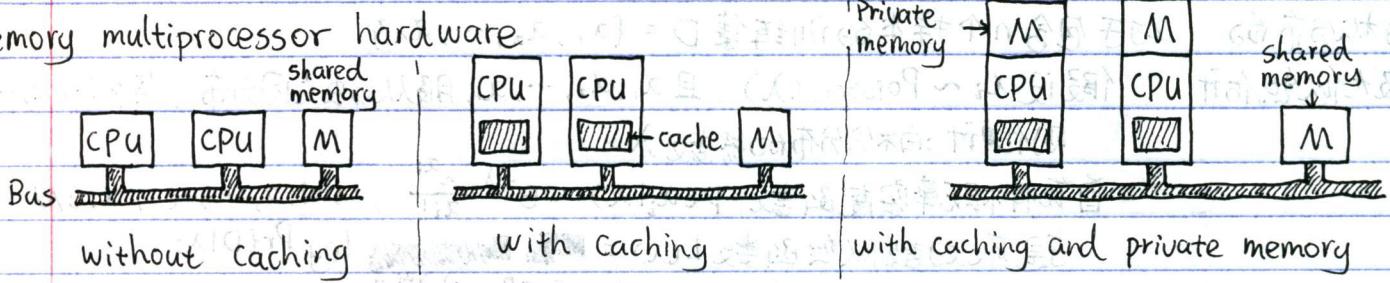


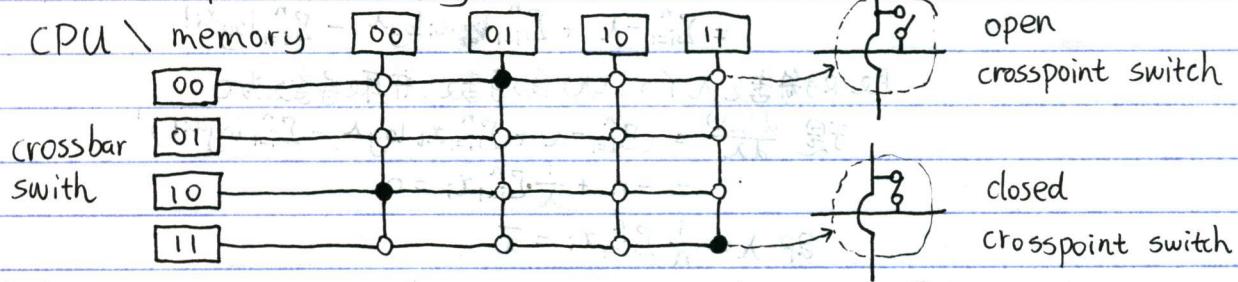
Operating System - P123

shared memory multiprocessor hardware



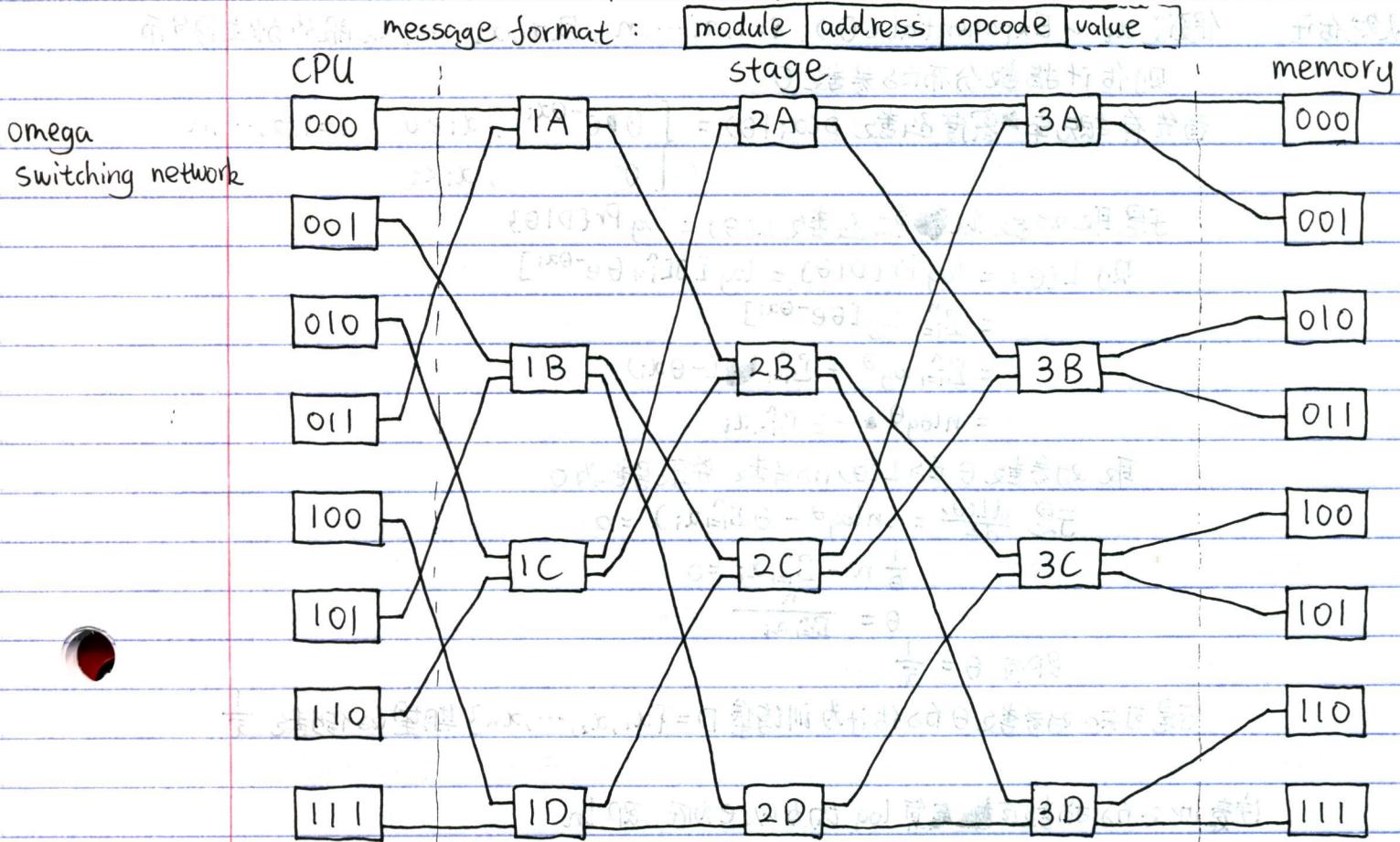
UMA (uniform memory access, 均匀访存模型), 又称统一寻址技术/统一内存存取架构

■ UMA multiprocessor using crossbar switch



UMA multiprocessor using multistage switching network

switch : input line → output line



Operating

System - P124

129 - secondary

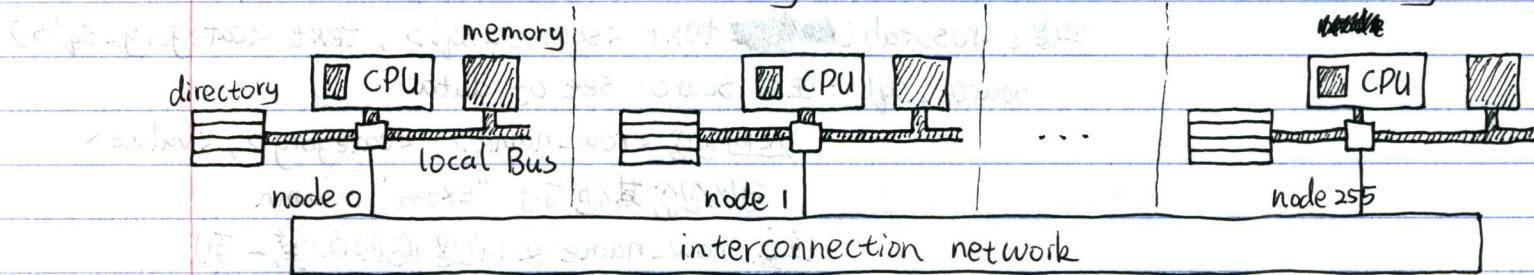
NUMA (non-uniform memory access, 非统一内存访问架构)

characteristic of NUMA multiprocessor machine

single address space visible to all CPU

access to remote memory via LOAD / STORE instruction

access to remote memory slower than access to local memory



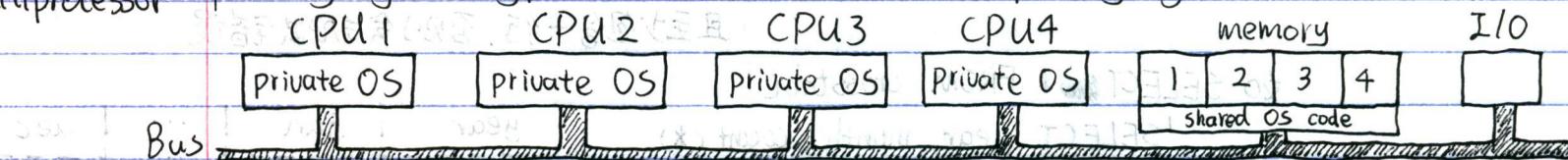
256-node directory-based multiprocessor

with direct memory access

such as subdivision of 32-bit memory address : 31 24|23 6|5 0

node block offset

multiprocessor operating system type, each CPU with own operating system

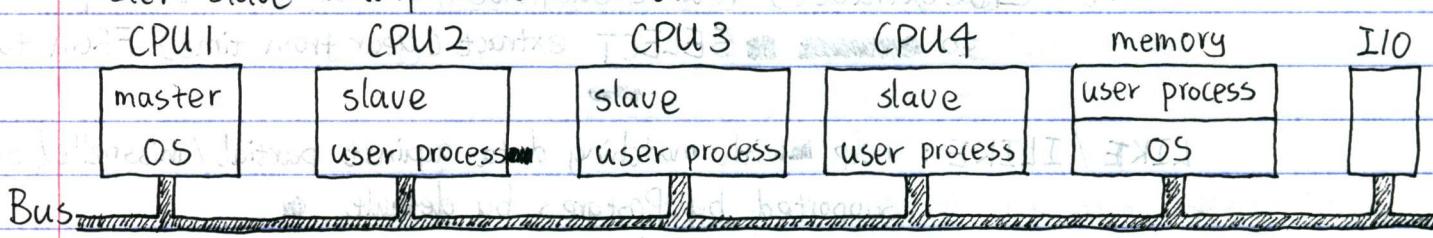


partitioning multiprocessor memory among each CPU

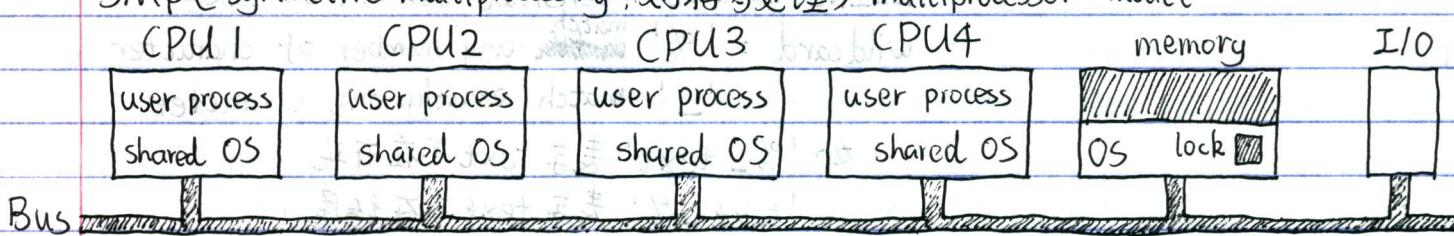
contain operating system private data for each CPU

sharing single copy of operating system code

master-slave multiprocessor model



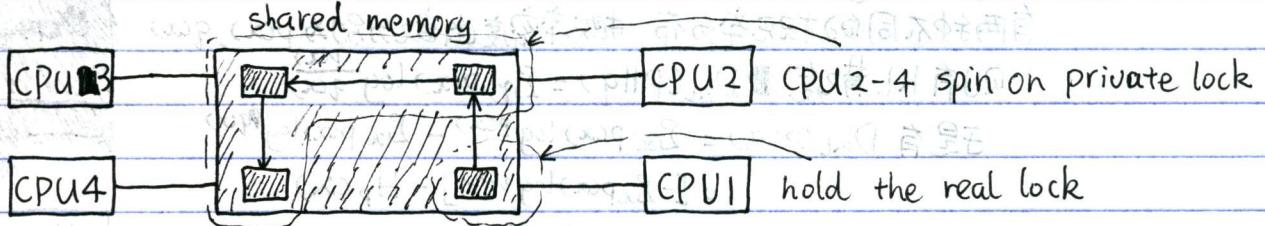
SMP symmetric multiprocessing (对称多处理) multiprocessor model



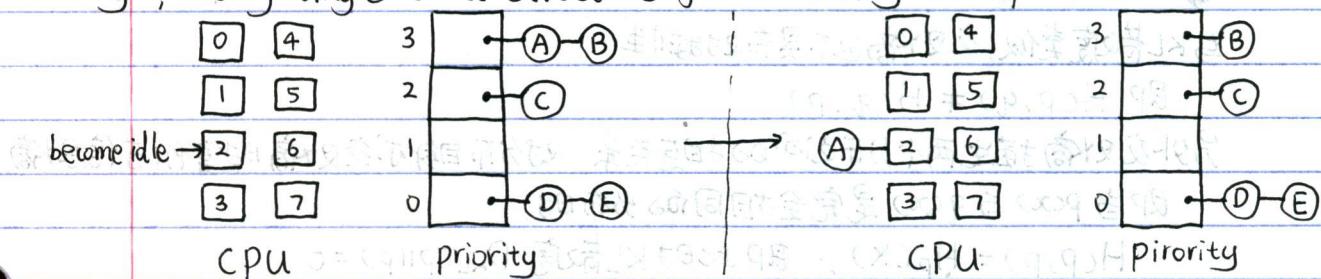
Operating

System - P125

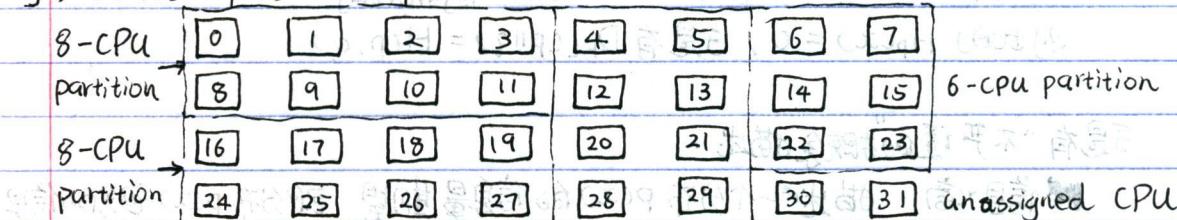
multiprocessor synchronization : using multiple locks to avoid cache trashing



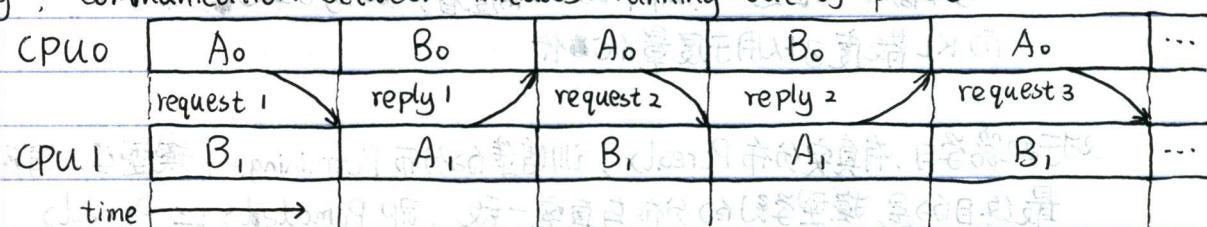
time sharing, using single data structure for scheduling multiprocessor



space sharing, CPUs split into partition 8-CPU partition



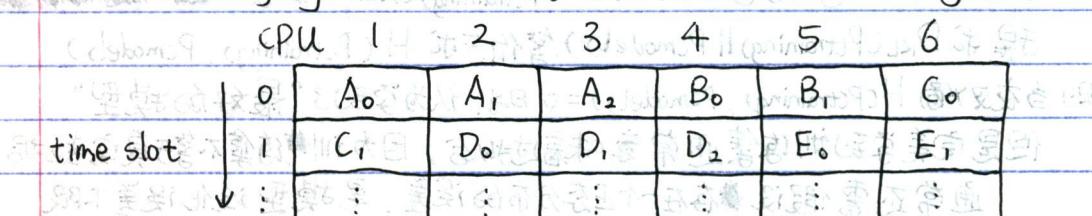
gang scheduling, communication between threads running out of phase



part of gang scheduling : group of related threads scheduled as one unit / gang

all member of gang run at once on different timeshared CPU

all gang member start and end time slice together



Operating System - P126

39 - Sandeep

milestone in parallel computing and supercomputer

1961 IBM 7030 "STRETCH", world's fastest computer 1961-1965

task parallelism : compute and I/O simultaneous

instruction pre-fetch and decode (4 instruction)

memory interleaving

500,000 IPS (instruction per second)

256K 64-bit word RAM (2MB)

predecessor to IBM System/360

FORTRAN and assembly language

1965 CDC 6600, world's fastest computer 1964-1969

successor CDC 7600 fastest 1969-1976

1 CPU (later CDC 6500 / 6700 have 2 CPUs)

sometimes considered the first RISC computer

very simple instruction and large register set

instruction cache

multiple, parallel functional unit (superscalar)

multiple instructions executed simultaneously

"Scoreboard" unit coordinated result/operand dependency

hardware memory protection

base address and length, RA/FL

CPU had no input/output capability

all I/O done by PPU (peripheral processing unit)

peak speed ~3M FLOPS (floating-point operation per second)

128K 60-bit word RAM (< 1MB)

10 PPUs : 4K 12-bit word each private memory

bulk transfer of data to/from CPU memory

shared time-sliced processor hardware

separate register set for each PPU ("Barrel")

no hardware synchronization

PPU 0 : master PPU, provide synchronization and other task in software

PPUs perform all I/O

OS ("Scope") run originally just in the PPUs

later extension use CPU

Operating

System - P127

notes/A

P9 - parallel

1971 ILLIAC-IV, early successful massively parallel computer design

one-off research project

design for DARPA (defense advanced research projects agency, 国防高级研究计划局)

256 processors SIMD (single instruction multiple data, 单指令流多数据流) design

only 64 processors built

designed for 1 GFLOPS, achieved 200 MFLOPS

pioneered parallel construct in programming language (IVTRAN)

1976 CRAY-1, single CPU

vector parallelism

80 MFLOPS

succeeded by multiprocessor version

1982 : CRAY X-MP (800 MFLOPS)

1988 : CRAY Y-MP (~2.5 GFLOPS)

1986 connection machine CM-1 (Thinking Machines Inc.)

64 K bit-serial SIMD processors

each processor has 4K bits of RAM

16-dimensional hypercube interconnect

successor : CM-2 / CM-2a / CM-200

2004 IBM Blue Gene/L

MIMD (multiple instruction multiple data, 多指令流多数据流) message-passing architecture

64K dual core nodes

each node contains: Power PC 440 dual-core processor

each core has a dual 2.8 GFLOPS FPU

512MB RAM

6 bidirectional ports to 3D torus interconnect

3 bidirectional ports to collective network

4 ports to barrier/interrupt network

200+ TFLOPS

Linux OS

successor : Blue Gene/P (1 PFLOPS), Blue Gene/Q (10 PFLOPS)

Operating System - P128

IBM Blue Gene Architecture

compute chip : 2 processors (2.8 GF/s / 5.6 GF/s)

4 MB eDRAM

compute card : FRU (field replaceable unit, 现场可更换单元)

(I/O card) 2 nodes (4 CPUs) (2x1x1) (2x2.8/5.6 GF/s)

2 x 512 MB DDR

node card : 16 compute cards, 0~2 I/O cards

32 nodes (64 CPUs) (4x4x2) (90/180 GF/s)

16 GB DDR

cabinet : 2 midplanes

1024 nodes (2048 CPUs) (8x8x16) (2.9/5.7 TF/s)

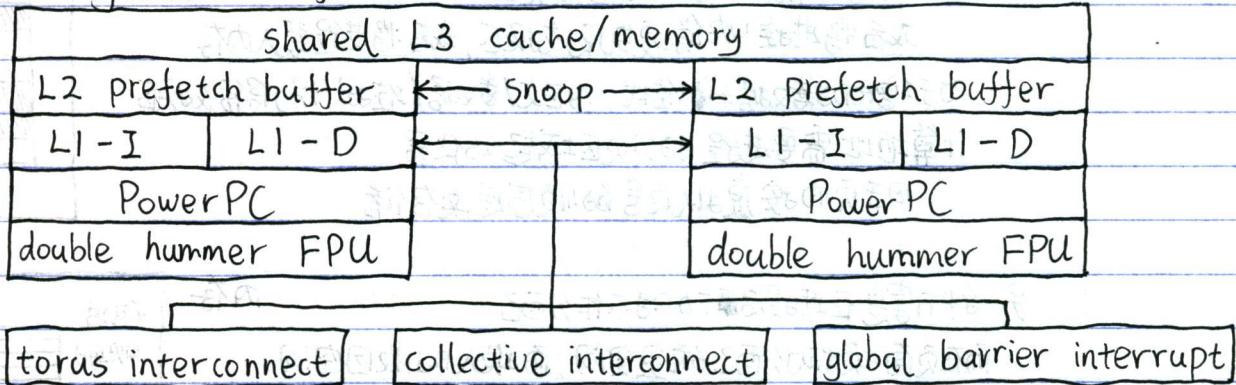
512 GB DDR

system : 64 cabinets

65536 nodes (131072 CPUs) (32x32x64) (180/360 TF/s)

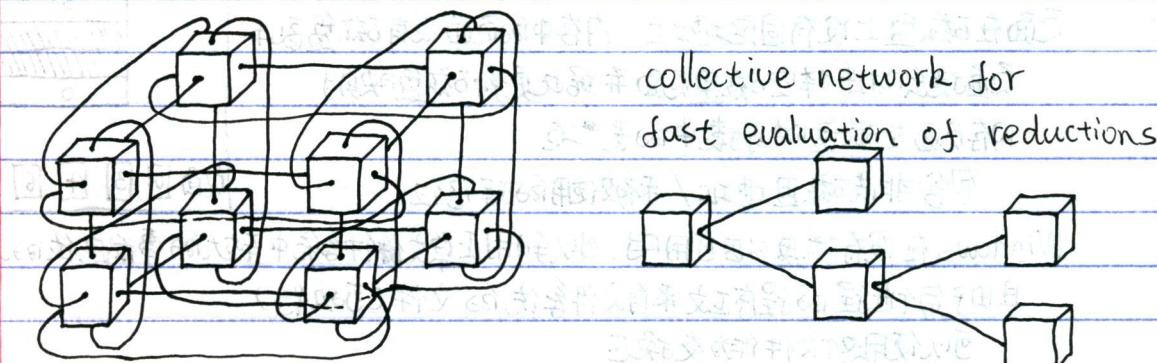
32 TB

logical organization of Blue Gene/L node



Blue Gene/L communication network

3D torus for standard inter-processor data transfer



Operating

System - P129

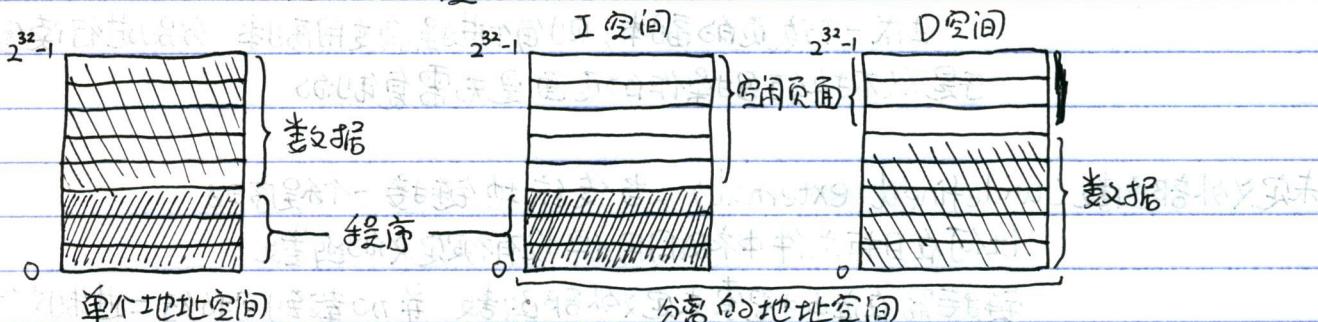
分离的地址空间，在 PDP-11 (16-bit) 上的用于地址空间不足以存放数据和程序的解决方案

W 空间：用于指令（程序正文）的地址空间

D 空间：用于数据的地址空间

另外，链接器 (linker) 必须知道何时使用分离的 W 空间和 D 空间

由于使用分离的地址空间时，数据不再是在程序之后开始，而是被重定位到虚拟地址 0



注意，两种地址空间都可以进行分页，且相互独立

分别有各自的页表，并分别完成虚拟页面到物理页框的映射

除了指令操作 / 数据访问 分别通过 W 空间 / 页表 / D 空间，页表不会引入其他任何复杂的设计

共享页面

用于在大型多道程序设计系统中，避免在内存中有一个页面的多个备份

仅有只读页面（程序正文）可以共享，数据页面不能共享

进程可以使用相同的 W 空间页表和不同的 D 空间页表来实现

在典型的指令共享的实现中，页表与进程表数据结构无关

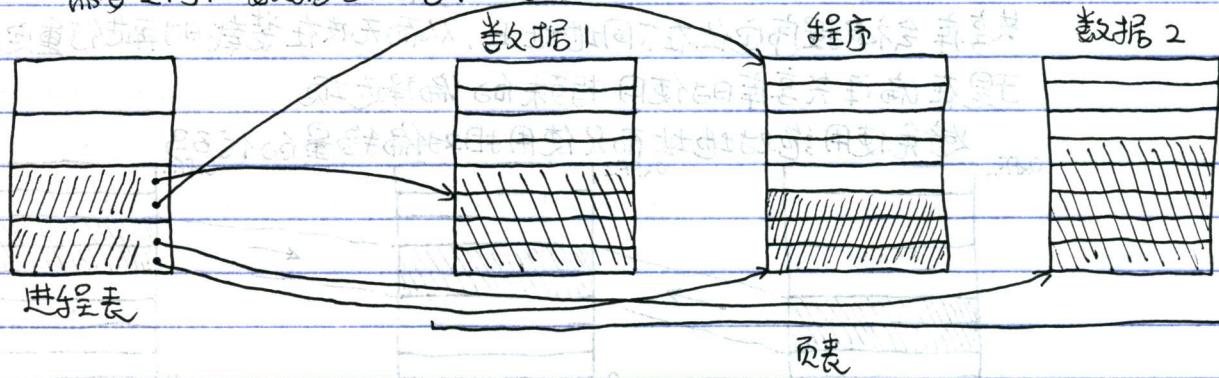
进程的进程表中有分别指向 W 空间页表和 D 空间页表的指针

当调度程序选择进程运行时，使用这些指针定位合适的页表，并设立 MMU

重要的是在调度时发现共享页面仍被使用，以防止释放后产生页错误

查找所有页表以确定某个页面是否共享的代价太大

需要专门的数据结构记录共享页面



Operating

System - P130

写时复制 (copy-on-write), 在共享数据时减少复制而提高性能

在UNIX中, 系统调用 fork 后 父进程与子进程共享程序文本和数据
进程分别拥有自己的页表, 但指向同一个页面集合

于是 fork 时不需要进行页面复制

所有映射到两个进程的数据页面都是只读的

而当进程更改数据时, 触发只读保护, 并引发操作系统的陷阱

生成一个该页的副本, 则每个进程有专用副本, 分别进行读写

于是从不执行写操作的页面是无需复制的

未定义外部函数 (undefined external), 当传统地链接一个程序时

任何在目标文件中被调用但没有被定义的函数

链接器在库中搜索未定义外部函数, 并加载到可执行二进制文件中

任何被未定义外部函数调用但不存在的函数也是未定义外部函数

在库中定义但是未被调用的函数不会被加载

静态链接库的程序会浪费大量磁盘空间, 装载也会浪费大量内存空间

动态链接库 (dynamic-link library, DLL), Windows 中的共享库技术

当程序与共享库链接时, 链接器不会加载被调用的函数

而是加载一段能够在运行时绑定被调用函数的存根例程 (stub routine)

共享库或者与链接的程序一起被装载

或者在其所包含的函数首次调用时被装载

如果已被装载, 则不必再次装载

并非一次性装载, 而是根据需要以页面为单位装载

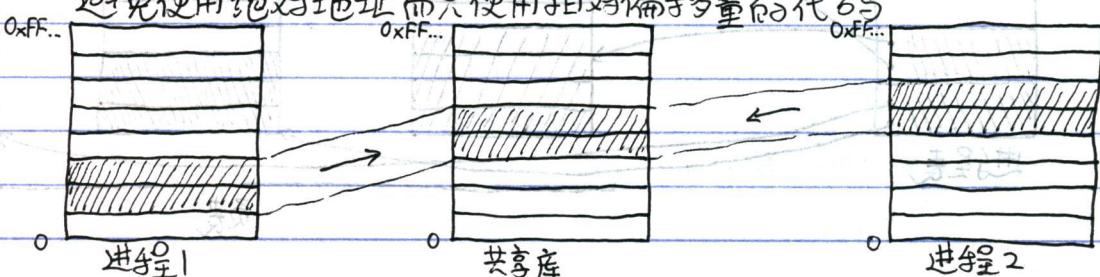
如果共享库更新, 并不需要重新编译调用库中函数的程序, 而只需装载新的共享库

位置无关代码 (position-independent code), 用于解决共享库的地址定位问题

共享库会被程序定位在不同地址上, 从而无法在装载时再进行重定位

于是在编译共享库时使用特殊的编译选项

避免使用绝对地址而只使用相对偏移量的代码



Operating System - P131

内存映射文件(memory-mapped file), 实际上更通用的机制, 一个特例是共享库

进程通过发起系统调用, 将文件映射到虚拟地址空间的一部分

多数实现中, 映射共享页面时不会实际读入页面内容

而是在访问页面时才会被每次一页地读入

磁盘文件则作为后备存储

进程退出或显式地解除文件映射时, 所有被改动的页面写回磁盘文件

提供 I/O 可选模型: 把文件当作内存中的大字符串数组进行访问

而非通过常规的读写操作来访问文件

当多个进程同时映射同一个文件, 则可以通过共享内存进行通信

即一个进程在映射文件的虚拟地址空间上执行的写操作

可以即刻被另一个进程在虚拟地址空间上的读操作看到

提供了一个进程间的高带宽通道

可以扩展到用于映射无名的临时文件

分页守护进程(paging daemon), 用于保证有足够的空闲页框的后台进程

分页系统最佳工作状态: 缺页中断发生时系统中有大量空闲页框

分页守护进程通常处于睡眠状态, 定期唤醒以检查内存状态

如果分页守护进程激活时空闲页框过少

则通过预定的页面置换算法选择页面换出内存

如果页面装入内存后被修改过, 则写回磁盘

注意在任何情况下, 页面中原先的内容都保留下来

当需要使用的是一个已被淘汰的页面

如果所在的页框尚未被覆盖, 则可以从空闲页框缓冲池移出即可恢复页面

保存一定数目的页框供给相比使用所有页框而在需要时搜索可用页框的性能更好

保证了所有空闲页框是“干净”的, 即进行页面置换时无需将页框写回磁盘

双指针时钟

前指针: 分页守护进程

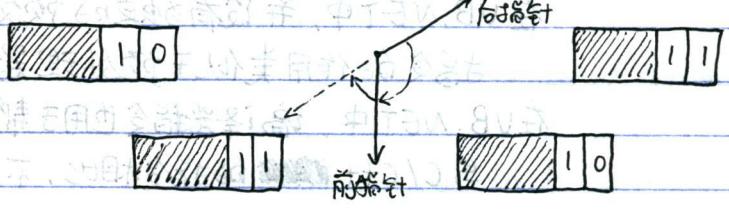
指向脏页面时,



则将页面写回磁盘

后指针: 页面置换

命中干净页面的概率增加



Operating System - P132

虚拟内存接口 (Virtual memory interface)，一些高级系统允许程序员对内存映射进行控制

于是可以通过非常规的方法来增强程序的行为

一个原因是 ~~为了~~ 为了允许两个或多个进程共享一部分内存

程序员可以通过对内存区域命名实现共享内存

通过使一个进程将一片内存区域的名称通知另一个进程

从而第二个进程可以将这片内存区域映射到其自身的虚拟地址空间

通过多个进程共享同一部分页面，从而实现高带宽的共享

进程可以通过控制页面映射来实现高性能的消息传递系统

发送进程清除包含消息的页面的映射，并复制页面名字

接收进程通过页面名字将页面映射进来，从而避免复制所有数据

分布式共享内存 (distributed shared memory)

允许网络上的多个进程共享一个页面集合，页面可能但不必要作为单个的线程共享地址空间

缺页中断产生时，内核空间/用户空间中的缺页中断处理程序对拥有页的机器定位

向该机器发送一条消息，请求其清除该页面的映射并通过网络发送

当页面抵达时，将页面映射进来并重新开始运行引起缺页中断的指令

分页相关工作

操作系统分别在不同阶段进行与分页 (paging) 相关的工作

进程创建：确定程序和数据的初始大小，并为其创建一个页表

在内存中为页表分配空间并初始化

进程换出时页表不需要驻留在内存，但进程运行时必须存在内存中

在磁盘区中分配空间，用于进程换出时放置于磁盘交换区

用程序正文和数据对交换区初始化，以便缺页中断时调入页面

一些系统直接从磁盘上的可执行文件对程序正文进行分页，以节省初始化时间

必须将有关页表和磁盘交换区的信息存储在进程表中

调度进程：必须为新进程重置 MMU，刷新 TLB，以清除以前进程遗留的痕迹

通过复制页表或将指向其的指针放入硬件寄存器使新进程页表成为当前页表

进程初始化时可以将进程全部/部分 (如 PC 指向) 的页面装入内存以减少缺页中断

缺页中断：通过读取硬件寄存器确定造成缺页中断的虚拟地址

通过虚拟地址计算需要的页面，并在磁盘上定位页面

找到合适框存放新页面，必要时置换旧页面

回退程序计数器以指向引起缺页中断的指令，并重新执行该指令

进程退出：必须释放进程的页表，页面和页面在硬盘上占用的空间

对于共享页面则必须等待最后一个使用的进程终止才能释放

Operating System - P133

- 缺页中断处理
- 硬件陷入内核，并在堆栈中保存程序计数器
大多数系统将当前指令的各种状态信息保存在特殊的CPU寄存器
 - 启动汇编代码例程保存通用寄存器和其他易失信息，以免被操作系统破坏
汇编代码例程将操作系统作为一个函数来调用
 - 当发现缺页中断时，操作系统尝试发现需要的虚拟页面
通常由一个硬件寄存器包含需要的虚拟页面信息
如果没有，则操作系统必须检索程序计数器并取出指令
并用软件分析指令在缺页中断时的行为
 - 当获取发生缺页中断的虚拟地址，操作系统检查地址是否有效，并检查存取保护是否一致
如果不一致，则向引发缺页中断进程发出信号或杀掉该进程
如果地址有效且没有发生保护错误，操作系统则检查是否有空页框
如果没有空闲页框，则执行页面置换算法寻找一个页面淘汰
 - 如果选择的是“脏”页框，则安排该页面写回磁盘
并发生上下文切换，挂起引起缺页中断的进程，运行其他进程直至磁盘传输结束
将该页框标记为忙，以免由于其他原因而被其他进程占用
 - 当页框状态为“干净”，操作系统查找所需页面在磁盘的地址，并通过磁盘操作装入
当页面装入时，依旧挂起引起缺页中断的进程，并运行其他进程
 - 当磁盘中断发生时，表明所需页面已经装入
此时页表已经更新并可以反映该页面的位置，页框也标记为正常状态
 - 恢复至发生缺页中断的指令以前的状态，程序计数器重新指向该指令
 - 调度引发缺页中断的进程，操作系统返回调用它的汇编代码例程
 - 汇编代码例程恢复寄存器和其他状态信息。
并返回用户空间继续执行，如同未发生过缺页中断

指令备份

在缺页中断处理完成后，OS需要知道该指令的首个字节的位置

但程序计数器的值依赖于引起缺页中断的操作数以及CPU微指令的实现方式

而准确判断指令起始位置对于OS通常是不可能的

特别是对于680x0体系结构，指令的副作用会增加/减少一个/多个寄存器

自动增量/减量可能在内存访问之前/之后完成

且随着指令和CPU模式的不同而不同

一种解决方案是利用隐藏的内部寄存器，在每条指令执行之前将程序计数器复制到寄存器

并有另一个寄存器记录寄存器已自动增加/减少以及增减的数量信息

MOVE.L #6(A1), 2(A0)	0x1000	2
	680x0: 0x99C	6
	0x998 MOVE	

← 32-bit →