μ-kernels

- The idea of μ-kernel is to minimize the kernel. I.e. to implement outside the kernel whatever possible.
- The μ-kernel concept is very old: Brinch Hansen's (1970) and Hydra (1974)

Advantages

- Fault isolation (independence). Kernel malfunction can cause a reboot. Less code is executed in Kernel Mode.
- More flexible. Different strategies and APIs (Application Programming Interface) can coexist in the system.
- Kernel recompilation is less needed when changes are done in OS. Also, there is no need to reboot the computer when a change outside the kernel is done.

Disadvantages

- Sometimes, when integrating code into the most existing μ-kernels, the performance of the code will be upgraded.
- Restricted flexibility. Non-integrated code will be rarely used, because of its bad performance. (weak argument).

Primitives in a μ-kernel

- Determining criterion: functionality
  - A concept is OK inside the μ-kernel, if left outside would prevent implementation of the system's required functionality.
- Assume
  - System must support
    - interactive processes.
    - not completely trustworthy processes.
  - Hardware implements page-based Virtual Memory (MMU).

Paging

- No reason to have the pager inside the kernel.
- On page fault:
  - The process interrupts the μ-kernel.
  - The μ-kernel wakes up the pager process.
  - The pager process brings the page from the disk to the main memory.
  - When the page is in the memory, the pager process interrupts the μ-kernel which changes the ownership of the page and wakes up the original process.

The Pager process

- The I/O handling is quite slow. What does the CPU do meanwhile?
  - The pager process blocks itself in the same mechanism a process blocks itself when making a "wait" operation on a semaphore.
  - Meanwhile other processes can be executed.
  - When the I/O interrupts the μ-kernel, the μ-kernel will wake up the pager process.

Device Drivers

- Device Drivers can also be taken out of the kernel.
- The Device Driver process needs the μ-kernel for the interrupt handling.
- Device Driver Process is more secure because it cannot access any address in the memory like the kernel.

Device Driver processes

- When a process performs read/write:
  - The process interrupts the μ-kernel which wakes up the device driver process.
  - The device driver process calls the I/O and blocks itself.
  - When the I/O is done, the μ-kernel gets the interrupt and wakes up the device driver process.
  - When the device driver process finishes, it interrupts the μ-kernel which changes the ownership of the fetched page(s) and wakes up the calling process.

UNIX Device Drivers

- Read/Write operations have too many kernel interrupts ⇒ Waste of time.
- Many versions of Unix implement the device drivers as kernel threads.
  - It reduces the context switches overhead.
  - Installing new device drivers does not require a reboot.
  - The threads are executed in kernel mode with the address space of the μ-kernel; hence they can corrupt the μ-kernel's memory; however not every malfunction in the device driver will cause a shutdown. Sometimes just the thread will die.

What is inside the μ-kernel?

- Scheduling cannot be done by a process.
  - Who will give the scheduling process a time slice?
- Processes cannot write into other processes' address space; Hence IPC should be done by the μ-kernel, which allocates them the space for the IPC.
- Similarly, Semaphores should have a common address space. Moreover, only the kernel is atomic.
- Interrupt handling cannot be done by a process.
  - What will happen if the process does not get a time slice?

What about Memory Management?

- The allocation and deallocation of memory space can be done by a process.
- The μ-kernel supports just 3 basic system calls:
  - Grant - Grant one of my pages to another process.
  - Map - Share one of my pages with another process.
  - Flush – Unshare one of my shared pages.

Memory Allocation

- The Pager process manages the memory.
- At system start time, the entire virtual memory (non-kernel RAM plus swap area) is held by the pager process.
- When a process asks for a new allocation, the pager process will *map* some of its pages to satisfy the request.
- when a process frees some of its space, the pager process will *flush* this pages.

New/Dead processes

- New processes are treated as new allocations for an existing process.
- Dead processes are treated as freed spaces of an existing process.
- Shared memory is obtained by mapping the same page to two or more processes.

The *grant* system call

- The *grant* system call is not used by the pager process.
  - If the pager process uses this system call, it will not be able to free the allocated memory.
- Usually, the *grant* system call is not used. The usage is just when page mappings should be passed through a controlling subsystem without burdening the controller's address space by all pages mapped through it.

Performance

- There are too many context switches to the new kernel aid-processes.
- benchmarking the context switch overhead is usually done by the execution time of getpid().
- Cost of kernel-user mode switches is very high on x86 processors.
  - 71 CPU cycles for entering kernel mode
  - 36 CPU cycles for returning to user mode.
- The cycles stem from the branch, the stack switch and the kernel-bit setting.
- There are many other CPUs that need less cycles.

Address Space Switches

- Cost of address-space switches is mostly because of the TLB switch.
- Pentiums processors have two TLBs one for code (32 entries) and one for data (64 entries). Each flush of a TLB entry is 9 cycles; hence replacing the whole TLB is 864 cycles.
- Using the complete TLB is usually unrealistic. The TLB is 4-way set associative. Using the the four slots of every entry is exceptional.

Address Space Switches (Cont.)

- In order to reduce the context switch overhead, Modern CPUs like the new versions of MIPS save the PID in the TLB. Then, there is no flushing of the TLB on context switches.
- Such a TLB is called *tagged TLB.*
- The system performance can notably benefit small processes like device driver processes.
- Switching the page table is just changing of one pointer; hence negligible.
- An address space switch with tagged TLB is less than 50 cycles.

Kernels vs. μ-kernels

- kernel-specific performance studies like UNIX on CMU Mach vs. pure UNIX, can be misleading.
- Most of the μ-kernels can perform as good as the monolithic kernel and sometimes even better, because of their small address space (a small number of TLB entries to flush on context switches).

Non-Portability

- Processors of competing families differ in:
  - Instructions set.
  - register architecture.
  - exception handling.
  - cache architecture.
  - TLB architecture.
- The IPC and the memory management are an essential part of any μ-kernel. Unfortunately, the memory model is very different even on processors of the same manufacturer.

The Memory Model

- The memory models differ in:
  - What is the maximum virtual address space?
  - Is the address space segmented or flat?
  - Does it support the segment model?
  - Is the TLB tagged?
  - What is the cache writing policy?
    - Write-through or write-back? Write-through policy does not require a dirty bit.

Memory Model - Page Management

- The memory model also includes the page management, which can be differ in:
  - Does it have multi-level page tables?
  - Does it have hash page tables?
  - Does it have more than a single page size?
  - What is the page protection strategic?
    - E.g. modern CPUs demand an explicit bit protection setting command from the kernel, before the kernel will be allowed to modify a write protected page. The very old 386 would do anything in kernel mode.

The μ-kernel Design

- The μ-kernel design is extensively depending on the CPU structure.
- The only μ-kernel feature that can be portable is the scheduler.
- Large monolithic kernel can have many more portable features.
- In the early 70's, large portions of the kernel have been started to be written in C, so they could be portable. Nowadays we return to the non-portable approach.

Some known μ-kernels

- Mach – Was developed at Carnegie Mellon University in the 80's. One of the first of its kind. It is the foundation of the Apple's MacOS.
  - Some newer versions: Utah-Mach, DP-Mach, Spin on Digital Alpha processor.
- Exokernel – Was developed at MIT in 1994-1995. Exokernel is tailored to the MIPS architecture and has some device drivers included in.
  - Exokernel was submitted on 1998 as the PhD thesis of Dawson R. Engler. He was the main designer of this kernel.

Some known μ-kernels (Cont.)

- L4 – Was developed in GMD (Germany's national Research Center for Information Technology). It was implemented on x86 platforms. It has only 12Kbytes of code and implements just 7 system calls. Very common in use as a platform for the Linux operating system.
  - Some newer versions for MIPS and Alpha processors have been developed during last years.

SPIN

- SPIN is a μ-kernel that was written on 1995.
- The users has the ability to integrate components into the kernel.
  - Like traditional μ-kernel, SPIN supports different strategic and different APIs.
  - Any fault of the user extensions will cause a reboot.

SPIN (Cont.)

- The main disadvantage of SPIN is reduced memory protection. If a user extension exceeds its memory allocations, the extension may damage the kernel memory.
  - Extensions has to be written in a special language (Modula-3) in order to check memory exceeding, but
    - The checking is time consuming. (The CPU checks memory exceeding, instead of the MMU).
    - Modula-3 does not always find the exceeding.

Users Programs in Kernel Mode

- On 2002 Maeda suggested to run users programs in kernel mode.
- This is exactly the opposite ideology of μ-kernels.
- It saves the time of the system calls' overhead.
  - Every system call is merely a function call.
- Any user can write to the kernel memory.
  - Maeda suggested to use TAL (Typed Assembly language) which checks the program before loading, but this check does not always find the memory leak.

I/O in Kernel Mode

- According to Maeda, when a process asks for an I/O, another process will be running without any context switch, because anything is kernel.
- No need for double buffering.
  - In the common Oses there are two buffers - one in the kernel space and one in the user space.
    - The user space can be swapped out, while the kernel space is non-swappable.
- The page table of the kernel is huge.
- Maeda reports on improvement of 14% when running "find".

Preemptable Kernel

- On 2000 the MontaVista version of the Linux Kernel 2.4 was introduced.
- When the kernel executes long transactions, it can be useful to be able to take over the CPU even when the kernel does not explicitly yield the CPU.
  - Can be more useful in real-time systems.
- Semaphores can not be used, because the kernel is not atomic $\Rightarrow$ use spinlock.
  - problem of busy waiting.

Preemptable Kernel (Cont.)

- An interrupt will not take over the CPU if the kernel:
  - holds a spinlock.
  - runs the scheduler.
- Linux Kernel 2.6.x (2004) is a partial preemptable kernel.
  - It enables preemptions just on some known points.