Memory Allocation Guide

Linux provides a variety of APIs for memory allocation. You can allocate small chunks using *kmalloc* or *kmem_cache_alloc* families, large virtually contiguous areas using *vmalloc* and its derivatives, or you can directly request pages from the page allocator with *alloc_pages*. It is also possible to use more specialized allocators, for instance *cma_alloc* or *zs_malloc*.

Most of the memory allocation APIs use GFP flags to express how that memory should be allocated. The GFP acronym stands for "get free pages", the underlying memory allocation function.

Diversity of the allocation APIs combined with the numerous GFP flags makes the question "How should I allocate memory?" not that easy to answer, although very likely you should use

```
kzalloc(<size>, GFP_KERNEL);
```

Of course there are cases when other allocation APIs and different GFP flags must be used.

Get Free Page flags

The GFP flags control the allocators behavior. They tell what memory zones can be used, how hard the allocator should try to find free memory, whether the memory can be accessed by the userspace etc. The :ref:`Documentation/core-api/mm-api.rst <mm-api-gfp-flags>` provides reference documentation for the GFP flags and their combinations and here we briefly outline their recommended usage:

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]memory-allocation.rst`, line 32); *backlink***
>
> Unknown interpreted text role "ref".

- Most of the time `GFP_KERNEL` is what you need. Memory for the kernel data structures, DMAable memory, inode cache, all these and many other allocations types can use `GFP_KERNEL`. Note, that using `GFP_KERNEL` implies `GFP_RECLAIM`, which means that direct reclaim may be triggered under memory pressure; the calling context must be allowed to sleep.
- If the allocation is performed from an atomic context, e.g interrupt handler, use `GFP_NOWAIT`. This flag prevents direct reclaim and IO or filesystem operations. Consequently, under memory pressure `GFP_NOWAIT` allocation is likely to fail. Allocations which have a reasonable fallback should be using `GFP_NOWARN`.
- If you think that accessing memory reserves is justified and the kernel will be stressed unless allocation succeeds, you may use `GFP_ATOMIC`.
- Untrusted allocations triggered from userspace should be a subject of kmem accounting and must have `__GFP_ACCOUNT` bit set. There is the handy `GFP_KERNEL_ACCOUNT` shortcut for `GFP_KERNEL` allocations that should be accounted.
- Userspace allocations should use either of the `GFP_USER`, `GFP_HIGHUSER` or `GFP_HIGHUSER_MOVABLE` flags. The longer the flag name the less restrictive it is.

  `GFP_HIGHUSER_MOVABLE` does not require that allocated memory will be directly accessible by the kernel and implies that the data is movable.

  `GFP_HIGHUSER` means that the allocated memory is not movable, but it is not required to be directly accessible by the kernel. An example may be a hardware allocation that maps data directly into userspace but has no addressing limitations.

  `GFP_USER` means that the allocated memory is not movable and it must be directly accessible by the kernel.

You may notice that quite a few allocations in the existing code specify `GFP_NOIO` or `GFP_NOFS`. Historically, they were used to prevent recursion deadlocks caused by direct memory reclaim calling back into the FS or IO paths and blocking on already held resources. Since 4.12 the preferred way to address this issue is to use new scope APIs described in :ref:`Documentation/core-api/gfp_mask-from-fs-io.rst <gfp_mask_from_fs_io>`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]memory-allocation.rst`, line 72); *backlink***
>
> Unknown interpreted text role "ref".

Other legacy GFP flags are `GFP_DMA` and `GFP_DMA32`. They are used to ensure that the allocated memory is accessible by hardware with limited addressing capabilities. So unless you are writing a driver for a device with such restrictions, avoid using these flags. And even with hardware with restrictions it is preferable to use *dma_alloc** APIs.

GFP flags and reclaim behavior

Memory allocations may trigger direct or background reclaim and it is useful to understand how hard the page allocator will try to satisfy that or another request.

- `GFP_KERNEL & ~__GFP_RECLAIM` - optimistic allocation without _any_ attempt to free memory at all. The most light weight mode which even doesn't kick the background reclaim. Should be used carefully because it might deplete the memory and the next user might hit the more aggressive reclaim.
- `GFP_KERNEL & ~__GFP_DIRECT_RECLAIM` (or `GFP_NOWAIT`)- optimistic allocation without any attempt to free memory from the current context but can wake kswapd to reclaim memory if the zone is below the low watermark. Can be used from either atomic contexts or when the request is a performance optimization and there is another fallback for a slow path.
- `(GFP_KERNEL|__GFP_HIGH) & ~__GFP_DIRECT_RECLAIM` (aka `GFP_ATOMIC`) - non sleeping allocation with an expensive fallback so it can access some portion of memory reserves. Usually used from interrupt/bottom-half context with an expensive slow path fallback.
- `GFP_KERNEL` - both background and direct reclaim are allowed and the **default** page allocator behavior is used. That means that not costly allocation requests are basically no-fail but there is no guarantee of that behavior so failures have to be checked properly by callers (e.g. OOM killer victim is allowed to fail currently).
- `GFP_KERNEL | __GFP_NORETRY` - overrides the default allocator behavior and all allocation requests fail early rather than cause disruptive reclaim (one round of reclaim in this implementation). The OOM killer is not invoked.
- `GFP_KERNEL | __GFP_RETRY_MAYFAIL` - overrides the default allocator behavior and all allocation requests try really hard. The request will fail if the reclaim cannot make any progress. The OOM killer won't be triggered.
- `GFP_KERNEL | __GFP_NOFAIL` - overrides the default allocator behavior and all allocation requests will loop endlessly until they succeed. This might be really dangerous especially for larger orders.

Selecting memory allocator

The most straightforward way to allocate memory is to use a function from the kmalloc() family. And, to be on the safe side it's best to use routines that set memory to zero, like kzalloc(). If you need to allocate memory for an array, there are kmalloc_array() and kcalloc() helpers. The helpers struct_size(), array_size() and array3_size() can be used to safely calculate object sizes without overflowing.

The maximal size of a chunk that can be allocated with *kmalloc* is limited. The actual limit depends on the hardware and the kernel configuration, but it is a good practice to use *kmalloc* for objects smaller than page size.

The address of a chunk allocated with *kmalloc* is aligned to at least ARCH_KMALLOC_MINALIGN bytes. For sizes which are a power of two, the alignment is also guaranteed to be at least the respective size.

Chunks allocated with kmalloc() can be resized with krealloc(). Similarly to kmalloc_array(): a helper for resizing arrays is provided in the form of krealloc_array().

For large allocations you can use vmalloc() and vzalloc(), or directly request pages from the page allocator. The memory allocated by *vmalloc* and related functions is not physically contiguous.

If you are not sure whether the allocation size is too large for *kmalloc*, it is possible to use kvmalloc() and its derivatives. It will try to allocate memory with *kmalloc* and if the allocation fails it will be retried with *vmalloc*. There are restrictions on which GFP flags can be used with *kvmalloc*; please see kvmalloc_node() reference documentation. Note that *kvmalloc* may return memory that is not physically contiguous.

If you need to allocate many identical objects you can use the slab cache allocator. The cache should be set up with kmem_cache_create() or kmem_cache_create_usercopy() before it can be used. The second function should be used if a part of the cache might be copied to the userspace. After the cache is created kmem_cache_alloc() and its convenience wrappers can allocate memory from that cache.

When the allocated memory is no longer needed it must be freed. You can use kvfree() for the memory allocated with *kmalloc*, *vmalloc* and *kvmalloc*. The slab caches should be freed with kmem_cache_free(). And don't forget to destroy the cache with kmem_cache_destroy().