Atomik Kernel API

Buddy allocator

One of the most important features of a kernel is to serve and manage system memory. Although this task is usually yielded to a memory management process in most microkernel implementations, Atomik implements it directly in the microkernel, getting closer to a hybrid kernel paradigm.

The most basic memory allocation operations provided by Atomik consists of the allocation of contiguous pages of physical memory and their freeing. This is made on top of a **buddy allocator** : Atomik maintains a global linked list of all ranges of usable physical memory, and for each of them a linked list of « buckets » of contiguous blocks of power-of-two free pages. Allocation and freeing operations are performed directly against this list, moving the remaining free pages to their corresponding buckets, mergin them to lower or higher order lists when necessary.

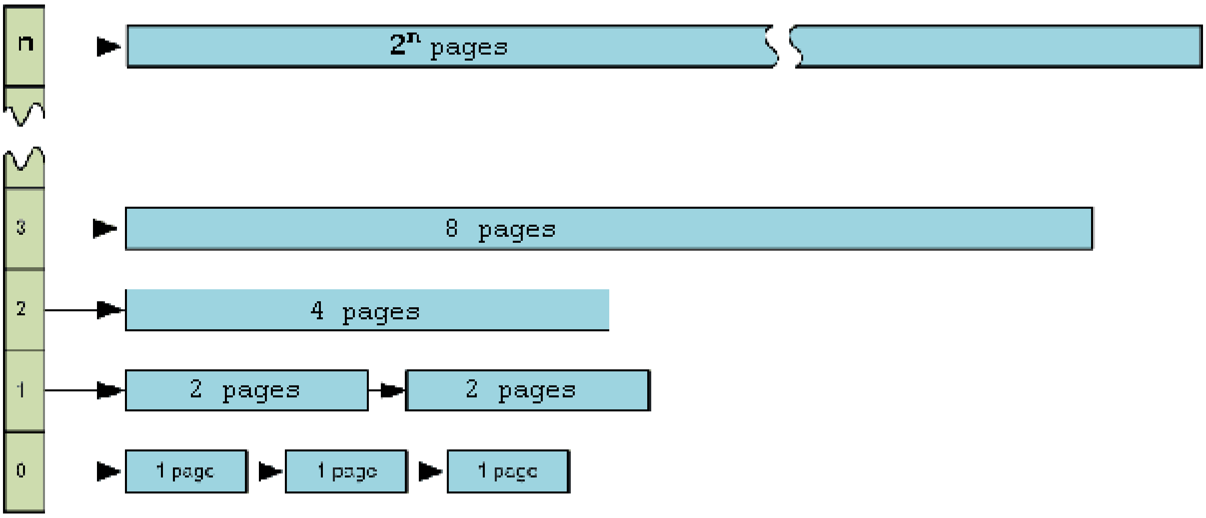

Ilustración 1: Atomik's buddy allocator

The buddy allocator provides two functions to kernel programmers. To use these functions, header file `<mm/regions.h>` must be used.

Functions

`physptr_t page_alloc (memsize_t pageno);`

Tries to allocate a range of *pageno* pages of physical and contiguous memory. By « physical » we mean that the virtual address returned by this function is mapped 1:1 in the kernel page table. Note that the kernel address space is still a virtual address space : in x86, the kernel is mapped in the higher half of the address space (usually starting at 0xd0000000). Lower virtual addresses are mapped 1:1 so they match with the underlying physical addresses.

This function returns a pointer to the first page (in this case, it's ensured it is aligned to `PAGE_SIZE`) or `NULL` if there are not enough contiguous free pages to perform this allocation.

This function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : no
Thread-safe : yes

`int page_free (void *start, memsize_t pageno);`

Returns pageno previously allocated pages beginning at start to the list of free pages of the system. Start must be aligned to `PAGE_SIZE`., but it doesn't need to be the the value returned by `page_alloc`. Any page contained within a previously allocated range can be freed aswell. Thus, this code :

```
void *page;

if ((page = page_alloc (5)) == NULL)
{
  /* Handle failure */
}

page_free (page + 2 * PAGE_SIZE, 3);
```

which frees the last three pages of the previously allocated range of 5 pages is perfectly correct.

This function is **atomic**, this is, interrupts are temporarily blocked during its execution (if they weren't) and restored to their previous state via critical sections. This makes this function suitable for execution both in interrupt context and task context.

Reentrant : no
Thread-safe : yes