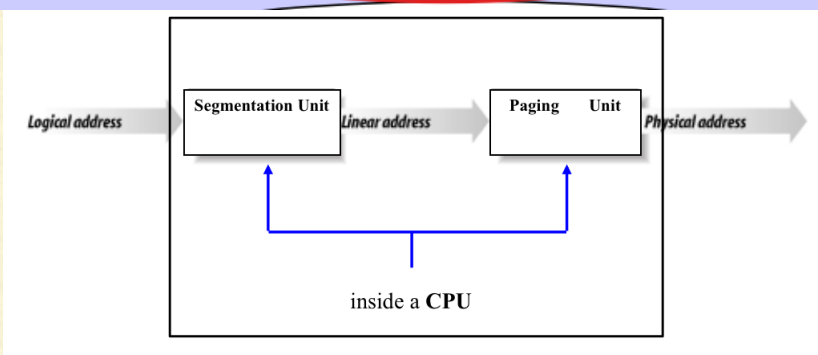
3-9-2

Memory Addressing



80386, IA-32 provides two logical

Real Mode
Compatibility with older processors
bootstrap
Protected Mode
In this chapter we only discuss this mode.

logical address is decided by

a16-bit ***segment selector*** (segment identifier)
and
a 32-bit ***offset*** within the segment identified by the segment selector.

CPU Privilege Levels

- The **cs** register includes a 2-bit field that specifies the Current Privilege Level (**CPL**) of the **CPU**.
  - The value 0 denotes the highest privilege level, while the value 3 denotes the lowest one.
- Linux uses only levels **0** and **3**, which are respectively called Kernel Mode and User Mode.

值越小, level 越大

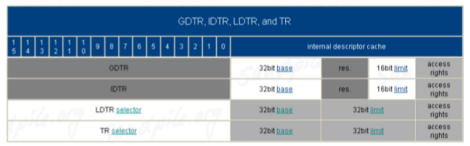
GDT vs. LDT

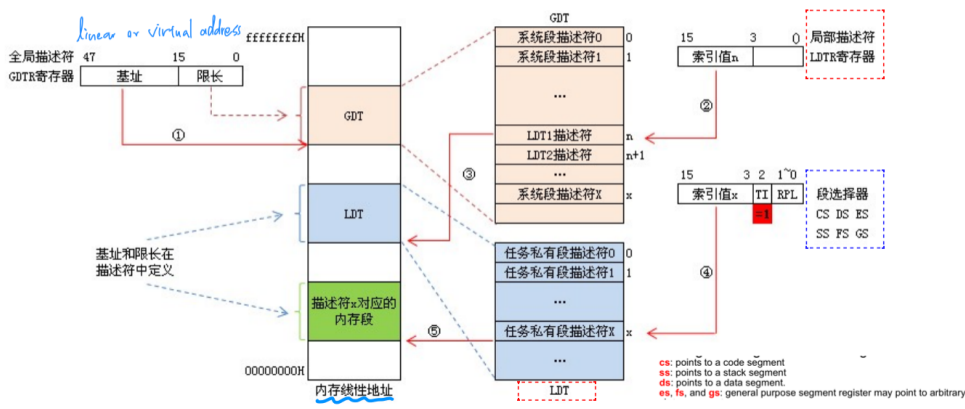
Segment Descriptors are stored either in the ***Global Descriptor Table*** (**GDT**) or in the ***Local Descriptor Table*** (**LDT**).

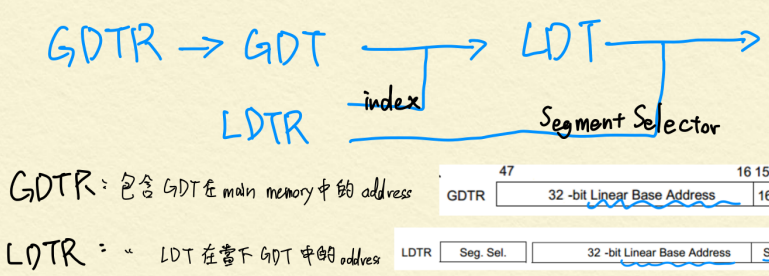
Usually only one **GDT** is defined, while each ***process*** is permitted to have its own **LDT** if it needs to create additional segments besides those stored in the **GDT**.

LDT可視為 GDT 的 cache
(∵ GDT很慢)

LDT Usage [松涛琴声]



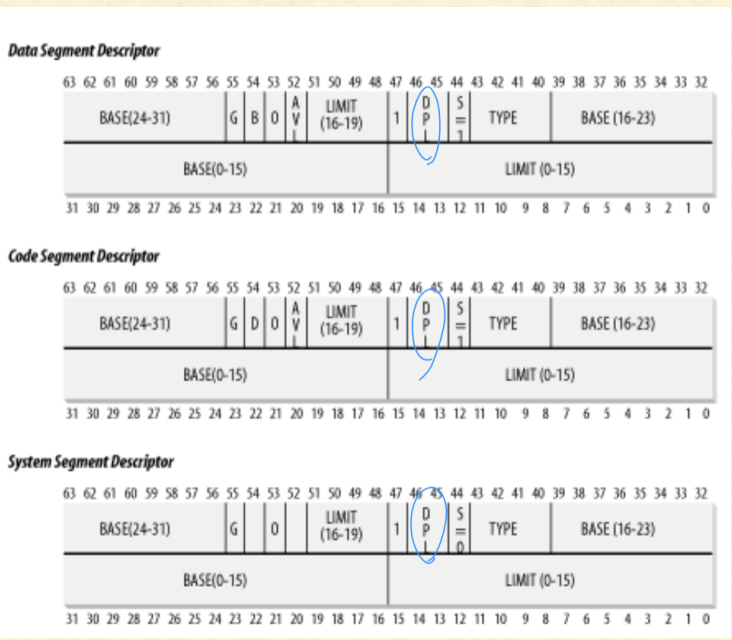


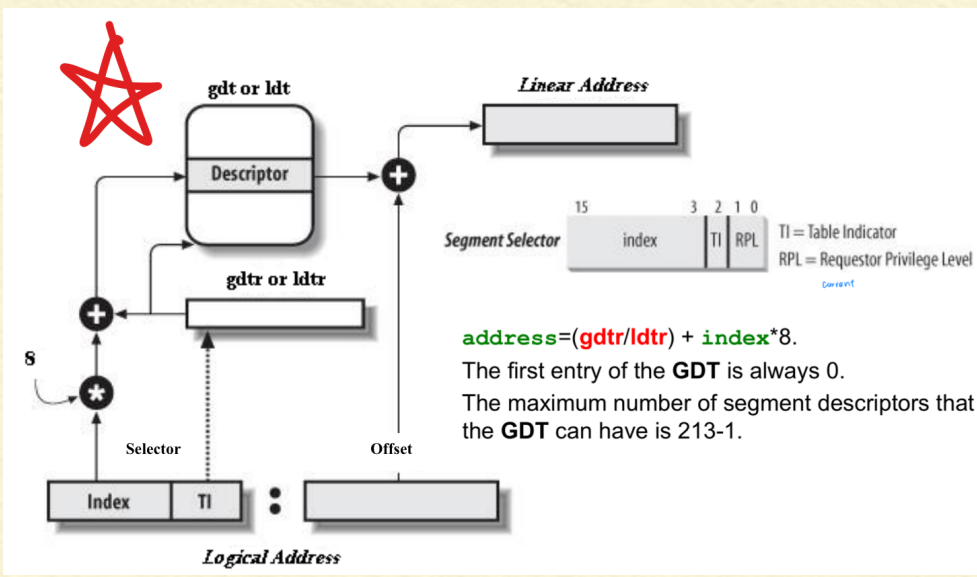


Segment Descriptor Format

Base field (32): *the linear address* of the first byte of the segment. (bits)
G granularity flag (1): 0 (byte); 1 (4**K** bytes).
Limit field (20).
S system flag (1): 0 (system segment); 1 (normal segment).
Type field (4): segment type and its access rights.
DPL (Descriptor privilege level) (2):
Segment-present flag
D/B flag
Reserved bit
AVL flag

segment need which privilege level





Segment	Base	G	Limit	S	Type	DPL	D/B	P
user code	0x00000000	1	0xffffff	1	10	3	1	1
user data	0x00000000	1	0xffffff	1	2	3	1	1
kernel code	0x00000000	1	0xffffff	1	10	0	1	1
kernel data	0x00000000	1	0xffffff	1	2	0	1	1

3.9-3

Each **GDT** includes
21 segment descriptors
and
11 null, unused, or reserved entries.

Linux's GDT	*Segment Selectors*
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	*Segment Selectors*
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
ESPFIX small SS	
per-cpu	
stack_canary-20	
not used	
not used	
double fault TSS	0xf8

The Paging Unit

Page

- Contiguous linear addresses are grouped in fixed-length intervals called ***pages***.
- The term "***page***" is also refer to:
  - A set of linear addresses
  - The data contained in this group of addresses.

V, S

Page Frame

- The ***paging unit*** thinks of all **RAM** as partitioned into fixed-length ***page frames*** (physical pages).
- The size of a page is equal to the size of a page frame.
  - Usually the size of a page frame is 4**KB**; however, sometimes a larger page frame size may also be used.

- Each active process must have a ***Page Directory*** assigned to it.
  - The *physical address* of the ***Page Directory*** of the active process is stored in the control register **cr3**.

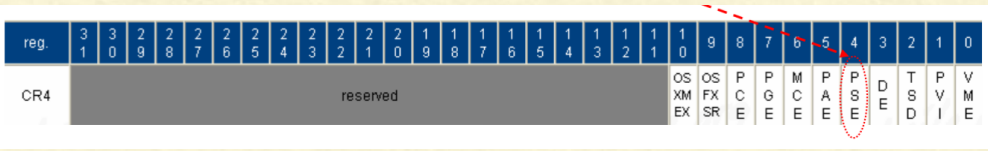
2^{10} 個 entries, (4 bytes per entry)

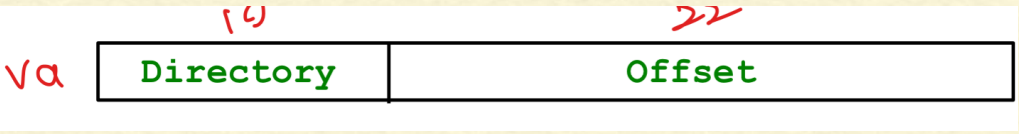
- Both ***Page Directory*** entries and ***Page Tables*** have the same structure.
  - **Present** flag
  - Field containing the 20 most significant bits of a page frame physical address.
  - **Access** flag
  - **Dirty** flag
  - **Read/write** flag
  - **User/Supervisor** flag
  - **PCD** and **PWT** flags
  - **Page size** flag
  - **Global** flag

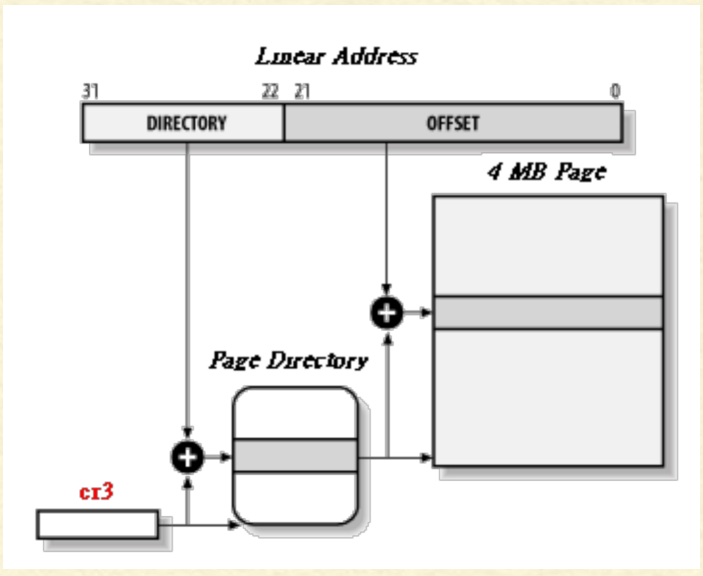
31 12	11 .. 9	8	7	6	5	4	3	2	1	0
ADDRESS	OS	0	0	D	A	0	0	U/S	R/W	P

Extended Paging

Allows page frames to be 4 MB instead of 4 KB





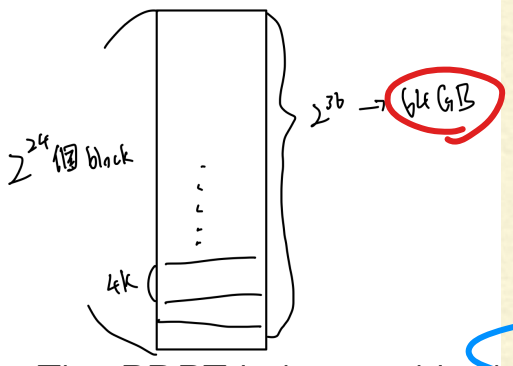


Hardware Protection Scheme

Read/Write flag:
- 0: can be read.
- 1: can be read and write.

- The ***segmentation unit*** uses four possible privilege levels to protect a segment (the two-bit request privilege levels, 0 for kernel mode, 3 for user mode).
- The ***paging unit*** uses a different strategy to protect ***Page Tables*** and page frames → the **User/Supervisor** flag.
  - **0** → **CPU**'s **CPL** must be less than 3 (i.e. for Linux, when the processor is in kernel mode.)
  - **1** → the corresponding ***Page Table*** or page frame can always be accessed.

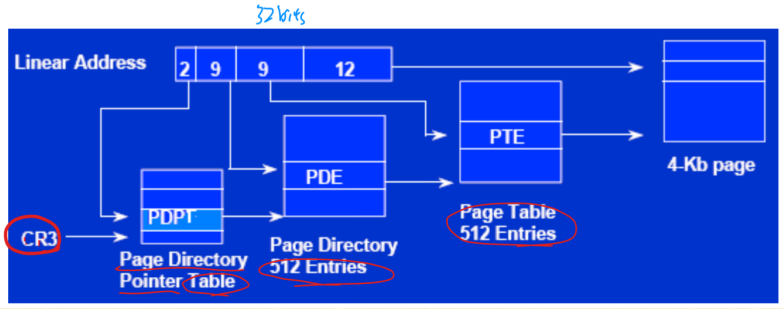
The Physical Address Extension (**PAE**) *Paging Mechanism*



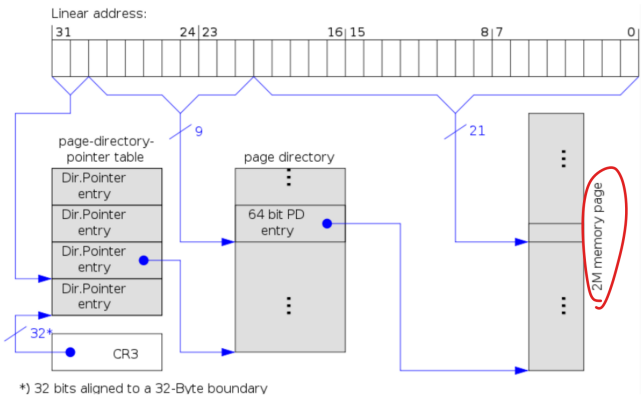
The PDPT is located in the first 4GB of RAM

- PAE is activated by setting the Physical Address Extension (PAE) flag in the cr4 control register.

PDPT(2 bits), **PD**(9 bits), **PT**(9 bits), **Offset**(12 bits).



2 M Page Size in PAE [REN]



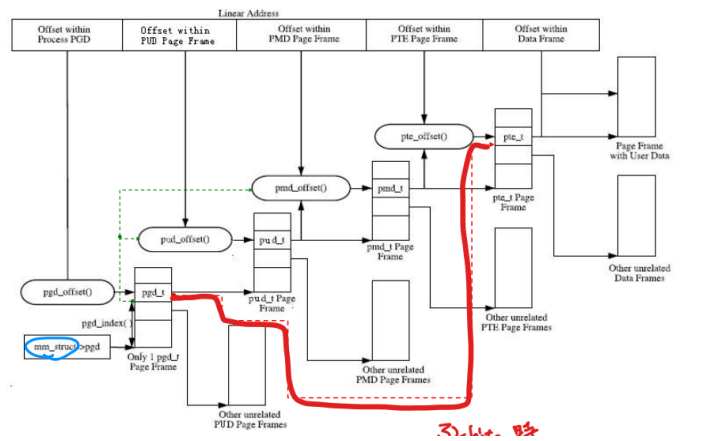
3-9-4

Linux essentially eliminates the ***Page Upper Directory*** and the ***Page Middle Directory*** fields by saying that they contain 0 bits.

- The kernel keeps a position for the Page Upper Directory and the Page Middle Directory by setting the number of entries in them to 1 and mapping these two entries into the proper entry of the Page Global Directory.

- Each process has its own Page Global Directory and its own set of page tables.

When a ***process switch*** occurs, Linux saves [kkto] the **cr3** control register in the ***descriptor*** of the process previously in execution and then loads **cr3** with the value stored in the descriptor of the process to be executed next.



Default Physical Addresses Used by Kernel [1]

The Linux kernel is installed in **RAM** starting from the physical address **0x01000000** (16**M**) by default.

- Linear addresses from **0x00000000** to **0xbfffffff** can be addressed when the process is in either User or kernel Mode.
- Linear addresses from **0xc0000000** to **0xffffffff** can be addressed only when the process is in kernel mode.

3-9-5

Memory Addressing

-- with the assistance of 江瑞敏 and 許齊顯

The content of the first entries of the ***Page Global Directory*** that map linear addresses lower than **0xc0000000** (the first **768** entries with **PAE** disabled, or the first **3** entries with **PAE** enabled) depends on the specific process.

Kernel Page Tables

After system initialization, the set of ***page tables*** are never directly used by any process or kernel thread.

How Kernel Initializes Its Own page tables

1. In the first phase, the kernel creates a limited address space including
   - the kernel's code segment
   - the kernel's data segments
   - the initial ***page tables***
   - 128 **KB** for some dynamic data structures.
2. In the second phase, the kernel takes advantage of all of the existing **RAM** and sets up the ***page tables*** properly.

Phase One

Assumption

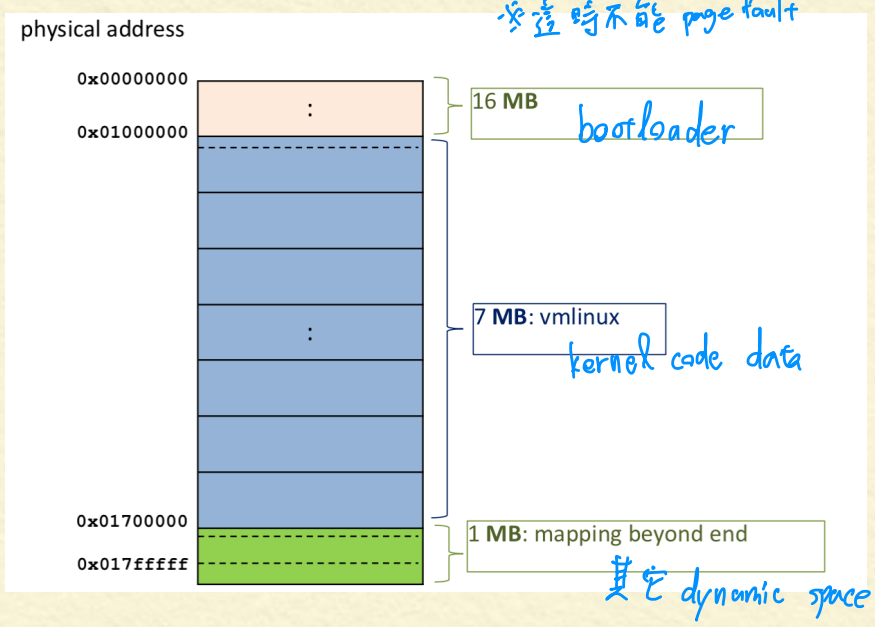
1. **CPU** architecture is **x86_32**.
2. **vmlinux**[wikipedia] size is 7**MB**.
   - On Linux systems, **vmlinux** is a statically linked executable file that contains the Linux kernel in one of the object file formats supported by Linux, which includes **ELF**, **COFF**, and **a.out**.
3. ***Boot loader*** put linux kernel at physical address **0x01000000**.

16 MB

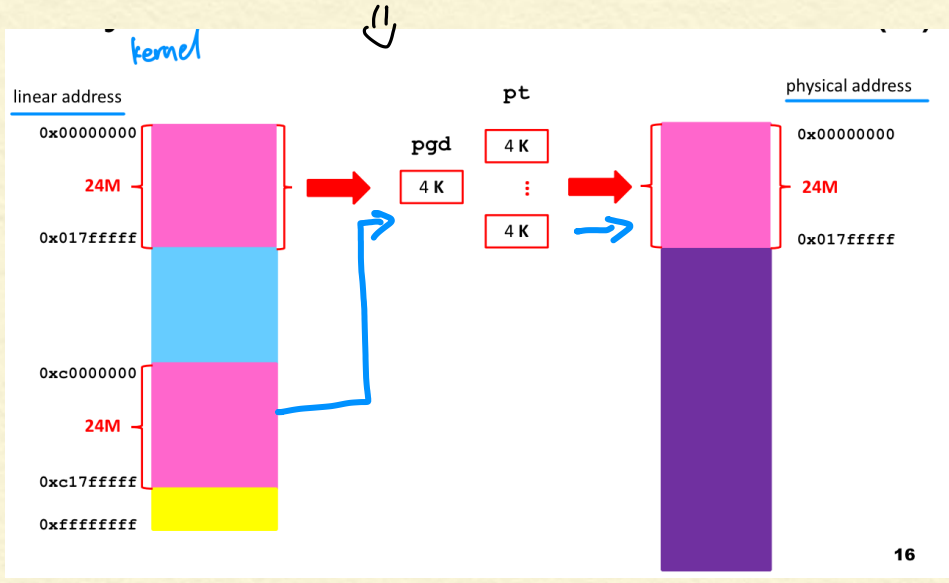
Phase One Mapping Size

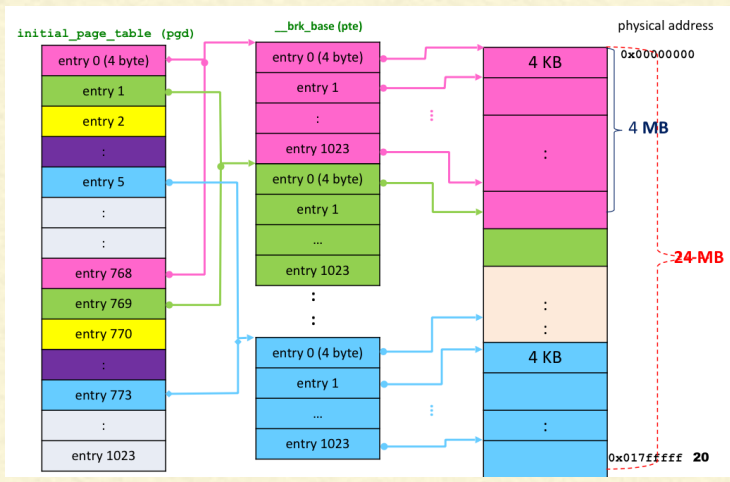
- In order to map 24 **MB** of **RAM**, 6 ***Page Tables*** are required.

16MB + 7MB + 1MB



initial_page_table in Phase One





cr3 points to `initial_page_table` → enable

Phase 2

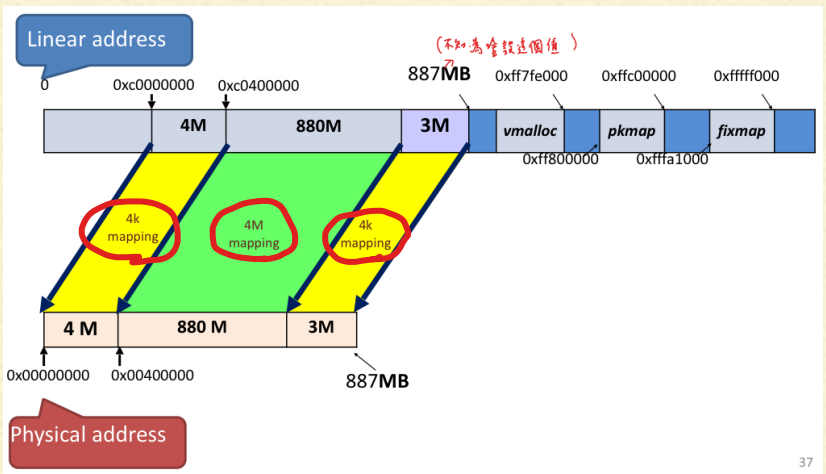
Kernel Initializes Its Own Page

- Finish the Page Global Directory
- *The final mapping provided by the kernel Page Tables must transform virtual addresses starting from* `0xc0000000` *to physical addresses starting from* `0x00000000`.
- There are two different configurations that will affect the size of the linear mapping region.
  - `CONFIG_HIGHMEM`
  - `CONFIG_NOHIGHMEM`

Phase 2
Case 1:
When **RAM** Size Is Less Than 887**MB**

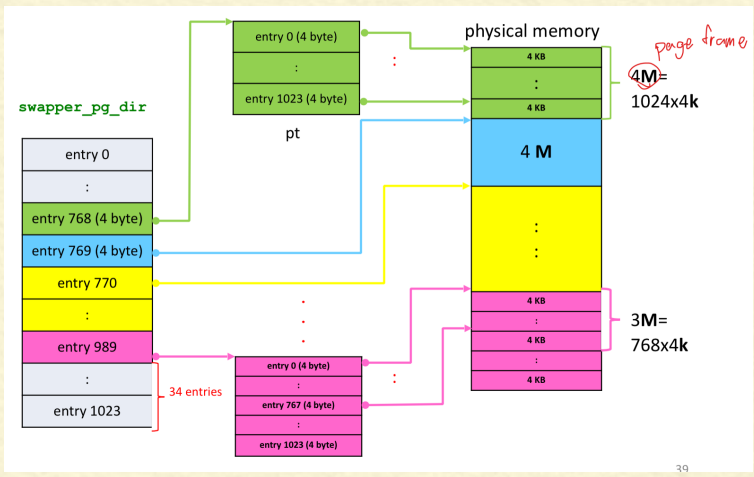
< 887MB

Assumption



37

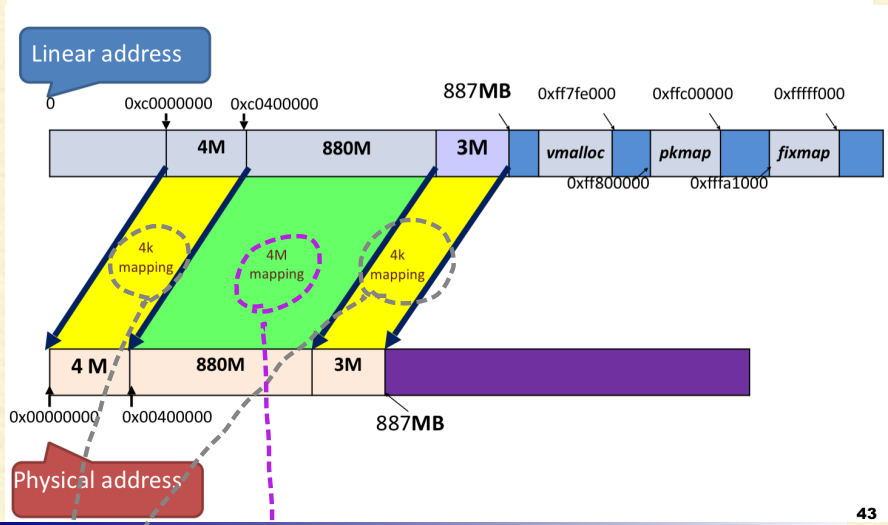
- We assume that the **CPU** is a 80x86 microprocessor supporting
  - 4 **MB** pages
    and
  - "global" **TLB** entries.
- Notice that the `User/Supervisor` flags in all ***Page Global Directory entries*** referencing linear addresses above `0xc0000000` are cleared,
  - thus denying processes in User Mode access to the kernel address space.
- Notice also that the `Page Size` flag is set
  - so that the kernel can address the **RAM** by making use of large pages.



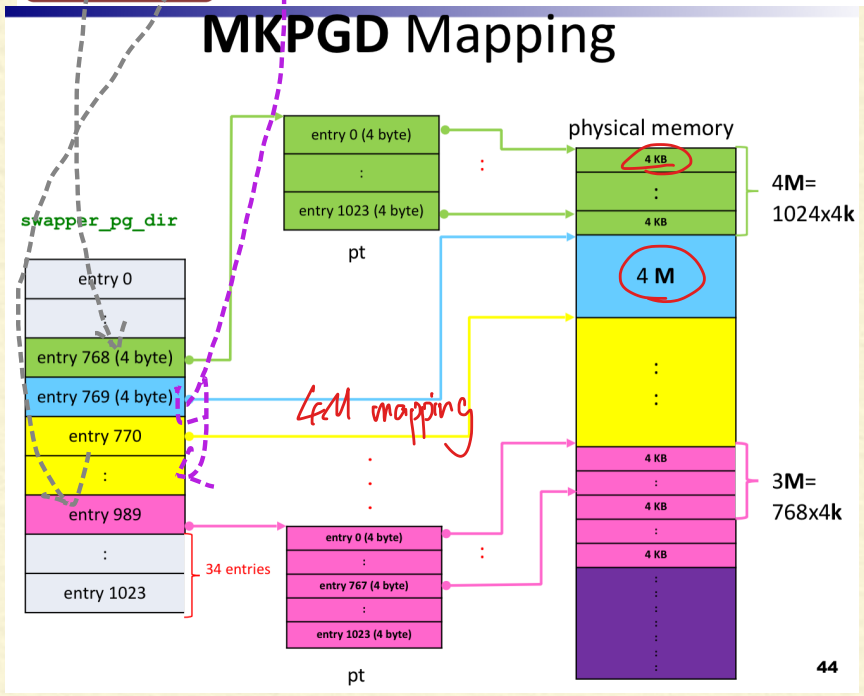
39

Phase 2
Case 2:
When **RAM** Size Is between 887**MB** and 4096**MB**

- Final kernel page table when **RAM** size is between 887 **MB** and 4096 **MB** :
  - In this case, the **RAM** CNNNOT be mapped entirely into the kernel linear address space, because the address space is only 1**GB**.
  - Therefore, during the initialization phase Linux **only** maps a **RAM** window having size of **887 MB** into the *kernel linear address space*.
  - If a program needs to address other parts of the existing **RAM**, some other linear address interval (from the 888th **MB** to the 1st **GB**) must be mapped to the required **RAM**.
    - This implies changing the value of some page table entries.

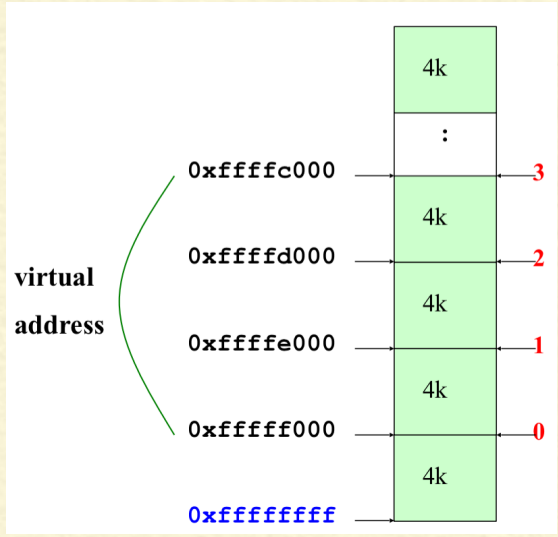


43



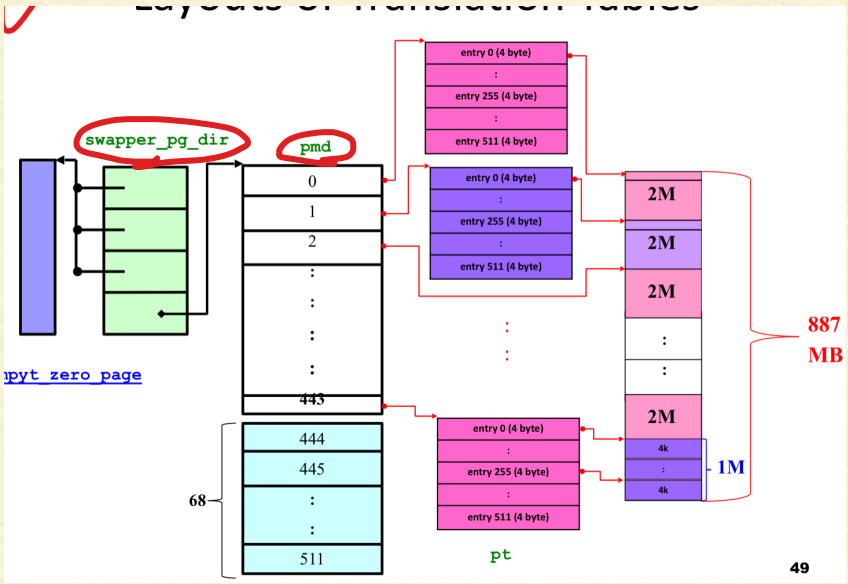
44

Phase 2
Case 3:
When **RAM** Size Is More Than 4096**MB**



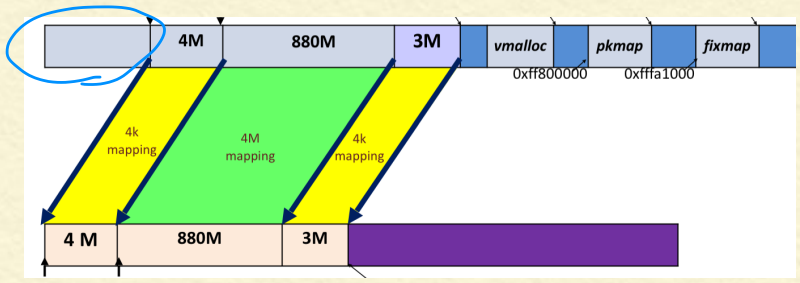
RAM Mapping Principle

- Although **PAE** handles 36-bit physical addresses, linear addresses are still 32-bit addresses.
- As in case 2, Linux maps a **887-MB RAM** window into the kernel linear address space; the remaining **RAM** is left unmapped and handled by dynamic remapping.



Fix-Mapped Linear Addresses

-- *with the assistance of* 黃建鴻, 李承恩 and 陳少宇



用在這一段的 mapping

但!! 有大約 137 MB 的 linear addr. 是永遠可用的，這 137 MB 用來

uses them to implement

- ***noncontiguous memory allocation***

and

- ***fix-mapped linear addresses***.

3-9-6 ☆特重要

20 points ↑

Processes

POSIX-compliant pthread libraries 標準(跨 UNIX平台)

- In Linux a ***thread group*** is basically a set of lightweight processes that
  - implement a multithreaded application
    and
  - act as a whole with regards to some system calls such as
    1. ■ `getpid( )`
    2. ■ `kill( )`
       and
    3. ■ `_exit( )`.

thread group leader

signal

clone : shared 的可指定

fork : shared 的都不可指定

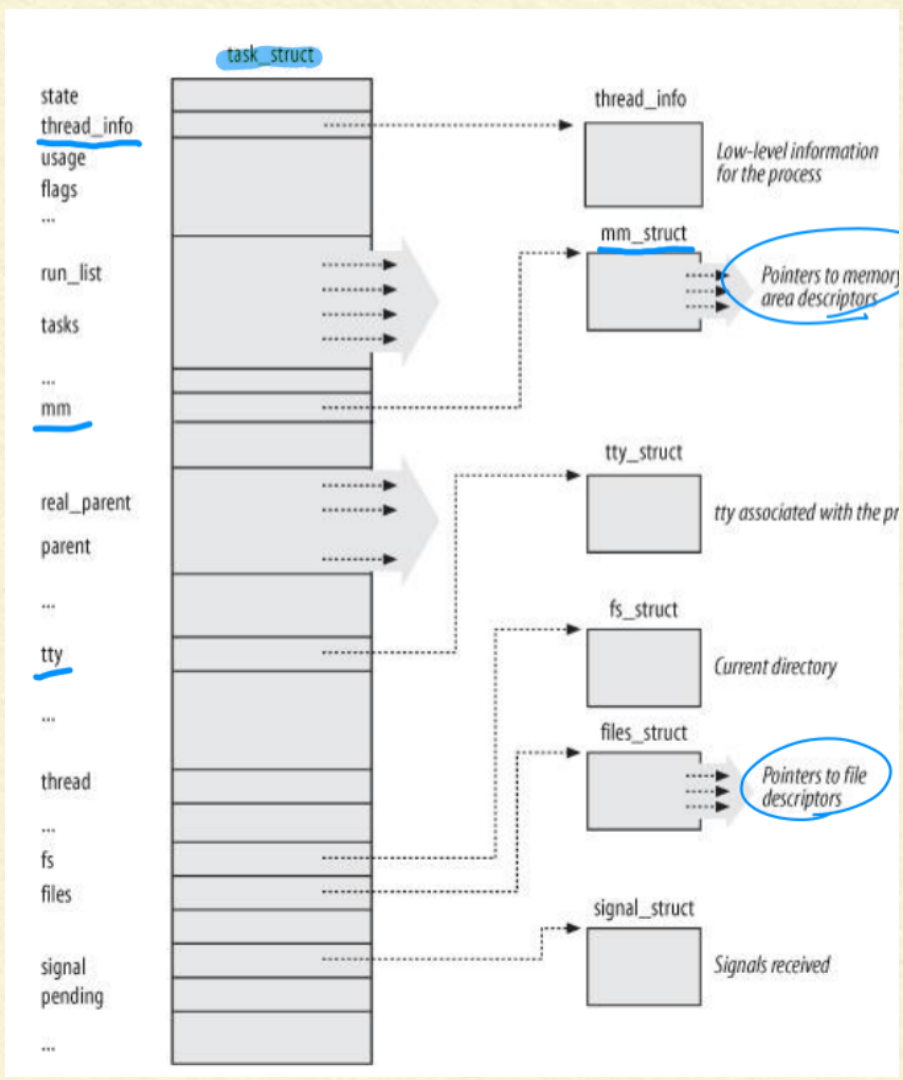
Why a Process Descriptor Is Introduced?

1. the process's priority
2. whether
   - it is running on a **CPU**
     or
   - blocked on an event
3. what address space has been assigned to it
4. which files it is allowed to address, and so on.

→ task_struct

track process 的 struct

※ task_struct 要加入新欄位時，必用 append，not insert
(∵ pointer 是用"+"的，kernel 是 hard code，用 offset 找)



Process State

TASK_RUNNING executing or not

TASK_INTERRUPTIBLE sleeping (until some condition)
ss

TASK_UNINTERRUPTIBLE
送 signal，都不會 wake up

__TASK_STOPPED

SIGSTOP signal
- Stop Process Execution

SIGTSTP signal
- SIGTSTP is sent to a process when
  - the ***suspend keystroke*** (normally **^Z**) is pressed on its controlling *tty*
    and
  - it's running in the foreground.

SIGTTIN signal
- Background process requires input

SIGTTOU signal.
- Background process requires output

When a ***stopped process*** receives **SIGCONT**, it starts running again.

__TASK_TRACED

stopped by debugger，put the process in this status

Fatal Signals 致命的

讓 kernel to kill the process 的 signal

SIGKILL is always fatal

TASK_WAKEKILL

```
#define TASK_KILLABLE   (TASK_WAKEKILL | TASK_UNINTERRUPTIBLE)
#define TASK_STOPPED    (TASK_WAKEKILL | __TASK_STOPPED)
#define TASK_TRACED     (TASK_WAKEKILL | __TASK_TRACED)
```

TASK_KILLABLE

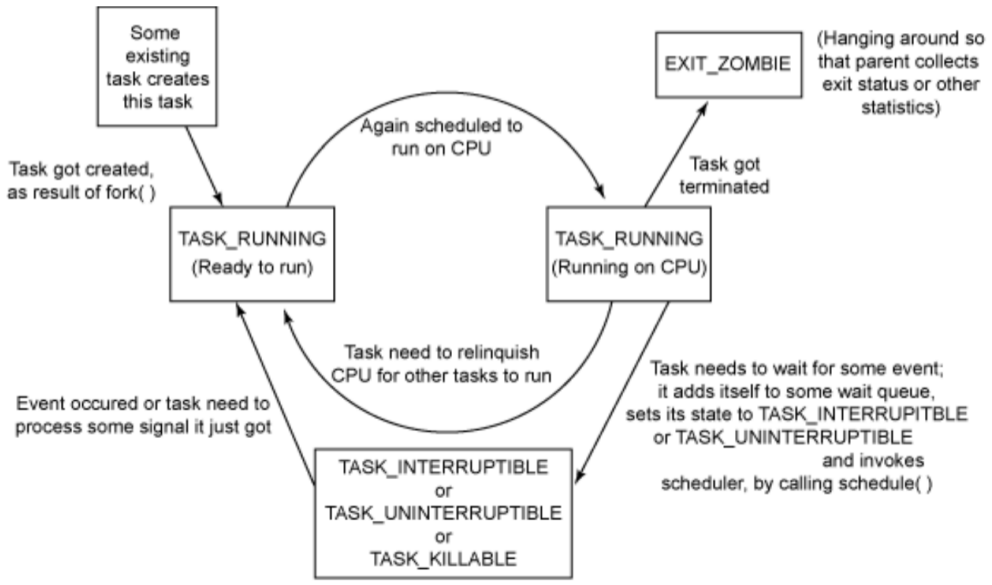
像 TASK_UNINTERRUPTIBLE
但可受 fatal signal control. (to kill)

New States Introduced in Linux 2.6.x.

EXIT_ZOMBIE

EXIT_DEAD

process 已結束且回收



Set the `state` Field of a Process

p->state = TASK_RUNNING;

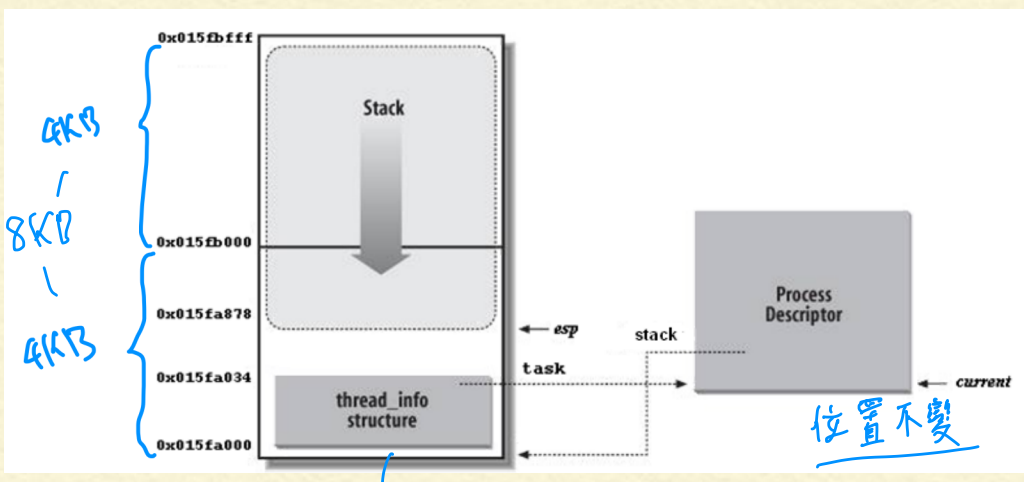
Identifying a Process

- The strict one-to-one correspondence between the process and process descriptor makes the 32-bit address of the `task_struct` structure a useful means for the kernel to identify processes.
- These addresses are referred to as ***process descriptor pointers***.
- Most of the references to processes that the kernel makes are through process descriptor pointers.

thread_info,

For each process, there are two data linked to process
1. thread_info
2. Kernel Mode process stack

```
union thread_union
{ struct thread_info thread_info;
  unsigned long stack[2048];
  /* 1024 for 4KB stacks */
};
```

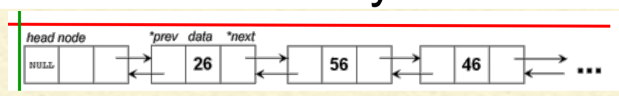


beginning of the memory area

current：目前正在執行的process

Linked List

Non-circular Doubly Linked Lists

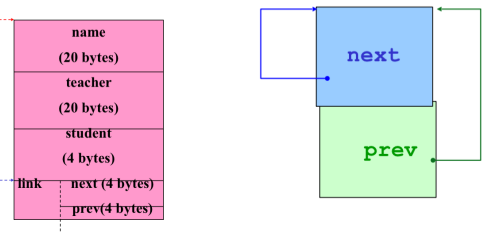


缺：waste of programmers' efforts
waste of memory to replicate

Data Structure `struct list_head`

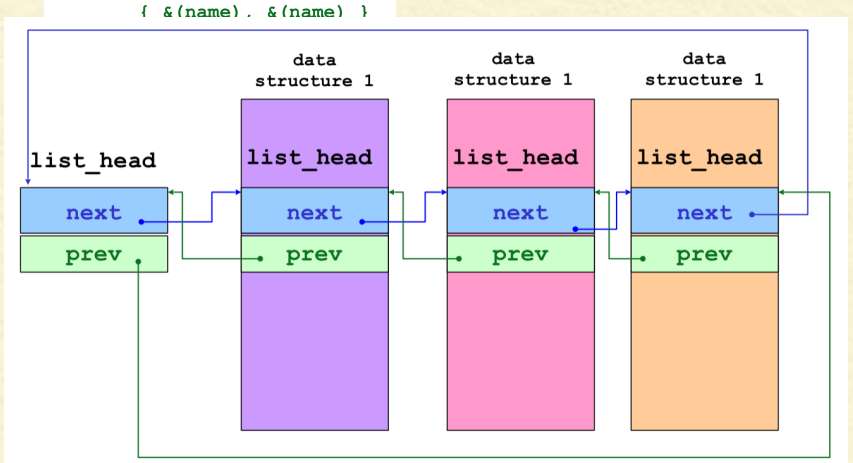
LIST_HEAD(my_list)

```
struct class{
   char  name[20];
   char  teacher[20];
   struct student_pointer  *student;
   struct list_head link;
};

struct class grad_1A;
struct list_head *poi;
poi=&(grad_1A.link);
list_entry(poi,struct class,link)
→  &grad_1A
```



`struct list_head my_list`

```
#define LIST_HEAD_INIT(name) /
        { &(name), &(name) }
```

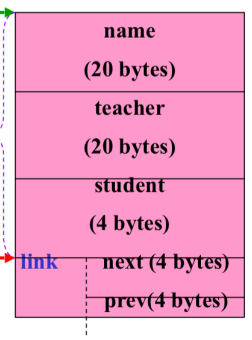



```
#define list_entry(ptr, type, member) \
((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

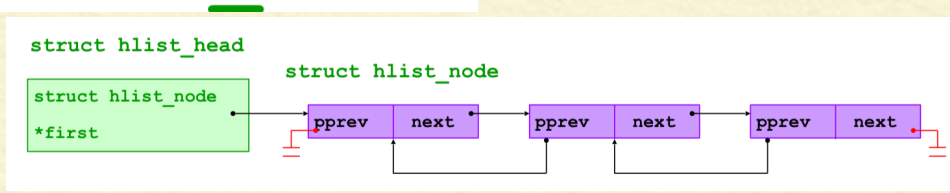
offset

```
list_entry(poi,struct class,link)
 ((struct class *)((char *)(poi)-(unsigned long)(&((struct class
*)0)->link)))
```

```
list_entry(...
)

poi - offset
= list_entry()
```



hlist_head



The Process List

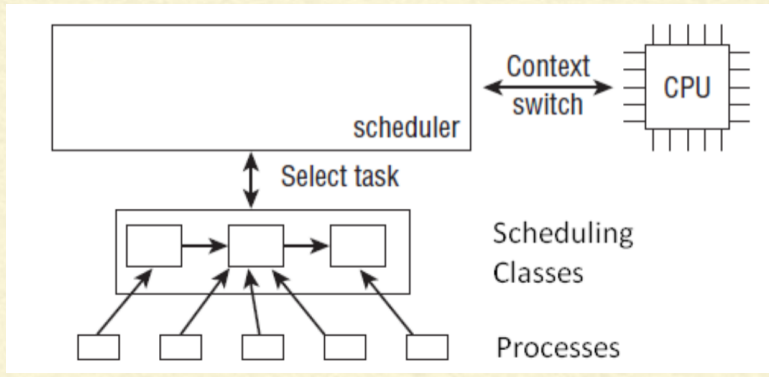
- The ***process list*** is a circular doubly linked list that links the process descriptors of all existing thread group leaders:
  - Each `task_struct` structure includes a `tasks` field of type `list_head` whose `prev` and `next` fields point, respectively, to the previous and to the next `task_struct` element's `tasks` field.

- The head of the ***process list*** is the `init_task` `task_struct` descriptor; it is the process descriptor of the so-called ***process 0*** or ***swapper***.
- The `tasks->prev` field of `init_task` points to the `tasks` field of the process descriptor inserted last in the list.

```
#define next_task(p) \
list_entry_rcu((p)->tasks.next, struct task_struct, tasks)

#define for_each_process(p) \
for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

- Earlier Linux versions put all runnable processes in the same list called ***runqueue***.
- Because it would be too costly to maintain the list ordered according to ***process priorities***, the earlier ***schedulers*** were compelled to scan the whole list in order to select the "best" runnable process.

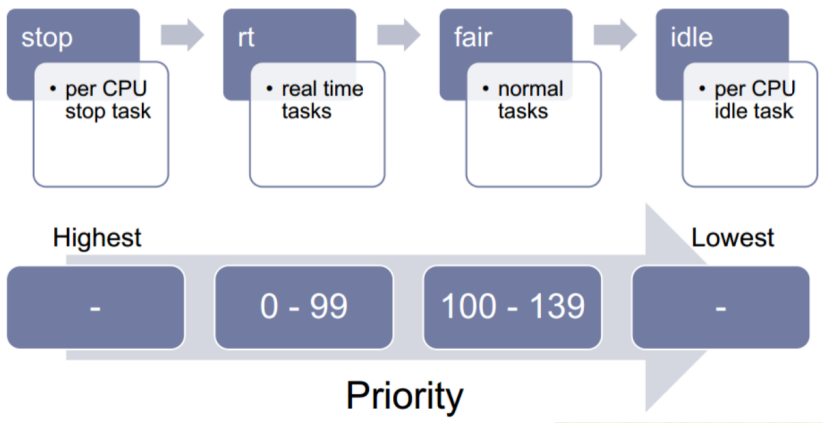
Linux 3.9 Scheduler



Scheduling Classes : used to decide which task runs next

policies

1. completely fair scheduling
2. real-time scheduling
3. scheduling of the *idle task* when there is nothing to do

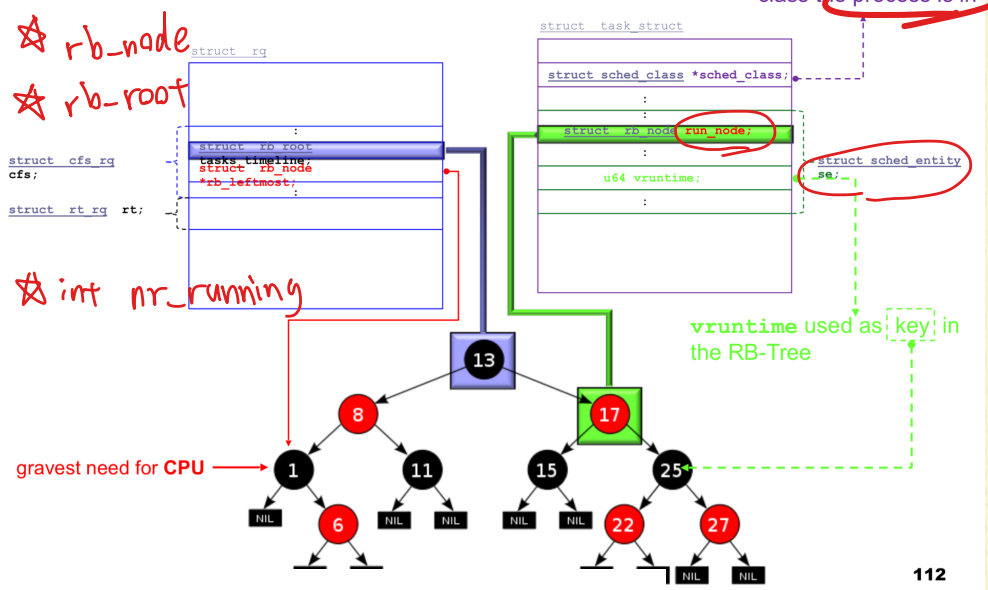


- The `stop_sched_class` is to stop **cpu**, using on **SMP** system, for
  - load balancing
    and
  - **cpu** hotplug.

CFS Run Queue

紅黑樹

CFS Run Queue of a CPU




```
struct rb_root tasks_timeline;
struct rb_node *rb_leftmost;
```

3-9-8

40 points!

Processes

runqueue

The runqueue lists group all processes in a **TASK_RUNNING** state.

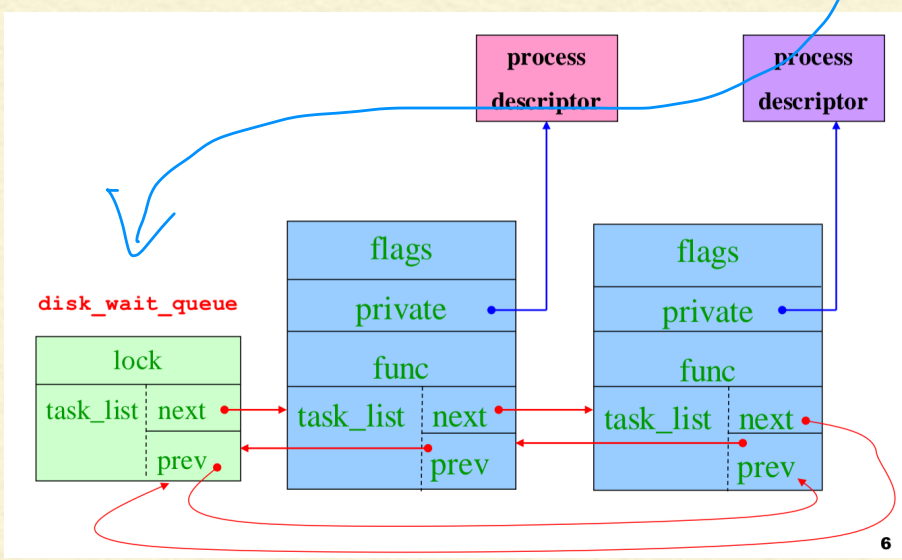
- Processes in a **TASK_STOPPED**, **EXIT_ZOMBIE**, or **EXIT_DEAD** state are not linked in specific lists.

Wait queue

- Processes in a **TASK_INTERRUPTIBLE** or **TASK_UNINTERRUPTIBLE** state are

Wait queues implement conditional waits on events:

- an interrupt, such as for a disk operation to terminate
- process synchronization
- timing: such as a fixed interval of time to elapse



wait_queue_head_t:

```
struct __wait_queue_head
{     spinlock_t lock;
      struct list_head task_list;
};
typedef struct __wait_queue_head  \
wait_queue_head_t;
```

for synchronization

the head of a list of waiting processes

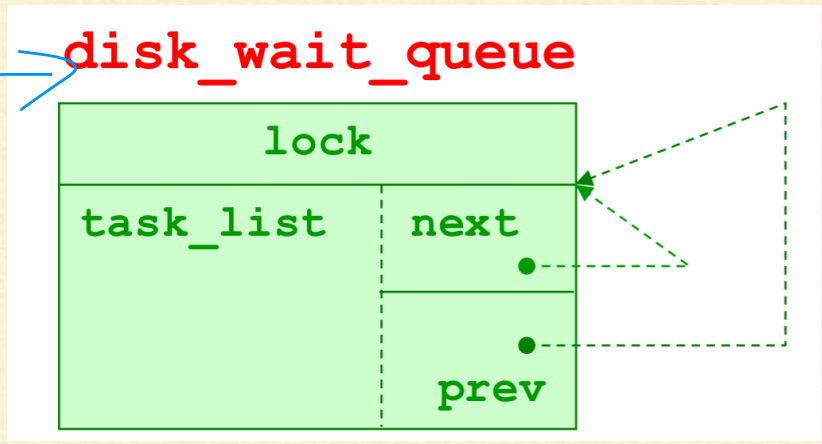
```
struct __wait_queue {
  unsigned int flags;
  #define WQ_FLAG_EXCLUSIVE        0x01
  void *private;
  wait_queue_func_t func;
  struct list_head task_list;
};

typedef struct __wait_queue wait_queue_t;
```

flags

- flags has the value **WQ_FLAG_EXCLUSIVE**
  - A set **WQ_FLAG_EXCLUSIVE** flag indicates that the waiting process would like to be woken up exclusively.

Declare a new Wait Queue Head

```
DECLARE_WAIT_QUEUE_HEAD(disk_wait_queue)
```



- The add_wait_queue( ) function inserts a nonexclusive process in the first position of a wait queue list.
- The add_wait_queue_exclusive( ) function inserts an exclusive process in the last position of a wait queue list.

- sleep_on( )
- interruptible_sleep_on( )
- sleep_on_timeout( )
- interruptible_sleep_on_timeout( )
- wait_event and wait_event_interruptible macros

```
有Bug
void sleep_on(wait_queue_head_t *wq)
{ wait_queue_t wait;

  init_waitqueue_entry(&wait, current);
  current->state = TASK_UNINTERRUPTIBLE;
  add_wait_queue(wq,&wait);
  /* wq points to the wait queue head */
  schedule( );
  remove_wait_queue(wq, &wait);
}
```

(過;度) → running 的 interruptible

(1) Remove **current** from the runqueue.

(2) In order to make **schedule()** resume its execution, there must be some other kernel control path setting this process back to **TASK_RUNNING** state and putting it back to the runqueue after (1) is executed.

- `prepare_to_wait( )`
- `prepare_to_wait_exclusive( )`

and

- `finish_wait( )` functions

way to put the current process to sleep in a wait queue.

prepare_to_wait( )

```
static inline int list_empty(const struct list_head *head)
{
    return head->next == head;
}

static inline void
__add_wait_queue(wait_queue_head_t *head, wait_queue_t *new)
{
    list_add(&new->task_list, &head->task_list);
}

void
prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)
{
    unsigned long flags;

    wait->flags &= ~WQ_FLAG_EXCLUSIVE;
    spin_lock_irqsave(&q->lock, flags);
    if (list_empty(&wait->task_list))
            __add_wait_queue(q, wait);
    set_current_state(state);
    spin_unlock_irqrestore(&q->lock, flags);
```

```
DEFINE_WAIT(wait);
for (;;) {
      :
  prepare_to_wait_exclusive(&wq, &wait, TASK_INTERRUPTIBLE);
  /* wq is the head of the wait queue */
    :
  if (condition)
    break;
  schedule();
    :
}
finish_wait(&wq, &wait);
```

```
if (!condition)
{
  DEFINE_WAIT(__wait);

  for (;;) {
    prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);
    if (condition)  break;
    schedule();
  }
  finish_wait(&wq, &__wait);
}
```

Remove **current** from the runqueue.

Process Resource Limits

The resource limits for the current process are stored in the `current->signal->rlim` field, that is, in a field of the process's signal descriptor.

```
struct rlimit
{ unsigned long rlim_cur;
  unsigned long rlim_max;
};
```

By using the `getrlimit( )` and `setrlimit( )` system calls, a user can always increase the `rlim_cur` of some resource up to `rlim_max`.

However, only the superuser can increase the `rlim_max` field or set the `rlim_cur` field to a value greater than the corresponding `rlim_max` field.

RLIMIT_AS

- The maximum size of process address space, in bytes.
- The kernel checks this value when the process uses `malloc( )` or a related function to enlarge its address space.
  - P.S.: See the section "The Process's Address Space" in Chapter 9.

RLIMIT_CORE

- The maximum core dump file size, in bytes.
- The kernel checks this value when a process is aborted, before creating a core file in the current directory of the process.
  - P.S.: See the section "Actions Performed upon Delivering a Signal" in Chapter 11.
- If the limit is 0, the kernel won't create the file.

RLIMIT_CPU

- The maximum CPU time for the process, in seconds.
- If the process exceeds the limit, the kernel sends it a `SIGXCPU` signal, and then, if the process doesn't terminate, a `SIGKILL` signal.

RLIMIT_DATA

- The maximum heap size, in bytes.
- The kernel checks this value before expanding the heap of the process.

RLIMIT_FSIZE

- The maximum file size allowed, in bytes.
- If the process tries to enlarge a file to a size greater than this value, the kernel sends it a `SIGXFSZ` signal.

RLIMIT_LOCKS

- Maximum number of file locks (currently, not enforced).

RLIMIT_MEMLOCK

- The maximum size of nonswappable memory, in bytes.
- The kernel checks this value when the process tries to lock a page frame in memory using the `mlock( )` or `mlockall( )` system calls

RLIMIT_MSGQUEUE

- Maximum number of bytes in POSIX message queues.

RLIMIT_NOFILE

- The maximum number of open file descriptors.
- The kernel checks this value when opening a new file or duplicating a file descriptor (see Chapter 12).

RLIMIT_NPROC

- The maximum number of processes that the user can own.

RLIMIT_RSS

- The maximum number of page frames owned by the process (currently, not enforced).

RLIMIT_SIGPENDING

- The maximum number of pending signals for the process.

RLIMIT_STACK

- The maximum stack size, in bytes.
- The kernel checks this value before expanding the User Mode stack of the process.

How the Resource Limits of a User Process Are Set?

- Whenever a user logs into the system, the kernel creates a process owned by the superuser, which can invoke `setrlimit( )` to decrease the `rlim_max` and `rlim_cur` fields for a resource.
- Each new process created by the user inherits the content of the `rlim` array from its parent, and therefore the user cannot override the limits enforced by the administrator.

- So before resuming the execution of a process, the kernel must ensure that each such register is loaded with the value it had when the process was suspended.
- The set of data that must be loaded into the registers before the process resumes its execution on the CPU is called the hardware context.

Process Switch

- **Process switching occurs only in Kernel Mode.**

1. kernel mode stack
2. translating table
3. process descriptor

  - local variable `prev` refers to the process descriptor of the process being switched out.
  - `next` refers to the one being switched in to replace it.

- The 80 x 86 architecture includes a specific segment type called the Task State Segment (TSS), to store hardware contexts.
- But Linux doesn't use TSS for hardware context switches.
- However Linux is nonetheless forced to set up a TSS for each distinct CPU in the system.

- When an 80 x 86 CPU switches from User Mode to Kernel Mode, it fetches the address of the Kernel Mode stack from the TSS.
- When a User Mode process attempts to access an I/O port by means of an `in` or `out` instruction, the CPU may need to access an I/O Permission Bitmap stored in the TSS to verify whether the process is allowed to address the port.

- In Linux, a part of the hardware context of a process is stored in the process descriptor, while the remaining part is saved in the Kernel Mode stack.

- The `tss_struct` structure describes the format of the TSS.
- The `init_tss` per- CPU variable stores one TSS for each CPU on the system.
- At each process switch, the kernel updates some fields of the TSS so that the corresponding CPU's control unit may safely retrieve the information it needs.
- But there is no need to maintain TSSs for processes when they're not running.

struct x86_hw_tss

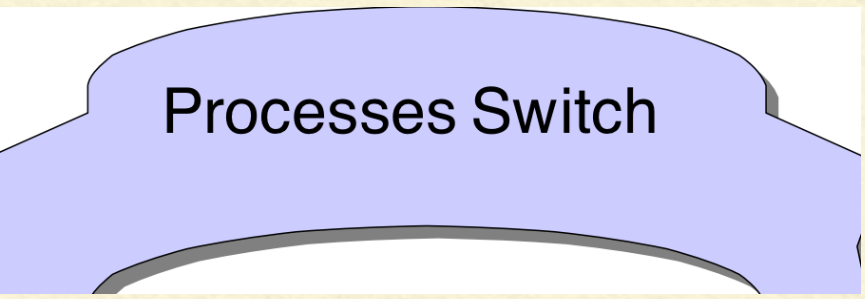
```
struct x86_hw_tss {
unsigned short          back_link, __blh;
unsigned long           sp0;
unsigned short          ss0, __ss0h;
unsigned long           sp1;
/* ss1 caches MSR_IA32_SYSENTER_CS: */
unsigned short          ss1, __ss1h;
unsigned long           sp2;
unsigned short          ss2, __ss2h;
unsigned long           __cr3;
unsigned long           ip;
unsigned long           flags;
unsigned long           ax;
unsigned long           cx;
unsigned long           dx;
unsigned long           bx;
unsigned long           sp;
unsigned long           bp;
unsigned long           si;
unsigned long           di;
unsigned short          es, __esh;
unsigned short          cs, __csh;
unsigned short          ss, __ssh;
unsigned short          ds, __dsh;
unsigned short          fs, __fsh;
unsigned short          gs, __gsh;
unsigned short          ldt, __ldth;
unsigned short          trace;
unsigned short          io_bitmap_base;
} __attribute__((packed));
```

Busy bit

- 1: if the process is being executed by a CPU
- 0: otherwise.

- In Linux design, there is just one TSS for each CPU, so the Busy bit is always set to 1.

3-9-9



`switch_to(prev,next,last)` macro:

- First of all, the macro has three parameters called `prev`, `next`, and `last`.
- The actual invocation of the macro in `context_switch()` is: `switch_to(prev, next, prev)`.
- In any process switch, three processes are involved, not just two, because a process P_{prev} uses the front part of `switch_to` called by P_{prev} to transfer CPU to another process P_{next}, and then uses the rear part of `switch_to` called by P_{last} to obtain CPU again from another process P_{last}.

37 points

1. How to compile linux kernel

(a)
1.make mrproper
2.make oldconfig or make menuconfig
3.make
4.make modules_install
5.make install
(b) suppose linux kernel version is 2.6.32.48 and source code in /usr/src/linux
compiled kernel in : /usr/src/linux/arch/x86/boot/bzImage
should put compiled kernel in : /boot/vmlinuz-2.6.32.48

2. 一個 user code data 時需要2個 virtual address

All Linux processes running in User Mode use the same pair of segments t address instructions and data.

These segments are called ***user code segment*** and ***user data segment***, respectively.

CS DS

Similarly, ***all*** Linux processes running in Kernel Mode use the same pair of segments to address instructions and data:

DPL = 0

they are called ***kernel code segment*** and ***kernel data segment***, respectively.

CS DS

Under the above design, it is possible to store all segment descriptors in the **GDT**.

3. 要加欄位要加在哪邊？ why?

ans: 有些欄位的 code 是用 offset 表示

hard-code

4. When an 80 x 86 CPU switches from User