Linux Kernel Development 05

工海系 b06501018 朱紹勳

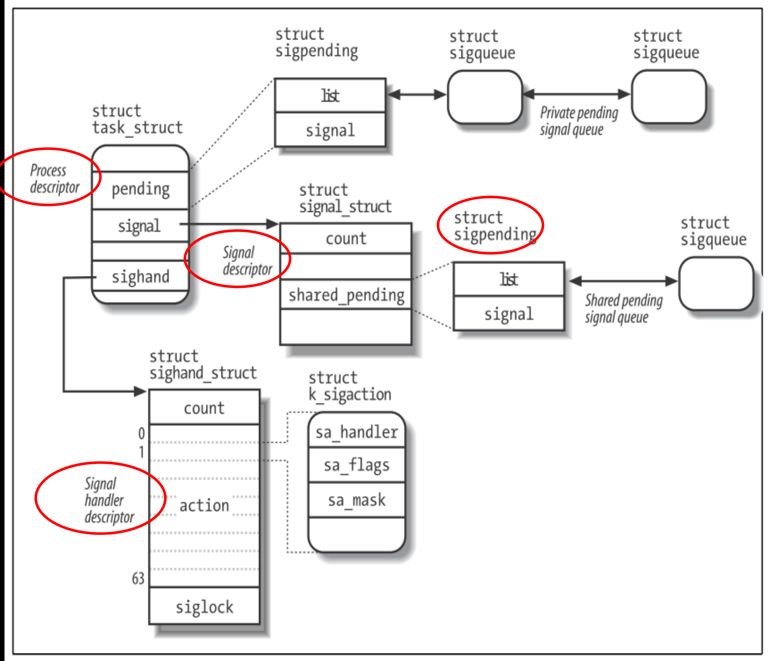
	Linux Kernel Dev 3rd	Understand Linux Kernel 3rd	
Signal			APUE : Chapter 10 CSAPP Chapter 8
Memory Management	Chapter 12	Chapter 8	CSAPP : Chapter 10
Process Address Space	Chapter 15	Chapter 9	CSAPP : Chapter 8 APUE : Chapter 7
Page Cache & Write Back	Chapter 16	Chapter 15, 17	

Signal - CSAPP

Signal

- Software Interrupt : notify process something or Execute a signal handler
- ex: press certain terminal keys(SIGINT), /0(SIGFPE), kill()(SIGKILL)…
- Kernel :   (1) Signal generation (pending)   (2) Signal delivery
- Process: (1) Receive signal   (2) Signal handling

- Signal sent but not received called pending signal
  - If one signal is pending, the later arrive will be discard
  - Blocked : can send but not receive
- POSIX & Multi-thread
- Data Structure of Signal



Send Signal

- Mechanism is based on process group `getpgrp(), setpgid(pid, pgid);`
- kill() `unix> kill .9 15213`
- keyboard `unix> ls | sort`
- Ex:

```c
#include "csapp.h"

void handler(int sig){
    static int beeps = 0;
    printf("BEEP\n");
    if(++beeps < 5) Alarm(1); /* next SIGALAM will be delivered in 1s*/
    else{
        printf("BOOM\n");
        exit(0);
    }
}
int main(){
    Signal(SIGALRM, handler);
    Alarm(1);/* next SIGALAM will be delivered in 1s*/
    while(1) /* signal handler returns control here each time*/
    exit(0);
}
```

signal set a signal handler function that is called asynchronously,

Interrupt the while loop in main()

Signal Receive

- When kernel return from exception handler, ready to pass control to process p, it checks the set of unblocked pending
  - if is empty, pass, else pick some signal k in the set and forces p to receive k
  - default action after receiving signal, can set by changing handler in signal()
    - process terminate
    - process terminate & dump core
    - process stops until restarted by a SIGCONT signal
    - process ignores signal

```c
#include "csapp.h"

void handler(int sig){ /* SIGINT handler */
    printf("Caught SIGINT\n");
    exit(0);
}
int main(){
    /* Install the SIGINT handler */
    if (signal(SIGINT, handler) == SIG_ERR)
        unix_error("signal error");
    pause(); /* Wait for the receipt of a signal */
    exit(0);
}
```

A program that catches the SIGINT signal. The default action for SIGINT is to immediately terminate the process.

In this example, we modify the default behavior to catch the signal, print a message, and then terminate the process.

Signal Handling Issues

- Problem when catches multiple signals
  - Pending signals are blocked
  - Pending signals are not queued
  - System calls can be interrupted

(1) Only two signals are captured(zombie process)

1st arrive, 2nd arrive(pending), 3rd arrive(drop)

Signals cannot be used to count events in other processes.

→ modify the SIGCHLD handler to reap as many zombie children as possible each time it is invoked

(2) Run on Solaris OS system

read() slow system call won't restart after interrupt by signal, different from Linux

→ Prevent system call return too early and manually restart EINTR indicates that the read system call returned prematurely after it was interrupted.

```
#include "csapp.h"

void handler2(int sig){
    pid_t pid;

    while ((pid = waitpid(-1, NULL, 0)) > 0)
        printf("Handler reaped child %d\n", (int)pid);
    if (errno != ECHILD)
        unix_error("waitpid error");
    Sleep(2);
    return;
}

int main() {
    int i, n;
    char buf[MAXBUF];
    pid_t pid;

    if (signal(SIGCHLD, handler2) == SIG_ERR)
        unix_error("signal error");

    /* Parent creates children */
    for (i = 0; i < 3; i++) {
        pid = Fork();
        if (pid == 0) {
            printf("Hello from child %d\n", (int)getpid());
            Sleep(1);
            exit(0);
        }
    }

    /* Manually restart the read call if it is interrupted */
    while ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
        if (errno != EINTR)
            unix_error("read error");

    printf("Parent processing input\n");
    while (1)
        ;

    exit(0);
}
```

Portable Signal Handling

- `int sigaction(signum, *act, *oldact);`

```
handler_t *Signal(int signum, handler_t *handler){
    struct sigaction action, old_action;

    action.sa_handler = handler;
    sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
    action.sa_flags = SA_RESTART; /* Restart syscalls if possible */

    if (sigaction(signum, &action, &old_action) < 0)
        unix_error("Signal error");
    return (old_action.sa_handler);
}
```

We use Signal function to wrapper the signal action
Which make Signal Potable in different system

Explicitly Blocking and Unblocking Signals

- `sigprocmask(), sigemptyset(), sigfillset(), sigaddset(), sigdelset()`

```
while (1) {
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    Sigprocmask(SIG_BLOCK, &mask, NULL); /* Block SIGCHLD */

    /* Child process */
    if ((pid = Fork()) == 0) {
    Sigprocmask(SIG_UNBLOCK, &mask, NULL); /* Unblock SIGCHLD */
    Execve("/bin/date", argv, NULL);
    }

    /* Parent process */
    addjob(pid); /* Add the child to the job list */
    Sigprocmask(SIG_UNBLOCK, &mask, NULL); /* Unblock SIGCHLD */
}
```

SIG_BLOCK: Add the signals in set to blocked
SIG_UNBLOCK: Remove the signals in set from blocked
SIG_SETMASK: blocked = set

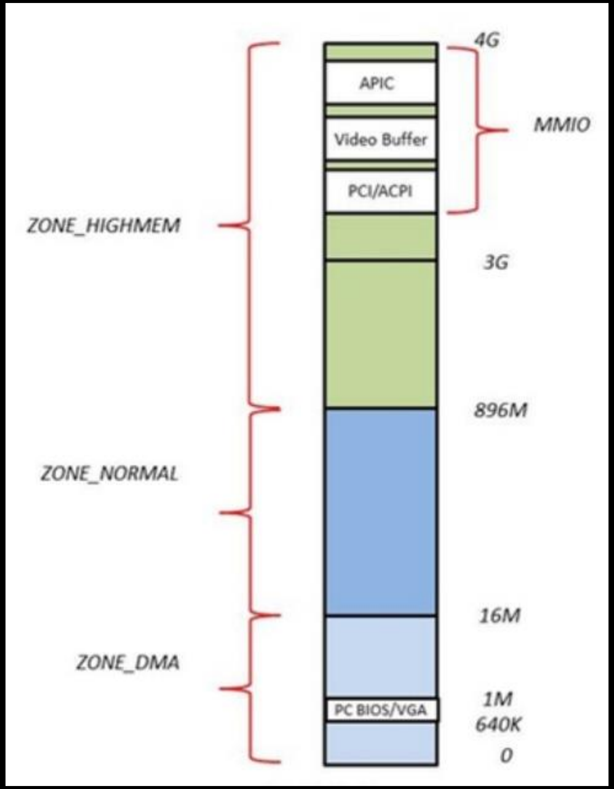
The example use sigprocmask to synchronize processes

Kernel Memory Management

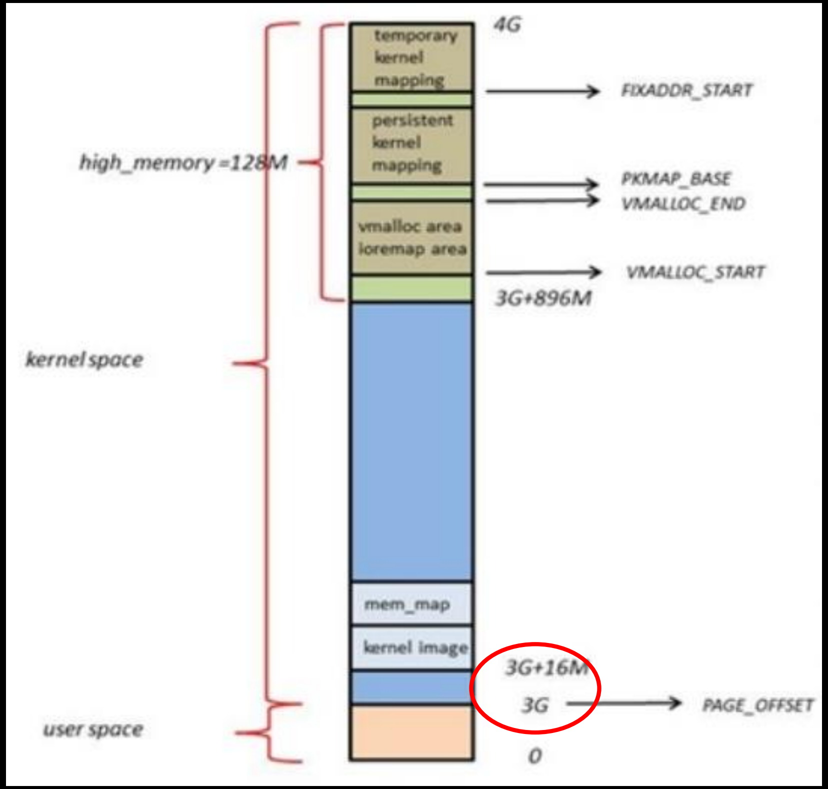
Page & Zone

- Page data structure is defined in <linux/mm_types.h>, 20MB for 500000+
  - atomic_t _count, void *virtual
- Kernel divides pages into different zones, defined in <linux/mmzone.h>
- Different between architectures
  - ZONE_DMA, ZONE_DMA32 (use for DMA) <16MB
  - ZONE_NORMAL (normal addressing) 16~896MB
  - ZONE_HIGHMEM (dynamic mapping) > 896MB

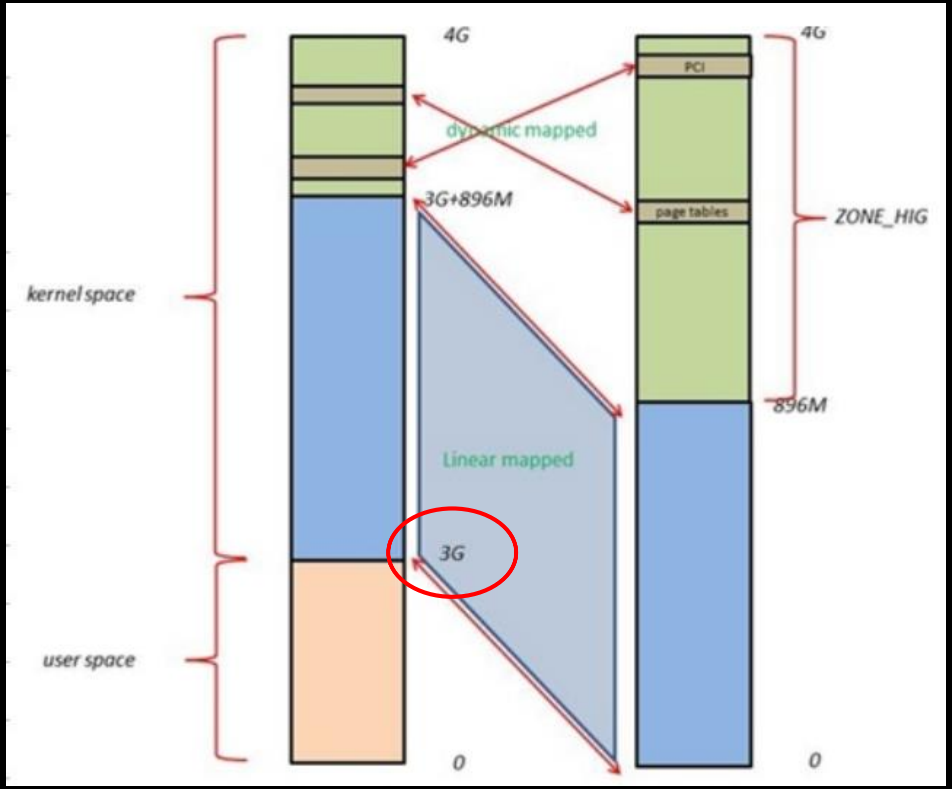
x86 physical memory



linux virtual memory



Mapping between virtual & physical



Getting Pages

- Use interface to allocate & free memory in kernel
  - `alloc_page(gfp_mask, n)`
    `// This allocates 2^n contiguous physical pages, return ptr to 1st page`
  - `void *page_address(*page)`
    `//  returns a ptr to the logical address of given physical page`
  - `long __get_free_page(gfp_mask)`
    `// directly returns the logical address of the first requested page`
  - `get_zeroed_page(gfp_mask)`
    `// filled with zeros`

  - `void __free_pages(struct page *page, unsigned int order)`
  - `void free_page(unsigned long addr)`
    `// free the page`

Method 1 & 2: kmalloc() & vmalloc()

- like malloc, a interface , defined in <linux/slab.h>
- `void *kmalloc(size_t size, gfp_t flags)`
  `// returns a pointer to a region of memory, has flag`
  `// flag including : action modifiers, zone modifiers, types`
  `// GFP_ATOMIC : use in irq, bh(can't sleep)`
  `// GFP_NOFS : can block, can turn on disk I/O`
  `// GFP_KERNEL : may block, safe in context switch…`
- `void kfree(const void *ptr)`
  `// release the memory`

- `void *vmalloc(unsigned long size)`
  `// continuous virtual memory block, but not in physical memory`
- `void vfree(const void *addr)`
  `// release`

Topic : Memory Allocation

- (1) free list (mentioned in intro of OS):
  - use linked list to link all the free memory, insert and remove
  - First fit, Best fit, Worst fit
  - May cause a lot of internal fragmentation
- (2) Memory pool:
  - Divide into different size of memory block
  - Reduce linear traverse time
  - How to determine the size of block?
- (3) Buddy system (concept is mentioned in intro of OS):
  - Memory pool, allow two nearby memory block to merge
  - If the memory block is too large, divide in half…, and move
  - reduce internal frag, overhead of merge and moving

Internal Fragmentation:

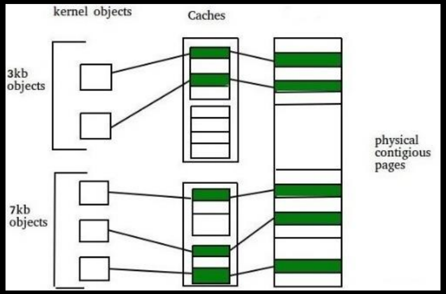
The allocate memory to process is larger than requested
Wouldn't use and can't use

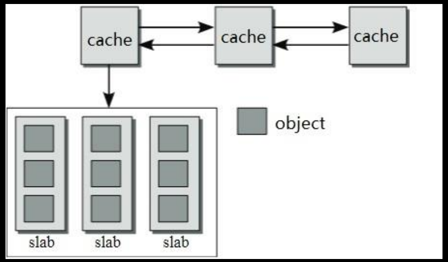
External Fragmentation:

The total free memory is larger than requested, but non-continuous

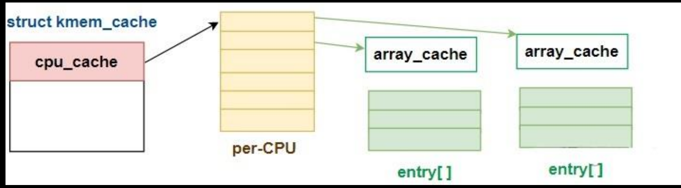
Topic : Memory Allocation - Method 3 : Slab

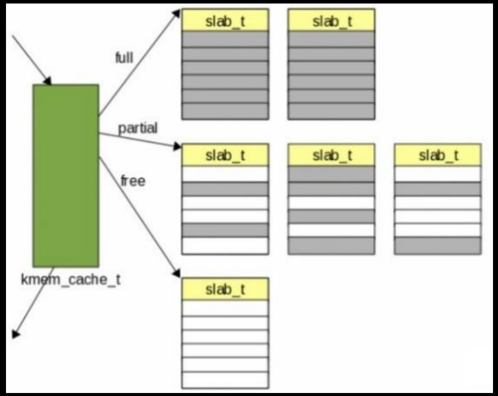
- buddy system is formed of page frame, slab use kernel object(inode)
- Slab allocator:
  - All object has a cache, allocate & release with cache(buddy)
  - cache is made up of several slabs
    - slab contain several page frames
    - coloring : Use offset to stagger the area (competing same cache line)
      improve the performance of reading

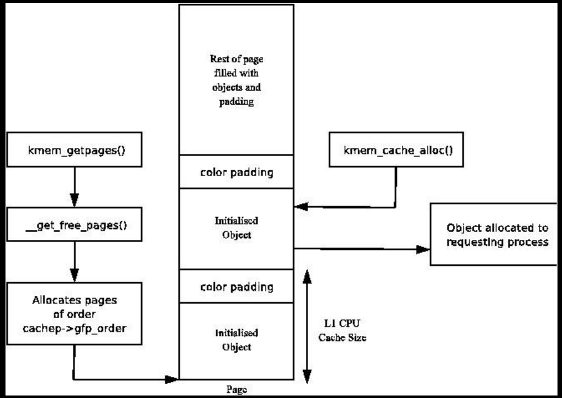




- Path
  - 1st level : Fast Path, 2nd level : Slow Path, 3rd level : Very Slow Path







Topic : Memory Allocation – Method 3 : Slab

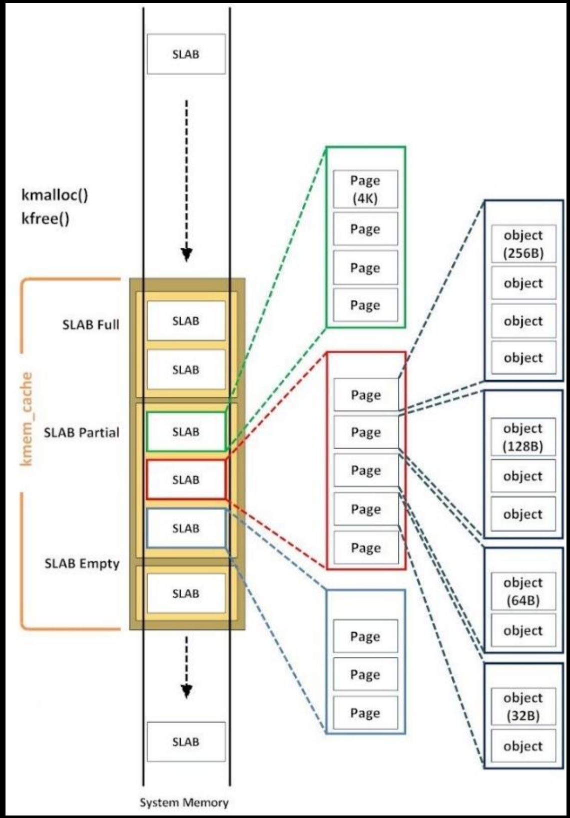
- High Utilization, easy to deal with frequently operation
- Slob, Slab, Slub
- Slab interface
  - `struct kmem_cache *kmem_cache_create(name, size, align, flags, *ctor)`

    `// slab descriptor, contain objs with same size, coloring`
  - `int kmem_cache_destroy(struct kmem_cache *cachep)`

    `// 1. make sure slab is free 2. can't access cache during time`
- Allocating from the Cache
  - `void *kmem_cache_alloc(*cachep, flags)`

    `// ptr to object from the given cache cachep`
  - `void kmem_cache_free(*cachep, *objp)`

    `//  free an object and return it to its originating slab`



Method 4: High Memory Mappings kmap()

- Physical memory higher than 896MB are HIGHMEM can temporary map to kernel address space (3GB~4GB)
- Permanent Mappings
  - `void *kmap(struct page *page)`
    `// use in both HIGH & LOW MEM, process context(can sleep)`
  - `void kunmap(struct page *page) // permanent mapping limited, free when no needed`
- Temporary Mappings
  - `void *kmap_atomic(struct page *page, enum km_type type)`
    `// atomic mapping, often use in interrupt routine`
  - `void kunmap_atomic(void *kvaddr, enum km_type type)`
    `// non blocking`

Allocation on kernel stack

- Single-Page Kernel Stacks
  - In early era, the kernel stack is the same size as page
  - Reason : In run time, prevent memory fragmentation
  - Because of the size, interrupt process is not put in the kernel stack
  - A new stack is created : Interrupt stack

- Make sure keeping stack usage to a minimum, ex: local variable
  - The kernel stack overflow is catastrophic; no warning, cover the nearby stack
  - Avoid using static allocation, use dynamic instead

CPU Allocation & Interface : SMP

```
unsigned long my_percpu[NR_CPUS];

int cpu;
cpu = get_cpu();    /* get current processor and disable kernel preemption */
my_percpu[cpu]++;   /* ... or whatever */
printk("my_percpu on cpu=%d is %lu\n", cpu, my_percpu[cpu]);
put_cpu();          /* enable kernel preemption */
```

Declare & access array store CPU info data

```
DEFINE_PER_CPU(type, name);

get_cpu_var(name)++;    /* increment name on this processor */
put_cpu_var(name);      /* done; enable kernel preemption */
per_cpu(name, cpu)++;   /* increment name on the given processor */
```

Per-CPU Data at Compile-Time

```
void *alloc_percpu(type);  /* a macro */
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(const void *);
```

Per-CPU Data at Runtime

Why using per-cpu data?

1. Reduction in locking requirements(only this cpu can access this data)
2. Reduces cache invalidation (Do not need synchronizations)

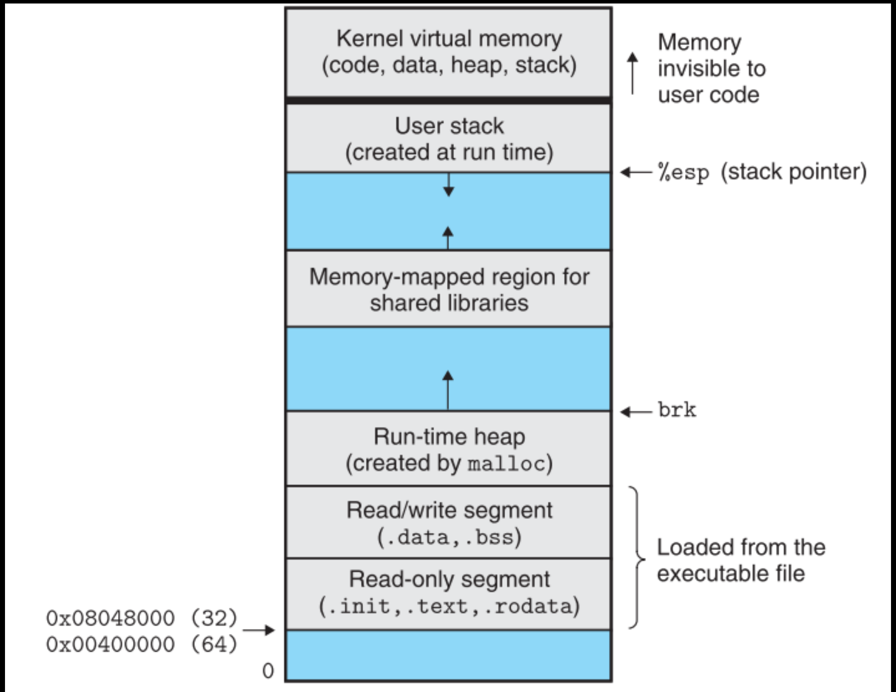
Final : Choosing Between Allocation Function

- Need continuous physical mem:
  - Use `kmalloc()`
  - GFP_ATOMIC & GFP_KERNEL
- Not need physically contiguous page, only virtually contiguous:
  - Use `vmalloc()`
  - Some overhead compare to kmalloc, guarantee continuous vmem(like user space)
- Creating and destroying many large data structures:
  - Use `slab layer`
  - Don’t need to allocate memory, often return object from cache
  - many method (like kmalloc is actually implement by slab)
- Allocate from high memory:
  - Use `kmap()`

Process Address Space

Process Address Space

- A virtual, continuous, individual memory section for every process
- Structure defined in <linux/scched.h>
  - mm_user, mm_count : # of thread share
  - mmap, mm_rb : memory area in address space
- Allocation
  - `fork() → copy_mm(), child : allocate_mm()`
- Destroy
  - `exit_mm() → mmput() → mmdrop() → free_mm()`
- Kernel thread
  - kernel thread has no PAS (no context)
  - mm→NULL



VM Areas & Manipulation

- Structure defined in <linux/mm_types.h>
- VMA flags & Operations <linux/mm.h>
  - Behavior and information about pages
- Operations
  - `void open(struct vm_area_struct *area)`
    `// memory area is added to an address space`
  - `int fault(struct vm_area_sruct *area, struct vm_fault *vmf)`
    `// page which not present in physical memory is accessed`
  - `int access(…)`
- Manipulation
  - `find_vma()`
  - `find_vma_prev()`
  - `find_vma_intersection()`

```c
struct vm_area_struct {
        struct mm_struct             *vm_mm;          /* associated mm_struct */
        unsigned long                vm_start;        /* VMA start, inclusive */
        unsigned long                vm_end;          /* VMA end , exclusive */
        struct vm_area_struct        *vm_next;        /* list of VMA's */
        pgprot_t                     vm_page_prot;    /* access permissions */
        unsigned long                vm_flags;        /* flags */
        struct rb_node               vm_rb;           /* VMA's node in the tree */
        union {          /* links to address_space->i_mmap or i_mmap_nonlinear */
                struct {
                        struct list_head        list;
                        void                    *parent;
                        struct vm_area_struct   *head;
                } vm_set;
                struct prio_tree_node prio_tree_node;
        } shared;
        struct list_head             anon_vma_node;     /* anon_vma entry */
        struct anon_vma              *anon_vma;         /* anonymous VMA object */
        struct vm_operations_struct  *vm_ops;           /* associated ops */
        unsigned long                vm_pgoff;          /* offset within file */
        struct file                  *vm_file;          /* mapped file, if any */
        void                         *vm_private_data;  /* private data */
};
```

Flag	Effect on the VMA and Its Pages
VM_READ	Pages can be read from.
VM_WRITE	Pages can be written to.
VM_EXEC	Pages can be executed.
VM_SHARED	Pages are shared.
VM_MAYREAD	The VM_READ flag can be set.
VM_MAYWRITE	The VM_WRITE flag can be set.
VM_MAYEXEC	The VM_EXEC flag can be set.
VM_MAYSHARE	The VM_SHARE flag can be set.
VM_GROWSDOWN	The area can grow downward.
VM_GROWSUP	The area can grow upward.
VM_SHM	The area is used for shared memory.
VM_DENYWRITE	The area maps an unwritable file.
VM_EXECUTABLE	The area maps an executable file.
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_SEQ_READ	The pages seem to be accessed sequentially.
VM_RAND_READ	The pages seem to be accessed randomly.
VM_DONTCOPY	This area must not be copied on fork().
VM_DONTEXPAND	This area cannot grow via mremap().
VM_RESERVED	This area must not be swapped out.
VM_ACCOUNT	This area is an accounted VM object.
VM_HUGETLB	This area uses hugetlb pages.
VM_NONLINEAR	This area is a nonlinear mapping.

mmap() & mummap() : Create & delete interval

- `do_mmap(*file, addr, len, prot, flag, offset)`
  - `will create anew virtual memory`
  - `do_mmap2(): call from user space`

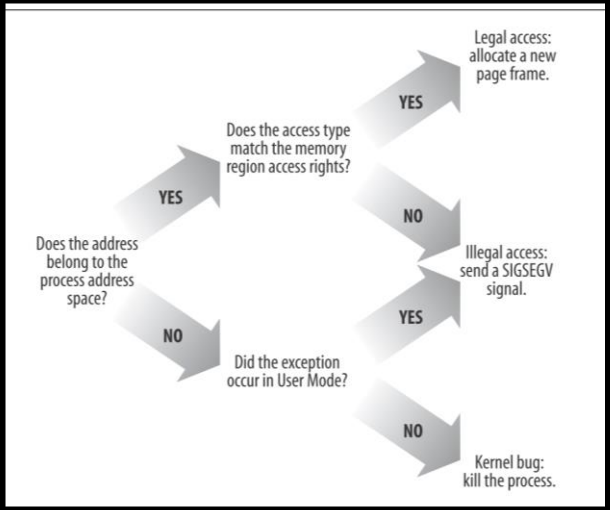
- `do_munmap(*mm, start, len)`
  - `defined in mm/mmap.c`
  - `delete specific address`

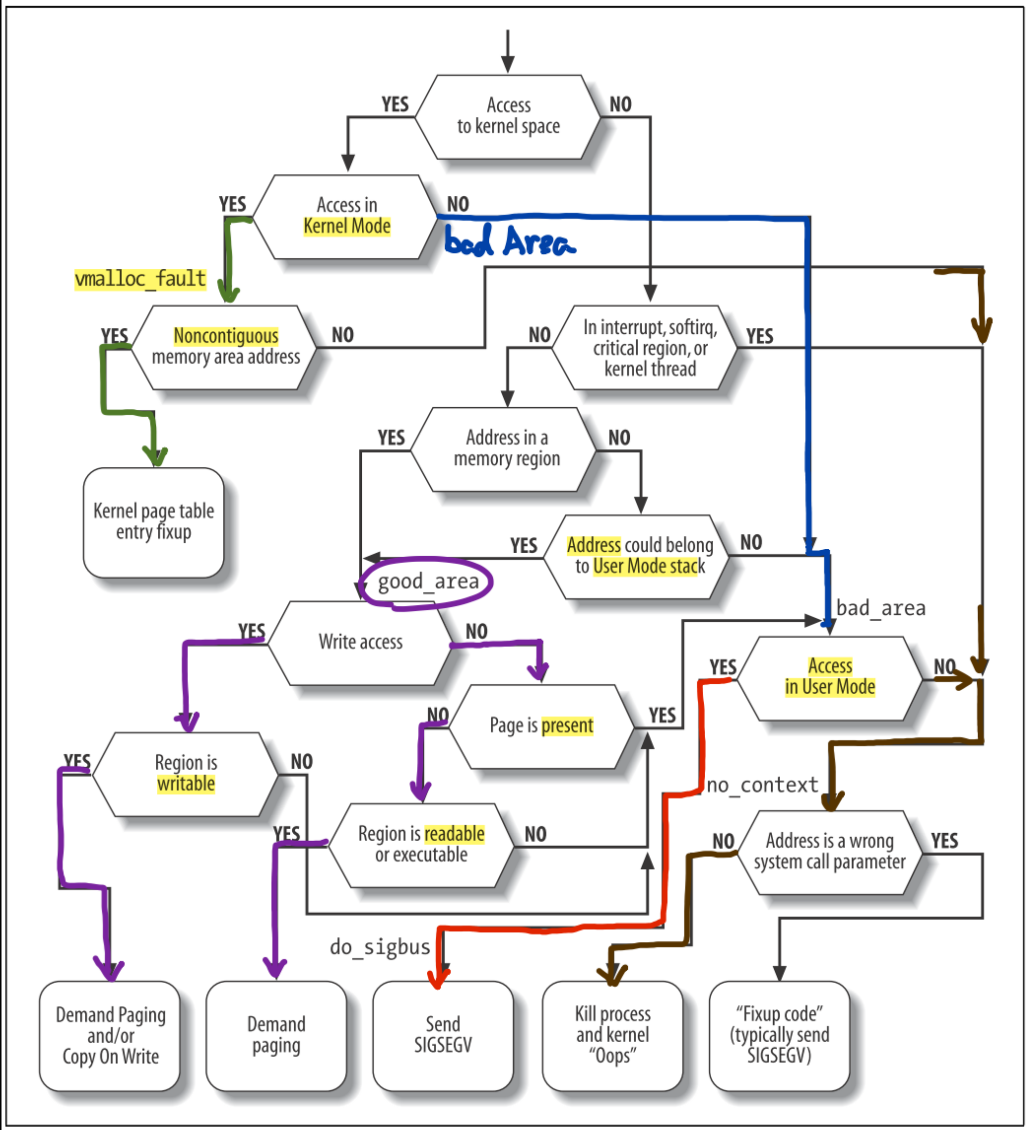
Heap management

- `void *malloc(size)`
  `// allocate space for specific size, the value is not initialized`
- `void *calloc(n,size)`
  `// allocate space with initial value`
- `void *realloc(*ptr, size)`
  `// resize the space allocate previously`
- `void *free(*ptr)`
  `// release the space which ptr point to`

Page Table, TLB & Page Fault

- Page Fault handling:
  - do_page_fault()
    - Faulty Address Inside the Address Space
    - Faulty Address Outside the Address Space
    - Demand Paging
    - Copy On Write
    - Noncontiguous Memory Area Accesses





Page Cache & Write Back

Page Cache

- Reduce the need of disk I/O manipulating
- Write :
  - direct write to cache, set dirty bit of the page(dirty list)
  - write back to disk periodically

Cache Eviction (page replacement)

- LRU
- Two-list Strategy (modified LRU)
  - Maintain active & inactive list(queue)

The Linux Page Cache

- Kernel can quickly return requested page from memory
- address_space object (vm_area_struct) will manage cache & page I/O
  - may contain multiple noncontiguous physical disk blocks
  - linux page cache use inode structure with I/O extension

- address_space manipulate
  - `int (*writepage)(struct page *, struct writeback_control *);`
    - `SetPageDirty(page); when the page is modified`
    - `Use writepage() later`
    - `check if the page needed is in cache, write request, copy from user space to kernel cache`
    - `write to disk`
  - `int (*readpage) (struct file *, struct page *);`
    - `page = find_get_page(mapping, index); allocated and added to the page cache`
    - `read data from disk, add to page cache and return`

- Radix Tree
  - defined in <linux/radix_tree.c>
  - to help checking if the page is in page cache
  - each address_space obj has it own radix tree
  - call by find_get_page()

Buffer Cache

- buffer is the in-memory representation of a single physical disk block
  - Manipulate file : page cache
  - Manipulate disk block : buffer cache

- ver2.4 unify the need of cache page & cache buffer

Flusher Threads

- Write back dirty page when:
  (1) too little free memory (2) time limited (3) sync(), fsync() system call

- flusher thread will take the job
  - free space < `dirty_background_ratio`
  - counter > `dirty_expire_interval`
  - defined in `mm/page-writeback.c, mm/backing-dev.c, fs/fs-writeback.c .`
  - Laptop mode, set two parameters larger

- Multi-thread (avoid congestion) after ver2.6
  - Threads are associated with a block (synchronization & simple)