CSE 486/586分布式系统
分布式共享内存

史蒂夫-高
计算机科学与工程
布法罗大学

概述

- 今天:分布式共享内存，从内存共享的一些背景说起
- 单一机器的内存共享
  - 线程和进程
- 不同机器的内存共享
  - 线程和进程

为什么是共享内存？

- 用于共享数据
- 数据共享有两种策略。
  - 信息传递
  - 共享内存
- 信息传递
  - 发送/接收原语
  - 显式共享，不需要同步（锁）。
- 共享内存
  - 内存读/写基元（在你的代码中，你可以使用常规变量）
  - 通常需要显式同步（锁）。
- 哪个更好？
  - 取决于你的使用情况。
  - 多个写手：也许是消息传递
  - （大部分）只读数据：共享内存

线程的内存共享

- 线程属于一个进程，所以所有线程共享相同的内存地址空间。
- 例如，Java线程

```
class MyThread extends Thread {
        HashMap hm;
        MyThread(HashMap _hm ) {
                this.hm = _hm;
        }
        public void run() {
                ...
                hm.put(key, value);
        }
}
```

```
HashMap hashMap = new HashMap();
MyThread mt0 = new MyThread(hashMap); // hashMap是共享的。
我的线程mt1 = new MyThread(hashMap);
mt0.start();
mt1.start();
```

内存。线程与进程

- 对于线程来说，没有必要建立特殊的机制来共享内存。
- 但是，一个进程有自己的地址空间，所以默认情况下，不同的进程不会共享内存。
- 进程(在同一台机器上)可以在其操作系统的支持下共享内存区域。

单台机器上的共享内存

- 共享内存是IPC（进程间通信）的一部分。
  - 什么是其他IPC机制？
  - 文件、（域）套接字、管道，等等。

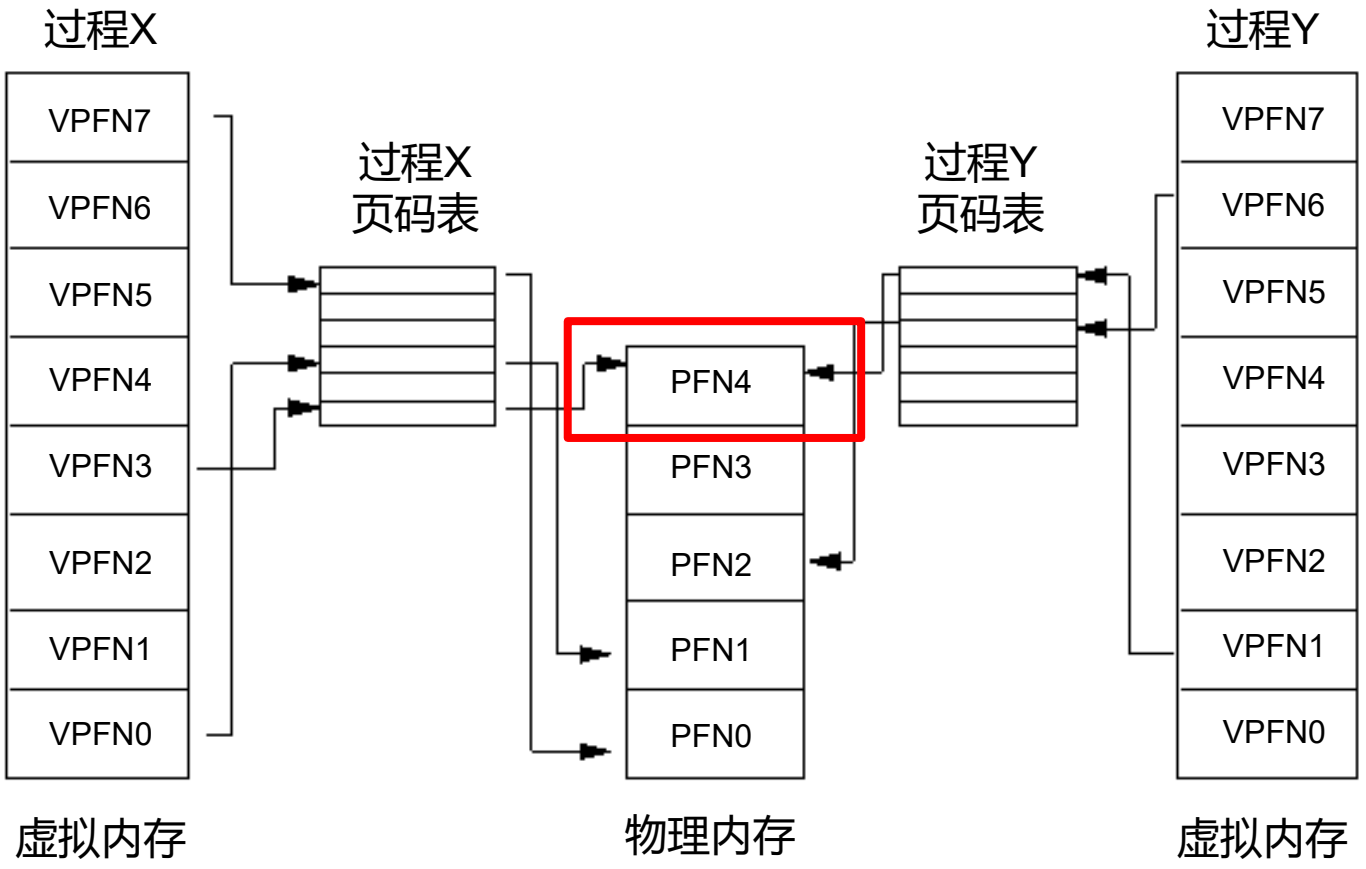
- 共享内存API（POSIX C）。
  - shm_open(): 创建并打开一个新的对象，或打开一个现有的对象。该调用返回一个文件描述符。
  - mmap(): 将共享内存对象映射到调用进程的虚拟地址空间。
  - ...和其他
- 拴马桩API(POSIX C)
  - sem_open(): 初始化并打开一个命名的信号器
  - sem_wait(): 锁定一个信号灯
  - sem_post(): 解锁一个信号灯
  - ...和其他

共享内存实例*（C语言）

```
int main() {
  const char *name = "shared"; // 与其他进程共享。
  int shm_fd;
  空白 *ptr;

  /*创建共享内存段，名称为共享。*/
  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666)。
  ...
  /*现在将共享内存段映射到进程的地址空间中
    进程的地址空间中的共享内存段 */
  ptr = mmap(0,SIZE, PROT_READ | PROT_WRITE,
                    MAP_SHARED, shm_fd, 0）。

  sprintf(ptr,"%s",message0)。

  返回0。
}
```

*改编自http://www.os-book.com

共享内存的实现

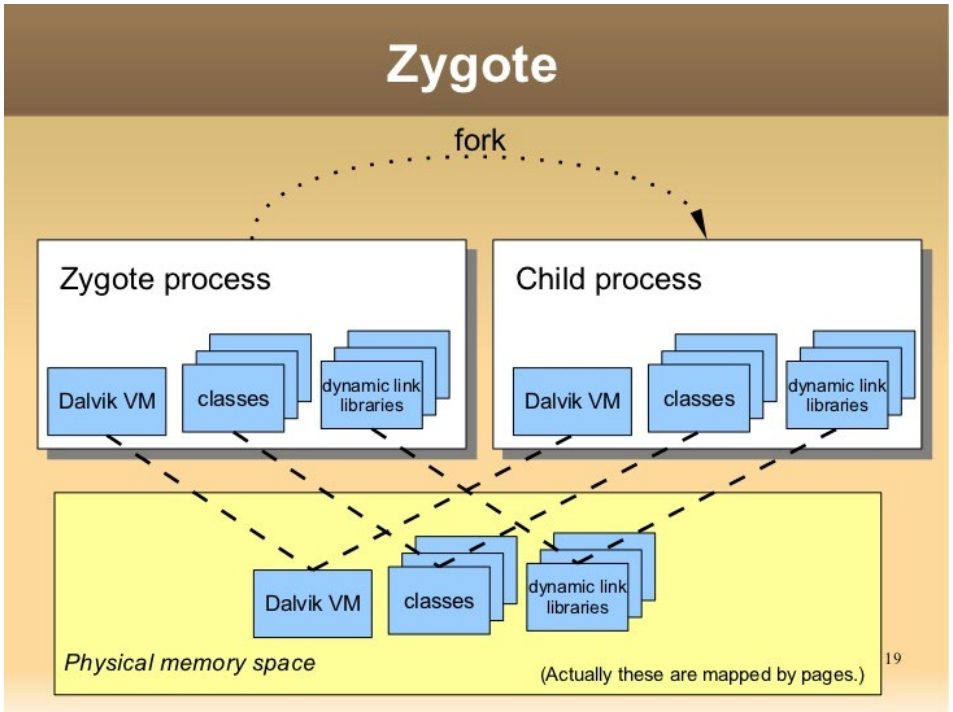


- VPFN: 虚拟页面帧号
- PFN: 物理页框号
- 改编自http://tldp.org/LDP/tlk/mm/memory.html

共享内存用例。安卓系统

- 所有的应用程序都需要框架API库、Java VM等。
  - 如果所有的应用程序进程都在其内存空间中单独拥有它们，则太昂贵了。
- 颧骨。一个启动其他一切的过程。
  - 所有应用程序进程都与Zygote共享内存。

图片来源：https://www.slideshare.net/tetsu.koba/android-is-not-just-java-on-linux/19-Zygote_forkZygote_process_Child_process



CSE 486/586 行政管理学

- PA3的成绩将在今天公布。
- PA4截止日期：5/10
  - 请早点开始。评级员需要很长很长的时间。
- 调查和课程评估
  - 调查：https://forms.gle/eg1wHN2G8S6GVz3e9
  - 课程评价：https://www.smartevals.com/login.aspx?s=buffalo
- 如果两者都有80%或更多的参与。
  - 对于你们每个人来说，我会在期中考试和期末考试之间选取较好的一个，并给较好的一个30%的权重，给另一个20%的权重。
  - (目前，期中考试为20%，期末考试为30%)。
- 今天没有背诵；用办公时间代替

分布式共享内存

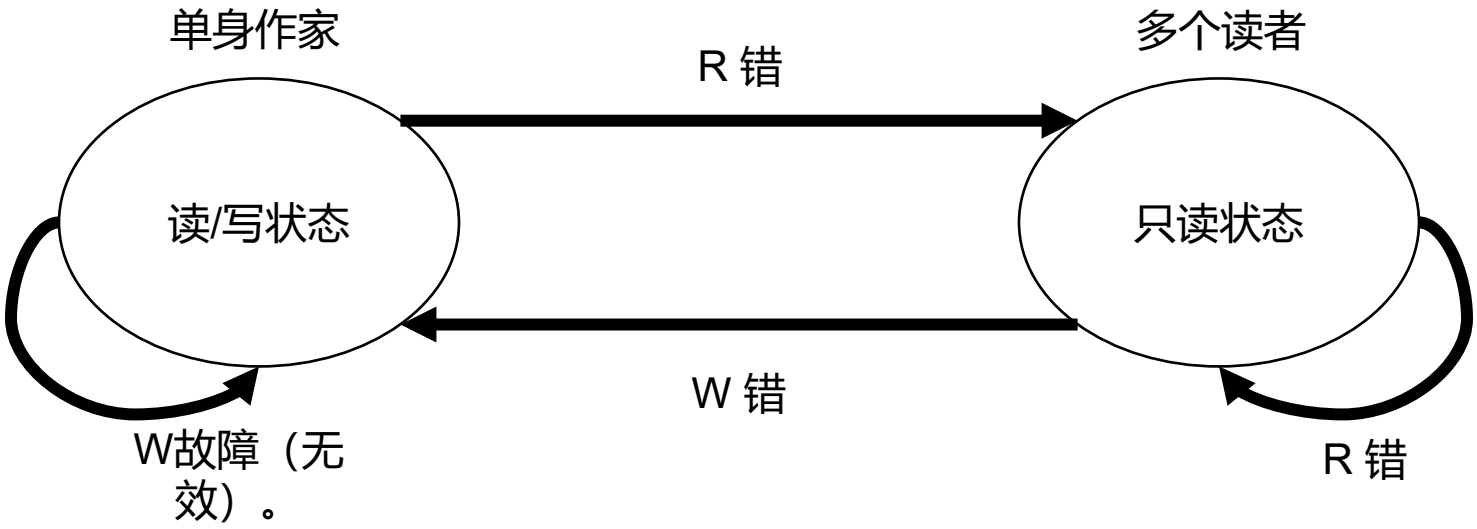
- 我们将讨论两个案例。
  - 过程的DSM
  - 线程的DSM
- 进程的DSM：在不同机器上运行的不同进程共享一个内存页。
- 共享内存页在不同的机器上被复制和同步。
  - 然而，复制并不是目的（例如，我们不是为了处理故障而保留复制）。
- 一个通用的方法是在操作系统层。
  - 与第8号幻灯片上的图类似，但在不同的机器上有流程

DSM同步化选项

- 撰写日期
  - 一个进程更新一个内存页。
  - 该更新被组播到其他副本。
  - 组播协议确定了一致性保证(例如，顺序一致性的FIFO-Total)。
  - 读取是便宜的(总是本地的)，但写入是昂贵的(总是多播的)。
- 写入-验证
  - 共享页面的两种状态：只读或读写
    - » 只读：该内存页有可能在两个或更多的进程/机器上被复制。
    - » 读和写：该内存页为该进程所独占(没有其他复制)。
  - 如果一个进程打算写到一个只读页，一个无效请求会被组播给其他进程。
  - 后来的写入可以在没有通信的情况下进行(廉价)。
  - 只有当另一个进程进行读取时，写才会被传播(对写来说很便宜，对读来说很昂贵)。
  - 但是写可以被无效化延迟(对写来说成本很高)。

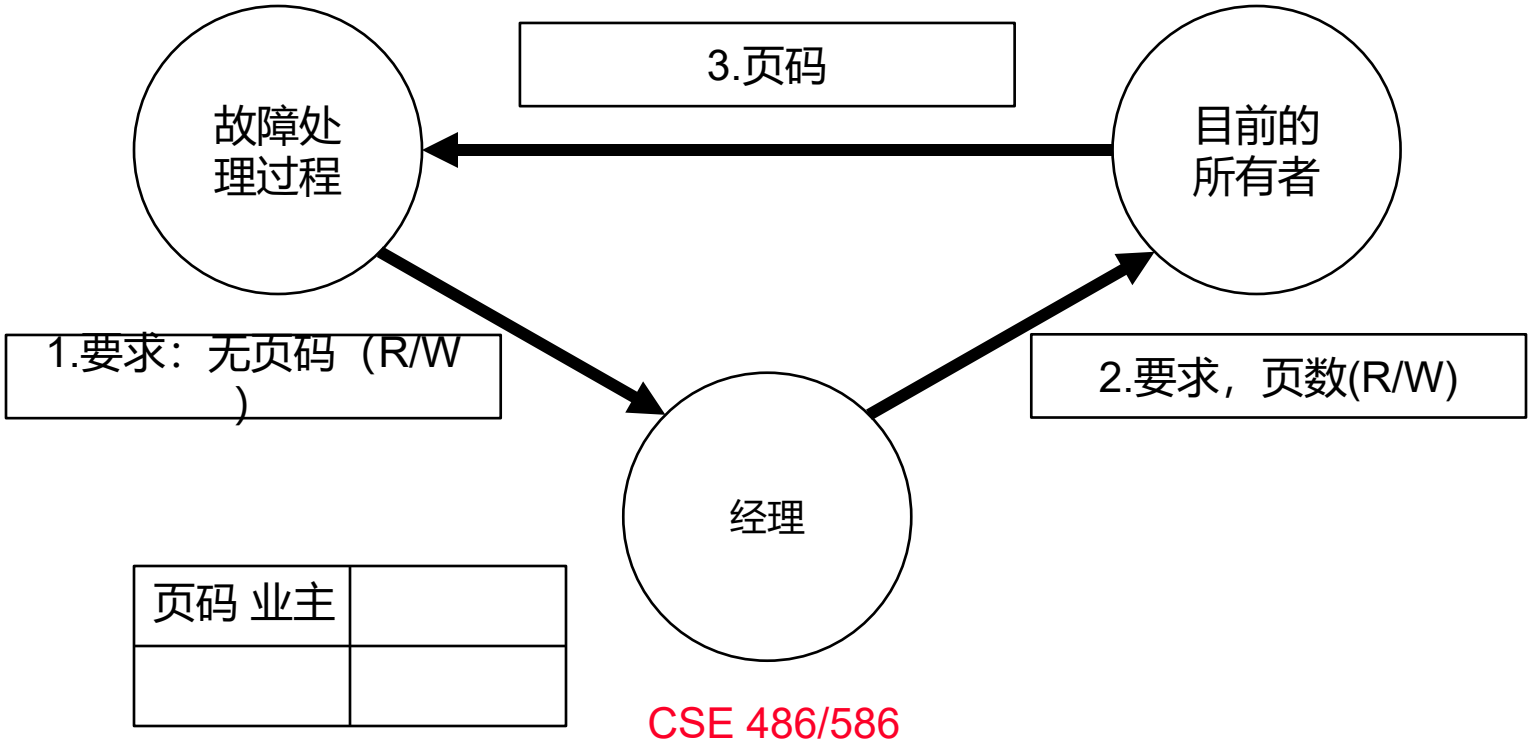
写入无效协议示例

- 注:R故障和W故障可以在任何过程中发生。



示例系统。常春藤

- 实施写入验证协议
  - 一个页面的所有者：拥有最新信息的过程
  - 一个页面的副本集：有副本的进程
  - 一个集中的管理者维护所有权信息。

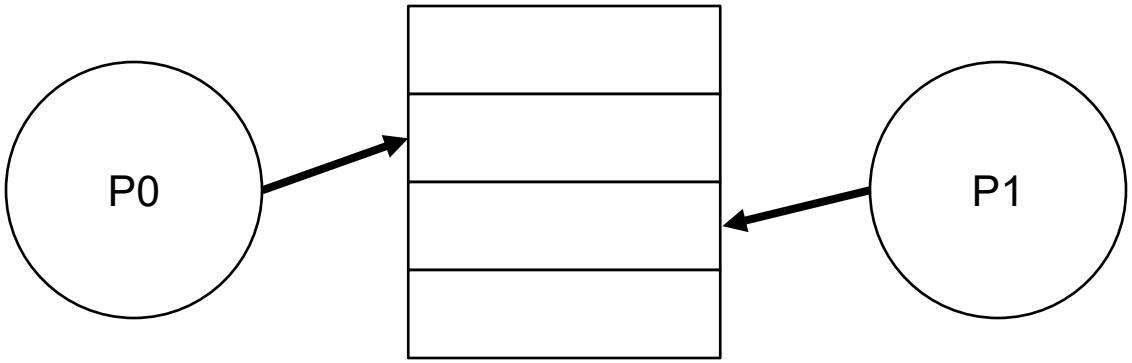


颗粒度问题

- 让我们假设，我们在页面层面上进行操作。
  - (但其他实现方式也有类似的问题)。
  - 作为参考，一个Linux内存页是4KB。
- 问题
  - 当两个进程(在两个不同的机器上)共享一个页面时，并不总是意味着它们共享页面上的一切。
  - 例如，一个进程从一个变量X中读出并写入，而另一个进程从另一个变量Y中读出并写入，如果它们在同一个内存页中，那么这些进程就共享这个页。

颗粒度问题

- 真正的分享
  - 两个进程共享完全相同的数据。
- 虚假分享
  - 两个进程并不共享完全相同的数据，但它们从同一个页面访问不同的数据。



- 虚假的分享问题
  - 写作无效：不必要的无效行为
  - 编写更新：不必要的数据传输

颗粒度问题

- 更大的页面尺寸
  - 更好地处理大量数据的更新(好)。
  - 由于要处理的单位/页面数量较少，管理开销较少(好)。
  - 虚假共享的可能性更大(坏)。
- 较小的页面尺寸
  - 与上述情况相反
  - 如果有大量数据的更新，它将被分解成许多小的更新，这导致更多的网络开销(坏)。
  - 较小的页面大小意味着更多的页面，这导致了更多的管理开销，即更多的读和写的跟踪(坏)。
  - 虚假共享的可能性较小(好)。

惊涛骇浪

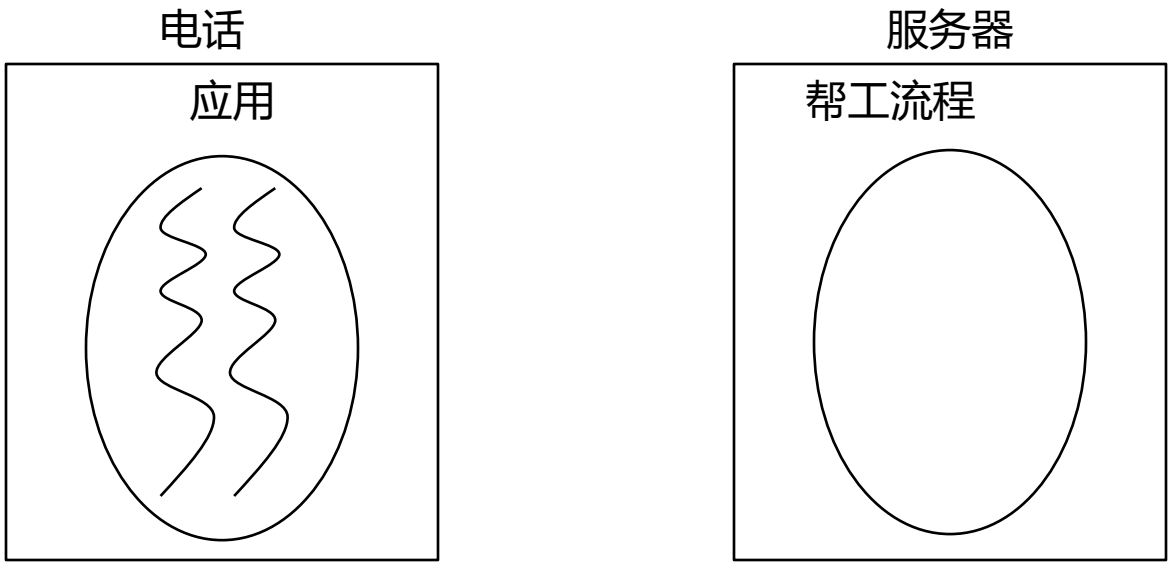
- 写入验证协议可能会发生抖动。
- *据说，当DSM花费了大量的时间来无效化和传输共享数据，而应用程序进程则花费了大量的时间来进行有用的工作时，就会发生Thrashing。*
- 当几个进程争夺一个数据项或虚假的共享数据项时，就会发生这种情况。

惊涛骇浪

- 常见情况：生产者-消费者模式
  - 数据由一个过程产生，由另一个过程使用。
  - 生产者将不断使消费者失效，而消费者将不断从生产者那里传输数据。
  - 写作日期对这种模式来说是比较好的。
- 鞭打的解决方案
  - 人工避免：程序员要避免惊动模式。
  - Timeslicing：一旦一个进程获得了对一个页面的写入权限，它就会在一段时间内保留它。其他进程的读/写请求在这段时间内被缓冲。

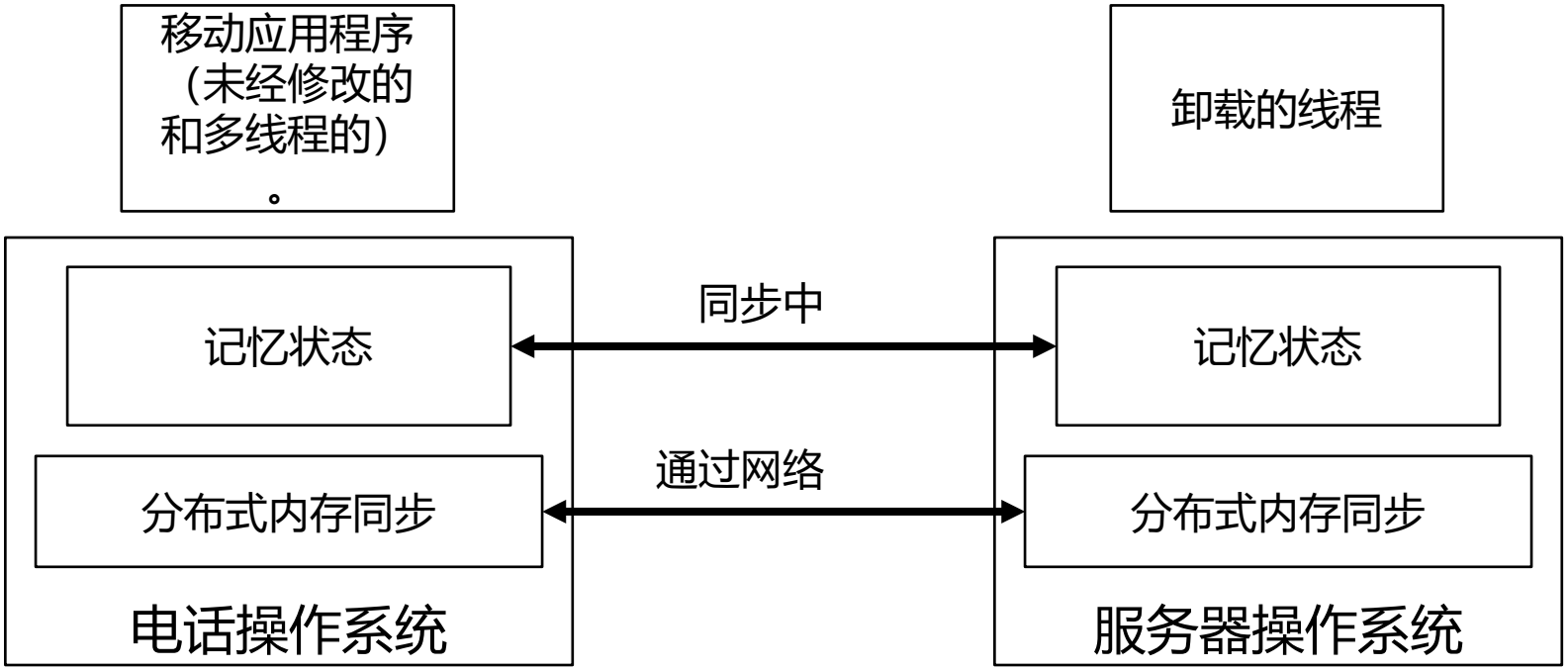
螺纹的DSM

- 不同机器上的线程之间共享内存。
- 用例：从智能手机到服务器的代码（线程）卸载
  - 低功耗的智能手机由高功率的服务器增强（计算和能源）。
  - 在某种意义上，它已经完成了（云后台），但DSM允许它不需要任何程序员的努力。



例子。彗星*

- 彗星允许在Java中对Android应用程序进行线程卸载
- Comet同步整个Java VM状态。



*https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gordon

Java代码执行的背景

- 内存：程序代码、堆栈、堆和CPU状态
- 堆栈和堆
  - 一般来说，程序栈处理静态分配的对象和方法调用的返回地址。
  - 堆是用于动态分配对象的。

```
public class Ex {
   public void method() {
            int i = 0; // 堆栈
            HashMap hm = new HashMap(); // heap
   }
}
```

- CPU状态
  - Android Java VM使用寄存器来执行指令。
  - 程序计数器（PC）指向下一条要执行的指令。
- 对于程序的执行，Java VM有一个执行循环。
  - 取出PC指向的下一条指令。
  - 执行新的指令
  - 在执行过程中，它使用寄存器、堆栈和堆。

彗星线迁移

- 彗星完全同步了两边的虚拟机(电话和服务器)。
  - 在Java中，程序执行所需的一切都存储在内存中。
  - 程序代码、堆栈、堆和CPU状态
  - DSM可以将这些同步化。
- 任何一方都可以执行一个线程，因为他们都知道程序执行所需的一切。
  - PC是同步的，所以双方都知道要执行的下一条指令。
  - 寄存器是同步的，所以它们都知道CPU的状态。
  - 堆栈和堆是同步的，所以它们知道内存状态。

摘要

- 线程间的内存共享
  - 默认情况下，它们共享同一个地址空间
- 进程间的内存共享
  - 共享内存API和信号API
  - 虚拟-物理内存映射实现了这一点。
- 跨机器的内存共享
  - 撰写日期
  - 写入-验证
- 不同机器上的线程共享内存
  - 用例：代码卸载

鸣谢

- 这些幻灯片包含由Indranil Gupta（UIUC）开发的材料，并拥有版权。