# *ECE391*
# *Computer System Engineering*
### *Lecture 22*

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Spring 2021

# *Lecture Topics*

- Memory allocation interfaces in the kernel
  - kmalloc
  - slab caches
  - vmalloc
  - buddy system

## *Aministrivia*

- **MP3 Checkpoint 3**
  - **Due by 6:00pm Monday, April 12**

# *Files for Memory Management*

- headers (all under linux/): gfp.h, slab.h, vmalloc.h, slab_def.h, slub_def.h
- sources: mm/slab.c, mm/vmalloc.c
- swap-related: mm/swap.c, mm/swapfile.c, mm/page_alloc.c

# *Memory Management*

- Paging translates virtual address to physical address
- Some portion of the physical memory permanently assigned to the kernel: stores code and data
- Rest of the physical memory can be allocated at run-time ➡ dynamic allocation
- Kernel must keep track of the status of each memory page in physical memory, e.g., is the page free

# *Memory Zones (1)*

- In allocating memory Linux kernel must deal with two important hardware constrains of the x86 architecture:
  - Direct Memory Access (DMA): older processor can address only the first 16MB of RAM
  - In 32-bit computers with a lot of memory, the processor cannot directly access all physical memory because the liner address apace is too small

# *Memory Zones (2)*

- To address hardware limitations Linux partitions the physical memory into three zones:
- ZONE_DMA
  - Contains memory pages below 16MB
- ZONE_NORMAL
  - Contains memory pages at and above 16MB and below 896MB
- ZONE_HIGHMEM
  - Contains memory pages at and above 896MB

# *Memory Allocation*

- Overview
  - a few small items → kmalloc
  - a lot of items, repeatedly → slab cache
  - a big, physically contiguous region → free pages
  - a big area of virtual memory → vmalloc
    (<u>not</u> necessarily physically contiguous)

- flags/allocation priorities (common to all interfaces)

**"get free pages"**

**`GFP_ATOMIC`** does not sleep; succeeds only if request can be satisfied with existing free resources (unlikely for large requests)

# Memory Allocation

may sleep to wait for pages

| | |
|---|---|
| `GFP_KERNEL` | by kernel, drivers, etc. |
| `GFP_NOFS` | no file system calls (avoids pushing pages to disk) |
| `GFP_NOIO` | no I/O operations at all |
| `GFP_USER` | on behalf of a user (low priority) |
| `__GFP_DMA` | DMA accessible (low physical addresses on some machines) |
| `__GFP_HIGHMEM` | high memory (PAE (phys. address extensions) on x86) is acceptable |

(two underscores)

# *Basic Interface*

```
void* kmalloc (size_t size, gfp_t flags);
```

- uses exponentially-sized slab caches (to be discussed)
  - ranging from 8B to several MB
  - up to 4MB in our kernel
- each allocation is contiguous in physical memory

# *Basic Interface*

- Managing a private cache of objects (slab cache)
  - frequent allocations/deallocations
  - one cache per item type
  - physical memory is contiguous
  - protocol
    - Creation returns  a page handle
    - Use handle to allocate/deallocate objects or to destroy the slab cache when done

# *Basic Interface*

```
kmem_cache* kmem_cache_create (const char* name,
        size_t size, size_t align, unsigned long flags,
        void (*constructor) (void*, kmem_cache*,
                                      unsigned long ignored),
       void (*ignored)([same as constructor]));
```

- name used to avoid >1 cache for same structure
- size is object size;  cache grows/shrinks automatically
- alignment specified for individual objects

# *Flags for* **slab** *cache allocation*

| | |
|---|---|
| SLAB_HWCACHE_ALIGN | align objects to cache lines (makes accesses a little faster) |
| SLAB_CACHE_DMA | use DMA-accessible memory |
| SLAB_POISON | fill new memory with 0xA5A5A5A5 |
| SLAB_RED_ZONE | bound objects with “red zones” (test buffer overruns) |

- Slab cache constructor function
  - Callback function used when memory is allocated for the slab cache
  - called for each object in new slab
  - NOT called for each object allocation (*kmem_cache_alloc*)
  - third argument used to be flags (now always 0)

# *Slab cache API*

- Slab cache allocation/deallocation and destruction

```
void* kmem_cache_alloc (kmem_cache*,
                        gfp_t flags);

void* kmem_cache_zalloc (kmem_cache*,
                         gfp_t flags);
```

  - flags are passed to lower allocator (*kmalloc)* iff a new slab is allocated
  - zalloc version zeroes memory in new object

```
void kmem_cache_free (kmem_cache*, void*);
```

# Slab cache API

```
void kmem_cache_destroy (kmem_cache*);
```

– fails silently (logs error message) if not empty

```
int kmem_cache_shrink (kmem_cache*);
```

– frees empty slabs; returns 0 if all slabs released

# *Getting big chunks of memory*

- multiples of page size (4kB on x86; ISA-dependent)
- physically contiguous

```
unsigned long get_zeroed_page (gfp_t flags);
unsigned long __get_free_page (gfp_t flags);
unsigned long __get_free_pages
            (gfp_t flags, unsigned int order);
```

- flags are same as for kmalloc
- order is log (base 2) of number of pages requested
- (the latter two function names start with two underscores)

# *Getting big chunks of memory*

```
void free_page (unsigned long);
void free_pages (unsigned long, int order);
```

- order <u>must match</u> value used when allocated!
- These functions do <u>not</u> check for you!

# *Getting big chunks of memory*

- Virtual memory allocation (request size in bytes, but allocates pages)
  - all functions return virtual addresses
  - but all other functions discussed today allocate physically contiguous regions
  - what if we don't care (or need a bigger region)?
  - use vmalloc (see linux/vmalloc.h)

```
void* vmalloc (unsigned long size);
void vfree (void* addr);
```

# *Memory Fragmentation*

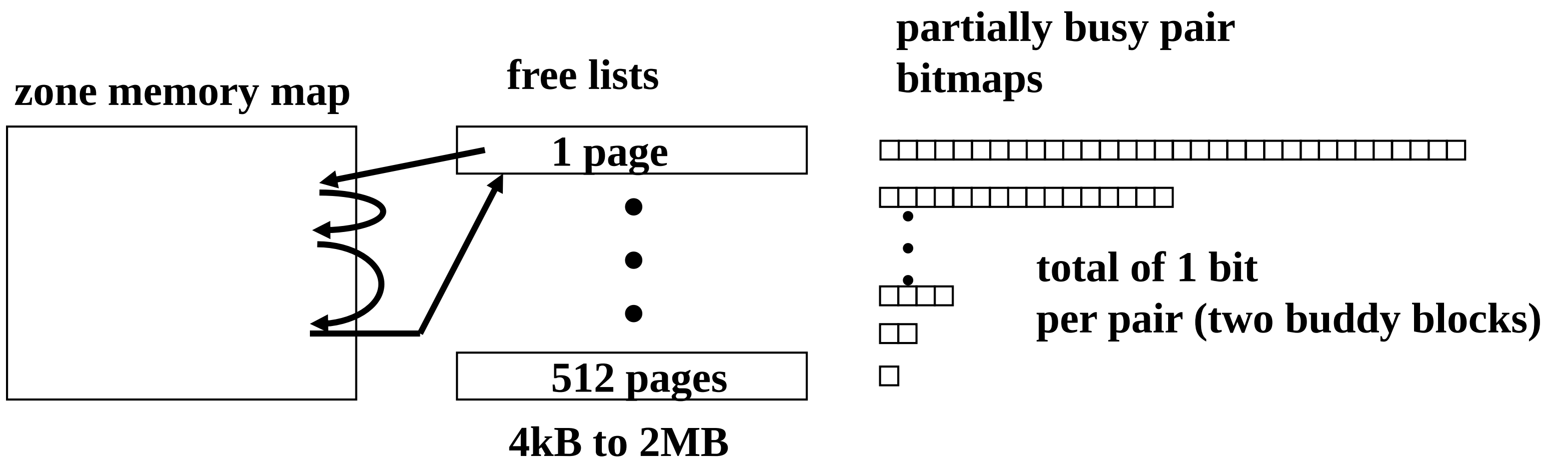
- ***External fragmentation:*** Frequent allocation/deallocation of group of continuous pages in physical memory of different sizes may lead to situation in which several small blocks of free pages are scattered inside blocks of allocated pages. As a result it may be impossible to allocate a large block of contiguous pages even if there is enough free pages.

- ***Internal fragmentation:*** caused by a mismatch between the size of the memory request and the size of the memory area allocated to satisfy the request

# *The Buddy System*

- Problem: how to implement memory allocation inside the kernel
  - need page alignment for allocations
  - may need contiguous regions of physical memory
  - need flexible allocation granularity
  - want to avoid always rewriting page tables
  - can’t easily add information to allocated (physical) memory chunks

# *The Buddy System*

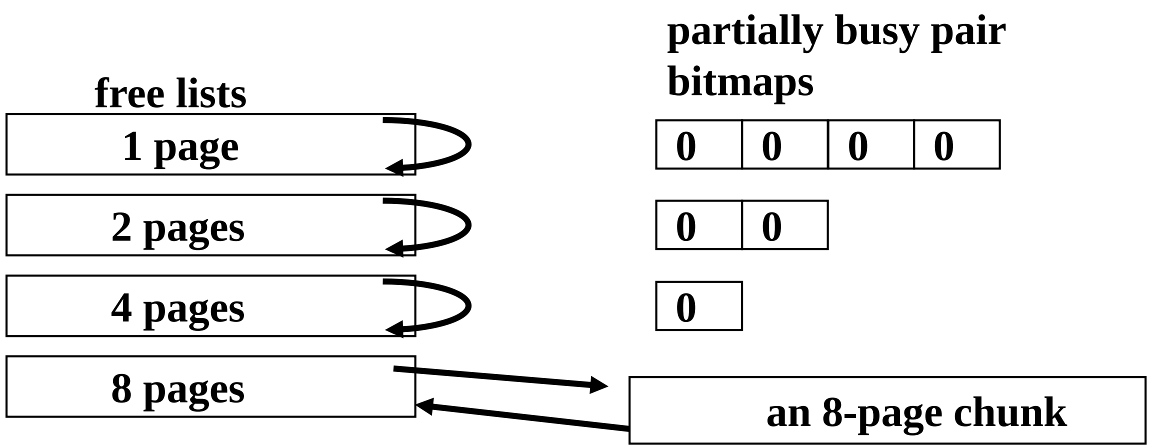


# *The Buddy System*

- Traditional simple answer
  - exponential bins: 1 page, 2 pages, 4 pages, etc.
  - buddy system extends with dynamic motion between bins
- Partially busy bit: 1 if exactly one buddy in use (0 if both/neither in use)
- Example using one group of eight pages
  - view also as two groups of four, four groups of two, or eight single pages
  - initially appears as a single chunk in 8-page free list
  - all other free lists are empty
  - all partially free bits are 0

# *Buddy system example*

- Initial configuration

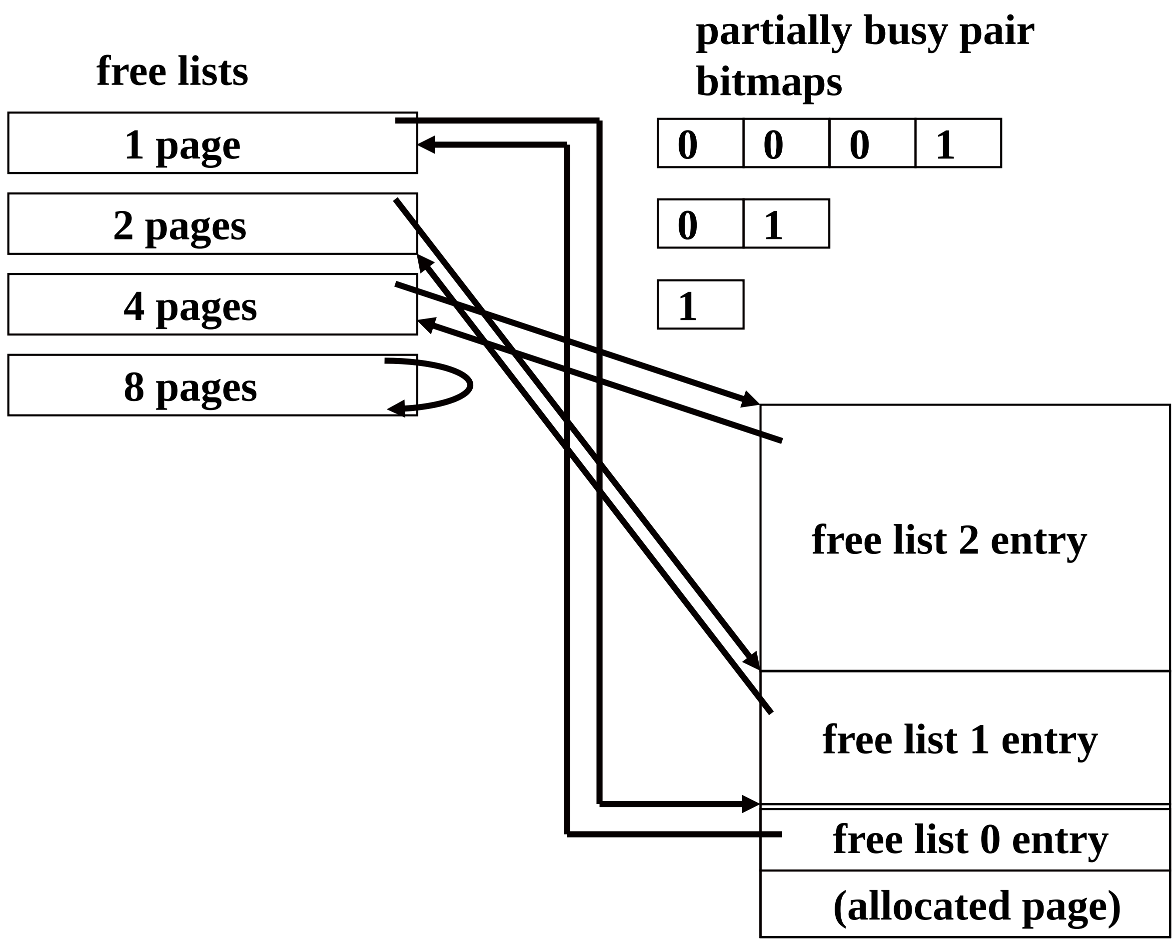



- Allocation
  - try the correct size free list
  - if empty, try the next larger size and break up a chunk

# Buddy system example

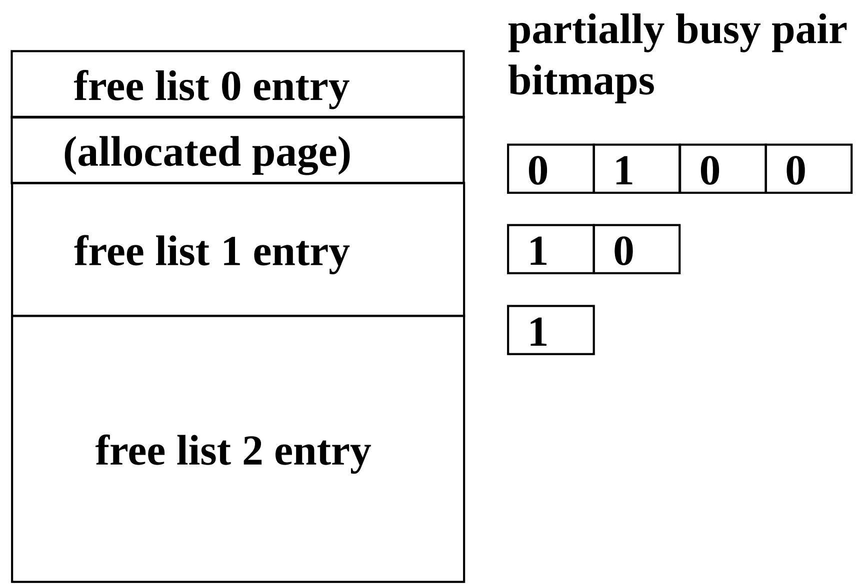
- Request one page of order 0
  (order is log of # of pages, i.e., one page here)
  - any in free list 0 (1 page)?  no…
  - any in free list 1 (2 pages)? no…
  - any in free list 2 (4 pages)? no…
  - any in free list 3 (8 pages)? yes!  split it up recursively…

# Buddy system example



# *Buddy system example*

- Deallocation
  - check if buddy is free (is pair bit = 1?)
  - if both free, merge and check again (recursively)
- initial configuration for free example



# *Buddy system example*

- Free element (order is 0)
  - check (& flip) buddy bit in order 0
  - buddy was free $\rightarrow$ remove buddy from free list and merge (address remains the same for now)
  - check (& flip) buddy bit in order 1
  - buddy was free $\rightarrow$ remove buddy from free list and merge (address changes to start of first 4-page block)
  - repeat for order 2, then done; end with initial (all free) configuration