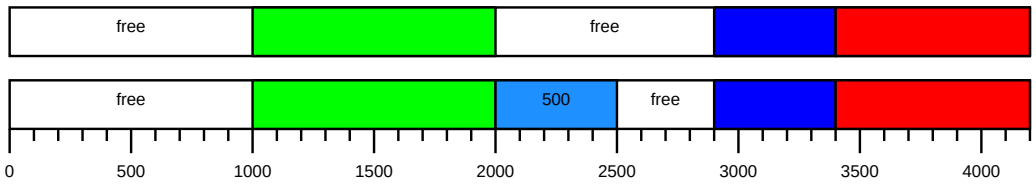
Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Question 12.1: Memory Allocation Policies

Given a system with 4200 memory cells and the following allocation of blocks in main memory: 1000 blocks starting at 1000, 500 blocks starting at 2900, and 800 blocks starting at 3400.

a. A program allocates additional blocks of memory of lengths 500, 1200, and 200 (in that order) according to the *best fit* policy. Show the memory pattern of allocated blocks and remaining holes after each allocation. Consider the program to halt if a request cannot be fulfilled.

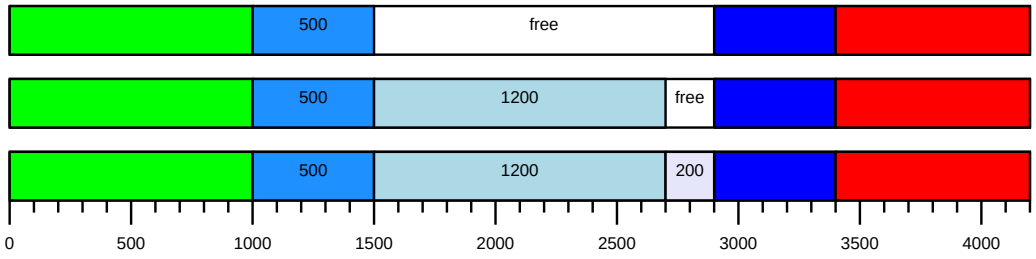
Solution:



The request for 500 cells is allocated to the second hole, since its size is 900, which is closer to the requested 500 than the size of the first gap (1000). The request for 1200 cells fails due to external fragmentation. The last request is not performed.

b. If the above does not succeed, try to create a sufficiently large hole by compacting allocated blocks towards address 0. Move allocated blocks in ascending order of their starting addresses and continue until the resulting hole is large enough.

Solution:



Moving the first two blocks yields a sufficiently large contiguous hole to accommodate the next allocation of 1200 cells. Afterwards, even the request for 200 cells can be fulfilled without any further compaction.

Question 12.2: Buddy Allocator

a. How does memory allocation using the buddy allocator work?

Solution:
A buddy system always manages and returns memory in units sized as a power of 2. To satisfy a request, the allocator rounds the requested size up to the next higher power of 2 and tries to find a free memory block of that size. If it cannot find such a block, it takes a block of the next higher order (i.e., twice as large) and splits it into two equally sized chunks, called buddies. The memory request is then satisfied by one of the two buddies. As soon as both buddies are free again, the buddy allocator merges them again to rebuild the memory chunk of the next higher order.

Note that searching for a memory chunk to be split may itself lead to memory blocks of even higher order to be split (up to the maximum order representing the full memory size, managed by the allocator).

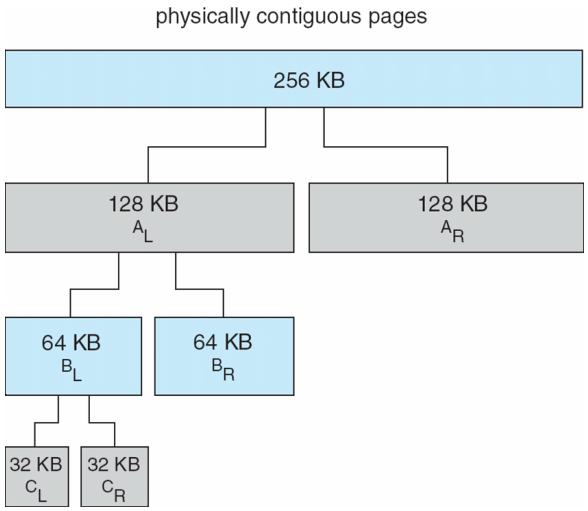
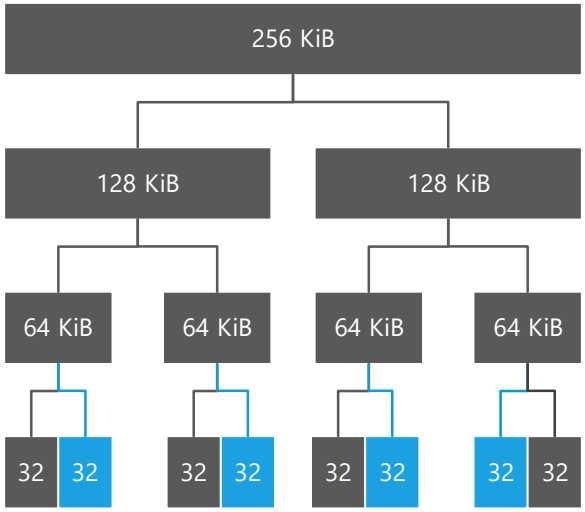


Abbildung 1: Buddy allocation scheme

b. Given a memory of size 2^k managed according to the buddy system. How many entries are there in the free list for a memory request size of 2^m maximally and minimally with $m < k$?

Solution:
Minimum: 0 – The reqeuest fails.

Maximum: $\frac{1}{2} \cdot 2^{k-m}$ *– For a memory chunk of size* 2^i *to be on the free list of level* i*, the 'parent' chunk of size* 2^{i+1} *needs to be split. That is the case, if there has been already a memory allocation with (at most) the same size (*2^i*) in the past, leaving the buddy in the free list. Hence, for the maximum number of chunks to get on the free list of level* i *all chunks of size* 2^{i+1} *must have been split with one buddy allocated and the other one already having been freed again.*



$2^k = 256K \quad k=18$

$2^m = 32K \quad m=15$

Allocated Free

Number of elements on level 2^m?

$2^{k-m} = 2^{18-15} = 2^3 = 8$

$\frac{1}{2}\ 2^{k-m}$

c. What type of fragmentation does the buddy system suffer from?

Solution:
External fragmentation can happen for example if multiple blocks of a certain level have been freed but cannot be merged because their respective buddies still contain valid allocations – you can only merge blocks that are buddies. In that case the total free memory might suffice to fulfill a memory request, but the memory is not contiguous.

Since the buddy allocator always allocates memory regions of size 2^n, internal fragmentation always occurs when memory gets requested in sizes that are not a power of 2. Internal fragmentation with the buddy allocator is a problem particularly when the requested memory is frequently big enough to just barely prevent the buddy allocator from splitting a chunk (e.g., a 129 KiB request will consume a whole 256 KiB chunk, wasting 127 KiB.).

d. Can you imagine a way to reduce fragmentation when using a buddy system?

Solution:
In general, allocators such as the buddy system work well for coarse-grain allocations, such as (ranges of) page frames. For finer-grained allocations you better combine them with other allocation schemes.

In practice, programs and the operating system often allocate multiple chunks of the same (non-power of 2) size, for example, to store instances of a certain type of object. A way to reduce fragmentation is to supplement the buddy system with a slab cache that manages objects this size.

Question 12.3: Solid-State Drives

a. Why is rewriting data on a flash-based solid-state drive an order of magnitude slower than writing?

Solution:
NAND memory needs to be erased before it can be written again. Rewriting thus requires reading an entire block, erasing it and writing the modified pages as well as all other pages which have not been modified back to the block. The erase cycle itself is very slow compared to reading.

- *Reading a page:* $\sim 25\,\mu s$
- *Writing a page:* $\sim 250\,\mu s$
- *Erasing a block:* $\sim 2\,ms$

b. How do spare blocks help with this matter?

Solution:
When previously erased blocks (the spare blocks) are available, erasure can be performed lazily when the disk is idle. This way, on a rewrite, the block is relocated to a spare-block and the old block is marked for later erasure.

c. Describe how the `trim` command helps improving slow rewrites.

Solution:
When a block is written, the information about the write is known to the disk. When a file is deleted in the filesystem, the information about the freed block is not transported to the block layer (and thus the SSD drive). This way, all pages need to be copied to the new or erased block on a write, whether they are in use or not. Unused pages are written with unneeded information this way. Writing to these pages which are marked as used in the block-layer but not used in the file-system layer requires a full erase cycle on the erase-block.

The `trim` command transports the information about freed pages to the ssd's firmware to enable it to keep these pages erased for faster writes in the future and to avoid unnecessary copy operations.

Question 12.4: Accessing Files and Directories

a. What is the difference between an absolute and a relative path name?

Solution:
An absolute path name starts at the root of the directory structure, whereas a relative path name defines a path starting from the current directory. The current directory is normally process-specific and called working directory.

b. What are the basic methods for accessing a file?

Solution:

Sequential access: *The file is accessed in order, one record (or byte) after the other. A read-operation reads the next n records (or bytes), and advances the position within the file accordingly. Similarly, writes are appended to the end of the file, and the position is set to the end of the file afterwards.*

Direct access: *Programs are allowed to read and write records of the file in any order. In contrast to sequential access, the programmer now has to specify explicitly which record to read/write.*

c. In Linux and Windows, random access on files is implemented via a special system call that moves the "current position pointer" associated with a file to a given position in the file. What are the names of these system calls in Windows and Linux?

Solution:
In Linux, the system call is called `lseek`*;*
in Windows, `SetFilePointer` *and* `SetFilePointerEx` *are provided.*

All functions allow the file offset to be set beyond the end of the file. If data is later written at this point, subsequent reads of the data in the gap return bytes of zeros (until data is actually written into the gap).

Linux:

```
off_t lseek(int fildes, off_t offset, int whence)
```

`off_t`*, being a 64 bit signed integer, sets the current position in the file* `fildes` *to* `offset`*, relative to the location specified by* `whence`*:*

SEEK_SET *The pointer is set to the* `offset`*'s byte.*

SEEK_CUR *The pointer is set to its current location plus* `offset` *bytes.*

SEEK_END *The pointer is set to the size of the file plus* `offset` *bytes.*

Windows:

```
BOOL SetFilePointerEx(
    HANDLE hFile,
    LARGE_INTEGER liDistanceToMove,
    PLARGE_INTEGER lpNewFilePointer,
    DWORD dwMoveMethod )
```

hFile *Handle to the file whose file pointer is to be moved.*

liDistanceToMove *The number of bytes to move the file pointer. A positive value moves the pointer forward in the file and a negative value moves the file pointer backward.*

lpNewFilePointer *A pointer to a variable to receive the new file pointer. If this parameter is* `NULL`*, the new file pointer is not returned.*

dwMoveMethod *Starting point for the file pointer move:*

FILE_BEGIN *The starting point is the beginning of the file.*

FILE_CURRENT *The starting point is the current value of the file pointer.*

FILE_END *The starting point is the current end-of-file position.*

d. Discuss alternative random access implementations without such a system call.

Solution:
One way is to add another parameter to the `read` *and* `write` *system calls, which supplies the offset to access. This would effectively prepend a seek operation to every* `read`*/*`write` *system call and, for truly randomly accessed files, save the extra system call required for* `lseek`*.*

On the downside, this approach adds an extra parameter to every `read`*/*`write` *system call, although most of the time, sequential access is desired. Furthermore, the application programmer or library code must now track the current file position to pass it on to the system calls every time.*

Another alternative is mapping the file (or parts thereof) into the virtual address space of the process. This allows for full random access without performing system calls for the accesses at all. Accesses, however, may trigger one or more page faults.

e. What system calls do you need to list the files in a directory in Linux?

Solution:
Before you get a list of file names, you first have to open the directory using the `opendir` *system call. You then call* `readdir` *repeatedly, receiving* `dirent` *structures that contain information on a single file in the directory or* `NULL` *when there are no more files. You then call* `closedir` *to close the used directory stream.*