Symmetric Multiprocess (SMP)

2002 Linux kernel trace seminar

陳嘉聖

jar_shen@seed.net.tw

Outline

1: SMP hardware introduction
2: Atomic Operation
3: Semaphores
4: Spinlocks
5: Schedule

Intel MP Overview

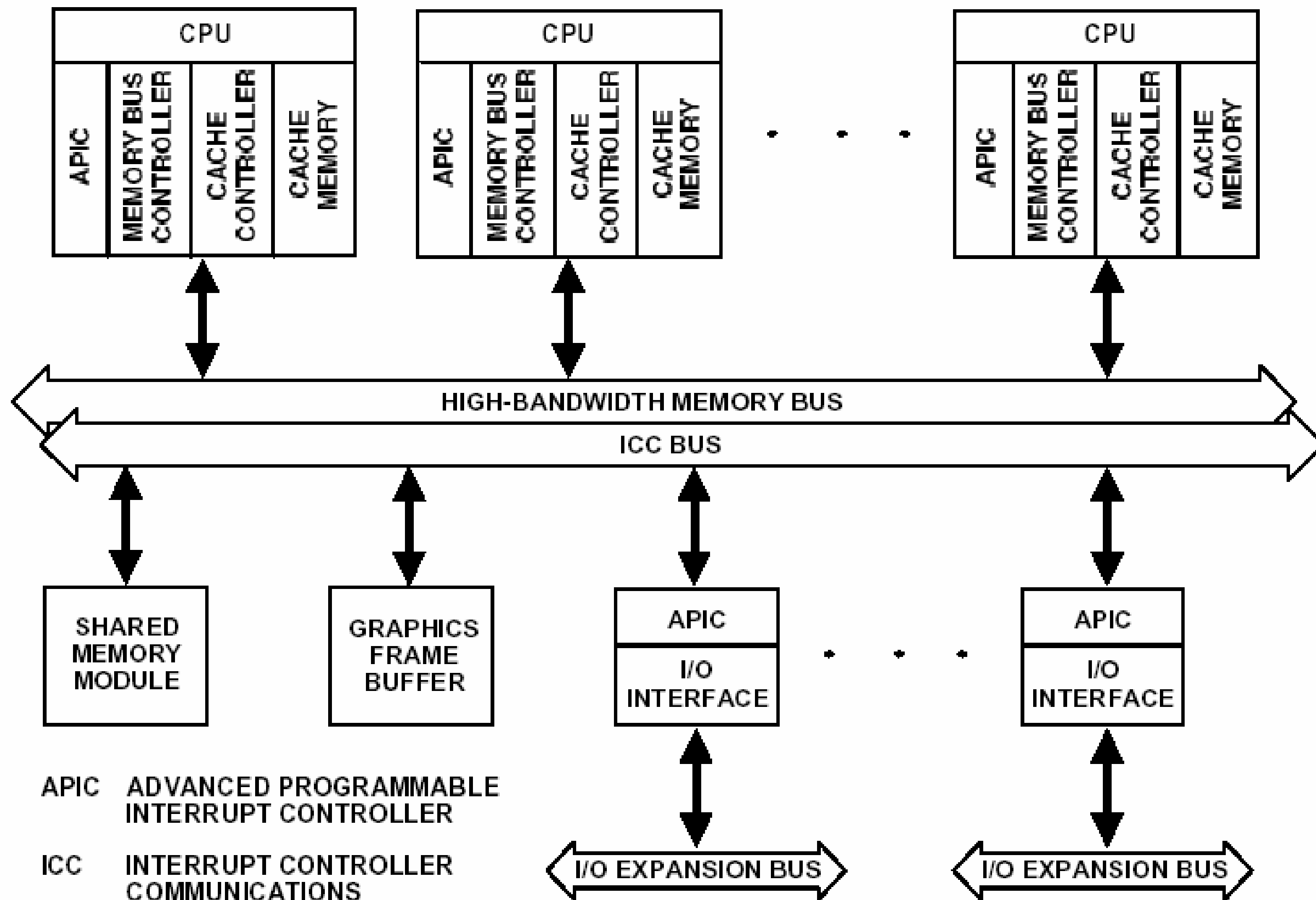
1997 Intel Multiprocessor specification 1.4

It is fully symmetric, all processors are functionally identical and equal Status, and each processor can communicate with every other processor. There is no hierarchy, no master-slave relationship.

1. Memory symmetry: All processor share the same memory space and access that space by the same address. And all processor execute a single copy of the operating system.

2. I/O symmetry: All processor share access the same I/O subsystem (including I/O ports and interrupt controllers) and any processor can receive interrupt from any source.

Multiprocessor System Architecture



Intel SMP Components

MP specification defines a system architecture base on the following components

1. One or more processors that are Intel architecture instruction set compatible.

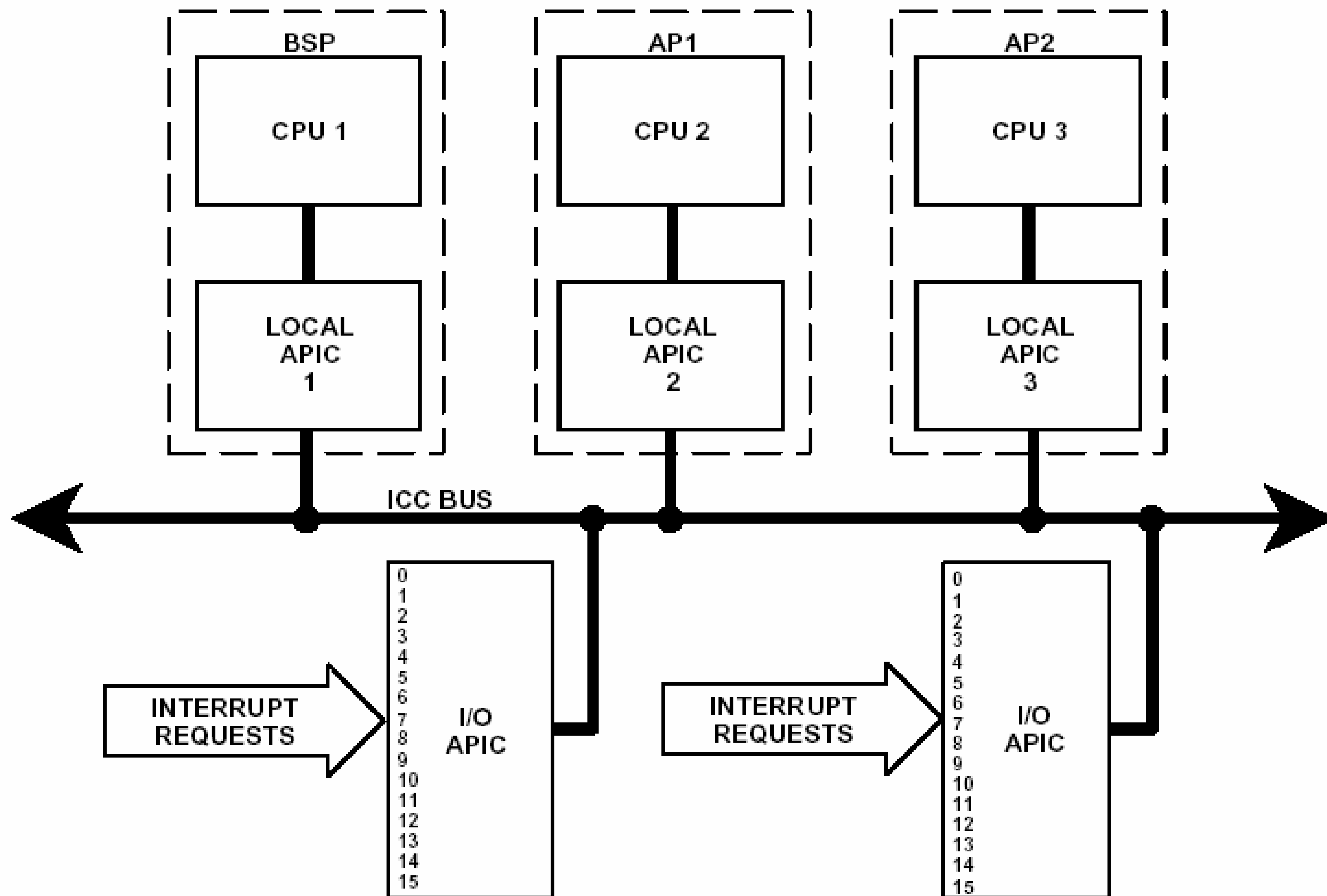
2. One or more APICs, such as Intel 82489DX Advanced Programmable Interrupt Controller or the Integrated APIC on the CPU.

3. Software-transparent cache and share memory subsystem.

4. Software-visible components of the PC/AT platform.

Processors can classifies two type: the bootstrap processor (BSP) and the Application processor (AP). Which processor is the BSP is determined by the Hardware or by the hardware in conjunction with the BIOS. These two different Only during the initialization and shutdown , BSP is responsible for initializing The system for booting the operating system, APs are activated only after the Operating system is up and running.

SMP APIC configuration



APIC and ICC bus

1. Off-loading interrupt-related traffic from the memory bus, making the memory Bus available for processor use.
2. Helping processor share the interrupt processing load with other processor.

The local APIC unit also provide inter-processor interrupt (IPI), which allow Any processor to interrupt any other processor or set of processor. (INIT IPI And STARTUP IPI are specifically designed for system startup and shutdown).

Each local APIC has a Local Unit ID Register and each I/O APIC has an I/O Unit ID Register. The ID serves as a physical name for each APIC unit. It is Used by software to specify destination information for I/O interrupt and Inter-processor interrupts, and is also used internally for accessing the ICC bus.

Local APIC is integrated with the CPU chip, and the I/O APIC may be integrate With the I/O chipset.

APIC Version

Each APIC has a version register that contains the version number of a specific] APIC implementation.

Table 3-2. APIC Versions

APIC Type	Local APIC Version Register (hexadecimal)	Integrated APIC Features
82489DX APIC	0x	
Integrated APIC, i.e., Pentium processors (735\90, 815\100)	1x	STARTUP IPI. See Appendix B.4.2 for details. Programmable interrupt input polarity

NOTE:

x is a 4-bit hexadecimal number.

To encourage future extendibility and innovation, the Intel APIC architecture definition is limited to the programming interface of the APIC units. The ICC bus protocol and electrical specifications are considered implementation-specific. That is, while different versions of APIC implementations may execute the same binary software, different versions of APIC components may be implemented with different bus protocols or electrical specifications. Care must be taken when using different versions of the APIC in a system.

Interrupt mode

Intel MP specification define three different interrupt modes

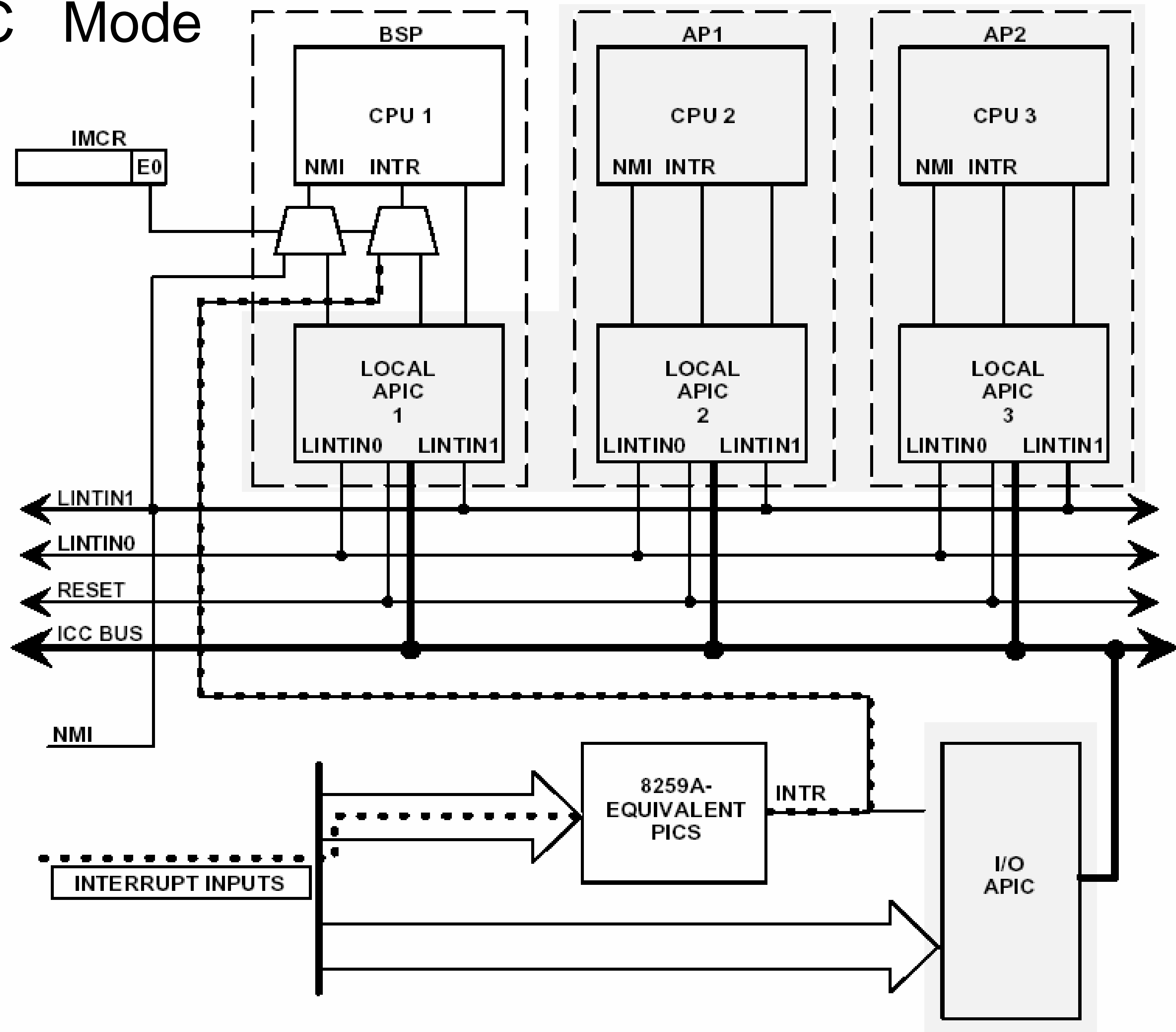
1. PIC Mode – effectively bypass all APIC components and forces the system to operate in single-processor mode.

2. Virtual wire Mode – uses an APIC as a virtual wire, but otherwise operates the same as PIC Mode.

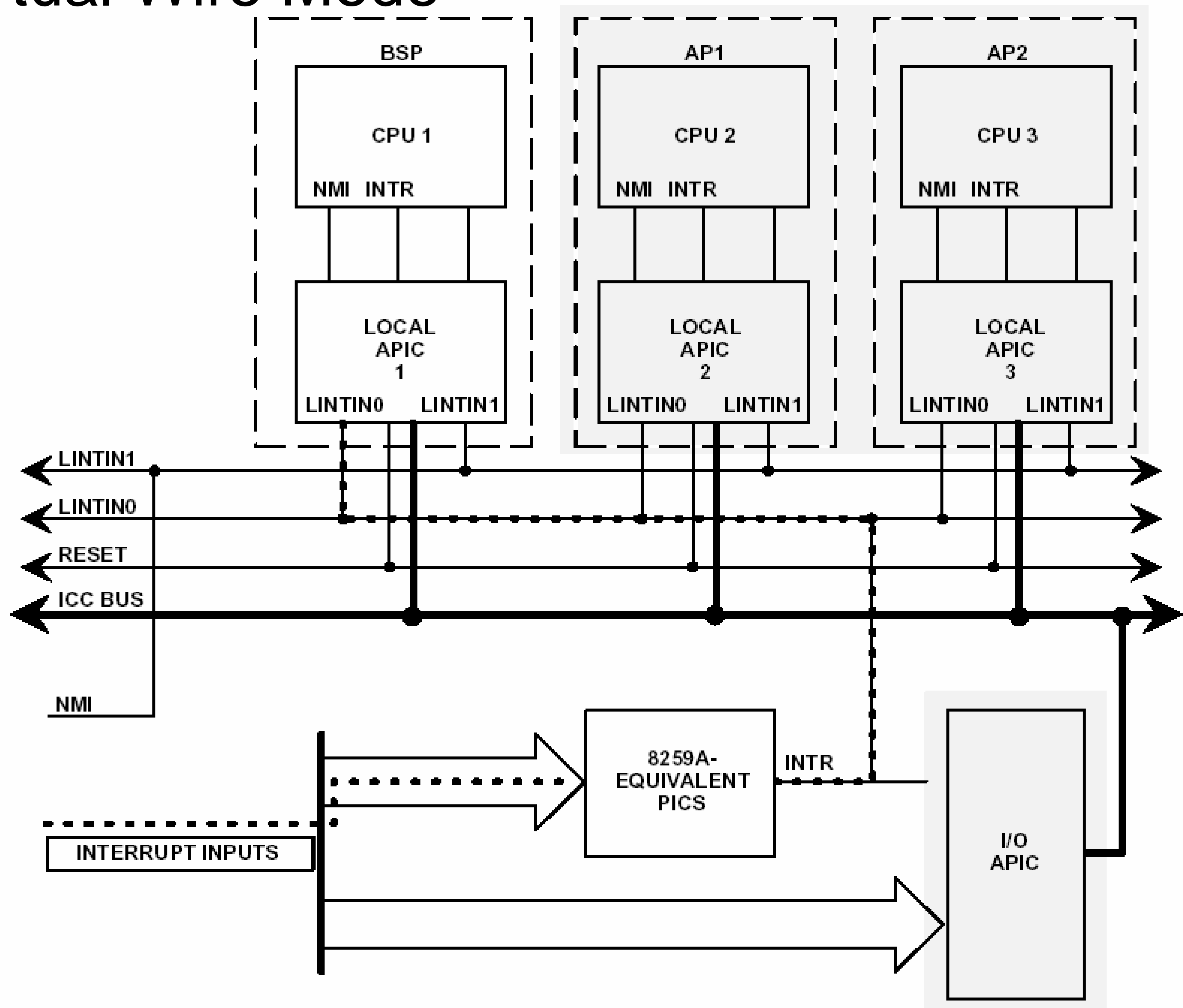
3. Symmetric I/O Mode – enables the system to operate with more than one processor.

An MP operating system is booted under PIC mode or Virtual wire mode, Later The operating system switches to Symmetric I/O Mode as it enters multi-Processor mode.

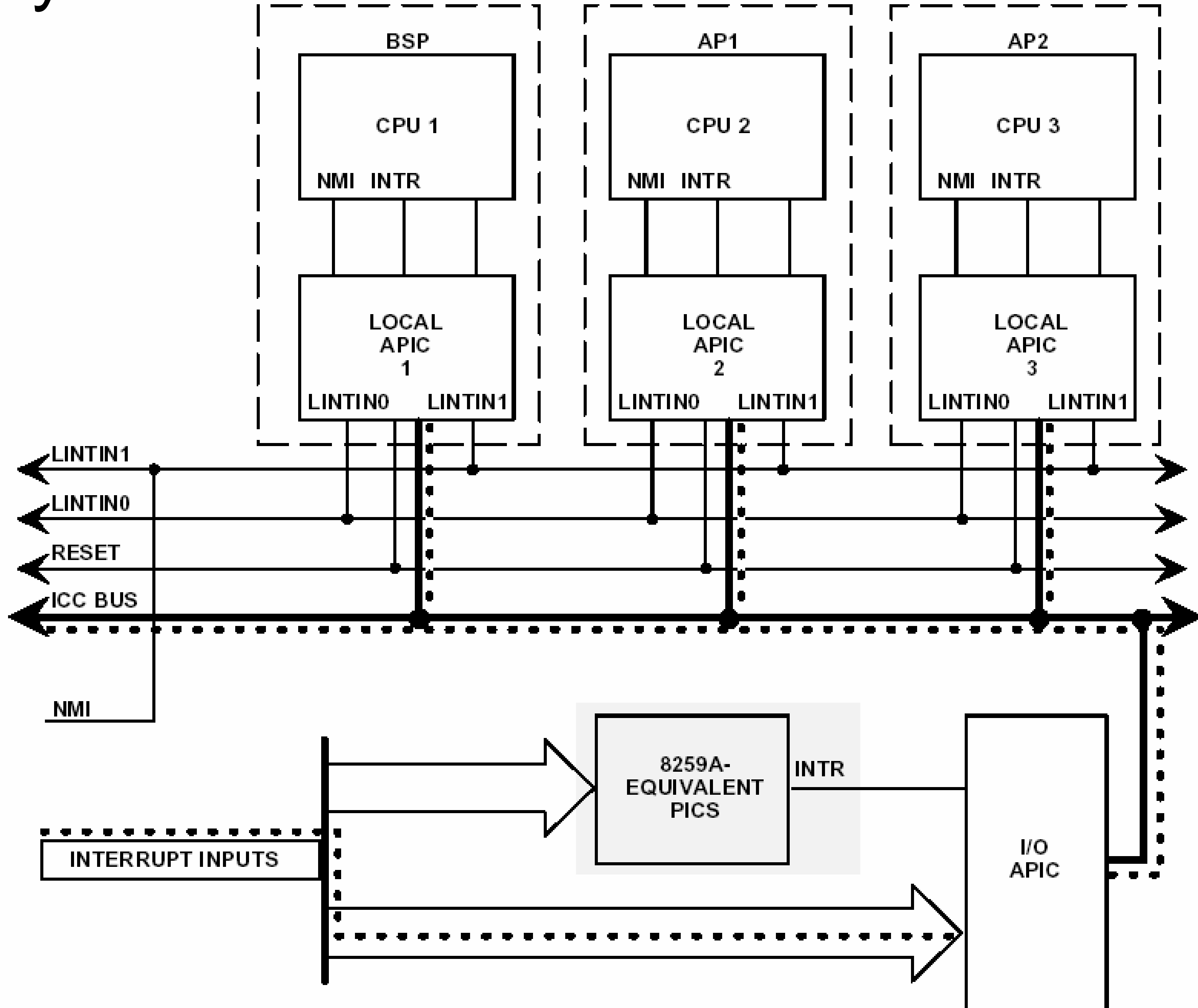
PIC Mode



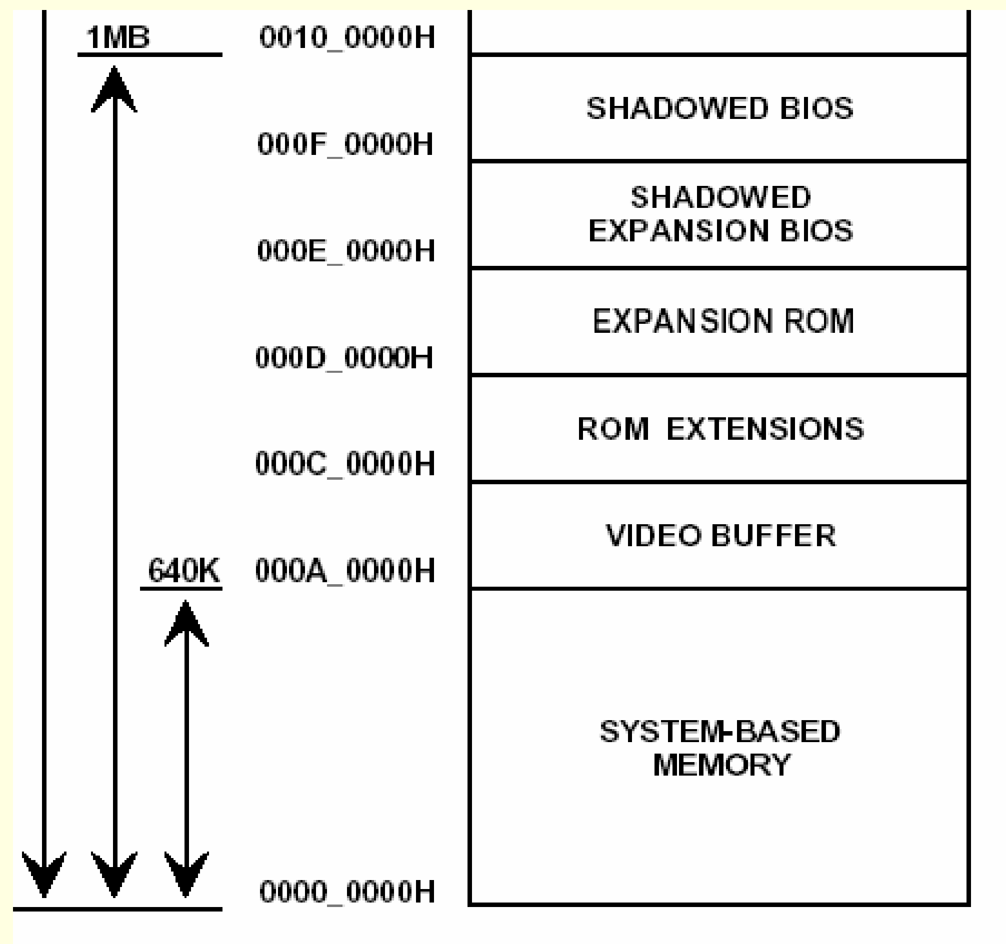
Virtual Wire Mode



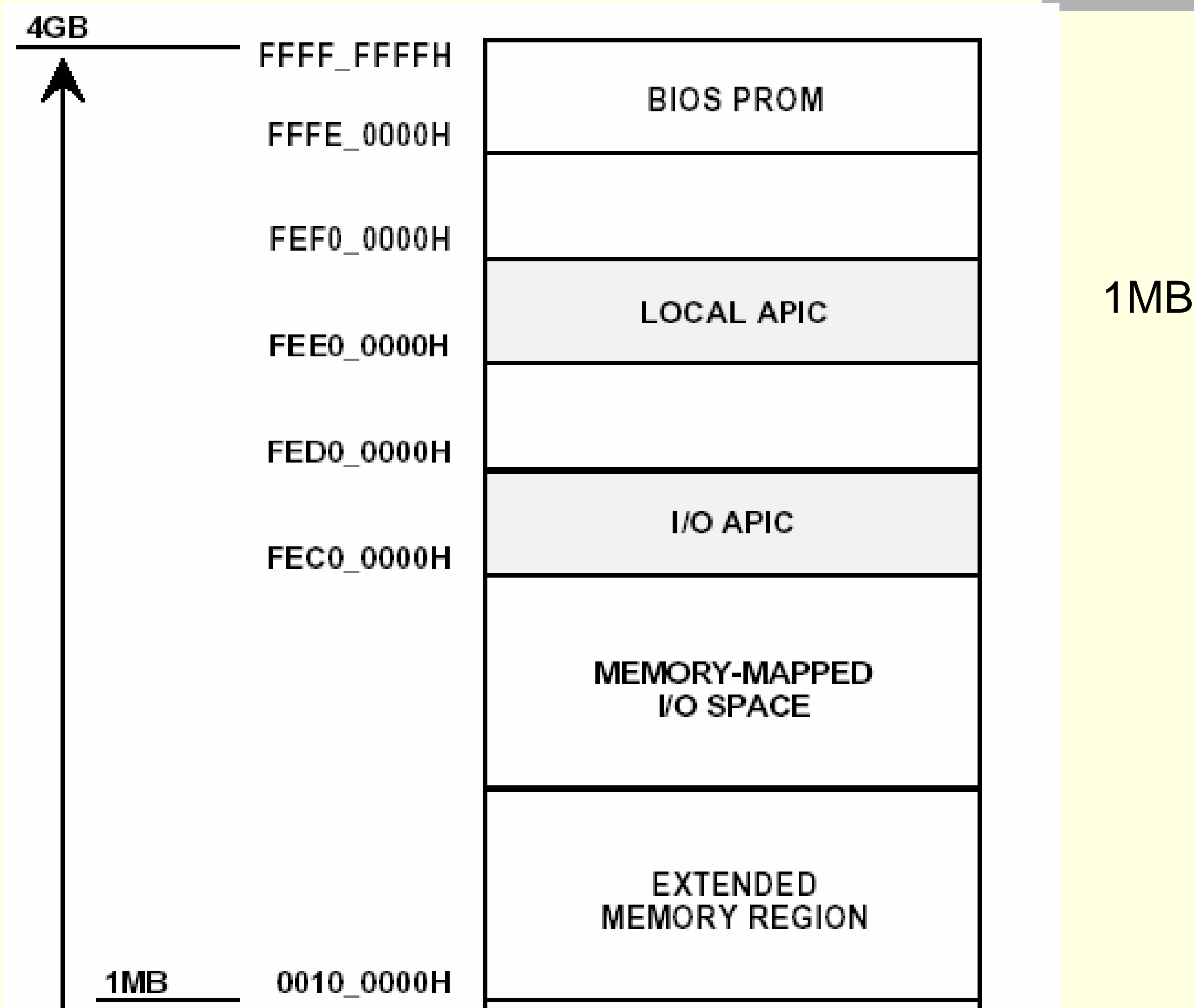
Symmetric I/O mode



System memory configuration



System memory configuration -- continue



1MB

Memory Cacheability Map

Table 3-1. Memory Cacheability Map

Addresses (in hex)	Size	Description	Shared by All Processors?	Cacheable?	Comment
0000_0000h – 0009_FFFFh	640KB	Main memory	Yes	Yes	
000A_0000h – 000B_FFFFh	128KB	Display buffer for video adapters	Yes	No	
000C_0000h – 000D_FFFFh	128KB	ROM BIOS for add-on cards	Yes	Yes	
000E_0000h – 000F_FFFFh	128KB	System ROM BIOS	Yes	Yes	
0010_0000h – 0FEBF_FFFFh		Main memory	Yes	Yes	Maximum address depends on total memory installed in the system.
Not specified.		Memory-mapped I/O devices	Yes[2]	Not specified	Top unused memory

Memory Cacheability Map -- continue

Addresses (in hex)	Size	Description	Shared by All Processors?	Cacheable?	Comment
0FEC0_0000h – 0FECF_FFFFh[1]		APIC I/O unit	Yes	No	Refer to the register description in the APIC data book.
0FED0_0000h – 0FEDF_FFFFh		Reserved for memory-mapped I/O devices	Yes[2]	Not specified	
0FEE0_0000h- 0FEEF_FFFFh[1]		APIC Local Unit	No	No	Refer to the register description in the APIC data book.
0FEF0_0000h – 0FFFD_FFFFh		Reserved for memory-mapped I/O devices	Yes[2]	Not specified	
0FFFE_0000h – 0FFFF_FFFFh	128KB	Initialization ROM	Yes	Not specified	

NOTES:

1. These addresses are part of this specification. The other address regions in this table are shown for reference only, and should not be construed as the sole definition of a PC/AT-compatible address space format or cache.
2. Any memory-mapped device should be shareable unless the nature of the device is that there is one device per processor.

Memory LOCK

To protect the integrity of certain critical memory operation, Intel processor Provide an output signal called LOCK. For any given memory access, LOCK is Asserted once, but may remain asserted for as many memory bus cycle as Required to complete the memory operation. It is the responsibility of the system hardware designers to use this signal to control memory accesses among Processor.

It must lock at least the area of memory defined by the destination operand. But A specific implementation may lock a broader area– it may even lock the entire bus. Therefore, software must consider this behavior.

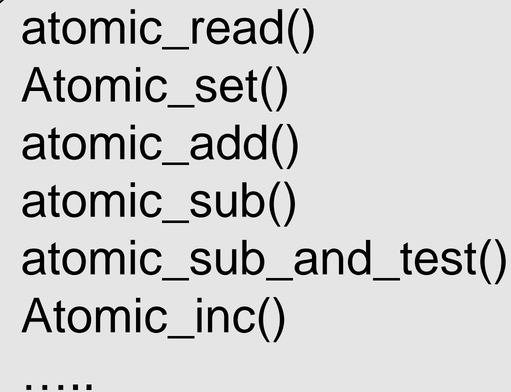
Atomic Operation

1:SMP need kernel support, when accessing their shared resource. SMP specific portion of the code are compiled out for UP(UniProcessor)

2: Atomic Operation : must be perfromed without interruption. Such operations must be indivisible. atomicity can't be guaranteed by softwore— not without special hekp from the Hardware. In X86 platform , "lock" instruction provides exactly this help."lock" instruction locks the memory bus

Atomic Operation

```
Include/asm-i386/atomic.h
#ifdef CONFIG_SMP
#define LOCK "lock ;
#else
#define LOCK  "“
#endif
```




```
static __inline__ void atomic_dec(atomic_t *v)
{
        __asm__ __volatile__(
                LOCK "decl %0"
                :"=m" (v->counter)
                :"m" (v->counter));
}
```

Atomic Operation 2

Include /asm-i386/system.h

```
#define xchg(ptr,v)
((__typeof__(*(ptr)))__xchg((unsigned long)(v),(ptr),sizeof(*(ptr))))
```

test_and_set 也定成 Macro

```
#define tas(ptr) (xchg((ptr),1))
```

Semaphore 1

Kernel semaphore is different to user semaphore

Kernel semaphore 的使用

```
struct semapho
sema_init(&sem, 1);
down(&sem);
…
…
up(&sem);
```

```
struct semaphore {
        atomic_t count;
        int sleepers;
        wait_queue_head_t wait;
};
```

count: Tracks the number of keys still available, if 0, the key is taken, if negative , the key is taken and additional application are waiting for it.

Sleepers: used during the “down”operation . initially 0, when not 0, it is ether 1 or 2, no matter how many processes are waiting to acquire the semaphore.

Wait : the queue of processes that had to be suspended, waiting for this semaphore.

Semaphore 2
sema_init( )

```
static inline void sema_init (struct semaphore *sem, int val)
{
        atomic_set(&sem->count, vl);  // 設定 counter 起始值, 1 為 binary
                                      //   semaphore, >1 counted semaphore
        sem->sleepers = 0;
        init_waitqueue_head(&sem->wait);
}
```

/include/linux/wait.h

```
static inline void
init_waitqueue_head(wait_queue_head_t *q)
{
q->lock = WAITQUEUE_RW_LOCK_UNLOCKED;
INIT_LIST_HEAD(&q->task_list);
}
```

```
#define INIT_LIST_HEAD(ptr)
do {
(ptr)->next = (ptr);
(ptr)->prev = (ptr);
} while (0)
```

只做一次

semaphore 3
down( )

include/asm-i386/semaphore.h

```
static inline void down(struct semaphore * sem)
{
    __asm__ __volatile__(
    LOCK “decl %0\n\t”  /* sem->count  值減 1 */
    "js 2f\n“                   /* if count < 0, then jump to 2: (goto sleep) */
    “1:\n“                      /* process got the semaphore , 結束 down()*/
    “.subsection 1\n“   /* 與 GNU assembler 相關 ,將下面的 code 搬到其他 section*/
    “.ifndef _text_lock_” __stringify(KBUILD_BASENAME) “\n“ /* 所以 1:/n 是程式尾端
    "_text_lock_" __stringify(KBUILD_BASENAME) ":\n“
    ".endif\n“
    "2:\tcall __down_failed \n\t“  /* wait for semaphore , when got semaphore then return*/
    "jmp 1b\n“
    ".subsection 0\n“
    :"=m" (sem->count)
    :"c" (sem)
    :"memory");
}
```

Semaphore 4
_down_failed (can not get semphore)

arch/i386/kernel/semaphore.c

```
asm(
".text\n“
".align 4\n“
".globl __down_failed\n“
"__down_failed:\n\t“
        "pushl %eax\n\t“
        "pushl %edx\n\t“
        "pushl %ecx\n\t“
        "call __down\n\t“
        "popl %ecx\n\t“
        "popl %edx\n\t“
        "popl %eax\n\t“
        "ret“
 );
```

```
void __down(struct semaphore * sem)
{
    struct task_struct *tsk = current;
    DECLARE_WAITQUEUE(wait, tsk);
    tsk->state = TASK_UNINTERRUPTIBLE;          // not interrupt by signal
    add_wait_queue_exclusive(&sem->wait, &wait);  // 加入 wait queue 中 only 1awaken
    spin_lock_irq(&semaphore_lock);
    sem->sleepers++;
    for (;;) {                    // process will not exit,until it acquires the semaphore
        int sleepers = sem->sleepers;
        if (!atomic_add_negative(sleepers - 1, &sem->count)) {
                sem->sleepers = 0;
                break;
        }
        sem->sleepers = 1; /* us - see -1 above */
        spin_unlock_irq(&semaphore_lock);
        schedule();
        tsk->state = TASK_UNINTERRUPTIBLE;
        spin_lock_irq(&semaphore_lock);
    }
    spin_unlock_irq(&semaphore_lock);
    remove_wait_queue(&sem->wait, &wait);
    tsk->state = TASK_RUNNING;
    wake_up(&sem->wait);
}
```

add and test if negative

Hand off CPU to other process, this process will be awaken when the semaphore owner release the semaphore.

Remove itself from the wait queue

Awaken next waiting process

```
void __down(struct semaphore * sem)
{
    struct task_struct *tsk = current;
    DECLARE_WAITQUEUE(wait, tsk);
    tsk->state = TASK_UNINTERRUPTIBLE;
    add_wait_queue_exclusive(&sem->wait, &wa
    spin_lock_irq(&semaphore_lock);
    sem->sleepers++;
    for (;;) {
        int sleepers = sem->sleepe
        if (!atomic_add_negative(sleepers - 1, &
                sem->sleepers = 0;
                break;
        }
        sem->sleepers = 1; /* us - see -1 above */
        spin_unlock_irq(&semaphore_lock);
        schedule();
        tsk->state = TASK_UNINTERRUPTIBLE;
        spin_lock_irq(&semaphore_lock);
    }
    spin_unlock_irq(&semaphore_lock);
    remove_wait_queue(&sem->wait, &wait);
    tsk->state = TASK_RUNNING;
    wake_up(&sem->wait);
}
```

add and test if negative

```
static __inline__ int
atomic_add_negative(int i, atomic_t *v)
{
    unsigned char c;
    __asm__ __volatile__(
    LOCK "addl %2,%0; sets %1"
    :"=m" (v->counter), "=qm" (c)
    :"ir" (i), "m" (v->counter) : "memory");
  return c;
}
```

Semaphore 5
up( )

```
static inline void up(struct semaphore * sem)
{
__asm__ __volatile__(
    "# atomic up operation\n\t"
    LOCK "incl %0\n\t"
    "jle 2f\n"
    "1:\n"              // 程式結束點
    ".subsection 1\n"
    ".ifndef _text_lock_" __stringify(KBUILD_BASENAME) "\n"
    "_text_lock_" __stringify(KBUILD_BASENAME) ":\n"
    ".endif\n"
    "2:\tcall __up_wakeup\n\t"
    "jmp 1b\n"
    ".subsection 0\n"
    :"=m" (sem->count)
    :"c" (sem)
    :"memory");
}
```

將 count 值加 1
If count = 1 , nobody wait for this semaphore
If count = 0 , someone was waiting to awake

```
void __up_wakeup(struct semaphore *sem)
{
    wake_up(&sem->wait);
}
```

Semaphore 6

atomic_add_negative()

Why _down() 需要執行 wake_up( )
假設有 4 個 CPU 及 4 個 process A, B, C, D 同時要求一個 semaphore

1: A call down() got semaphore --- count = 0
B call down() -> _down() -> goto sleep --- count = -3 , sleepers = 1
C call down() -> _down() à goto sleep --- count = -2 , sleepers = 1
D call down() -> _down() -> goto sleep --- count = -1, sleepers = 1

2: A call up() release semaphore --- count = 0 執行 __up_wakeup()
B awaken and 執行 atomic_add_negative() -- count = 0, sleepers = 0, 叫醒 C, C 執行 atomic_add_negative() -- count = -1 , sleepers = 1 , C goto sleep

3: 與步驟 2 相同, 但是 C got semaphore 並且馬上呼叫 up() 來 release semaphore, 此時 count 值將會爲 1 , 因此 up( )將不會執行 __up_wakeup() 來叫醒 D process , 爲避免此情況發生, 所以在 _down() 的尾端要 awaken next waiting process.

Spinlocks 1

Why spinlock

1: semaphore is expensive to use, they put process to sleep. So they are a heavyweight solution for a problem of protecting a quick check on a status variable.

2: spinlock implementation is empty when compiling code for a uniprocessor system.

Use spinlock

1: spinlock will never put a process to sleep, if a lock is not available, it will simple retry, over and over. (spin 的含意)

2: processes should hold spinlocks for the minimum time possible, and must never sleep while holding a lock.

Spinlocks 2

How to use spinlock (simple case)

```
#include <linux/spinlock.h>

spinlock_t lock;
spin_lock_init( &lock);
spin_lock( &lock);
…
…
spin_unlock( &lock);
```

```
typedef struct
{
volatile unsigned long lock;
} spinlock_t
```

```
#define spin_lock_init(x)
do { *(x) =SPIN_LOCK_UNLOCKED; } while(0)

// SPIN_LOCK_UNLOCKED 爲 1
```

Spinlocks 3
Spin_lock( )

```
static inline
void spin_lock(spinlock_t *lock)
{
   __asm__ __volatile__(
      spin_lock_string:
"=m" (lock->lock) : : "men
}
```

A: --lock 如果爲 0 則取得 lock, 離開此函數
B: -- lock 結果爲負值, 跳至 2 處執行 loop

Lock 還不是正值則繼續 Loop, 等到 lock 爲正值後跳至 1 處, 重新開始

```
#define spin_lock_string \
    "\n1:\t" \
    "lock ; decb %0\n\t" \    將 lock low byte減1
    "js 2f\n" \
    ".subsection 1\n" \
    ".ifndef _text_lock_"
__stringify(KBUILD_BASENAME) "\n" \
       "_text_lock_"
__stringify(KBUILD_BASENAME) ":\n" \
    ".endif\n" \
    "2:\t" \
    "cmpb $0,%0\n\t" \
    "rep;nop\n\t" \
    "jle 2b\n\t" \
    "jmp 1b\n" \
    ".subsection 0\n"
```

Spinlocks 4
spin_unlock( )

```
static inline
void spin_unlock(spinlock_t *lock)
{
        __asm__ __volatile__(
            spin_unlock_string );
 }
```

將 lock 值設爲 1

```
#define spin_unlock_string \
        "movb $1,%0" \
        :"=m" (lock->lock) : : "memory"
```

Spinlocks 5

1: Spin lock 種類有很多, 但方式都類似, 目標在盡量減小 lock 的範圍

spin_lock()
spin_lock_irqsave()
spin_lock_irq()
spin_lock_bh()

spin_unlock()
spin_unlock_irq()
spin_lock_irqrestore()

2: 在 uniprocessor 時, spin lock 沒做任何事

```
#define spin_lock_init(lock)        do { } while(0)
#define spin_lock(lock)             (void)(lock) /* Not "unused variable". */
#define spin_unlock(lock)           do { } while(0)
```

Schedule( )   1   (2.4.18 Kernel)

```
asmlinkage void schedule(void)
{
        struct schedule_data * sched_data;
        struct task_struct *prev, *next, *p;
        int this_cpu, c;
  ……
need_resched_back:
    prev = current;
    this_cpu = prev->processor;
    release_kernel_lock(prev, this_cpu);
    spin_lock_irq(&runqueue_lock); /* 爲了 schedule ,所以 lock runqueue */
   ………
    /* move an exhausted RR process to be last.. */
    if (unlikely(prev->policy == SCHED_RR))
                if (!prev->counter) {  // counter 爲零, 表示 time slice 結束,
                                       //所以移到 runqueue 的尾端
                        prev->counter = NICE_TO_TICKS(prev->nice);
                        move_last_runqueue(prev);
                }
```

Schedule( ) 2

```
…..    /* 針對 task 呼叫 schedule() 的情 … I/O) 來做處理 */
switch (prev->state) {
        case TASK_INTERRUPTIBLE:
                if (signal_pending(prev)) {
                        prev->state = TASK_RUNNING;
                        break;
                }
        default:
            del_from_runqueue(prev);
        case TASK_RUNNING:;
 }
……
```

static inline int
signal_pending(struct task_struct *p)

```
{
    return (p->sigpending != 0);
}
```

如果 task 不再是 pending, 設回 running 狀態

將此 task 由 runqueue 中移除(例如果是等待 I/O , 則由 runqueue 中移除)

```
static inline void
del_from_runqueue(struct
task_struct * p)
{
    nr_running--;
    p->sleep_time = jiffies;
    list_del(&p->run_list);
    p->run_list.next = NULL;
}
```

Schedule( ) 3

```
……
repeat_schedule:
    /*  Default process to select.. */
    next = idle_task(this_cpu);
    c = -1000;

    /* iterate runqueue 並取得第一個 task */
    list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
            if (can_schedule(p, this_cpu)) {
                int weight = goodness(p, this_cpu, prev->active_mm);
                if (weight > c)
                    c = weight, next = p;
rs? */
```

// iterate over runqueue list

```
#define list_for_each(pos, head)
 for (pos = (head)->next,
      prefetch(pos->next);
      pos != (head);
      pos = pos->next,
      prefetch(pos->next)    )
```

```
#define can_schedule(p,cpu) ((p)->cpus_runnable
& (p)->cpus_allowed & (1 << cpu))
```

// get the struct for this entry

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

Schedule( ) 4

/* from this point on nothing can prevent us from switching to the next task

```
……
  /* from this point on nothing can prevent us from
     switching to the next task, save this fact in sched_data.*/
   sched_data->curr = next;
   task_set_cpu(next, this_cpu);

   spin_unlock_irq(&runqueue_lock);

   /* 如果下一個要執行的 task 與
  目前的相同則不做 context switch */
   if (unlikely( prev == next)) {
           /* We won't go through the normal tail, so do this by hand */
           prev->policy &= ~SCHED_YIELD;
           goto same_process;
   }
   …….
```

設定 next task 是在這個 CPU 上執行

```
static inline void task_set_cpu(struct
task_struct *tsk, unsigned int cpu)
{
        tsk->processor = cpu;
        tsk->cpus_runnable = 1UL << cpu;
}
```

Schedule( ) 5

```
#ifdef CONFIG_SMP
  /* maintain the per-process 'last schedu
  * (this has to be recalculated even if we
  * the same process) Currently this is o
  * and it's approximate, so we do not h
  * it while holding the runqueue spinlo

  sched_data->last_schedule = get_cycle

  /* We drop the scheduler lock early (it's
  * thus we have to lock the previous pro
  * rescheduled during switch_to(). */
 #endif /* CONFIG_SMP */

  kstat.context_swtch++;
```

/* Standard way to access the cycle counter on i586+ CPUs. Currently only used on SMP. */
/* Get the current time, this information helps reschedule_idle decide which CPU has been idle the longest */

```
static inline cycles_t get_cycles (void)
{
#ifndef CONFIG_X86_TSC
    return 0;
#else
    unsigned long long ret;
    rdtscll(ret);
    return ret;
 #endif
}
```

Schedule( ) 6

```
……
/* 準備作 context switch 的一些
prepare_to_switch();
{
 ……..
 }

/* This just switches the registe
switch_to(prev, next, prev);
__schedule_tail(prev);

same_process:
reacquire_kernel_lock(curren
if (current->need_resched)
    goto need_resched_back;
 return;

} /* end of schedule */
```

Do context switch

```
#define switch_to(prev,next,last)
do {
asm volatile("pushl %%esi\n\t"  "pushl
%%edi\n\t"
"pushl %%ebp\n\t“
 "movl %%esp,%0\n\t" /*save ESP register*/
 "movl %3,%%esp\n\t" /* restore ESP */
"movl $1f,%1\n\t" /* save EIP */
"pushl %4\n\t" /* restore EIP */
"jmp __switch_to\n" /* switch cacheline …. */
“1:\t“   /* 應該是返回時才執行 */
"popl %%ebp\n\t"
"popl %%edi\n\t"
"popl %%esi\n\t" :"=m" (prev-
>thread.esp),"=m" (prev->thread.eip),"=b"(last)
:"m" (next->thread.esp),"m" (next->thread.eip),
"a" (prev),  "d" (next),
"b" (prev));
} while (0)
```

__schedule_tail( ) 1

```
/*schedule_tail() is getting called from the fork return path. It do two thing
  1: clears prev's cpus_runnable flag
  2: find another CPU for prev to run on.

static inline void __schedule_tail(struct task_struct *prev)
{
#ifdef CONFIG_SMP
        int policy;
        policy = prev->policy;
        prev->policy = policy & ~SCHED_YIELD;
        wmb();  /* Force strict CPU ordering.  for intel CPU 此function沒作用 */

        task_lock(prev);              /* 即 spin_lock(&prev->alloc_lock) */
        task_release_cpu(prev);   /* 即 prev->cpus_runnable = ~0UL */
        mb();                               /*  * Force strict CPU ordering. */
        /* 如果 prev 是 RUNNING, 爲 prev task 找另一個 CPU 來執行 */
        if (prev->state == TASK_RUNNING)
                goto needs_resched;
```

__schedule_tail( )　　2

```
out_unlock:
        task_unlock(prev);
        return;
needs_resched:
    {
            if ((prev == idle_task(smp_processor_id())) ||(poli           D_YIELD))
                        goto out_unlock;

            spin_lock_irqsave(&runqueue_lock, flags);
            if ((prev->state == TASK_RUNNING) && !task_has_cpu(prev))
                        reschedule_idle(prev);  /* relocate prev task if possible */
            spin_unlock_irqrestore(&runqueue_lock, flags);
            goto out_unlock;
    }
#else
        prev->policy &= ~SCHED_YIELD;  /* Uni Processor 只做此項 */
#endif /* CONFIG_SMP */
}
```

```
#define task_has_cpu(tsk) ((tsk)->cpus_runnable != ~0UL)
```

reschedule_idle( )   1

reschedule_idle tries to schedule task p on a different CPU, preferably An idle one, reschedule_idle( ) find a destination CPU with the follow algorithm.

1: Run task on whatever CPU it ran on last, if that CPU is now idle.

2: Otherwise, if another CPU is idle, run task on it. If more than one CPU is idle, run one that has been idle for the longest time.

3: otherwise, if task has a higher goodness than some other currently running process, allow task to preempt the process.

4: Otherwise, all CPUs are running processes that have a higher goodness than this task, then just gives up.

reschedule_idle( )　　　2

rule 1: Run task on whatever CPU it ran on last, if that CPU is now idle

```
static void reschedule_idle(struct task_struct * p)
{    struct task_struct *tsk, *target_tsk;
#ifdef CONFIG_SMP
    …….
    best_cpu = p->processor;            /* 上次task p 執行時所使用的  CPU */
    if (can_schedule(p, best_cpu)) {
        tsk = idle_task(best_cpu);          /* get best_cpu 的 idle task struct*/
        if (cpu_curr(best_cpu) == tsk) { /* a CPU is idle if it is exec its idle task */
            int need_resched;
send_now_idle:
            need_resched = tsk->need_resched;
            tsk->need_resched = 1;  /* notice idle process to give away the CPU */
            If ((best_cpu != this_cpu) && !need_resched) /*idle CPU is not current CPU*/
                smp_send_reschedule(best_cpu); /* send a RESCHEDULE_VECTOR
                                                  IPI to this idle CPU */
            return;
        }
    }
………………
```

```
/*We align per-CPU scheduling data on cacheline boundaries, to prevent cacheline ping-pong*/
static union {
        struct schedule_data {
                struct task_struct * curr;
                cycles_t last_schedule;
        } schedule_data;
        char __pad [SMP_CACHE_BYTES];
} aligned_data [NR_CPUS] __cacheline_aligned { {{&init_task,0}}};

#define cpu_curr(cpu)       aligned_data[(cpu)].schedule_data.curr
```

```
struct task_struct * init_tasks[NR_CPUS] = {&init_task, };
#define idle_task(cpu)    (init_tasks[cpu_number_map(cpu)])
```

```
tsk = idle_task(best_cpu); /* get best_cpu 的 idle task struct*/
        if (cpu_curr(best_cpu) == tsk) {/* a CPU is idle if it is exec its idle task */
int need_resched;
send_now_idle:
          need_resched = tsk->need_resched;
          tsk->need_resched = 1;  /* notice idle process to give away the CPU */
         If ((best_cpu != this_cpu) && !need_resched) /*idle CPU is not current CPU*/
             smp_send_reschedule(best_cpu); /* send a RESCHEDULE_VECTOR
                                                  IPI to this idle CPU */

          return;
       }
   }
 ................
```

```
void smp_send_reschedule(int cpu)
{
    send_IPI_mask(1 << cpu, RESCHEDULE_VECTOR);
}
```

reschedule_idle( )   3

Rule 2: Otherwise, if another CPU is idle, run task on it. If more than one CPU is idle, run one that has been idle for the longest time.

Rule 3: Otherwise, if task has a higher goodness than some other currently running process, allow task to preempt the process.

```
oldest_idle = (cycles_t) -1; /* oldest_idle the most idle CPU */
target_tsk = NULL; /*target_tsk 是 task p 所想要取代的 process */
max_prio = 0;      /*max_prio 幫助決定 task p 是否要 preempt a non-idle process

for (i = 0; i < smp_num_cpus; i++) {
    cpu = cpu_logical_map(i);
    if (!can_schedule(p, cpu))
      continue; /* 此process 被禁止在此 CPU 上執行 */
    tsk = cpu_curr(cpu); /* 目前正在執行的 task */

    if (tsk == idle_task(cpu)) { /* 如果目前這個task 是 idle task */
       /* 下面的 if 與 for loop 配合, 找到所有 idle CPU 中 idle 最久的 */
       if (last_schedule(cpu) < oldest_idle) {
           oldest_idle = last_schedule(cpu);
           target_tsk = tsk; /* target_tsk 目前為 idle 時間最久的 task */
       }
    } else /* if CPU is not idle */
  .........
} /* end of loop for */
```

Walks over all CPU, checking any of them is idle, or ,if not, whether P should preempt the non-idle process

reschedule_idle( ) 3 continue
Rule 2,3

```
oldest_idle = (cycles_t) -1; /* 此爲 MAX value of cycles_t , 値越小idle 越久*/
max_prio = 0;      /*max_prio 幫助決定 task p 是否要 preempt a non-idle process
for (i = 0; i < smp_num_cpus; i++) {
    if (tsk == idle_task(cpu)) { /* 如果目前這個task 是 idle task */
        ……..
        tsk = cpu_curr(cpu); /* 目前正在執行的 task */
    }
    else {   /* if CPU is not idle */
        if (oldest_idle == -1ULL) {  /* 還沒有任何的 idle CPU 被找到 */
            int prio = preemption_goodness(tsk, p, cpu); /* p與tsk的 goodness的差値
            if (prio > max_prio) { /*與    loop 配合, 找的 goodness 値與p差最多的task
            max_prio = prio;
            target_tsk = tsk;          ss 差值最大的 running task */
        }
    }
}
```

```
static inline int preemption_goodness(struct task_struct * prev, struct task_struct * p, int cpu)
{  return goodness(p, cpu, prev->active_mm) - goodness(prev, cpu, prev->active_mm); }
```

reschedule_idle( ) 4

```
target_tsk = NULL; /*target_tsk 是 task p 所想要取代的 process */

tsk = target_tsk;
if (tsk) {
    if (oldest_idle != -1ULL) {  /* 有 idle CPU 存在 */
        best_cpu = tsk->processor;
        goto send_now_idle;
    }
    tsk->need_resched = 1;  /* 要 preempt running task , raise flag
    if (tsk->processor != this_cpu)
        smp_send_reschedule(tsk->processor);
}
return;
#else /* UP */
int this_cpu = smp_processor_id();
struct task_struct *tsk;
tsk = cpu_curr(this_cpu);
if (preemption_goodness(tsk, p, this_cpu) > 0)
tsk->need_resched = 1;
#endif   /* CONFIG_SMP */
} /* end of reschedule_idle( )  */
```

Send RESCHDULE_VECTOR IPI to this processor

Reference

1. Linux Core Kernel Commentary -- second edition

2. Cross Referencing Linux – http://lxr.linux.no

3. Intel MP Specification V1.4
   - www.intel.com/design/pentium/datashts/24201606.pdf