4. Memory Management

We'll focus on Intel's memory management.

- Process level (logical level)
- Kernel level (mapping from logical level to physical level)
- Device level (physical level)

Memory Management from a process' perspective

A process is running some program code.

The process can change this relationship using the exec*() system call family.

The running program code is required to be in the memory, since the CPU needs to fetch the instructions from the memory for execution.

Global variables and constants are stored in the object file.

The number of global variables and constants are fixed at compile time.

When a new program code starts running (started by an exec*() system call), the global variables and the constants are created in the memory.

Constants are read-only and should be protected.

Local variables are bound to a function only.

When a function is invoked, the local variables are created. When a function returns, the local variables are abandoned.

Surprisingly, the kernel is not involved during the function calls. The compiled code contains all the memory management.

Local variables are organized as a stack.

Function parameter passing is also done through the stack. A function refers to the parameters as if they are local variables.

When a function returns, all the local variables and function parameters are popped out.

The frame pointer (a.k.a. base pointer or BP) register points to the stack top as it was when the function was just invoked.

No more space left in the stack — stack overflow. Use dynamically allocated memory (heap).

Allocating memory is done by the brk() system call, which grow or shrink the allocated area, or mmap() system call, which can map files or devices into memory.

malloc() only manages free space returned by the kernel. It internally needs memory to manage that, so it asks for more bytes using brk() or mmap().

Memory Management from the kernel' perspective

The kernel knows...

- How many processes are in the system.
- How much space each process needs.
- How much memory is in the system.

A portion of the hard disk will be reserved to serve as the memory. We call this the swap.

Swapping increases the time in context switching. It causes external fragmentation — use defragmentation, a.k.a., memory compaction.

Address space: when the CPU wants to read/write a piece of memory, a memory address is needed.

The same variable may have different addresses at different times (e.g. swapping).

Instead of physical addresses, the compiled code uses logical addressing. A logical address can be translated to a physical address by the OS kernel or the CPU.

base address (physical address, one for each process) + offset (logical address) = target address

Virtual memory

Virtual memory is a memory management technique that solves all these problems...

- External fragmentation.
- Memory growth.
- Memory size limit.

Virtual memory allows...

- The physical address space of processes can be noncontiguous.
- Processes can run even when they are only partially in the memory!

Every process has the entire address space.

The physical memory is partitioned into fixed-size (e.g. 4KB) blocks called frames (page frames).

A process' logical memory is also partitioned into blocks of the same size called pages.

The virtual memory allows a process' memory to grow. The free space inside the process is so virtual that it's not even pre-allocated. It's indeed free space.

Virtual memory internals

The big problem: the CPU runs in a fetch-decode-execute cycle.

We need to translate virtual addresses into physical addresses by memory management unit (MMU), a chip inside the CPU.

Page table: stores the memory mapping.

- It tells which pages are in the physical memory.
- It's stored in the memory and used by the MMU.

The content of the page table depends on the paging mechanism.

Demand paging loads a page from the disk to the physical memory when that page is demanded.

It's similar to swapping, but the system will not load the entire process into the memory, but only the pages required. This allows more processes to be hosted in the system.

A translation lookaside buffer (TLB) is an address-translation cache inside the MMU. It stores the recent translations of virtual memory to physical memory.

The page table must be contiguous → too large.

Multilevel page tables: a page table of page tables.

Page replacement

Condition 1: The process requests a page that does not exist in the physical memory.

Condition 2: There are no free frames available in the physical memory

Step 1: decide the victim page in the physical memory

Step 2: swap out the victim and write it back to the disk

Step 3: swap in the desired frame

The kernel needs to know…

- The current frame allocation status.
- page reference string: the order of the pages that are being referenced by the running process.

Page replacement algorithm: to locate which page is the victim page.

- First-in first-out (FIFO) page replacement.
- Optimal page replacement: replace the page that will not be used for the longest period of time in the future, but impossible since we don't know the future.
- Least-recently-used (LRU) page replacement.
    - Give every frame an age.
    - If the frame is just used, reset the frame's age to 0.
    - Other frame's ages are incremented by 1.

Performance issues

Bélády's anomaly: more memory, sometimes worse performance.

Principle of locality

- Temporal locality: Recently-accessed items are likely to be accessed in the near future.

- Spatial locality: Items whose addresses are near one another tend to be referenced close

together in time.

So a page that is being accessed will have a very high probability to be accessed again.

More processes in the system → more CPU cycles are wasted. This is called thrashing.