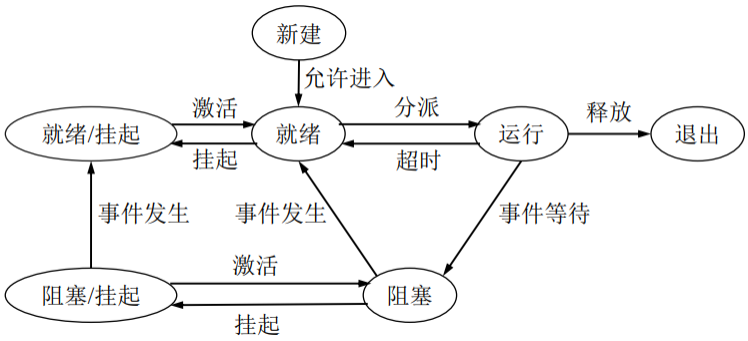
操作系统

- 操作系统课堂检测知识点
  - 操作系统的目标：方便性/有效性/可维护性/开放性
  - 处理器管理包括：进程管理/处理器调度/寄存器管理，不包括内存越界
  - 文件管理包括：目录管理/存储空间管理/文件访问管理等，不包括文件加密
  - 从系统的角度来看，操作系统不关注使用方便性
  - 系统调用是用户程序或其他系统程序 访问硬件资源的唯一途径。
  - 系统调用向系统传递参数——通过寄存器/内存/堆栈
  - 微内核功能包括低级存储管理/进程间通信/IO和中断管理，不包括页面管理
  - 中断既可以由硬件触发也可以由软件触发
  - 多道程序的度表示内存中的进程数
  - 进程的就绪态和阻塞态不能相互转换
  - 新建进程不能直接成为就绪状态进程——新建态
  - 当系统“启动系统服务”时，当前进程被阻塞
  - 线程特定数据——复制且不与父进程共享的数据
  - 套接字不是高级通信方案
  - 可以创建没有任何内核级支持的线程库
  - 一种是没有获取到锁的线程就一直循环等待判断该资源是否已经释放锁，这种锁叫做自旋锁，它不用将线程阻塞起来(NON-BLOCKING)；还有一种处理方式就是把自己阻塞起来，等待重新调度请求，这种叫做互斥锁
  - 一条原语作为一个不间断的单元执行
    - 原语可能不止包括一条机器指令
  - 有限等待：对要求访问临界资源的进程，应该能保证能在有限时间内进入自己的临界区
  - 原子性：每个事物中的所有操作要么全部完成，要么就像全部没有发生一样。
  - 信号量可以用于防止进入临界区前出现忙等
  - 高级通信机制：共享内存/管道通信/消息传递
  - 临界区是访问或者更新共享数据的代码段。
  - 操作系统通常“认为永远不会发生死锁”
  - 指令执行结束不是进程调度的时机
  - 在FCFS中，护航效应指执行时间短的进程长时间等待执行时间长的进程
  - 进程调度中操作系统的工作
    - 保存运行进程的现场信息
    - 在就绪队列中选择一个进程使其占用CPU
    - 为新选中的进程恢复现场

- 丢弃任务在实时系统中很少发生
- CPU生成的地址为逻辑地址
- 逻辑地址到物理地址的映射是通过内存管理单元（MMU）实现的
- 重定位寄存器：存的是基址
- 为确保将数组元素全部赋值完成后再对其进行运算，需要使用DSB
- LRU是大多数系统上实现的算法
- 位示图，从左到右，从上到下
- 多进程可同时打开同一文件的环境中
  - 系统打开文件表
  - 进程打开文件表
- 访问磁盘每次数据传输是——一个磁盘块
- 树形目录结构是最常见的目录结构
- 文件的逻辑结构
  - 堆
  - 顺序结构
  - 散列结构
- 缓冲技术中的缓冲XX在内存中
- 字符流设备一次传输一个字节数据
- 主机可以写入I/O端口的“控制”寄存器以启动命令或更改设备的模式
- DMA可以从主CPU窃取内存访问周期
- 大多数的低速设备都属于“独享”设备
- spooling技术是采用空间换时间的技术
- 字节多路通道连接大量低速或中速的I/O设备
- 在使用恒定线速度的介质上，每次到的比特密度恒定——线速度恒定
- 磁盘定位时间=寻道时间+旋转时间
- FCFS不考虑磁头的当前位置
- 磁盘读写依次经历：
  - 等待设备空闲
  - 等待数据通道可用
  - 定位磁道
  - 定位扇区
- 磁盘控制器通常设置有缓存
- 磁盘存储器的最小读写单位是磁盘块
- DMA传送结束后的处理由中断服务程序完成
- 用户级线程间的切换比内核级线程间的切换效率高
-

- 复习大纲普通知识点
  - 操作系统地位
    - 是计算机系统中的基础系统软件
    - 是计算机硬件上的第一层软件
    - 其他软件通过操作系统使用计算机系统硬件
  - 操作系统功能
    - 处理机管理：把CPU的时间合理地分配给各个执行的程序，使其得到充分有效的利用
    - 存储管理：管理计算机的内存，保证程序运行互不冲突
    - 设备管理：对系统中所有I/O设备进行管理
    - 文件管理：有效的管理外存储器空间存储及结构
    - 进程调度：操作系统负责进程的调度管理
    - 用户接口：为用户使用计算机系统提供了良好的工作环境和用户界面
  - 操作系统结构
    - 整体式结构
      - 接口简单，系统效率高，无可读性，不具备可维护性
    - 层次式结构
      - 难点：合理的层次划分
    - 微内核结构
      - 最基本的功能保留在内核
      - 功能
        - 低级存储管理
        - 进程间通信
        - I/O和中断管理
      - 只有一个模块——微内核—运行在内核态上
    - 模块化结构
      - 模块化结构是指利用面向对象编程技术来生成模块化的内核
  - 用户接口
    - 命令接口
      - 图形界面
      - 命令行
    - 系统调用
    - API
  - 处理机管理
    - 进程控制和管理
      - 对系统中所有进程从创建、执行到撤销的全过程实行有效的管理和控制

    - 进程同步和互斥管理
        - 提供软件或硬件的机制，来确保多个进程按照预想的步调顺利推进
    - 进程通信管理
        - 提供软件或硬件的机制，来确保多个进程按照预想的步调顺利推进
    - 处理机调度
        - 进程是动态的，而程序是静态的
        - 以满足系统目标（如响应时间、吞吐量、效率）的方法，把进程分配到一个或多个处理器中执行
    - 线程控制和管理
        - 对线程进行必要的控制与管理
- 并发执行
    - 间断性
    - 失去封闭性
    - 不可再现性
- 进程状态
    - 5个基本状态
        - 新建态
        - 就绪态
        - 运行态
        - 阻塞态
        - 退出态
    - 挂起
        - 就绪/挂起
        - 阻塞/挂起
    -



- 互斥与同步
    - 临界资源
        - 各进程采取互斥的方式，实现共享的资源成为临界资源
    - 临界区

        - 每个进程中访问临界资源的那段代码称为临界区
    - 实现
        - 信号量
            - 互斥：（—∞，1）
            - 范围（ - ∞，+∞）
        - 管程
            - 只有一个进程在管程内部
            - 外部调用外部，内部调用内部
            - signal和wait与普通的P，V不一样
- 信号量值
    - 当 x.value>0，表示当前有 x.value 个资源可以使用
    - 当 x.value=0，表示当前没有个资源可以使用
    - 当 x.value<0，表示当前有 |x.value| 个进程等待使用资源
- 生产者/消费者问题，P108
- 进程的前驱后继问题，p113

- 复习大纲标红知识点
    - 微内核结构的操作系统
        - 鸿蒙系统，IOS，macos，redox
    - 系统调用与API的区别与联系
        - 区别
            - API能让程序的可移植性变高
            - 系统调用比API更注重细节且更加难用
            - API是函数的定义，规定了函数的功能，与内核无直接关系，而系统调用是通过中断向内核发出请求，实现内核提供的某些服务
        - 联系
            - API通常为应用程序员调用实际的系统调用
            - 一个API可能会需要一个或多个系统调用来完成特定的功能。
    - 进程与线程
        - 进程与程序的区别与联系
            - 区别
                - 进程有一定的生命期，而程序是指令的集合，程序本身没有生命期
            - 联系
                - 进程是计算机中的程序关于某数据集合上的一次运行活动
        - 进程与线程的区别与联系
            - 区别
                - 进程是资源分配的基本单位，线程是调度和分配的基本单位

        - 进程间切换，系统开销较大；线程间切换系统开销较小
        - 线程共享本进程的资源，不利于资源的管理和保护，而进程之间的资源是独立的，能很好的进行资源管理和保护
    - 联系
        - 进程具有并发性，线程也具有并发性，从而使操作系统具有更好的并发性，提高了吞吐量
- 写时复制
    - 父进程创建子进程时，最初父子进程共享内存空间
        - 当子进程修改数据时才真正分配内存空间，这是对程序性能的优化，可以延迟甚至是避免内存拷贝，当然目的就是避免不必要的内存拷贝
- 死锁问题
    - 必要条件
        - 互斥条件
            - 每个资源每次只能分配给一个进程使用
        - 占有且等待
            - 进程占有一部分资源后还允许继续申请新的资源
        - 不可抢占
            - 任一个进程不能从另一个进程那里抢占资源，只能由占用进程自己释放
        - 循环等待
            - 存在一条循环的等待序列
    - 死锁预防——破坏四个必要条件
    - 死锁避免——银行家算法
        - 系统进行资源分配前，先计算资源分配的安全性，若此次分配不会导致系统进入不安全状态。
    - 死锁的检测
        - 进程申请资源时
        - 定时检测
        - 系统资源利用率下降
    - 死锁的解除（原则：最小代价）
        - 撤销所有死锁进程
        - 把所有死锁进程恢复到前面的某个检查点
        - 有选择地撤销死锁进程，知道不存在死锁
        - 剥夺资源，直到不存在死锁
- 页面置换算法
    - FIFO，LRU（最常用），OPT（无法实现，用于评价其他算法），CLOCK（二次机会页面置换算法）
    - FIFO：简单，易于实现，常用

- OPT：理想的算法，不能实现，可以作为算法比较的标杆
- LRU：最常用的算法，若干变种，需硬件支持
- CLOCK：效率较高，易于实现

- 磁盘调度算法
  - 评价指标：磁头移动的磁道总数
  - 调度算法
    - 先进先出（FIFO）
    - 最短查找时间优先 （SSTF）
    - 扫描算法（SCAN）
    - 循环扫描（C-SCAN）
    - LOOK & C-LOOK算法
  - 算法的选择
    - SSTF 是常见的，并且具有自然的吸引力，因为它比 FCFS 具有更好的性能。
    - 对于磁盘负荷较大的系统，SCAN 和 C-SCAN 表现更好
    - SSTF 或 LOOK 是默认算法的合理选择
- 访问b字节的平均时间：
  -
    - 寻道时间：T_s
    - 旋转速度：r转/秒
    - 每条磁道N字节
    - 访问b字节的平均时间：T_a

$$T_a=T_s+1/(2\times r)+b/(r\times N)$$

- I/O数据传输控制（原理/方法/特点）
  - 轮询
    - 当前进程需要输入操作时，CPU发出命令，外设接到命令后，开始准备数据。当外设准备数据的过程中，CPU会不断和循环检测I/O控制器的状态，若数据已经准备好了，则CPU就将I/O控制器中的数据寄存器的读入数据放至内存中；若需要读入的数据还没有完毕，下一轮继续读入。
    - 特点：简单易控制，且控制器外围接口控制逻辑少。
    - 缺点：CPU只能和外设进行串行工作，效率低。速度慢
  - 中断
    - 站在有需求的进程角度：CPU向I/O控制器发生读命令，当前进程会变成阻塞状态，然后CPU执行调度程序从就绪队列中调度一个合适的就绪进程继续执行。在新进程执行过程中，每执行一条执行，CPU就会检测一次中断，当接收到来自外设I/O控制器的中断请求信号时，CPU保存当前那进程的上下文转向执行外设的中断服务程序，CPU将数据寄存器中的数据传送到特定的内存单元中。传送过程结束后，恢复中断现场，继续执行中断前的进程。

    - 站在I/O设备的角度，当I/O控制器接受到来自于CPU的读命令后，开始准备数据，将数据放在数据寄存器中。当数据准备好了后，I/O控制器向CPU发送中断信号。
    - 优点
        - 极大的提高了CPU的利用率，初步实现了CPU与外设的并行工作。
    - 缺点
        - 由于I/O控制器中的数据寄存器中的数据与内存之间的传输仍然需要CPU参与，所以仍会消耗大量CPU时间
        - 传输数据量大，并且外设的速度较快的情况下，容易造成在多次中断的情况下出现数据丢失的现象。
- DMA
    - 当一个进程要求设备输入数据时，
        - 1、CPU对DMA进行初始话工作：比如将存放数据的内存起始地址至DMA控制器的内存地址寄存器中（传送字节数）。修改其他控制信号（如DMA启动位置等等）
        - 2、该进程放弃CPU，进入阻塞状态。进程调度程序调度其他进程执行。与此同时，由DMA控制器控制整个数据的传输。
        - 3、当输入设备将一个数据送入DMA控制器的数据缓冲寄存器之后,DMA控制器取代CPU接管数据地址总线（CPU工作周期挪用），将数据送至相应的内存单元。DNA 控制器的传输字节寄存器计数减一。回复CPU 对数据地址总线的控制权。不断重复输入传输过程直到传输完毕。
        - 4、当一批数据传输完成后，DMA控制器向CPU发送中断信号，请求中断运行进程并转向执行中断处理程序。中断处理程序保存被中断进程的现场，唤醒等待输入数据的那个进程，让他变为就绪态，然后回复现场，返回被中断的进程继续执行。
    - 优点：
        - DMA方式中，由于I/O设备直接同内存发生成块的数据交换，因此I/O效率比较高。
        - DMA 是一种比较令人满意的处理方式，通过 DMA 设备的引入将 CPU 从繁重的 I/O 操作中解放了出来。排除了CPU因并行设备过多而来不及处理以及因速度不匹配而造成数据丢失等现象。
    - 缺点
        - 1、对于不连续的数据块的传输，则需要多次DMA过程才能完成。
        - 2、DMA控制器功能的强弱，是决定DMA效率的关键因素。DMA控制器需要为每次数据传送做大量的工作，数据传送单位的增大意味着传送次数的减少。另外，DMA方式窃取了时钟周期，因为其占据了访问内存的数据总线，CPU处理效率降低了，要想尽量少地窃取始终周期，就要设法提高DMA控制器的性能，这样可以较少地影响CPU处理效率。
- 通道

        - 运行过程
            - 1、当一个进程要求输入数据时，CPU根据请求形成相应的通道程序，然后执行输入指令启动通道工作。
            - 2、申请输入的进程放弃CPU进入阻塞等待状态。进程调度程序调度其他进程运行
            - 3、通道开始执行CPU 放在主存内的通道程序，独立负责外设与主存的数据集奥换。
            - 4、数据交换后，通道向CPU发出中断请求信号，中断重在运行的进程，转向中断服务程序。
            - 5、中断处理程序保存被中断进程的现场，唤醒等待输入数据的那个进程，让他变为就绪态，然后回复现场，返回被中断的进程继续执行。
        - 优点
            - 1、进一步减少了CPU对数据传输的控制，对一个数据块的读写干预->对一组数据块的读写干预。
            - 2、实现CPU、通道和I/O的并行操作。
- 位示图
    - （i，j）= ixn+j
    - 优点：可以比较容易的找到一组连续的空间块，适用于所有类型的物理结构文件
- SPOOLING
    - 虚拟分配最成功的技术是spooling技术，该技术把一台独占设备变换为若干逻辑设备，供若干个用户（进程）同时使用。
    - 是慢速字符设备与计算机主机交换信息的一种技术
    - 以空间换时间
    - spooling技术是对脱机输入、输出系统的模拟。
    - 好处：
        - 字符设备和虚拟设备间的数据交换由spooling进程统一调度，以并行方式进行，从而提高并发，同时减少用户进程的等待时间。
        - 使得设备的利用率和系统效率都能得到提高
    - 技术特点：
        - 提高了I/O速度
        - 将独占设备改造为共享设备
        - 实现了虚拟设备功能
    - 主要特点：
        - 输入井和输出井（外存）
            - 在磁盘上开辟的两大存储空间
            - 输入井模拟脱机输入，暂存I/O设备输入的数据
            - 输出井模拟脱机输出，暂存用户程序的输出数据

- 输入缓冲区和输出缓冲区
  - 缓和CPU和磁盘之间速度不匹配的矛盾
  - 输入缓冲区：暂存由输入设备送来的数据，以后再传入输出井
  - 输出缓冲区：暂存从输出井送来的数据，以后再传送给输出设备

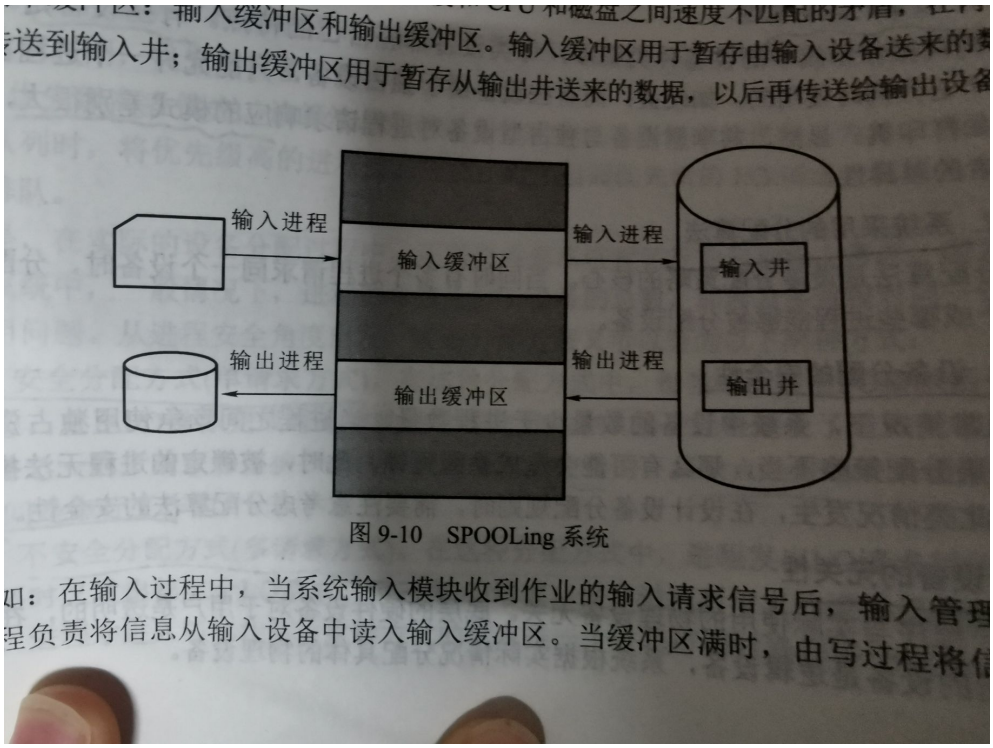


图 9-10 SPOOLing 系统

- 华为OpenEuler
  - 锁
    - 自旋锁
      - openeuler中的锁

进程A:	进程B：
……	……
while(turn!=0);	while(turn!=1);
临界区；	临界区；
turn=1;	turn=0；
……	……

能否实现进程A与进程B互斥访问临界区？
优点？缺点？

OpenEuler中的互斥与同步

- NUMA
  - NUMA（Non-Uniform Memory Access Architecture），非统一内存访问架构。
  - NUMA模式是一种分布式存储器访问方式，处理器可以同时访问不同的存储器地址，大幅度提高并行性。NUMA模式下，处理器被划分成多个"节点“，每个节点被分配有本地存储器空间。

NUMA感知队列自旋锁的引入

- 不同线程并发地访问共享地址空间时，如果这些线程占用CPU的时机或者顺序不同，产生的计算结果不同。
- 为了解决这个问题，操作系统提供互斥机制与同步机制。其中互斥机制主要使用自旋锁来实现。
- openEuler提供“NUMA感知队列自旋锁”实现互斥机制以减小MUMA体系结构中使用自旋锁的开销。

OpenEuler中的互斥与同步

- Qspinlock
  - 在队列自旋锁（Qspinlock）机制中，如果跨NUMA节点进行锁传递，将导致锁变量从一个NUMA节点迁移到另一个NUMA节点。
  - 在NUMA体系结构中，访问本地节点比访问远程节点快得多。同时，由于缓存失效，将导致额外的性能开销。因此，如果尽可能连续地在同一个NUMA节点上进行传递锁，将有效地降低使用锁带来的开销。
  - 基于这个思想，在NUMA体系结构中，openEuler采用CNA（Compact NUMA-Aware Lock，紧凑NUMA感知锁）队列代替Qspinlock中的MCS队列。

OpenEuler中的互斥与同步

- MCS 数据结构

```
struct mcs_spinlock{
    struct mcs_spinlock *next;    //指向节点的后继节点
    int locked;    //用来判断当前节点是否为队头节点
    int count;    //记录当前CPU获取锁的数目
};
```

  - 每个CPU共定义4个MCS节点，这些节点保存在每个CPU的qnodes数组中
  - 只有争用锁的线程大于2个，才会启用MCS队列。

OpenEuler中的互斥与同步

- CNA队列
  - CNA队列是MCS队列的一种变体。
  - MCS将等待获取锁的线程组织在一个队列中，而CNA则将等待获取锁的线程组织为两个队列：一个主队列，一个辅助队列。
  - 主队列的线程队头运行在相同的 NUMA 节点上。辅助队列的线程与主队列队头运行在不同NUMA节点上。
- NUMA-aware Qspinlock
  - 当一个线程试图获取CNA锁时，它将先被加入主队列。当锁被释放时，与队头不处于同一个NUMA节点的线程可能将被移动到辅助队列。
  - 这种类型的锁称为NUMA感知队列自旋锁，即NUMA-aware Qspinlock。

OpenEuler中的互斥与同步

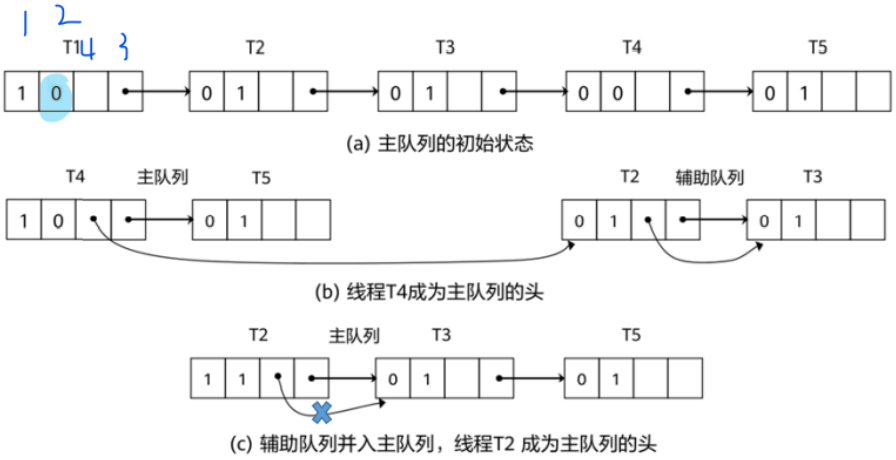
CNA的数据结构

```
//源文件：kernel/locking/qspinlock_cna.h
struct cna_node {
    struct mcs_spinlock mcs; // MCS 锁的数据结构，记录了锁状态 locked
                             // 和指向下一个MCS 节点的指针next
    int    numa_node;        //numa节点的序号
    u32    encoded_tail;     //主队列中尾节点的位置
    struct cna_node *tail;   //指向辅组队列队尾
};
```

  - 4个主要成员
    - mcs.locked、numa_node、tail、mcs.next

OpenEuler中的互斥与同步

CNA排队示例 (1)



OpenEuler中的互斥与同步

如上图(a)、(b)、(c)说明如下：

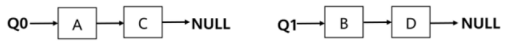
- 开始有5个线程排队，线程T1位于序号为0的NUMA节点上并在队头自旋以等待获取锁；
- 线程T1成功获取锁，遍历主队列寻找队头的继承者。遍历至节点T4时，由于线程T4也位于序号为0的NUMA节点上，故其成为队头的继承者，遍历结束；
- 线程T1将线程T4的锁状态locked置1，使其成为主队列队头。线程T2与T3位于序号为1的NUMA节点上，故其将被加入到辅助队列中。由于辅助队列当前为空队列，所以线程T2成为辅助队列的队头，线程T3成为辅助队列的队尾；
- 线程T4成功获取锁时，其遍历主队列寻找继承者。线程T4的后继节点T5位于节点序号为1的NUMA节点上，主队列中没有线程T4期望找到的继承者，辅助队列的节点被并入主队列中。辅助队列的队头线程T2成为主队列的队头，线程T5成为队尾。

- 多核调度
  - 架构
  - ARMv8架构
    - 每个集群又包含了多个CPU核，
    - 每一个CPU核都有自己的L1 Cache

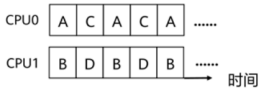
- 同一个集群中的多个CPU共享一个L2 Cache
- 多个集群共享一个L3 Cache
- L3 Cache 通过总线与内存通信

- 多核处理器面临的问题
  - 缓存一致性问题
  - 缓存亲和性问题
  - 核间数据共享
  - 负载均衡
- 迁移进程

多队列调度



openEuler使用多队列调度策略。图中队列Q0由CPU0维护，Q1由CPU1维护。



采用轮转调度算法，调度之初的情况。

一段时间后，假设A、C两个进程都结束，此时负载失衡。

---

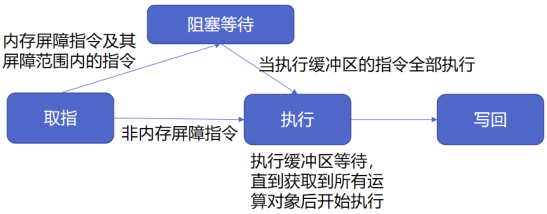
openEuler中的迁移线程

- 解决多CPU负载不均衡问题最直接的方法就是让就绪进程跨CPU迁移。比如上例中将进程D迁移到CPU0上运行后，CPU0和CPU1则实现了负载均衡。
- 在openEuler中，每个处理器都有一个迁移线程（称为migration/CPUID），每个迁移线程都有一个由函数组成的停机工作队列。如上例：
  1. CPU0向CPU1的停机工作队列中添加一个工作函数，并唤醒CPU1上的迁移线程；
  2. 该迁移线程不会被其他进程抢占，故其第一时间从停机工作队列中取出函数执行，即将进程从CPU1迁移到CPU0；
  3. 此时已实现负载均衡。

- 内存屏障
  - 内存屏障指令

基本原理：在合适的位置插入内存屏障指令，确保程序语义上的顺序依赖关系。
主要作用：禁止内存屏障指令的前后指定类型的指令的乱序执行

硬件原理：处理运行时的内存乱序访问（以**Tomasulo**算法的三个阶段为例）



- 绝对顺序模型：禁止所有优化导致的乱序执行，所有内存操作串行排队。
- 强内存顺序模型：以x86为代表，只允许store-load（即先执行store指令，再执行load指令）乱序执行。
- 弱内存顺序模型：以ARM为代表，允许所有情况下的指令乱序执行。

- HRRN：R=(W+S)/S，W：等待时间，S：执行时间
- RR常用于分时系统
- 固定分区分配：主存利用率低，会产生内部碎片（程序比固定分区小）
- 动态分区分配：会产生外部碎片
- belady：有些页面置换算法在某些进程调度顺序时，可能出现缺页数随页帧增加而增加
- 系统抖动：同一时间进程过多，每个进程占用的帧数下降，缺页率增加，CPU利用率降低，而操作系统认为是进程数太小的原因导致，所以增加进程数，恶性循环的现象
  - 缺页率显著提高，系统吞吐量下降

以上内容整理于 幕布文档