALUATION

The greatest strength of this approach is its simplicity: it shows that even during a software system crash, volatile RAM serves as reliable repository for files. The approach is high-performance, and works to protect against software system crashes as well as some memory corruption errors (when protection is included).

In my opinion, this approach has one fatal flaw, depending on asystem's reliability requirements: RIO presumes that a hardware error never occurs. Their approach appears sufficient to protect against software crashes, but they weasel around hardware errors by saying, if the system board fails, it should be possible to move the memory board to a different system without losing power or data. Their approach does nothing to protect against presumably rare catastrophic

hardware failures, especially those that affect memory or the memory bus. But maybe I'm being a stick in the mud.

FYI, the authors responses to common objections are at:

http://www.eecs.umich.edu/Rio/faq.html

----------------------------------------------------------------------

The most important idea turns out to be that of the warm reboot. It is rather unexpected that this idea does so well. Warm reboots are the only new idea introduced in the paper. The others have minimal impact and importance.

----------------------------------------------------------------------

The originality of the idea sounds interesting. The results in performance seem to show that the implementation was successful. I am curious to see how the system performs in a real environment since many of the experiments were done introducing artificial failures so that the system might crash.

----------------------------------------------------------------------

Another cool trick done on a commodity operating system - retrofit always deserves high marks. I wonder how much of this technique depends on the fact that digital UNIX keeps its buffer cache in the same locating in physical memory on every boot, though. How applicable would this be to Linux kernels, or Windows kernels?

----------------------------------------------------------------------

The clear strengths of the approach are that the resulting system can combine both reliability and performance. The filesystem modifications allow for a recovery after system crash of file data \*without\* the need to constantly write to disk. Essentially, the authors use the protection mechanism of the virtual memory system to prevent unauthorized modifications to the file cache. I don't actually consider this a very "language-based" system, but it serves as a good example of an application of memory safety.

One aspect of the paper that was unconvincing was the testing

portion, which introduced seemingly arbitrary faults with no clear explanation for why they were chosen. The authors correctly point out the difficulty in simulating arbitrary system failure, but the work really focuses on a narrow set of faults in the grand scheme of things. They rule out (for good reason) power failure and hardware failures, but don't provide any good evidence for the commonality or nature of OS faults that result in warm reboot.

----------------------------------------------------------------------

The goal of the Rio project is to use memory as a reliable file store across OS crashes. Their approach is to protect the memory from interference during normal operation, even from the kernel, and leave information for bootstrapping the memory cache in a well known location in physical memory. They preferred to achieve memory protection through hardware means, but they also allowed for the use of program rewriting (called "code patching" in the paper) to achieve protection (at a performance cost). First and foremost, with their system they provide fast and reliable storage at low cost (as opposed to using a solid state disk that has similar characteristics but is priced out of the budget of most people).

Second, they use the technique of fault insertion, where they transform the code to introduce faults. This seems novel to me.

Like the SPIN paper, they use a dead architecture. Moreover, their system won't port to x86, the most popular architecture, because the memory isn't usable across warm boots. This severely limits Rio's usability in a practical sense.

Furthermore, in my experience (which is admitedly anecdotal), most sudden faults come from faulty disk drives and power supplies, not bad kernel modules (excepting faults in DFS on Solaris systems).

----------------------------------------------------------------------

The authors make a case for using memory as a persistent store by making it complex to access particular regions. Their argument is that disks are perceived to be safer because of the inherent complexity in accessing them. They therefore try to ensure that memory regions are protected by mechanisms which force software to go through steps (which makes the operation possibly as complex) so that disk and

memory are equally likely to be corrupted in crashes. Somehow this does not seem to be very convincing. Disk are considered much more reliable probably because we know that they are not a volatile store. So, the entire motivation for the paper seems flimsy at a first glance.

----------------------------------------------------------------------

RESEARCH

I wonder if using RIO with protection created by SFI or some other scheme would give an even greater performance boost. I know they claim that RIO is all you would need, but would some sort of hybrid asynchronous/RIO approach give you some higher level of protection against hardware faults, without all of the RIO performance gains.

----------------------------------------------------------------------

It would be interesting to know if there are any other software based methods for achieving the same isolation results.

----------------------------------------------------------------------

I'm curious to know what data structures on Linux might be made persistent across reboots in this way. Linux kernel memory isn't paged, so it seems there might be some chance. However, I fear that most interesting data structures may be heap-allocated, and may consequently not be in the same location on each boot.

----------------------------------------------------------------------

Extension of this technique to other modules and the kernel in general would be nice. Data other than just that of the filesystem gets corrupted on system failure and new techniques such as microreboot are interesting ideas for compartmentalizing, protecting, and restoring systems while keeping system state alive and well.

----------------------------------------------------------------------

I would like to see an implementation on other operating systems and architectures, such as Solaris and Linux (except on x86, of course). Also, perhaps this could be adapted to x86 by modifying the way the kernel handles faults -- perhaps it could wipe all non-cache kernel memory and re-initialize by itself, instead of requiring and actual boot. I would also like to see this sort of protection for other things across boots, such as network connections. In terms of the software protection mechanisms, it might be useful to see if static

analysis techniques can improve the performance (if its necessary -- it seems that all major architectures implement the hardware protection needed).

----------------------------------------------------------------------

Would be interesting to explore what issues arise and what can be done in the case of deliberate faults in the software that corrupt memory.

========================================

Another version:

Key ideas: avoid reliability-related file system writes to disk; instead, make RAM used for files reliable by:

1) adding battery backup to deal with power failures

2) protecting the the file cache using either SFI or VM tricks

3) supporting "warm reboot" in which old file cache memory is restored following a failure.

Point 2) uses VM in that all all accesses to file cache pages go through the virtual memory system (i.e. the TLB and page tables, though they just say the TLB); we do not permit direct physical addressing. This way, we can set the pages as read-only except when they must actually be written to by the file cache subsystem.

Using VM, rather than SFI, is great since it incurs no additional overhead (except the cost of actually flipping the bits on the table): this is because the kernel is already running in supervisor mode, and thus does not require a trap. Using SFI would add extra overhead.

Protection in general is more useful than disk-related reliability mechanisms because any illegal attempt to write to a file cache page will be fail-stop: the system will panic immediately, rather than corrupt the datastructure and permit bad data eventually to get written to disk.

Point 3) requires some reworking of the way the file cache is written so that the datastructure is always in a stable state. This can be done by making changes to "shadow copies" and then atomically linking the change into the persistent data structure.

This is basically like general-purpose persistence, but without language support: no transactions, no garbage collection. However, language support would make a lot of sense, particularly transactions. The GC part is not needed since all RAM is persistent

(though not all CONSISTENT), and thus we don't need the reachability part.