CS353 Linux Kernel

Chentao Wu吴晨涛
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

上海交通大學

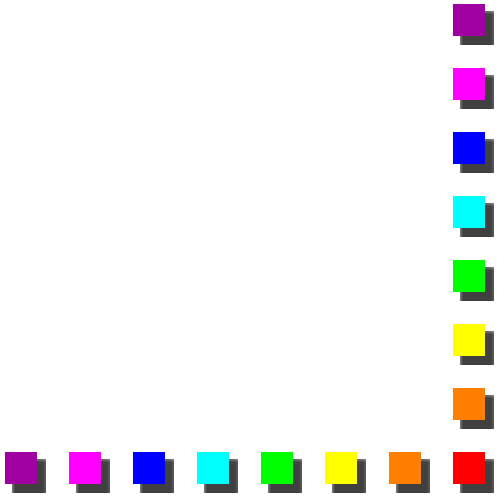
5. Symmetric Multiprocessing (SMP)

Chentao Wu
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

上海交通大學

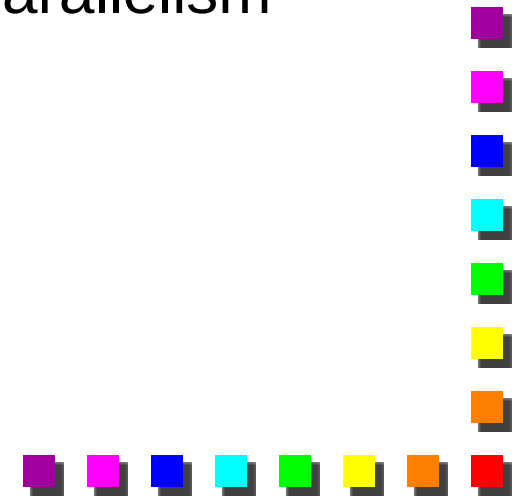
Outline

- Introduction on SMP
- SMP and NUMA
- Process Scheduling with SMP
- Synchronization Problem with SMP



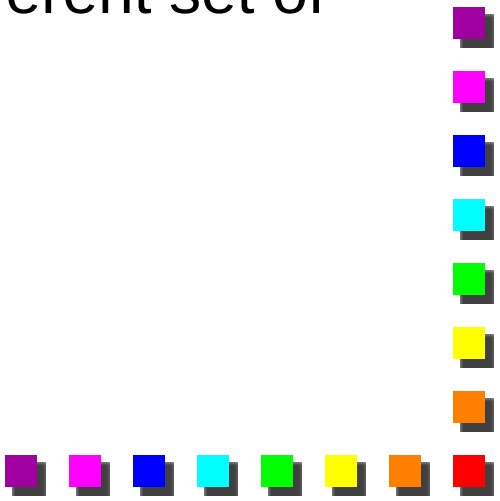
Traditional View

- Traditionally, the computer has been viewed as a sequential machine.
  - A processor executes instructions one at a time in sequence
  - Each instruction is a sequence of operations
- Two popular approaches to providing parallelism
  - Symmetric MultiProcessors (SMPs)
  - Clusters



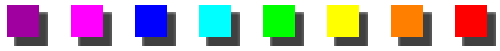
Categories of Computer Systems

- Single Instruction Single Data (SISD) stream
  - Single processor executes a single instruction stream to operate on data stored in a single memory
- Single Instruction Multiple Data (SIMD) stream
  - Each instruction is executed on a different set of data by the different processors

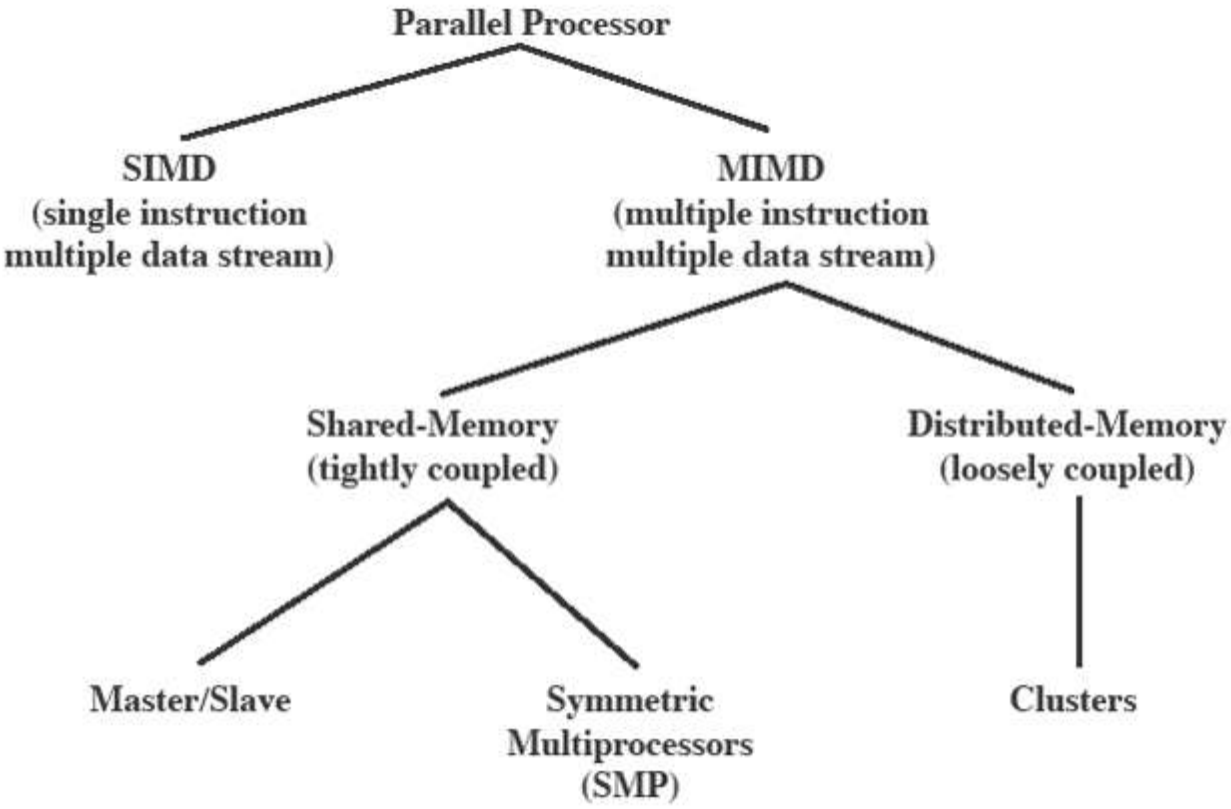


Categories of Computer Systems

- Multiple Instruction Single Data (MISD) stream (Never implemented)
  - A sequence of data is transmitted to a set of processors, each of execute a different instruction sequence
- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets

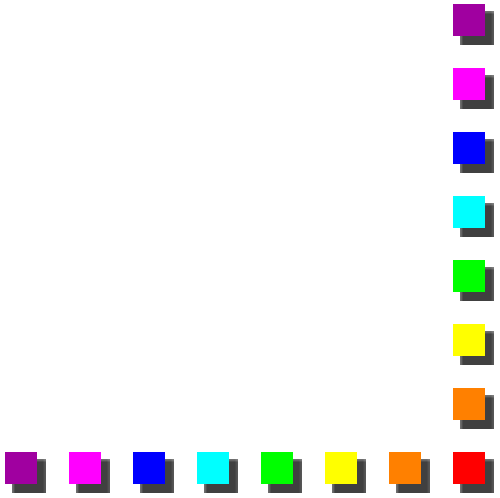


Parallel Processor Architectures

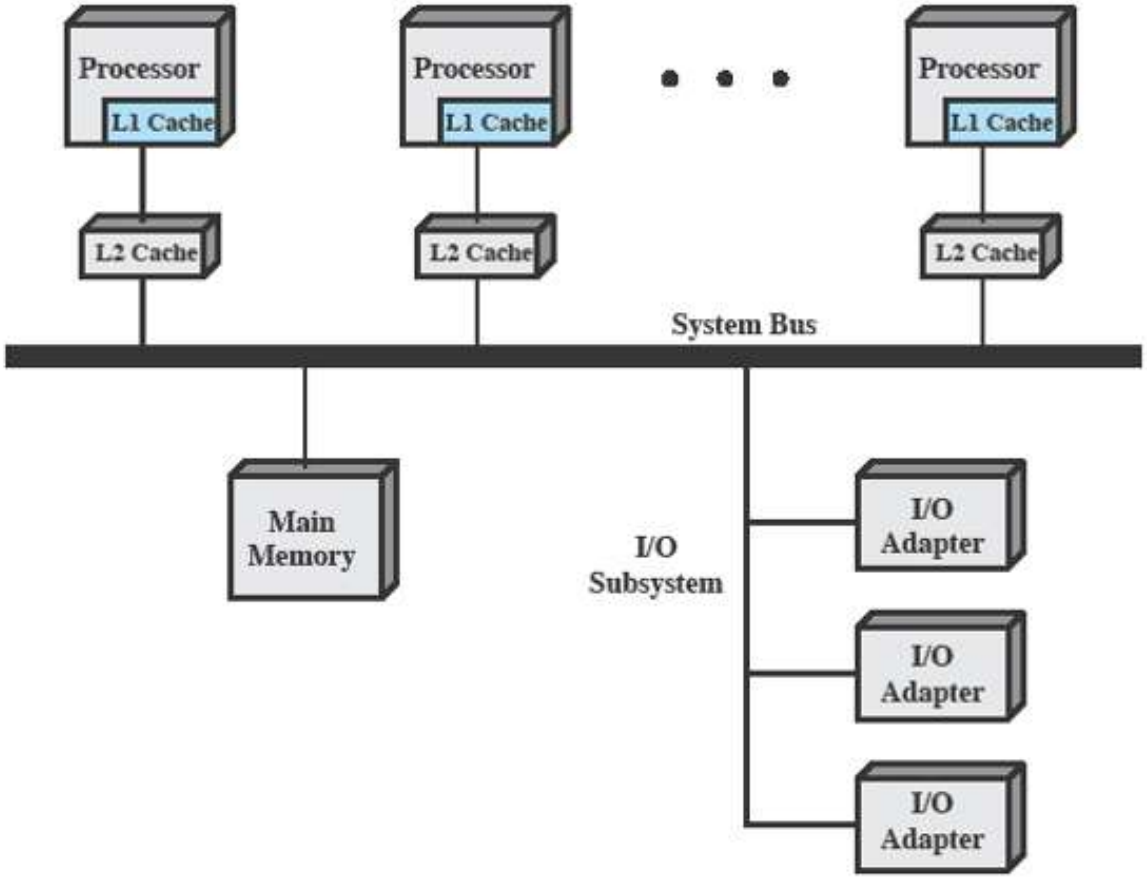


Symmetric Multiprocessing

- Kernel can execute on any processor
  - Allowing portions of the kernel to execute in parallel
- Typically each processor does self-scheduling from the pool of available process or threads



Typical SMP Organization

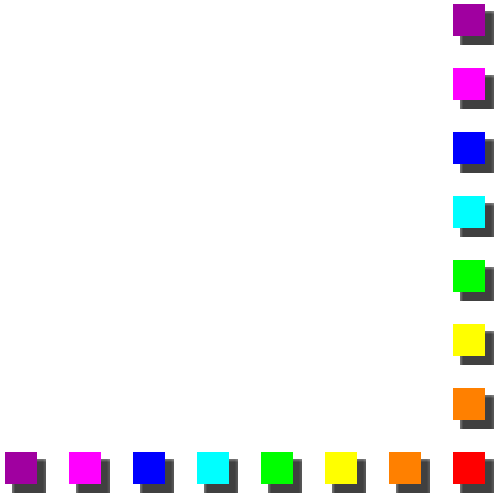


Multiprocessor OS Design Considerations

- The key design issues include
  - Simultaneous concurrent processes or threads
  - Scheduling
  - Synchronization
  - Memory Management
  - Reliability and Fault Tolerance

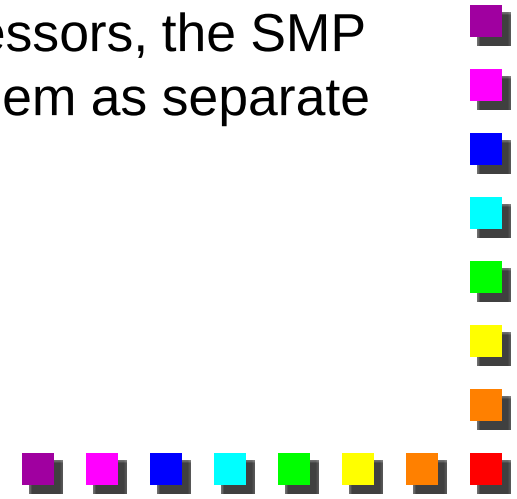
Outline

- Introduction on SMP
- SMP and NUMA
- Process Scheduling with SMP
- Synchronization Problem with SMP

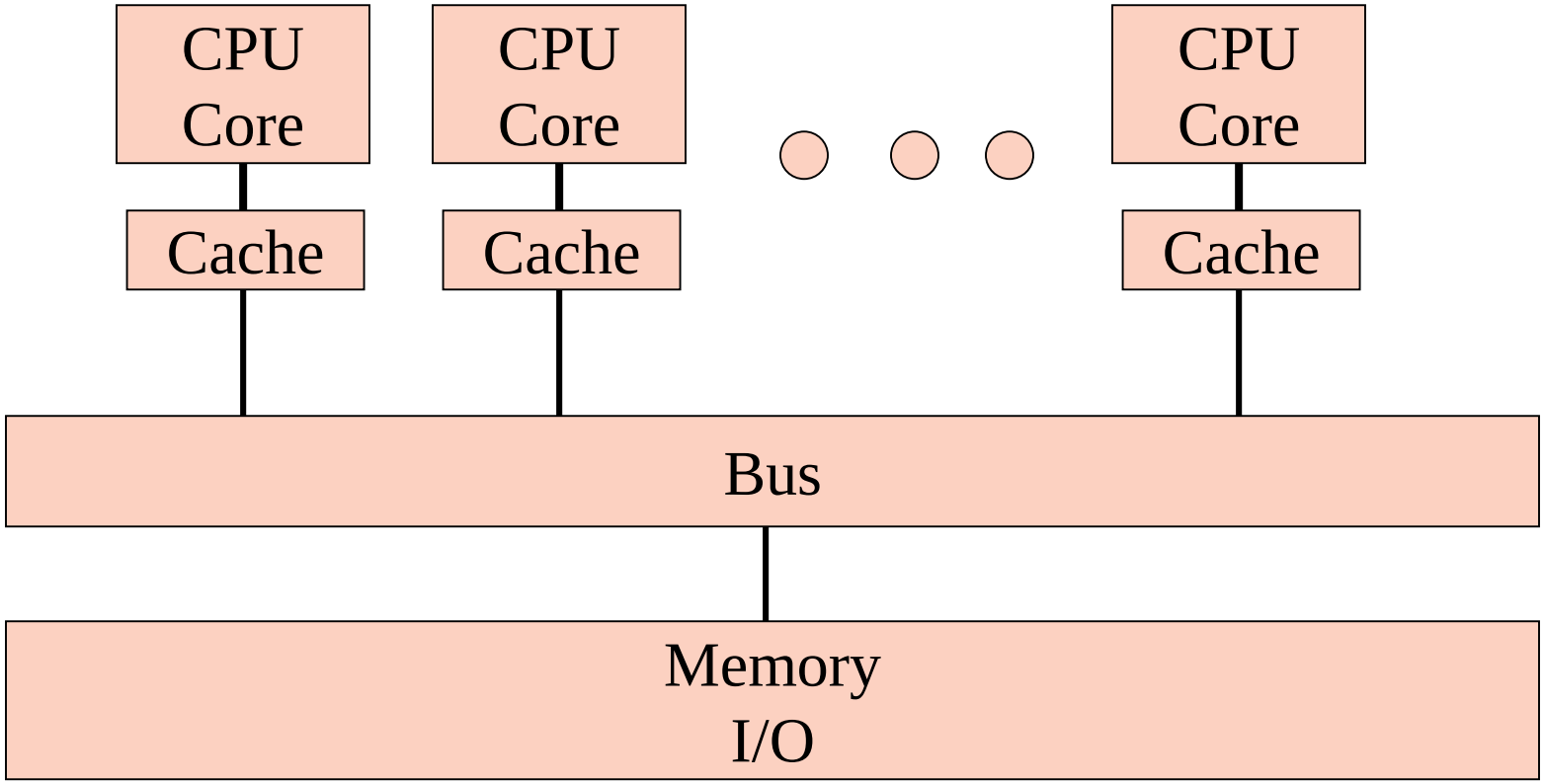


SMP (1)

- **Symmetric multiprocessing** (**SMP**) involves a multiprocessor computer hardware and software architecture where two or more identical processors connect to a single, shared main memory, have full access to all I/O devices, and are controlled by a single OS instance that treats all processors equally, reserving none for special purposes.
  - Most multiprocessor systems today use an SMP architecture. In the case of multi-core processors, the SMP architecture applies to the cores, treating them as separate processors.



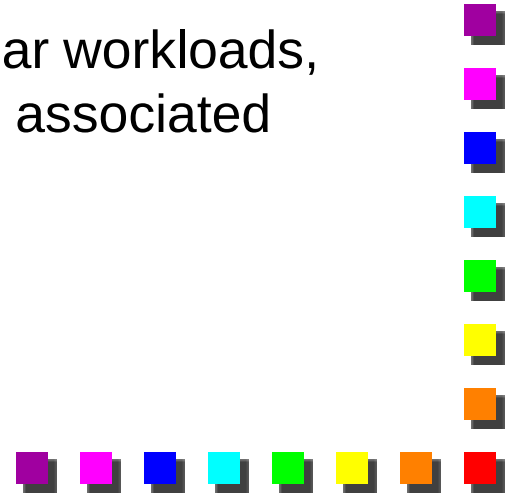
SMP (2)



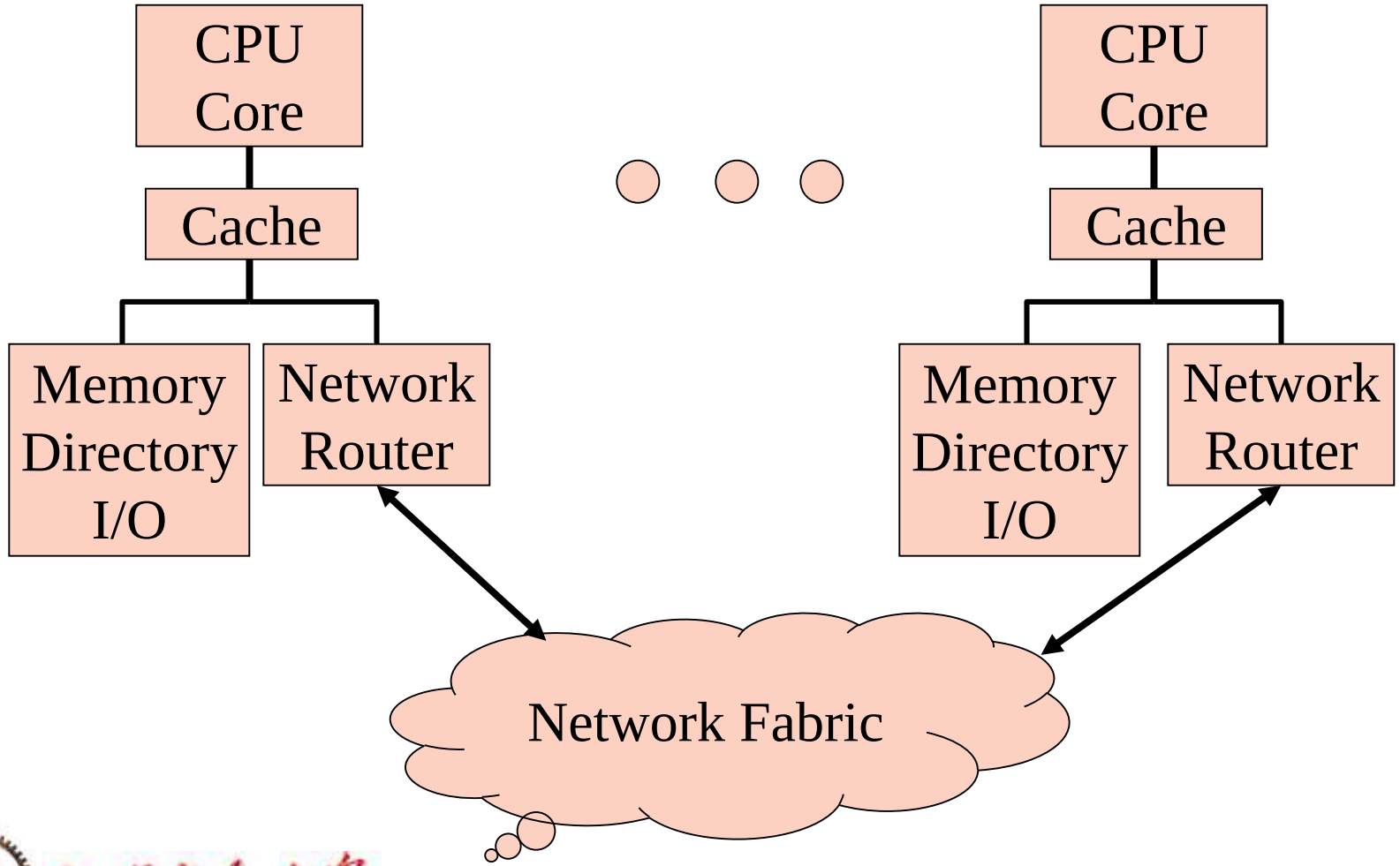
上海交通大學

NUMA (1)

- **Non-uniform memory access** (**NUMA**) is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to the processor.
  - Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors).
- **Benefits**
  - The benefits of NUMA are limited to particular workloads, notably on servers where the data are often associated strongly with certain tasks or users

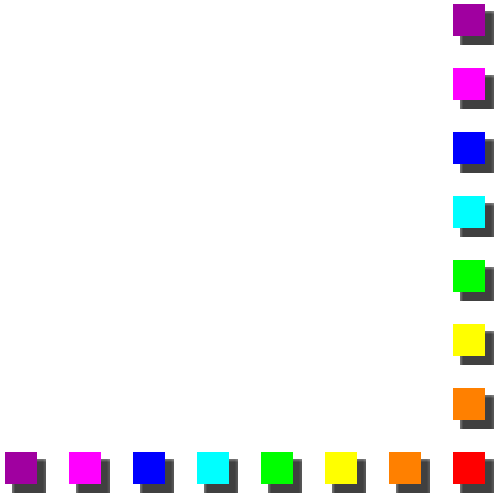


NUMA (2)

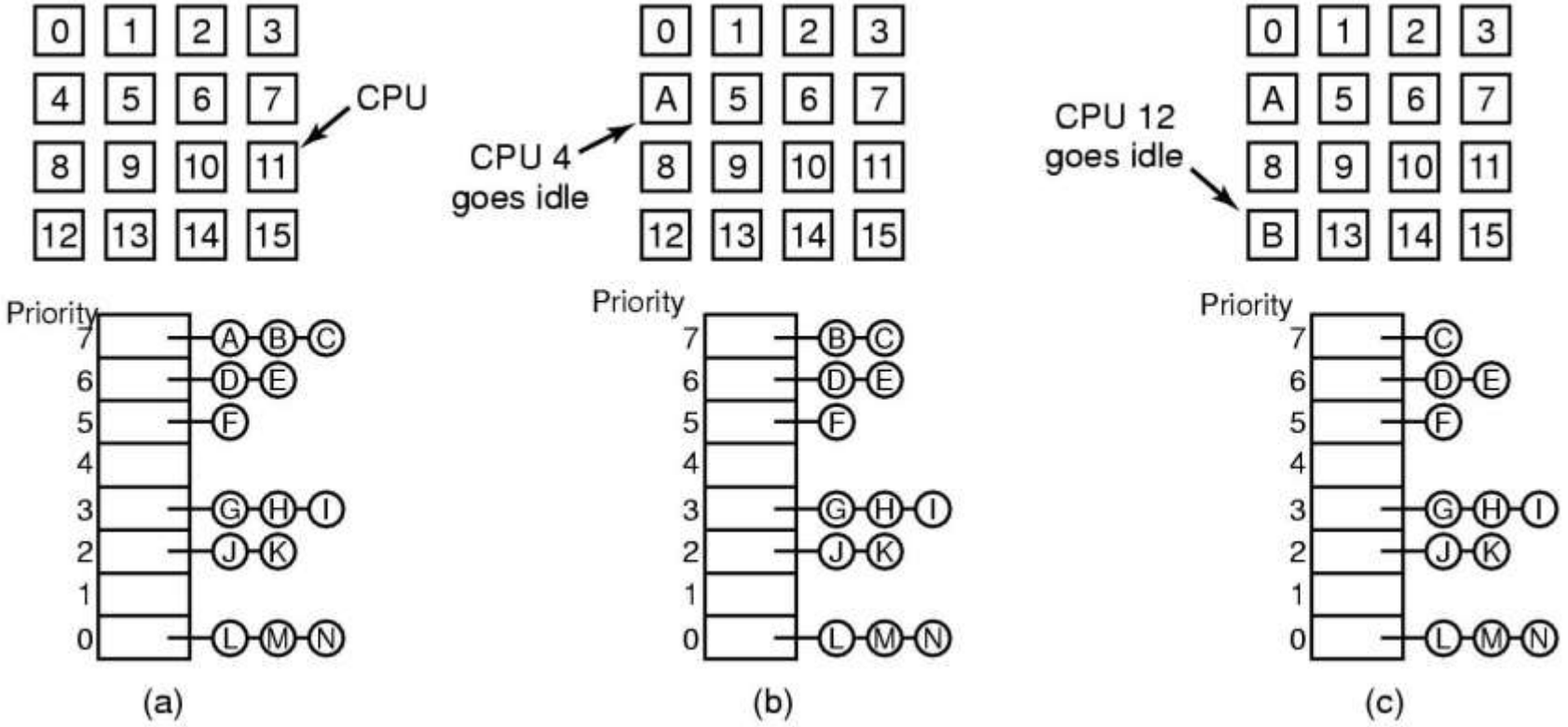


Outline

- Introduction on SMP
- SMP and NUMA
- Process Scheduling with SMP
- Synchronization Problem with SMP

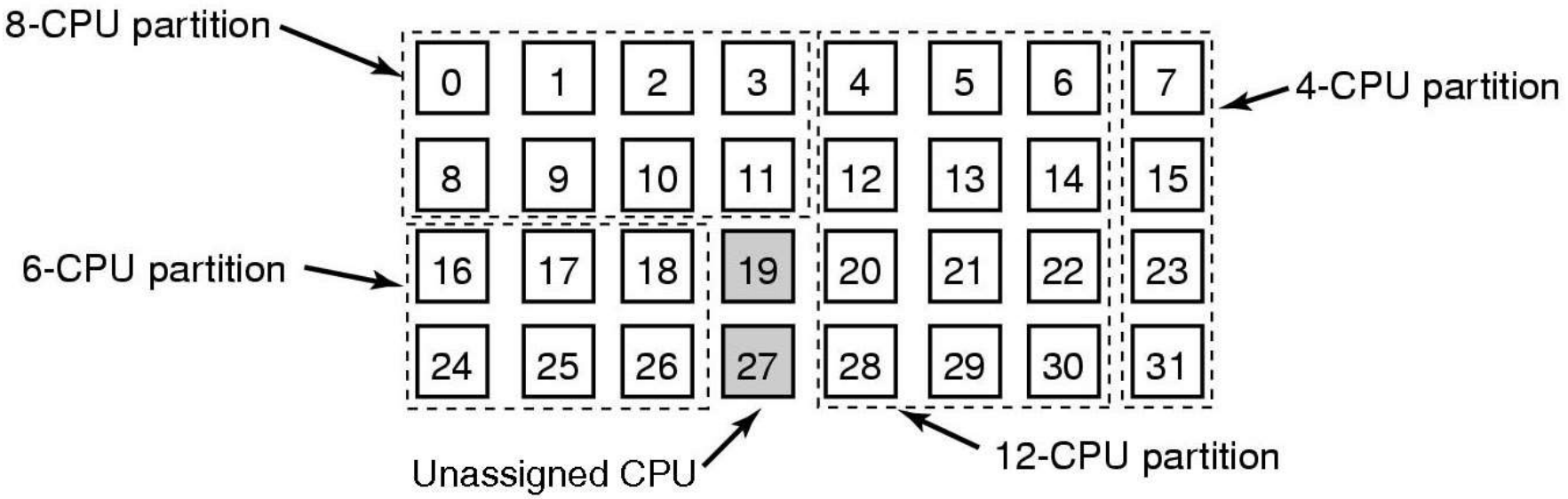


SMP Scheduling (1)



- Timesharing
  - note use of single data structure for scheduling

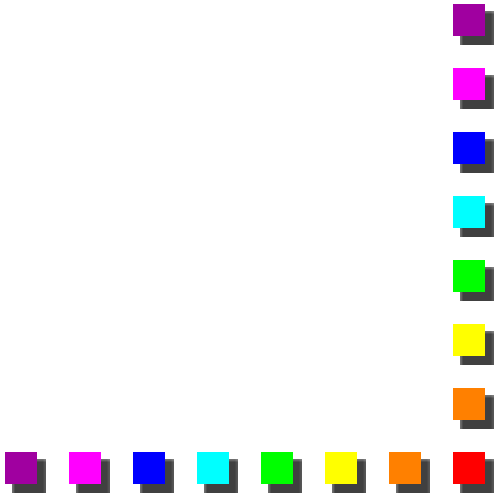
SMP Scheduling (2)



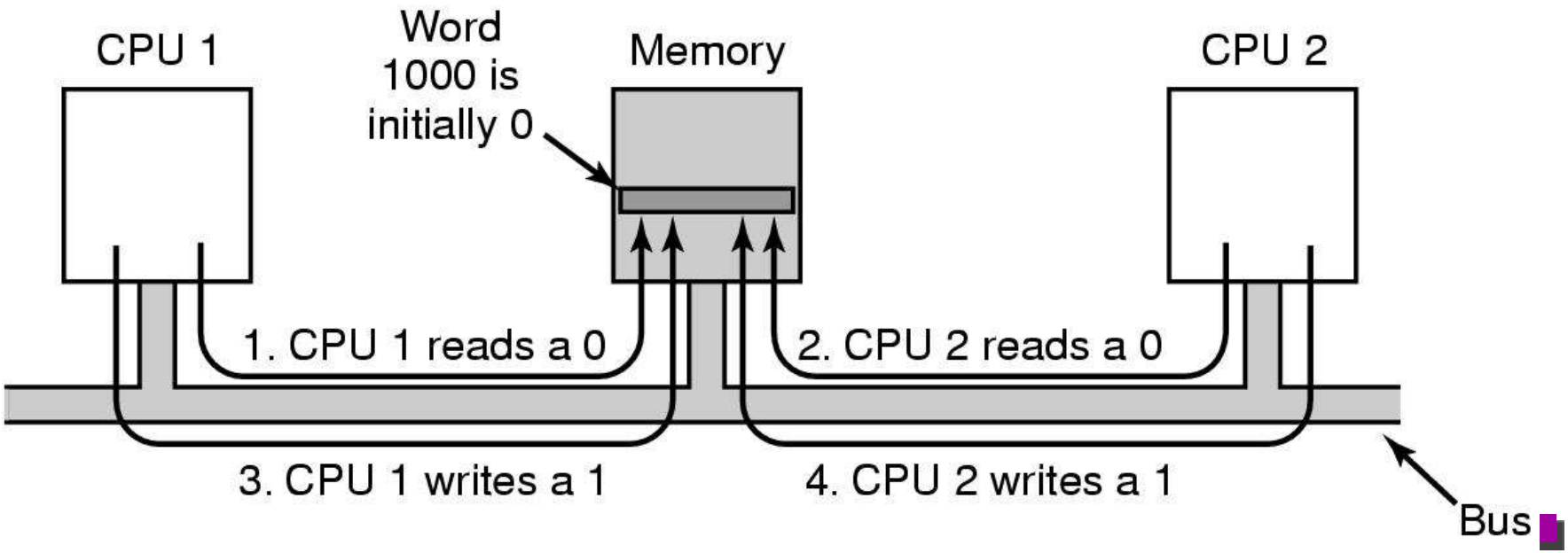
- Space sharing
  - multiple threads at same time across multiple CPUs

Outline

- Introduction on SMP
- SMP and NUMA
- Process Scheduling with SMP
- Synchronization Problem with SMP

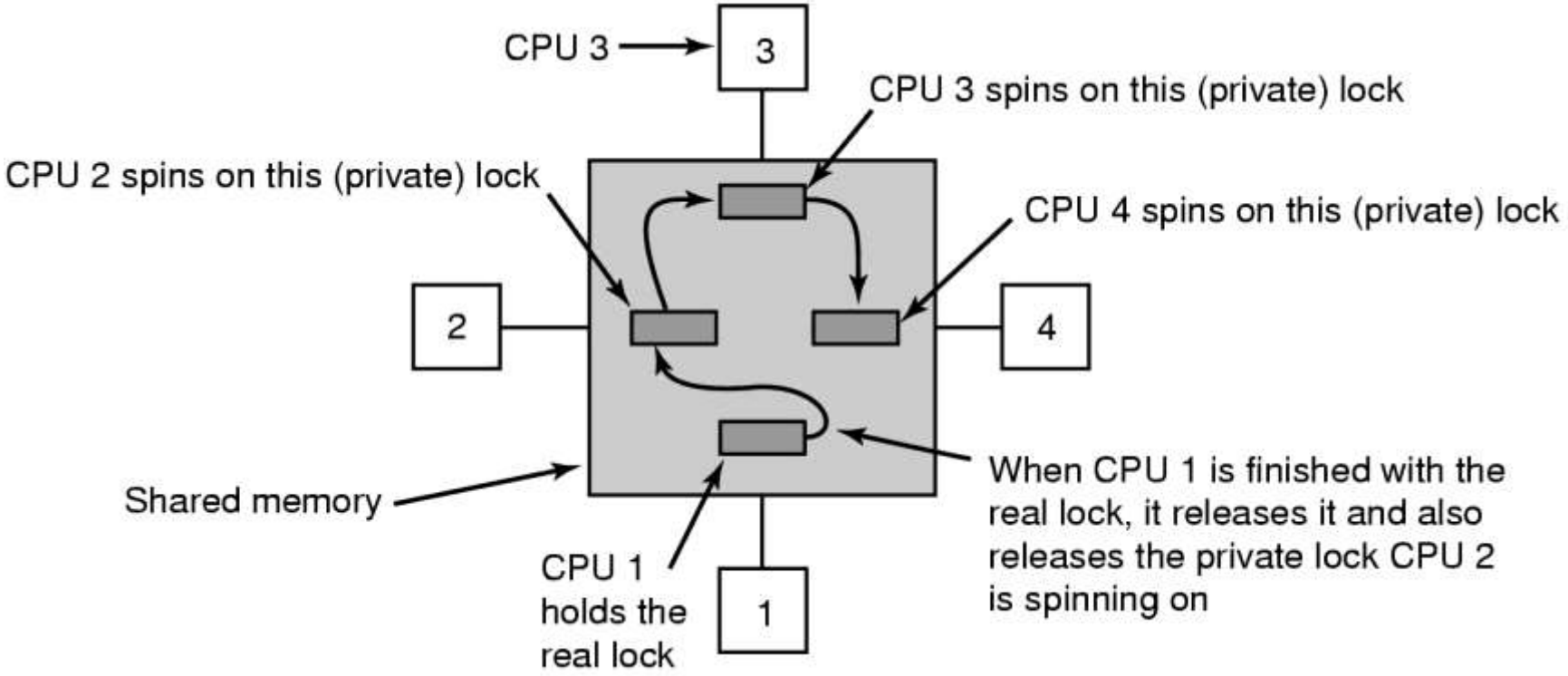


Synchronization Problem (1)



Instruction can fail if bus already locked

Synchronization Problem (2)



Multiple locks used to avoid cache thrashing

5. Reading Source Code -- SMP

Chentao Wu
Associate Professor
Dept. of CSE, SJTU
wuct@cs.sjtu.edu.cn

一、三个问题

- 在SMP机器上，Linux的启动过程是怎样的？
- 在SMP机器上，Linux的进程调度如何进行？
- 在SMP机器中，中断系统有何特点？

二、Linux启动过程（基本概念）

- SMP机器中，有以下几个基本概念：
  - BSP：也叫BP，是Bootstrap Processor的缩写，即启动CPU，在操作系统启动过程的前期，只有BSP在执行指令。
  - AP：Application Processor的缩写，即应用CPU。
  - APIC：高级可编程中断控制器，分为本地APIC和IO APIC。
  - IPI：处理器间中断，用于处理器之间的通信。

Linux启动过程（续）

- 由于BIOS代码并不是支持多线程的，所以在SMP中，系统必须让所有AP进入中断屏蔽状态，不与BSP一起执行BIOS代码。为了达到这一目的，可以利用两种手段：1、利用系统硬件本身进行处理；2、系统硬件与BIOS程序一起处理。在后一种方法中，BIOS程序将其它AP置于中断屏蔽状态，使其休眠，只选择BSP执行BIOS代码中的后继部分。BIOS要同时完成对APIC以及其他与MP相关的系统组件初始化过程，并建立相应的系统配置表格，以便操作系统使用。

Linux启动过程（主要流程）

1. BIOS初始化（屏蔽AP，建立系统配置表格）。
2. MBR里面的引导程序（Grub，Lilo等）将内核加载到内存。
3. 执行head.S中的startup_32函数（最后将调用start_kernel）。
4. 执行start_kernel，这个函数相当于应用程序里面的main，在早期的内核中，这个函数就叫main。
5. start_kernel进行一系列初始化，最后将执行
   smp_init() //启动各个AP，关键的一步
   rest_init() //调用init()创建1号进程，自身执行cpu_idle()成为0号进程
6. 1号进程即init进程完成余下的工作。

Linux启动过程（smp_init()函数）

- static void __init smp_init(void) {
  smp_boot_cpus();
  smp_threads_ready=1;
  smp_commence(); //让各AP开始执行指令
  }

- smp_boot_cpus()函数初始化各AP，设置为待命模式（holding pattern，就是处于等待BSP发送IPI指令的状态），并为之建立0号进程。

- smp_threads_ready=1表示各AP的idle进程已经建立。

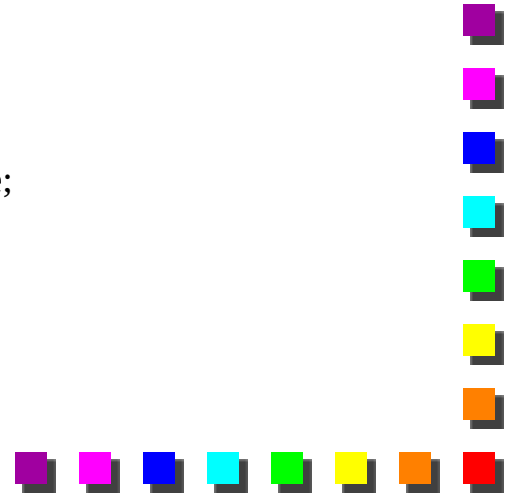
- smp_commence()函数让各AP开始执行指令。

- 注意：在smp机器中，有几个CPU，就有几个idle进程（0号进程），但1号进程即init进程只有一个。

Linux启动过程（smp_boot_cpus()函数）

```
void __init smp_boot_cpus(void)
{
  for (apicid = 0; apicid < NR_CPUS; apicid++) {
    if (apicid == boot_cpu_id)  continue; // 是BP，因为上面已经初始化完毕，就不再需要初始化
    if (!(phys_cpu_present_map & (1 << apicid)))  continue; // 如果CPU不存在，不需要初始化
    if ((max_cpus >= 0) && (max_cpus <= cpucount+1))  continue; //如果超过最大支持范围，不需要初始化
    do_boot_cpu(apicid);// 对每个AP调用do_boot_cpu函数
  }
}

void __init smp_boot_cpus(void)
{
   init_cpu_to_apicid();
   for (bit = 0; bit < BITS_PER_LONG; bit++){
      apicid = cpu_present_to_apicid(bit);
      if (apicid == BAD_APICID)    continue;
      if (apicid == boot_cpu_apicid)     continue;
      if (!(phys_cpu_present_map & apicid_to_phys_cpu_present(apicid)))  continue;
      do_boot_cpu(apicid);
    }
}
```



Linux启动过程（do_boot_cpu(cpuid)函数）

```
static void __init do_boot_cpu (int apicid)  //  linux/arch/i386/kernel/smpboot.c
{
  struct task_struct *idle; // 空闲进程结构
  if (fork_by_hand() < 0) …. //在每个cpu上建立0号进程，这些进程共享内存
  idle->thread.eip = (unsigned long) start_secondary;
  // 将空闲进程结构的eip设置为start_secondary函数的入口处
  start_eip = setup_trampoline(); // 得到trampoline.S代码的入口地址
  stack_start.esp = (void *) (1024 + PAGE_SIZE + (char *)idle);
  *((volatile unsigned short *) phys_to_virt(0x469)) = start_eip >> 4;
  *((volatile unsigned short *) phys_to_virt(0x467)) = start_eip & 0xf;
  // 将trampoline.S的入口地址写入热启动的中断向量(warm reset vector)40:67
  apic_write_around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));// 确定发送对象
  apic_write_around(APIC_ICR, APIC_INT_LEVELTRIG | APIC_DM_INIT);   // 发送INIT IPI
  apic_write_around(APIC_ICR2, SET_APIC_DEST_FIELD(apicid));  //确定发送对象
  apic_write_around(APIC_ICR, APIC_DM_STARTUP | (start_eip >> 12));
  //发送STARTUP IPI
}
```

较新的内核中，后面的几个apic_write_around()放在wakeup_secondary_via_NMI()或wakeup_secondary_via_INIT()中完成

Linux启动过程（AP的启动过程）

- 1. AP响应IPI中断，跳转到trampoline.S的入口，装入gdt和idt，然后跳转到head.s入口执行startup_32()函数。
- 2. startup_32()中有一个ready变量，startup_32()每执行一次ready加1，ready初始值为0，所以BSP时ready为1，将跳到start_kernel()，AP时ready大于1，所以AP不会执行start_kernel()，而是跳至initialize_secondary()。
- 3. 执行initialize_secondary()，转至current->thread.esp，跳至start_secondary()。
- 4. 执行start_secondary()函数。

Linux启动过程（initialize_secondary()函数）

```
void __init initialize_secondary(void)
{
  asm volatile(
    "movl %0,%%esp\n\t"
    "jmp *%1"
    :
    :"r" (current->thread.esp),"r" (current->thread.eip));
}
```

以上过程就是设置好堆栈指针和指令指针，因而函数返回之后就跳转到了start_secondary()函数。

Linux启动过程（start_secondary()函数）

```
int __init start_secondary(void *unused)
{
  cpu_init();
  smp_callin();
  while (!atomic_read(&smp_commenced))
    rep_nop();
  local_flush_tlb();
  return cpu_idle(); // 进入空闲进程
}
```

- 至此，AP启动完成。

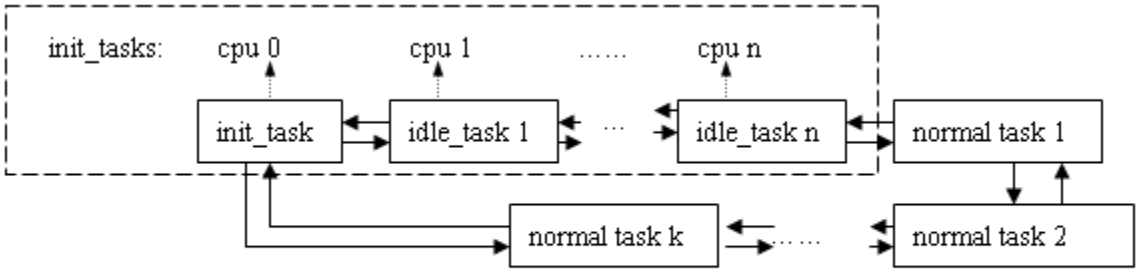
三、Linux进程调度

- 基本数据结构task_struct：表示一个任务（进程）。

```
struct task_struct{
  volatile long states;  //当前状态，包括可运行、可中断、不可中断、停止等
  long priority;    //静态优先级，实时进程忽略此成员
  long nice;    //用户可以控制的优先级
  unsigned long rt_priority;  //实时进程优先级
  long counter;  //剩余时间片，开始运行时初值等于priority
  unsigned long policy;  //调度策略，有
    SCHED_FIFO,SCHED_RR,SCHED_OTHER三种
  struct task_struct *next_task，*prev_task; //前（后）一个任务
  struct task_struct *next_run，*prev_run;  //前（后）一个可运行任务
  unsigned short uid，gid，euid，egid;  //用户id，组id，有效用户id，有效组id
  int pid;  //进程号
  struct thread_struct thread; //保存执行环境，包括一些寄存器内容
  struct mm_struct *mm;  //内存结构
  int processor;   //正在使用的CPU
  int last_processor;  //上次使用的CPU
  int lock_depth;  //内核锁的深度
}
```

Linux进程调度（续）

- 与task_struct有关的全局变量：
- task[NR_TASKS]：所有进程包括0号进程的数组，构成一个双向循环链表，表头是BSP的0号进程，即init_task。
- init_tasks[NR_CPUS]：所有CPU的0号进程的数组，构成一个链表，它是上一个链表的子链表，调度器通过idle_task(cpu)宏来访问这些idle进程。
- runqueue_head：所有就绪进程（状态为可运行的进程）构成一个链表，表头是runqueue_head。
- current：表示当前CPU的当前进程。



图：进程管理相关的数据结构示意图

（注：新进程总是添加到 init_task 的左端，即 prev 端，如图）

- 此图显示了task和init_tasks的关系，注意BSP的0号进程叫init_task而不是idle_task，但这个init_task不是1号进程init。

上海交通大学

Linux进程调度（续）

- 基本数据结构schedule_data：表示一个CPU。

```
static union {
   struct schedule_data {
      struct task_struct * curr; //此CPU上的当前进程
      cycles_t last_schedule;  //此CPU上次进程切换的时间
   } schedule_data;
   char __pad [SMP_CACHE_BYTES];
}
```

- 这种union被组织在一个叫aligned_data [NR_CPUS]的数组中，每个元素代表一个CPU。

Linux进程调度（续）

- 进程调度有关的主要函数和宏：
- schedule()：进程调度的主函数。
- switch_to()：schedule()中调用，进行上下文切换的宏。
- reschedule_idle()：在SMP系统中，如果被切换下来的进程仍然是可运行的，则调用reschedule_idle()重新调度，以选择一个空闲的或运行着低优先级进程的CPU来运行这个进程。
- goodness()：优先级计算函数，选择一个最合适的进程投入运行。

Linux进程调度（续）

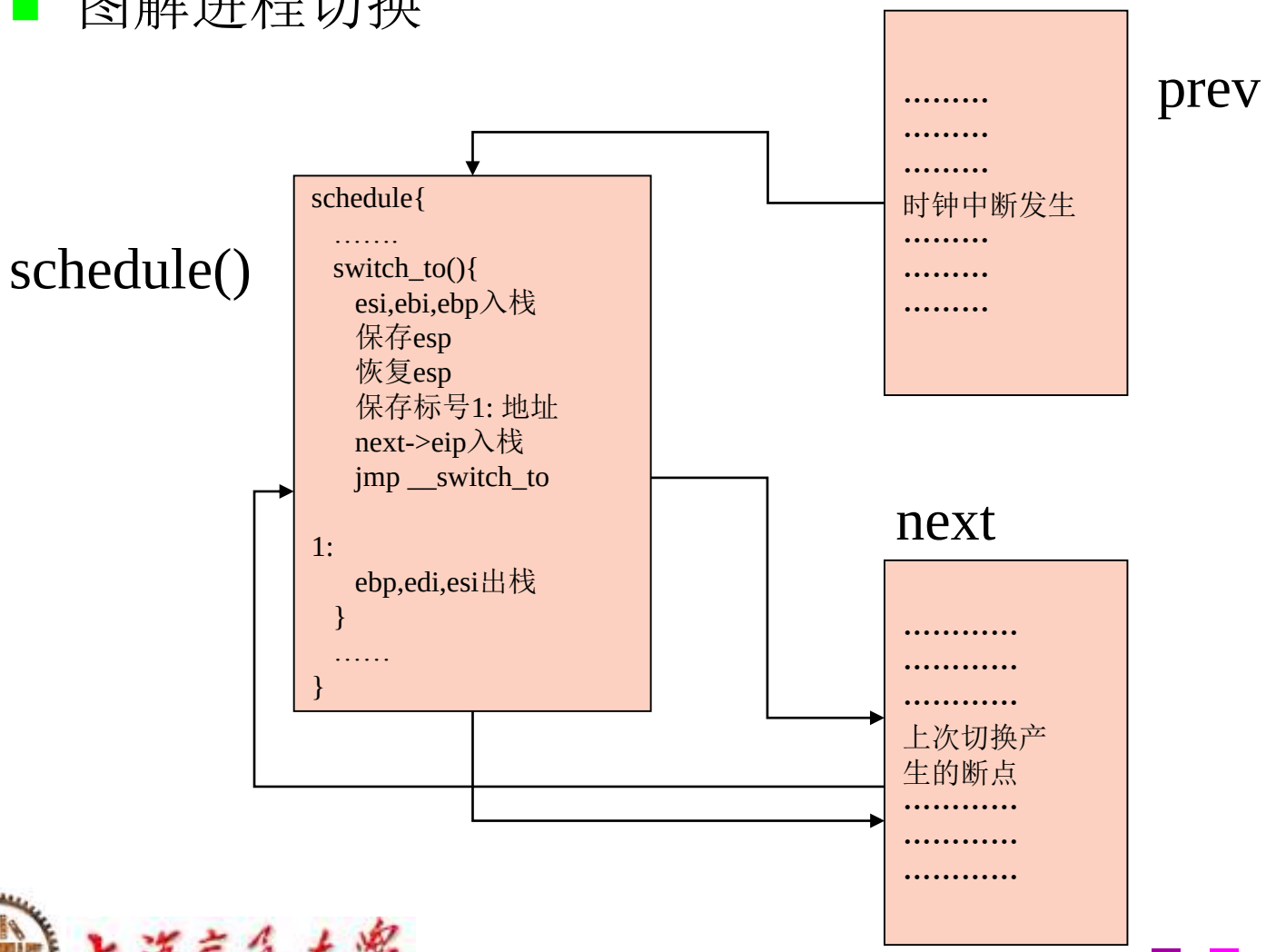
- schedule()函数的主要工作：
- 1. 从移走进程processor域读取cpu标识，存入局部变量this_cpu。
- 2. 初始化sched_data变量，指向当前处理器schedule_data结构。
- 3. 调用goodness()函数选取进程,对于上一次也在当前处理器的进程加上PROC_CHANGE_PENALTY的优先权。
- 4. 如果必要,重新计算动态优先权。
- 5. 设置sched_data->last_schedule值为当前时间。
- 6. 调用switch_to()宏执行切换。
- 7. 调用schedule_tail()，如果传入的prev进程仍是可运行的而且不是空闲进程，schedule_tail调用reschedule_idle()来选择一个合适的处理器。

Linux进程调度（续）

- switch_to(prev,next,last)的工作：
- 1. 将esi，edi，ebp压入堆栈。
- 2. 堆栈指针esp保存到prev->thread.esp。
- 3. 将esp恢复为next->thread.esp。
- 4. 将标号1：的地址保存到prev->thread.eip。
- 5. 将next->eip压入堆栈。
- 6. 无条件跳转到__switch_to()函数，切换LDT和fs，gs等寄存器，由于有了上一步的工作，因此本函数返回时已经切换到了next的上下文。
- 7. switch_to()中jmp指令以后的代码即标号1：的代码作的工作与第一步相反，即从堆栈弹出ebp，edi和esi。这部分的代码是下次该进程运行时最先执行的代码。

Linux进程调度（续）

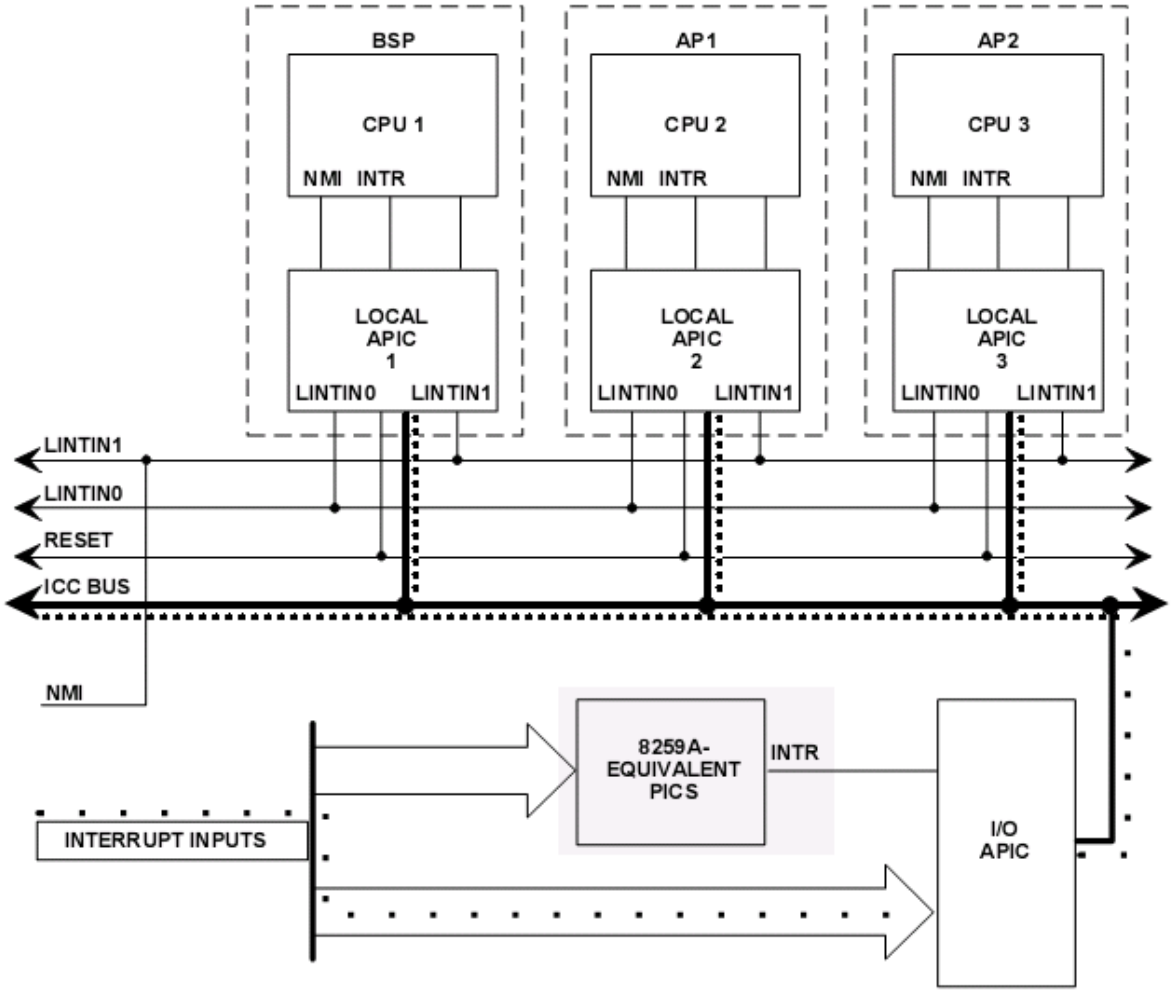
- 图解进程切换



Linux进程调度（续）

- reschedule_idle()的工作过程：
- 1. 先检查p进程上一次运行的cpu是否空闲，如果空闲，这是最好的cpu，直接返回。
- 2. 找一个合适的cpu，查看SMP中的每个CPU上运行的进程，与p进程相比的抢先权，把具有最高的抢先权值的进程记录在target_task中，该进程运行的cpu为最合适的CPU。
- 3. 如target_task为空，说明没有找到合适的cpu，直接返回。
- 4. 如果target_task不为空，则说明找到了合适的cpu，因此将target_task->need_resched置为1，如果运行target_task的cpu不是当前运行的cpu，则向运行target_task的cpu发送一个IPI中断，让它重新调度。

四、Linux中断系统



Linux中断系统（续）

- 本地APIC的作用：

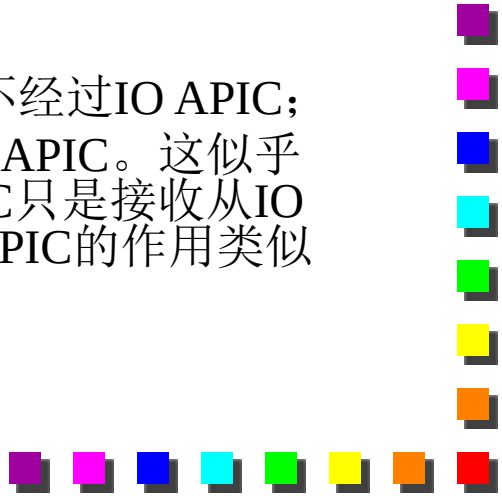
1. 接收理本地外部中断（直接连在LINTIN 0/1上的设备）。
2. 接收本地内部中断（除法错误等软件上的中断）。
3. 接收来自IO APIC的中断。

- IO APIC的作用：

1. 接收系统总线上的IPI消息。
2. 接收外部设备的中断。
3. 将接收到的中断分发给本地APIC。

- 注意：

1. 外设可以通过LINTIN0/1直接连在某一个本地APIC上，不经过IO APIC；
2. 处理器间中断先由IO APIC接收，然后分发给相应的本地APIC。这似乎暗示着中断的分发策略完全是IO APIC的事情，本地APIC只是接收从IO APIC发过来的中断，并不区分是IPI还是外部中断。IO APIC的作用类似于以太网交换机。



Linux中断系统（续）

- Linux启动过程中有一步是调用函数init_IRQ，作用是初始化各个中断向量。
- init_IRQ中，调用set_intr_gate(unsigned int n, void *addr)函数注册中断向量，n表示中断向量号，addr是中断响应函数的名字（地址）。
- 各种跟SMP相关的中断的入口函数是基本相同的，Linux提供了BUILD_SMP_INTERRUPT(x,v)宏来完成中断入口函数的定义和中断响应函数的声明，时钟中断入口函数的处理稍有不同，使用的是BUILD_SMP_TIMER_INTERRUPT(x,v)宏，x是中断响应函数的名字，v是中断向量号。中断入口函数名是call_前缀加上入口函数的名字，如：apic_timer_interrupt是时钟中断响应函数，对应的入口函数是call_apic_timer_interrupt。SMP系统的中断响应函数通常还需要加上smp_前缀，如smp_apic_timer_interrupt。

Linux中断系统（续）

- Linux针对IA32的SMP系统定义了五种主要的IPI：

1. CALL_FUNCTION_VECTOR：发往自己除外的所有CPU，强制它们执行指定的函数；
2. RESCHEDULE_VECTOR：使被中断的CPU重新调度；
3. INVLIDATE_TLB_VECTOR：使被中断的CPU废弃自己的TLB缓存内容。
4. ERROR_APIC_VECTOR：错误中断。
5. SPUROUS_APIC_VECTOR：假中断。

Linux中断系统（续）

- 发送IPI的函数主要有4个，分别是：
- send_IPI_self(int vector)；发送IPI给自己，中断类型由vector指定。
- send_IPI_mask(int mask,int vector)；发送IPI给某一个或某几个CPU，发送目标由掩码mask决定，中断类型由vector决定。
- send_IPI_all(int vector)；发送IPI给所有CPU，参数意义同上。
- send_IPI_allbutself(int vector)；发送IPI给除自己以外的所有CPU，参数意义同上。
- 早期曾经存在一个发送给单个CPU的send_IPI_single的函数，在IA32体系中现已废弃。

五、结论

- Linux对SMP的支持主要体现在三个方面：
- 启动过程：BSP负责操作系统的启动，在启动的最后阶段，BSP通过IPI激活各个AP，在系统的正常运行过程中，BSP和AP基本上是无差别的。
- 进程调度：与UP系统的主要差别是执行进程切换后，被换下的进程有可能会换到其他CPU上继续运行。在计算优先权时，如果进程上次运行的CPU也是当前CPU，则会适当提高优先权，这样可以更有效地利用Cache。
- 中断系统：为了支持SMP，在硬件上需要APIC中断控制系统。Linux定义了各种IPI的中断向量以及传送IPI的函数。