Linux and Real-Time

Linux 4.7 釋出

tracing 方面的改進:

1. 終於可以 透過 eBPF 控制 tracepoint 了，由於有更多的資料內容讓我們可以更方便的建構更複雜的工具。
2. 上次有提到的 hist trigger。在 /sys/kernel/debug/tracing 多了 histogram 相關的檔案。我們可以對 tracepoint 定義的任意欄位組合拿來當作 key，並統計所需的數值的 histogram。請參考
3. perf trace 可同時顯示被觸發的 syscall 的 call stack。

專案開發與「品味」議題

- Linus Torvalds 日前在 TED 的演說，值得一看，裡頭 Linus 提到對程式碼的「品味」，絕對不是「能夠正確運作」而已，就像裡頭展示的這兩段程式碼，前者和後者功能相同，但後者更精簡更優雅。
  - 延伸閱讀: http://debug-life.net/entry/185

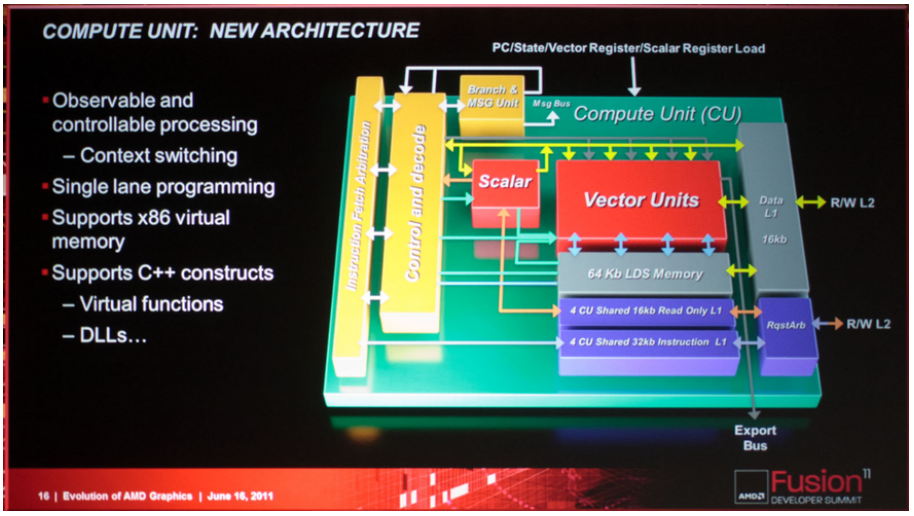
☐ 不優雅

```
remove_list_entry(entry)
{
    prev = NULL;
    walk = head;

    // Walk the list

    while (walk != entry) {
        prev = walk;
        walk = walk->next;
    }

    // Remove the entry by updating the
    // head or the previous entry

    if (!prev)
        head = entry->next;
    else
        prev->next = entry->next;
}
```

☐ 品味: 避免了 head special case

```
remove_list_entry(entry)
{
    // The "indirect" pointer points to the
    // *address* of the thing we'll update

    indirect = &head;

    // Walk the list, looking for the thing that
    // points to the entry we want to remove

    while ((*indirect) != entry)
        indirect = &(*indirect)->next;

    // .. and just remove it
    *indirect = entry->next;
}
```

- 避開盲目「為需求而升級」的發展道路
  - Linux 早期內建 in-kernel httpd，也就是 khttpd，後來更名為 TUX，不過就沒有持續維護，為什麼呢？
  - 十幾年前，Linux 核心的效能仍然不夠好，而且無法發揮 SMP 的優勢，thread 的實現還是很拙劣 (不是現今的 NPTL，而是彆腳的 LinuxThread)，很多方面沒有達到 POSIX 的高效能指標，因此當時的開發者就鑽進瞭解決性能瓶頸的牛角尖裡頭。
  - 過往 Linux 核心開發者們將 web 伺服器單獨加速，也就是在核心內部中實做一個 web 加速器，若按照這條路走下去，Linux 還會把圖形處理搬到核心內部，一如 Windows NT 所為，那就真的是悲劇了。因為 Windows NT 最早是 microkernel (CMU Mach) 的設計，但後來一堆原本位於 userspace 的系統服務搬進核心本身，所以核心就膨脹到超過 20 MB (WTF!)。
    - 延伸閱讀: 淺談 Microkernel 設計和真實世界中的應用
  - 不過 Linux 2.6 以來，核心發展重新找到自己的方向，沒必要透過特化需求去克服局部的效能瓶頸。Linux-2.6 核心的進展是整體性思考的，像是新的排程演算法 (`O(1)` scheduler)，像是 linux-2.6.17 中的 splice 和 tee 系統呼叫，都讓 khttpd 沒有存在的必要
  - [共筆] 實驗目的是在新的 Linux 核心驗證這件事，並重新體會到，當年這些開發者是如何透過大量的效能改進，讓高效能網路伺服器不需要仰賴 khttpd 這類「特化」設計。



面試題目分析

給定一個 singly-linked list 結構:

```
typedef struct _List {
struct _List *next;
int val;
} List;
```

要求：初始化 10,000 個 element，使其內含值為介於 1 到 10000 之間的亂數，彼此不重複

執行結果 (以 1-100 示意)

random list: 77, 12, 51, 84, 50, 8, 42, 23, 49, 100, 29, 35, 96, 22, 7, 41, 11, 30, 63, 31, 73, 60, 1, 75, 56, 71, 45, 39, 80, 43, 18, 55, 40, 5, 37, 98, 20, 82, 27, 59, 90, 76, 10, 53, 33, 68, 85, 32, 65, 83, 97, 87, 46, 66, 3, 58, 17, 74, 89, 70, 69, 94, 38, 6, 21, 81, 19, 47, 86, 44, 25, 61, 28, 64, 2, 48, 92, 4, 95, 52, 54, 57, 13, 79, 16, 99, 88, 24, 34, 14, 36, 93, 78, 26, 15, 62, 91, 67, 72, 9,

效能分析

分析工具: Linux 效能分析工具: Perf

- **方法一：使用每次random就比對是否重複的方式**

  Performance counter stats for './test':

  2,581 cache-misses # 0.001 % of all cache refs [74.93%]
  209,877,933 cache-references [50.18%]
  4,672,016,085 instructions # 0.92 insns per cycle [75.09%]

5,100,727,922 cycles [74.90%]

2.135942019 seconds time elapsed

分析 : 由於每次random都必須在從頭在跑一遍看是否重複,若重複的話就在重新random,那在list越長的情況下重複的機率就越大,就必須不斷的重新random,因此執行時間將會變得無法預期

- **方法二 : 使用card shuffle algorithm**

Performance counter stats for './test':

3,626 cache-misses # 0.015 % of all cache refs [75.27%]

24,664,838 cache-references [50.54%]

354,096,373 instructions # 0.71 insns per cycle [75.26%]

495,513,610 cycles [74.24%]

0.210719471 seconds time elapsed

分析 : card shuffle alogrithm複雜度為O(n*n) , 因此效率比方法一要好很多

延伸討論

(以下由 YouTube 工程師 YC 提供)

延伸提問：

- 假設資料大小為 N：初始化N個元素的 singly linked list 並隨機不重複地填入 -N 的數字。
- 假設忽略 PRNG 與 RNG 的差異，要求 N 個數字 N! 種組合出現的機率相同。

1. 除了linked list本身的佔用的空間外，若允許額外使用O(N)的空間，怎麼做較好呢？

- 初始化一個陣列並填入1-N，需時O(N)。
- 使用陣列的shuffling演算法，可在O(N)時間內完成。（每取一個元素時，亂數一個未處理元素互換）
- 將陣列內容複置回 linked list，需時O(N)。

可在 O(N) 內完成，典型的空間換時間。

2.若允許額外使用 O(log(N)) 的空間，怎麼做較好呢？（call stack 的空間也算）

先初始化linked list並填入1~N的整數。

再使用merge sort的變形，但不依靠比較大小，而以亂數決定順

序。Pseudo Code連結
這個程式結構跟merge sort一模一樣，只是以機率取代比大小而已
worst case time complexity是O(N*log(N))，space complexity是O(log(N))用在call stack上。

機率部份的計算比較奇怪一些，這是為了讓N!種組合出現的機率都相同。
比方說 left 剩 3 個 right 剩 2 個時，應該要有 60% 的機率從 left 拿取元素。

3.若僅能允許額外使用 O(1) 的空間，怎麼做較好呢？

使用方法二 O(N^2) 的方法 (可討論)

4.方法一雖然不如方法二，其平均時間複雜度是多少呢？

這個方法的 worst case time complexity 是沒有理論上限的
就算是完美的 RNG，理論上還是有可能連續連續出現同一個數字幾百萬次。
但是 average case time complexity 呢？
假設使用的 PRNG 夠亂 (當作 RNG 討論)，如果需要的平均時間能用一個算式model
應該就可以分析其時間隨資料量的成長趨勢。

* 先討論一個簡單的機率問題
重複做一件成功機率為p的事（每次機率皆為p，每次為獨立事件），一成功就停手
「做幾次會成功」的期望值是多少呢？

有p的機率會一次成功
有1-p的機率會在第一次失敗，然後再需要E次嘗試才會成功
設此期望值為E，可列等式E = p + (1-p) * (1+E)
化簡後可得E = 1/p，次數的期望值為成功率的倒數
具體的例子：連續丟銅板丟到正面才停，需要的平均次數是2次。連續丟六面骰子丟到1才停，平均需要的次數是6次。

* 接下來，考慮取一個元素的cost：
分析已經取了x個元素，要取第x+1個元素的狀況
從1~N中任取一個元素，有x/N的機率會取到重複的元素
試問：「取第幾次元素會取到不重複元素」的期望值是多少呢？
以p = 1 - x/N代入前式，需嘗試的次數期望值為N / (N - x)
那麼，共要做多少次整數比較呢？顯然，若選到已重複的元素，

需比較次數的期望值為x/2
兩式相乘為N * x / (N - x) / 2

* 最後，考慮取得所有元素的cost總和
累加選擇每個元素，所需的比較次數則為：

$$\sum_{x=0}^{N-1} \frac{Nx}{2(N-x)} < N^2 \sum_{x=0}^{N-1} \frac{1}{N-x} = N^2 \sum_{x=1}^{N} \frac{1}{x}$$

最後一項其實是 harmonic數列 的前N項。以積分的面積思考，其值不會超過 1 + ln(N - 1) （見下式，可參考此圖，但此圖是求下限，考量每個區塊的右邊界則可求上限）

$$\sum_{x=1}^{N} \frac{1}{x} = 1 + \sum_{x=1}^{N-1} \frac{1}{x+1} < 1 + \int_{x=1}^{N-1} \frac{1}{x} dx = 1 + ln(N-1)$$

故整體 cost 為 O(N^2*log(N))。

(13:30-15:00)

作業系統演進

- [ ] The Evolution of Operating Systems

Tag: CTSS, Multics, Unix, BSD, Linux, Android, mbed

搭配閱讀「從 Revolution OS 看作業系統生態變化」

早期: serial processing 軍事用途 (彈道模擬)
二戰期間，賓州大學和阿伯丁彈道研究實驗室共同負責為陸軍每日提供 6 張火力表，每張表都要計算幾百條彈道，一個熟練的計算員計算一條飛行時間 60 秒的彈道要花 20 小時。儘管改進了微分分析儀，聘用 200 多名計算員， 一張火力表仍要算兩三個月。這是電子計算機發展的迫切需求

- computer 早期指「計算員」，後來才變成專指「機器」
- 第一台計算機 ENIAC

分時多工系統/期望的硬體特徵

- Memory protection for monitor
- Timer
- Privileged instructions
- Interrupts

作業系統任務: 分配資源

「電腦科學」之所以不算是「科學」的緣故：先有明確工程需求和實作, 才有整理過的理論發表

process 可定義為

- a program in execution
- an instance of a running program
- the entity that can be assigned to, and executed on, a processor
- a unit of activity characterized by a single sequential thread of
- execution, a current state, and an associated set of system resources

Linux Kernel 提供的基本資料結構

- Linked lists
- Queues
- Maps
- Binary trees (Red-black trees)

Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	-	O(1)	O(1)	O(1)	-	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	-	O(log(n))	O(log(n))	O(log(n))	-	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)

Linked lists

適合使用情況

1. 需要遊走所有的資料
2. 當效能不重要時
3. 儲存的資料項數量不多

linux實作方式

- <linux/list.h>裡頭定義了list_head資料結構

```
struct list_head {
    struct list_head *next;
    struct list_head *previous;
}
```

- 可直接在新增自己的(doubly) linked list時使用list_head結構, 如以下範例:

```
struct list_element {
    char name[16];
    int id;
    struct list_head list;
}
```

優點: 可直接使用linux kernel提供的API(如下所示)，且這個API不管節點資料結構為何都適用
都是利用 list_head 在操作的. 方便許多且比較容易規格化, 不會受到 node 的 data structure 所影響.

- `container_of` macro
  用已知某一結構成員的位址計算出該結構的起始位址(可接著用指標對其他結構成員進行操作)

```
#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \
    (type *)( (char *)__mptr - offsetof(type,member) );})
```

- `list_entry` macro
  - linked list版本的 `container_of` macro
    #define list_entry(ptr, type, member) container_of(ptr, type, member)
  - 此macro被用來建立、操作與管理與linked list有關的API
- 常用操作API
  - 注意環狀串列沒有頭尾節點的概念，傳入的 `head` 可以為任何節點

```
/* LISTHEAD */
#define LIST_HEAD(name) struct list_head name = LIST_HEAD_INI
T(name)
/* INIT_LIST_HEAD */
static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

使用範例

- 假設我們使用struct fox當作節點的資料結構

```
struct fox {
    unsigned long tail_length;
    unsigned long weight;
    bool is_fantastic;
    struct list_head list;
}
```

- 宣告並初始化某個節點

```
struct fox *red_fox;
red_fox = kmalloc(sizeof(*red_fox), GFP_KERNEL);
red_fox->tail_length = 40;
red_fox->weight = 6;
red_fox->is_fantastic = false;
INIT_LIST_HEAD(&red_fox->list);    /* pass a pointer to a list_head st
ructure */
```

- 通常會使用一個 `list_head` 指標來參照使用的linked list (而非使用串列節點)

```
static LIST_HEAD(INIT_LIST_HEAD);
```

- 在串列中加入節點

```
list_add(&red_fox->list, &fox_list);
```

- 遊走linked list

(1) 使用list_for_each函式

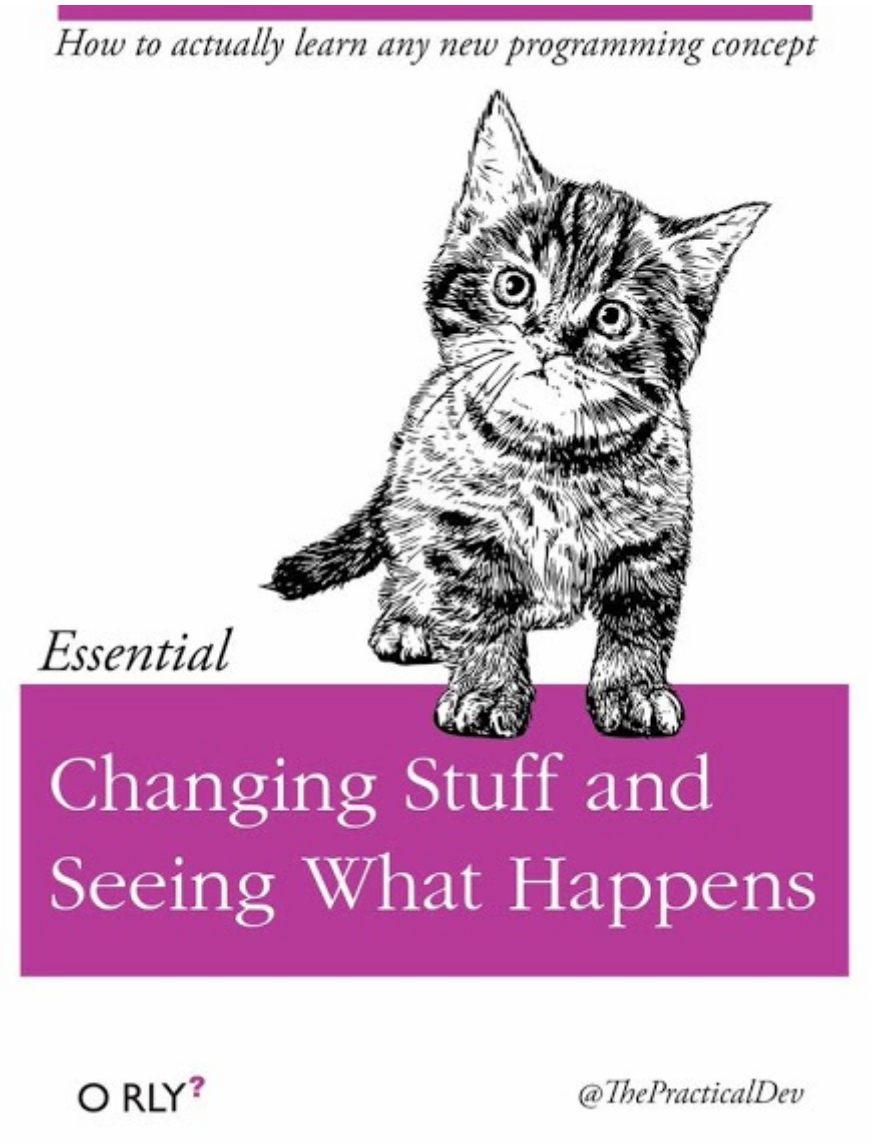
```
struct list_head *p;
struct fox *f;

list_for_each(p, &fox_list)
{
  f = list_entry(p, struct fox, list); // f points to a fox struct, not a list stru
ct
  /* Do Something */
}
```

(2) 使用list_for_each_entry函式

```
struct fox *f;

list_for_each_entry(f, &fox_list, list)
{
    /* Do Something */
}
static struct inotify_watch *inode_find_handle(struct inode *inode, stru
ct inotify_handle *ih)
{
    struct inotify_watch *watch;
```

```
}
```

實驗和統計的重要



[source]

[source]

「Process 和 Thread 有什麼差異？」

大部份的學生很快就可以「背誦」作業系統課程給予的「心法」，回答一長串，可是，其中多少人「看過」Process 和 Thread 呢？多少人知道這兩者怎麼實做出來？又，為何當初電腦科學家和工程師會提出這兩個概念呢？

書本說 thread 建立比 process 快，但你知道快多少？是不是每次都會快？然後這兩者的 context switch 成本為何？又，在 SMP 中，是否會得到一致的行為呢？

之前選修課程的學生透過一系列的實驗，藉由統計來「看到」process 與 thread。物理學家如何「看到」微觀的世界呢？當然不是透過顯微鏡，因為整個尺度已經太小了。統計物理學 (statistical physics) 指的是根據物質微觀夠以及微觀粒子相互作用的認知，藉由統計的方法，對大量粒子組成的物理特性和巨觀規律，做出微觀解釋的理論物理分支。今天我們要「看到」context switch, interrupt latency, jitter, ... 無一不透過統計學！

- Solving Problems at Google Using Computational Thinking (video)

以 Linux 為分析對象

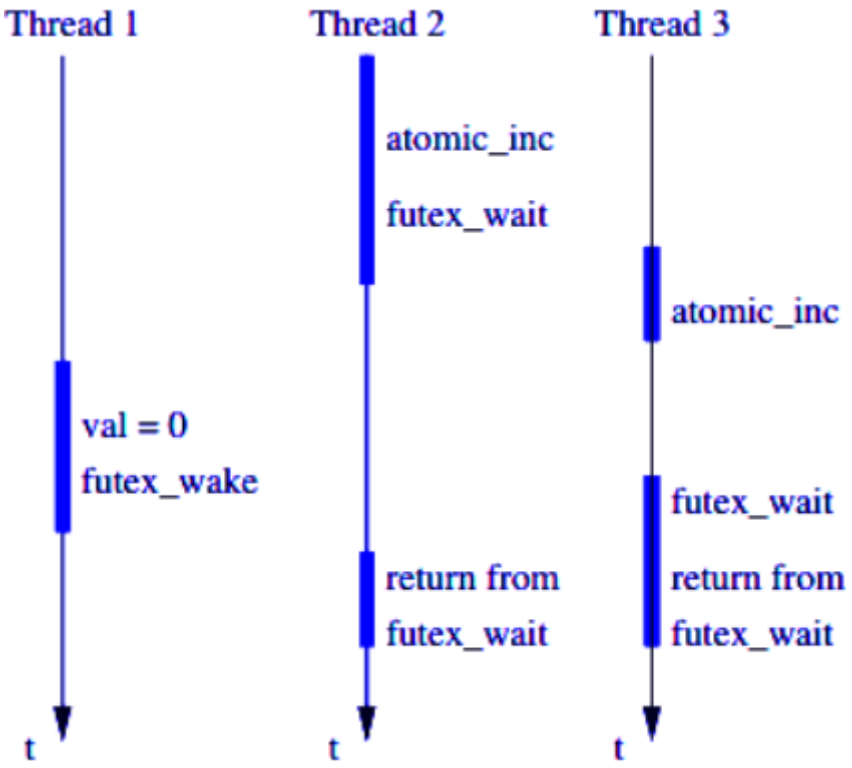
[Thread & Synchronization]

- Shared Code and Reentrancy

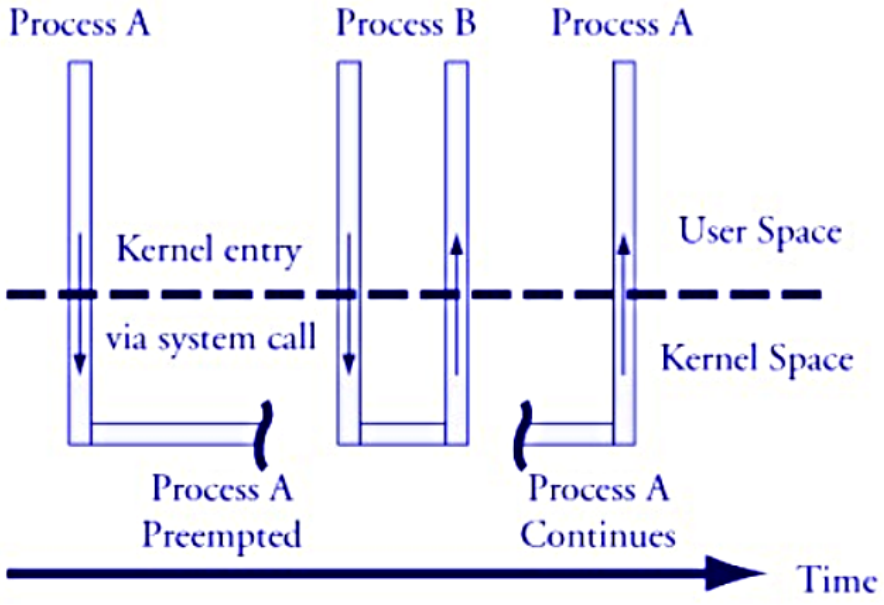
    - reentrancy 會造成問題的案例: strtok()
    - 解決辦法: strtok_r()
    - newlib 實做:
        - strtok
        - strtok_r
            - 沒有全域變數
            - 有個工作區，去保存變數

```
char *
_DEFUN (strtok_r, (s, delim, lasts),
        register char *s _AND
        register const char *delim _AND
        char **lasts)
{
        return __strtok_r (s, delim, lasts, 1);
}
```

    - Use condition variables to atomically block threads until a particular condition is true.
    - Always use condition variables together with a mutex lock
- Mutex in Linux: Various implementations for performance/function tradeoffs
    - Speed or correctness (deadlock detection)
    - lock the same mutex multiple times
    - priority-based and priority inversion
    - forget to unlock or terminate unexpectedly
    - Priority Inversion on Mars
- FUTEX (fast mutex)
    - Lightweight and scalable
    - In the noncontended case can be acquired/released from userspace without having to enter the kernel.



  - invoke `sys_futex` only when there is a need to use futex queue
  - need atomic operations in user space
  - race condition: atomic update of unlock and system call are not atomic
- Kernel preemption



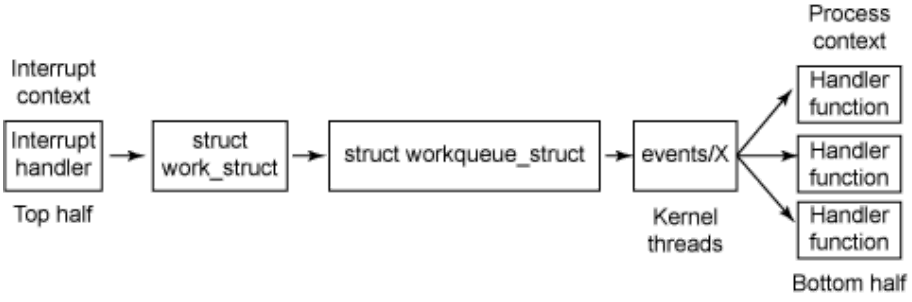
  - a process running in kernel mode can be replaced by another process while in the middle of a kernel function
  - process B may be waked up by a timer and with higher priority
  - 考量點: 降低 dispatch latency
- Linux kernel thread

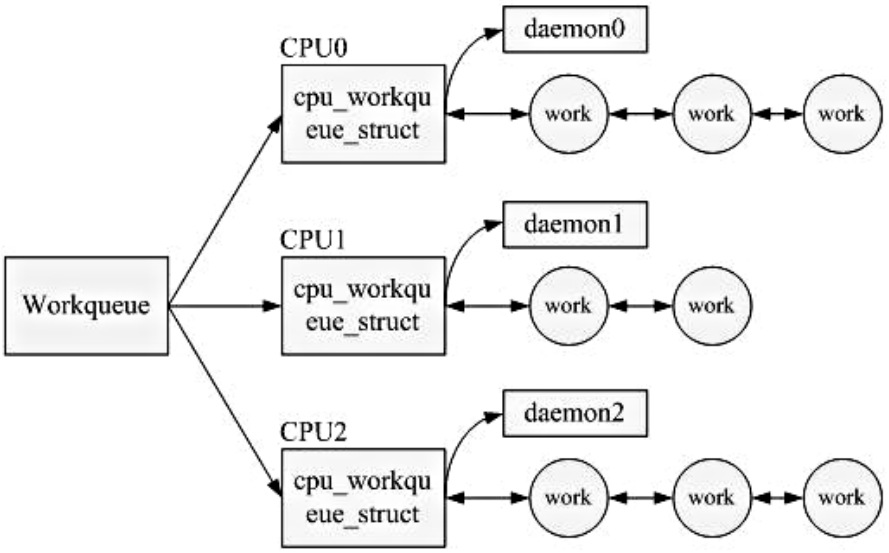
```
static struct task_struct *tsk;
static int thread_function(void *data)
```

```
{
    int time_count = 0;
    do {
        printk(KERN_INFO "thread_function: %d times", ++time_count);
        msleep(1000);
    } while(!kthread_should_stop() && time_count<=30);
    return time_count;
}
static int hello_init(void)
{
    tsk = kthread_run(thread_function, NULL, "mythread%d", 1);
    if (IS_ERR(tsk)) {  …. }
}
```

Work Queue

```
static irqreturn_t event_handler(int irq,
                        void *dev_info)
{
        /* software clear */
        __raw_writel(2, syscfg0_base + SYSCFG_CHIPSIG_CLR_OFF
SET);

        wake_up(&waitq);

        return IRQ_HANDLED;
}
```





    - tasklets execute quickly, for a short period of time, and in atomic mode
    - workqueue functions may have higher latency but need not be atomic
    - Run in the context of a special kernel process (worker thread)
        - more flexibility and workqueue functions can sleep.
        - they are allowed to block (unlike deferred routines)
        - No access to user space
- Atomic Operations
    - provide instructions that are:
    - executable atomically;
    - without interruption
    - Not possible for two atomic operations by a single CPU to occur concurrently
    - ARM: SWP (早期); LDREX/STREX (ARMv6+) [source]
        - support multi-master systems, for example, systems with multiple cores or systems with other bus masters such as a DMA controller.
        - Their primary purpose is to maintain the integrity of shared data structures during inter-master communication by preventing two masters making conflicting accesses at the same time, i.e., synchronization of shared memory.
    - 注意 DMB 的使用
    - 在某些 ARM core (in-order, 如 Cortex-A9) 提供 L2 Lockdown 機制，降低 cache replacement policy 對於特定 isolated cpu 的效能影響。Cortex-A15 這樣 out-of-execution processor 沒有 L2 lockdown
- Spinlock
    - 作用: Ensuring mutual exclusion using a busy-wait lock

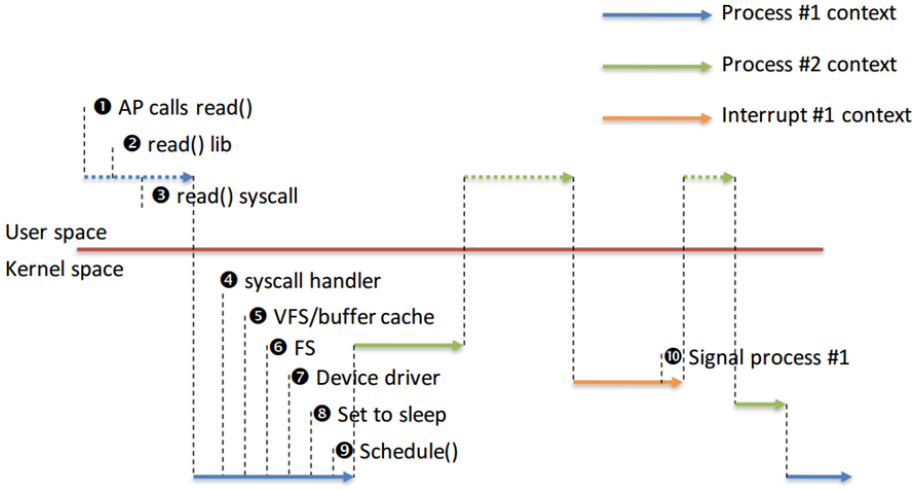
        - really only useful in SMP systems
    - Spinlock with local CPU interrupt disable

```
spin_lock_irqsave(&my_spinlock, flags);
/* critical section */
spin_unlock_irqrestore(&my_spinlock, flags);
```

    - Reader/writer spinlock (存在效能議題，近年許多替代方案被提出)
- Semaphore
    - blocked thread can be in TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE (by timer or signal)
    - 之後探討 real-time Linux 時，這部份會重新分析
- Blocking Mechanism
    - ISR can wake up a block kernel thread which is waiting for the arrival of an event
    - Wait queue
    - wait_for_completion_timeout
        - specify "completion" condition, timeout period, and action at timeout
        - "complete" to wake up thread in wait queue
        - wake-one or wake-many
- Reader/Writer
    - 這部份**很重要**

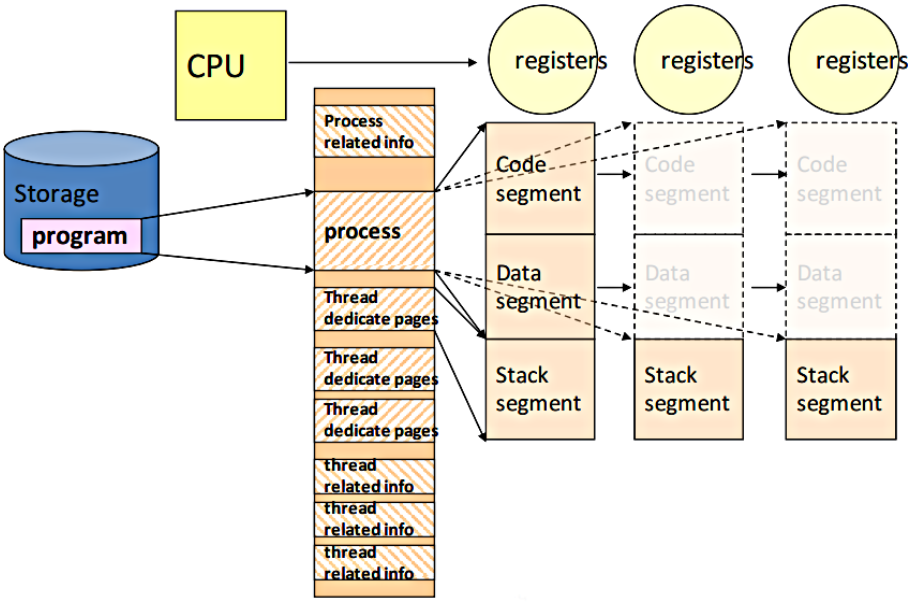
[Process Management]回顧

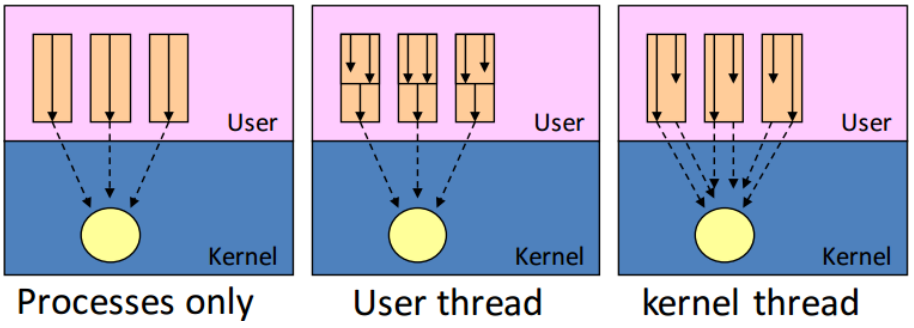
- Scheduling
    - Find the ***next suitable*** process/task to run
- Context switch
    - Store the context of the current process, restore the context of the next process
- Process context + interrupt context



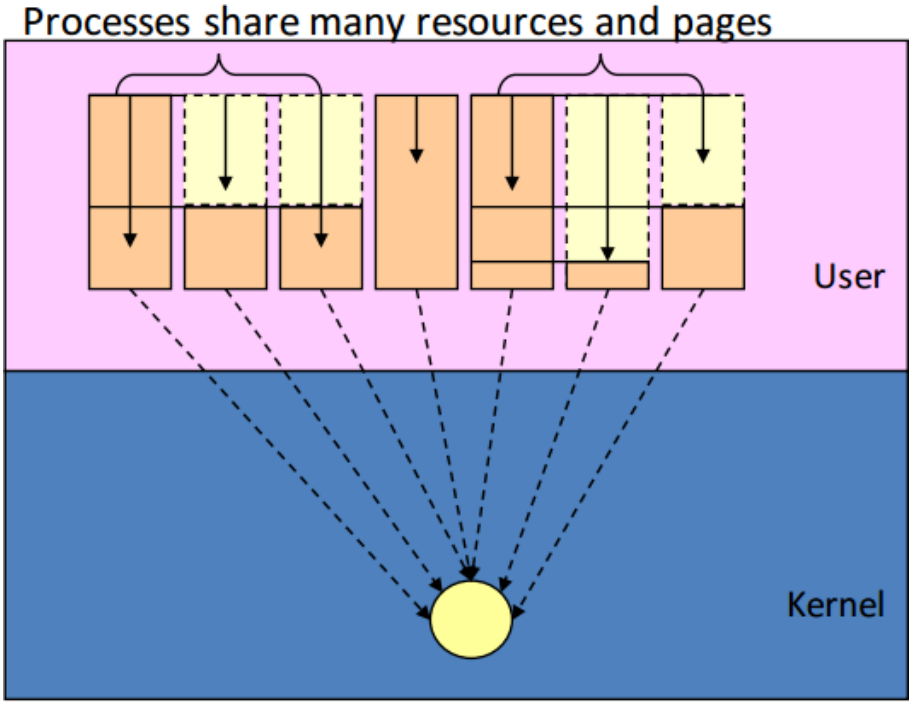
- Linux kernel is possible to preempt a task at any point, so long as the kernel does not hold a lock
- Kernel can be interrupted ≠ kernel is preemptive

    - Non-preemptive kernel, interrupt returns to interrupted process
    - Preemptive kernel, interrupt returns to any schedulable process
- [Page 48 - 51] Process vs. Thread

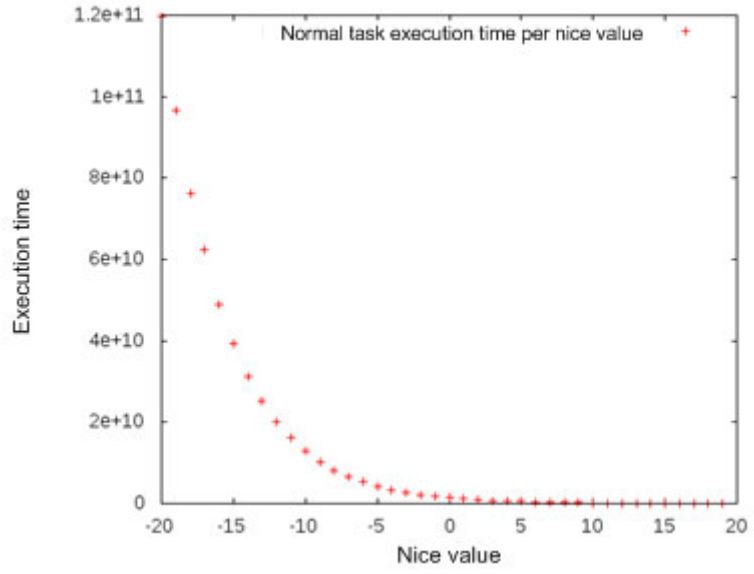




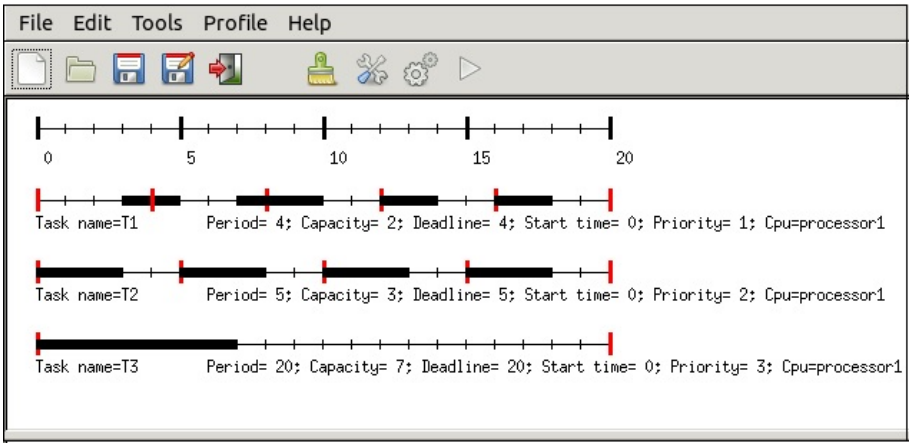
(和作業系統和計算機組織結構的設計有極大的落差)



- Scheduling simulation
    - LinSched: Simulating the Linux Scheduler in user space



- Cheddar project : real-time scheduling analyzer



對照去年選修課程學生的 ARM-Linux 技術報告

Real-Time Linux

- Making Linux do Hard Real-time
- Real time Linux Memory Allocation

- sched_yield vs. CFS

high resolution timer
clock_gettime

- [ ] 作業系統概念和文藝復興

案例探討

- server-framework 效能強化
- epoll experiments
- concurrent B+ tree

int is_aligned(uint8_t byte_count, const uint8_t *p) { ... }

reference: Alignment in C (Page 4)