6410: Microkernels

Presented By: Dan Williams

(some content borrowed from previous years: Ken Birman (2007) and Saikat Guha (2005))
Outline

- Background
- Mach
- L4
- Summary
A short history of kernels

- Early kernel: a library of device drivers, support for threads
- Monolithic kernels: Unix, VMS, OS 360…
  - Unstructured but fast…
  - Over time, became very large
- Pure microkernels: Mach, Amoeba, Chorus…
  - OS as a kind of application
- Impure microkernels: Modern Windows OS
  - Microkernel optimized to support a single OS
  - VMM support for Unix on Windows and vice versa
Monolithic Kernels vs Micro Kernels

User-Kernel Split
Monolithic Kernels vs Micro Kernels

Monolithic Kernel

- Filesystem
- Kernel
- Pipe

Processes:
- Process 1
- Process 2
- Process 3
- Process 4
- Process 5
Monolithic Kernels vs Micro Kernels

Microkernel
Microkernels

- Minimal services
- Usually threads or processes, address space, and inter-process-communication (IPC)
- User-space filesystem, network, graphics, even device drivers sometimes
The great μ -kernel debate

• How big does it need to be?
  • With a μ -kernel protection-boundary crossing forces us to
    – Change memory-map
    – Flush TLB (unless tagged)
  • With a macro-kernel we lose structural protection benefits and fault-containment

• Debate raged during early 1980’s
Monolithic Kernels: Advantages

- Kernel has access to everything
  - All optimizations possible
  - All techniques/mechanisms/concepts can be implemented
- Extended by simply adding more code
  - Linux has millions of lines of code
- Tackle complexity
  - Layered kernels
  - Modular kernels
  - Object oriented kernels. Do C++, Java, C# help?
Microkernels: Advantages

• Minimal
  • Smaller trusted computing base
  • Less error-prone
  • Server malfunction easily isolated

• Elegant
  • Enforces modularity
  • Restartable user-level services

• Extensible
  • Different servers/APIs can exist
Microkernels

- 1\textsuperscript{st} generation
  - Mach, Chorus, Amoeba, L3
- 2\textsuperscript{nd} generation
  - Spin, Exokernel, L4
Papers

- The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System
  - Young et al.
  - Mach microkernel
  - SOSP 1987
- The Performance of µ-Kernel-based Systems
  - Härtig et al.
  - L4 microkernel
  - SOSP 1997
Outline

- Background
- Mach
- L4
- Summary
Summary of First Paper

• Summary of Mach
• Memory object
  • Manage system services like network paging and filesystem support outside the kernel
  • Kernel acts as cache manager
• Memory via communication
• Performance
  • Does not prohibit caching of physical memory
  • More space for caching without copying
Mach Abstractions

- Task
  - Basic unit of resource allocation
  - Virtual address space, communication capabilities
- Thread
  - Basic unit of computation
- Port
  - Communication channel for IPC
  - Need port capability
- Message
  - May contain port capabilities, pointers
- Memory Object
Messages and Ports

msg_send(message, option, timeout)
   Send a message to the destination specified in the message header.

msg_receive(message, option, timeout)
   Receive a message from the port specified in the message header, or the default group of ports.

msg_rpc(message, option, rcv_size, send_timeout, receive_timeout)
   Send a message, then receive a reply.

port_allocate(task, port)
   Allocate a new port.

port_deallocate(task, port)
   Deallocate the task’s rights to this port.

port_enable(task, port)
   Add this port to the task’s default group of ports for msg_receive.

port_disable(task, port)
   Remove this port from the task’s default group of ports for msg_receive.

port_messages(task, ports, ports_count)
   Return an array of enabled ports on which messages are currently queued.

port_status(task, port, unrestricted, num_msgs, backlog, receiver, owner)
   Return status information about this port.

port_set_backlog(task, port, backlog)
   Limit the number of messages that can be waiting on this port.
Virtual Memory

vm_allocate(task, address, size, anywhere)
   Allocate new virtual memory at the specified address or anywhere space can be found (filled-zero on demand).

vm_deallocate(task, address, size)
   Deallocate a range of addresses, making them no longer valid.

vm_inherit(task, address, size, inheritance)
   Specify how this range should be inherited in child tasks.

vm_protect(task, address, size, set_max, protection)
   Set the protection attribute of this address range.

vm_read(task, address, size, data, data_count)
   Read the contents of this task’s address space.

vm_write(task, address, count, data, data_count)
   Write the contents of this task’s address space.

vm_copy(task, src_addr, count, dst_addr)
   Copy a range of memory from one address to another.

vm_regions(task, address, size, elements, elements_count)
   Return a description of this task’s address space.

vm_statistics(task, vm_stats)
   Return statistics about this task’s use of virtual memory.
External Memory Management

- No kernel-based filesystem
  - Kernel is just a cache manager
- Memory object
  - Aka “paging object”
- Pager
  - Task that implements memory object
External Memory Management

vm_allocate_with_pager(task, address, size, anywhere, memory_object, offset)
   Allocate a region of memory at the specified address. The specified memory object provides the initial data values and receives changes.

• Call by application program to cause a memory object to be mapped into its address space

pager_init(memory_object, pager_request_port, pager_name)
   Initialize a memory object.

pager_data_request(memory_object, pager_request_port, offset, length, desired_access)
   Requests data from an external data manager.

pager_data_write(memory_object, offset, data, data_count)
   Writes data back to a memory object.

pager_data_unlock(memory_object, pager_request_port, offset, length, desired_access)
   Requests that data be unlocked.

pager_create(old_memory_object, new_memory_object, new_request_port, new_name)
   Accept responsibility for a kernel-created memory object.

• Calls made by kernel on data manager
External Memory Management

- pager_data_provided(pager_request_port, offset, data, data_count, lock_value)
  Supplies the kernel with the data contents of a region of a memory object.
- pager_data_lock(pager_request_port, offset, length, lock_value)
  Restricts cache access to the specified data.
- pager_flush_request(pager_request_port, offset, length)
  Forces cached data to be invalidated.
- pager_clean_request(pager_request_port, offset, length)
  Forces cached data to be written back to the memory object.
- pager_cache(pager_request_port, may_cache_object)
  Tells the kernel whether it may retain cached data from the memory object even after all references to it have been removed.
- pager_data_unavailable(pager_request_port, offset, size)
  Notifies kernel that no data exists for that region of a memory object.

- Calls made by data manager on Mach kernel to control use of memory object
(Copy-on-Write) Filesystem Example

```c
char *file_data;
int i, file_size;
extern float rand(); /* random in [0,1] */

/* Read the file -- ignore errors */
fs_read_file("filename", &file_data, file_size);

/* Randomly change contents */
for (i = 0; i < file_size; i++)
    file_data[(int)(file_size*rand())]++;

/* Write back some results -- ignore errors */
fs_write_file("filename", file_data, file_size/2);

/* Throw away working copy */
vm_deallocate(task_self(), file_data, file_size);
```

- Read file maps file into address space
- Explicitly write contents back to file
(Copy-on-Write) Filesystem Example

```c
return_t fs_read_file(name, data, size)
    string_t name;
    char **data;
    int *size;
{
    port_t new_object;

    /* Allocate a memory object (a port), */
    /* and accept request */
    port_allocate(task_self(), &new_object);
    port_enable(task_self(), new_object);

    /* Perform file lookup, find current file size,*/
    /* record association of file to new_object */
    ...

    /* Map the memory object into our address space*/
    vm_allocate_with_pager(task_self(), data, *size,
                            TRUE, new_object, 0);

    return(success);
}
```

- Server maps file into own address space
- Kernel will issue pager_init
(Copy-on-Write) Filesystem Example

```c
void pager_data_request(memory_object, pager_request,
                         offset, size, access)
    port_t memory_object;
    port_t pager_request;
    vm_offset_t offset;
    vm_size_t size;
    vm_prot_t access;
{
    char *data;

    /* Allocate disk buffer */
    vm_allocate(task_self(), &data, size);

    /* Lookup memory_object; find actual disk data*/
    disk_read(disk_address(memory_object, offset),
              data, size);

    /* Return the data with no locking */
    pager_data_provided(pager_request, offset, data,
                         size, VM PROT NONE);

    /* Deallocate disk buffer */
    vm_deallocate(task_self(), data, size);
}
```

- Give memory to kernel to act as cache
Lots of Flexibility

- E.g. consistent network shared memory
  - Each client maps X with shared pager
  - Use primitives to tell kernel cache what to do
    - Locking
    - Flushing
Problems of External Memory Management

- External data manager failure looks like communication failure
  - e.g. need timeouts
- Opportunities for data manager to deadlock on itself
Performance

• Does not prohibit caching
• Reduce number of copies of data occupying memory
  • Copy-to-user, copy-to-kernel
  • More memory for caching
• “compiling a small program cached in memory ... twice as fast”
• I/O operations reduced by factor of 10
• Context switch overhead?
Outline

- Background
- Mach
- L4
- Summary
Context Switches

Monolithic Kernel

Multiple User Address Spaces

Kernel Address Space

User Process

File System

IPC

Process Management

Network Stack

Synchronization

Microkernel

Multiple User Address Spaces

Kernel Address Space

User Process

Network Stack

IPC

Process Management

2

1

2

1

4

3
The Performance of μ-Kernel-based Systems

- Evaluates an L4 based system
  - Second generation microkernel
- Ports Linux to run on top of L4
- Suggests improvements
The L4 Microkernel

- Similar to Mach
  - Started from scratch, rather than monolithic
  - More strictly minimal
- Uses user-level pagers
- Tasks, threads, IPC
<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>filesystem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>network</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L4 Backend</td>
</tr>
<tr>
<td>L4 Kernel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
L4Linux

• Linux source has two cleanly separated parts
  • Architecture dependent
  • Architecture independent

• In L4Linux
  • Architecture dependent code is modified for L4
  • Architecture independent part is unchanged
  • L4 not specifically modified to support Linux
L4Linux (continued)

- Linux kernel as L4 user service
  - Runs as an L4 thread in a single L4 address space
  - Creates L4 threads for its user processes
  - Maps parts of its address space to user process threads (using L4 primitives)
  - Acts as pager thread for its user threads
  - Has its own logical page table
  - Multiplexes its own single thread (to avoid having to change Linux source code)
L4Linux – System Calls

- The statically linked and the shared C libraries are modified
  - System calls in the lib call the Linux kernel using IPC
- For unmodified native Linux applications there is a “trampoline”
  - The application traps
  - Control bounces to a user-level exception handler
  - The handler calls the modified shared library
- Binary compatible
A note on TLBs

- Translation Lookaside Buffer (TLB) caches page table lookups
- On context switch, TLB needs to be flushed
- A tagged TLB tags each entry with an address space label, avoiding flushes
- A Pentium CPU can emulate a tagged TLB for small address spaces
Performance – The Competitors

- Mach 3.0
  - A “first generation” microkernel
  - Developed at CMU
  - Originally had the BSD kernel inside it
- L4
  - A “second generation” microkernel
  - Designed from scratch
Performance – Benchmarks

- Compared the following systems
  - Native Linux
  - L4Linux
  - MkLinux (in-kernel)
    - Linux ported to run inside the Mach microkernel
  - MkLinux (user)
    - Linux ported to run as a user process on top of the Mach microkernel
Performance - Microbenchmarks

![Diagram showing microbenchmark results.]

Figure 6: *lmbench* results, normalized to native Linux. These are presented as slowdowns: a shorter bar is a better result. *[lat]* is a latency measurement, *[bw⁻¹]* the inverse of a bandwidth one. Hardware is a 133 MHz Pentium.
Performance - Macrobenchmarks

- AIM Benchmark Suite VII simulates “different application loads” using “Load Mix Modeling”.

- This benchmark has fallen out of favor but included various compilation tasks.

- Tasks are more representative of development in a systems lab than production OS in a web farm or data center.
Figure 9: *AIM Multiuser Benchmark Suite VII*. Jobs completed per minute depending on *AIM* load units. (133 MHz Pentium)
Performance – Analysis

- L4Linux is 5% - 10% slower than native for macrobenchmarks
- User mode MkLinux is 49% slower (averaged over all loads)
- In-kernel MkLinux is 29% slower (averaged over all loads)
- Co-location of kernel is not enough for good performance
L4 is Proof of Concept

- Pipes can be made faster using L4 primitives
- Linux kernel was essentially unmodified
  - Could be optimized for microkernel
- More options for extensibility
Outline

- Background
- Mach
- L4
- Summary
Summary

- Microkernel has attractive properties
  - Extensibility benefits
  - Minimal/elegant
- Microkernel can perform well