整数编码

$\sim x+1 == -x$

$\sim x+x == -1$

补码 $|T_{Min}| = T_{Max}+1$

无符号 $U_{Max} = 2*T_{Max}+1$

有无符号混用时有符号数转换为无符号数.

$$TAdd(x,y)=\begin{cases} x+y-2^w, & T_{Max_w} < x+y \quad \text{正溢} \\ x+y, & T_{Min_w} \le x+y \le T_{Max_w} \quad \text{正常} \\ x+y+2^w, & x+y < T_{Min_w} \quad \text{负溢} \end{cases}$$

无符号数乘积最多2w位, [保留低w位]

补码最小值最多需要2w-1位,

最大值最多需要2w位。

负数除以2的整数幂的商:

就计算 $\lceil x/2^k \rceil$ (向0舍入),

按 $\lfloor (x+2^k-1)/2^k \rfloor$ 计算.

(表达式: (x+(1<<k)-1)>>k

·巧用无符号数向下计数

for(i=cnt-2; i<cnt; i--)

·阿贝尔群

封闭性 交换性 结合性

单位元 每个元素都有逆元

程序的机器级表示

·机器级编程的两种抽象: ①由指令集体系结构或指令集架构来定义机器级程序的格式和行为

②机器级程序使用的内存地址是虚拟地址.

·程序员可视的状态: ①程序计数器 ②内存 ③寄存器文件 ④条件码

·汇编指令后缀: b w l q (1, 2, 4, 8)

·6个参数寄存器: rdi rsi rdx rcx r8 r9

edi esi edx ecx r8d r9d

di si dx cx r8w r9w

dil sil dl cl r8b r9b

rax 返回值 rsp 栈指针

rbx, rbp, r12~r15 被调用者保存

r10, r11 调用者保存

·movabsq I, R (R←I传送绝对的四字)

·cltq 把%eax符号扩展到%rax

·movzxx 零扩展传送

·movsxx 符号扩展传送

·特殊算术操作: imulq S R[%rdx]:R[%rax]← S×R[%rax] 有符号全乘法

mulq S ~ 无符号全乘法 cqto ~:~←符号扩展(R[%rax])

idivq S R[%rdx]← ~:~ mod S R[%rax]← ~:~ ÷S 有符号除法

divq S 同理,无符号除法

·条件码相关:

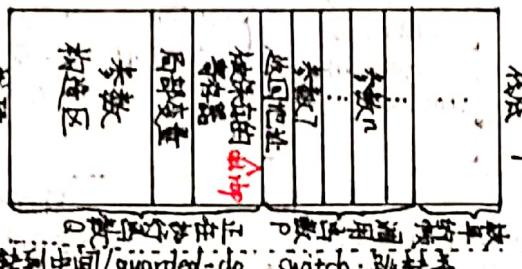
e: ZF ne: ~ZF a: ~CF&~ZF

s: SF ns: ~SF ae: ~CF

g: ~(SF^OF)&~ZF b: CF

ge: ~(SF^OF) be: CF|ZF

·for循环: loop while循环: 跳转到中间/guarded-do switch: 跳转表



①十进制转化为二进制 ②表示为科学记数法 ③指数→阶码 ④符号位 ⑤尾码 ⑥内存中倒序

(除2取整法) 化导为1

浮点表示

	s	exp	frac
单精度:	1	8	23
双精度:	1	11	52
扩展精度:	1	15	63/64

单精分类:

①规格化: 阶码≠0&&≠255

②非~: — 为0

③无穷大: 阶码255 尾数0

④NaN: 阶码255 尾数≠0

规格化数: E=exp-bias

非规格化数: E=1-bias

bias = $2^{k-1}-1$ [偏置值编码]

·向偶数舍入: "偶数": 最后有效位值为0

"中间值": 舍入位置右侧的位都是0

·浮点加法: 二进制小数点对齐.

尾数 M≥2: M右移1位, E加1

M<1: M左移k位, E减k

E超范围溢出

M舍入以符合精度

局部性原理

程序倾向于使用与最近使用过的数据的地址接近或相同的指令和数据,包括时间局部性和空间局部性.

·空间局部性: 1.对数据的引用: 顺序访问数组元素 (步长为1的引用模式)

2.对指令的引用: 顺序读取指令

·时间局部性: 1.对数据的引用: 变量~

2.对指令的引用: 重复循环执行for循环体…

专注在核心函数和内循环

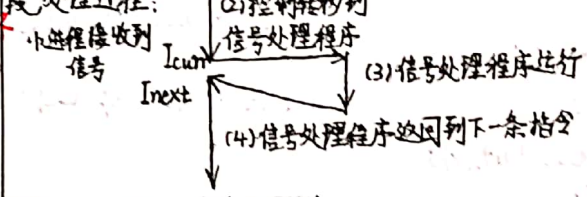
反复引用变量

步长为1的参考模式

信号处理

Linux通过软中断(陷阱)方式实现信号机制,包括信号发送和信号接收两个阶段.

处理过程:



小进程接收到信号

信号处理程序的编写原则:

①处理程序尽可能简单

②确保其他处理程序不会覆盖当前的errno

③只调用异步信号安全的函数

④阻塞所有信号保护对共享全局数据结构的访问

⑤用volatile声明全局变量

⑥用sig_atomic_t声明标志

程序优化的方法

①代码移动: 通过将代码从循环中移出减少计算执行的频率

②简单计算替代复杂计算,移位、加法替代乘法等.

③共享共用子表达式、重用表达式的一部分

④使用局部变量作为累积量

⑤循环体展开减少循环次数

⑥消除了不必要的内存引用

⑦多个累积量、重新结合以提高指令级并行性

⑧尽量缩短关键路径

⑨减少过程调用

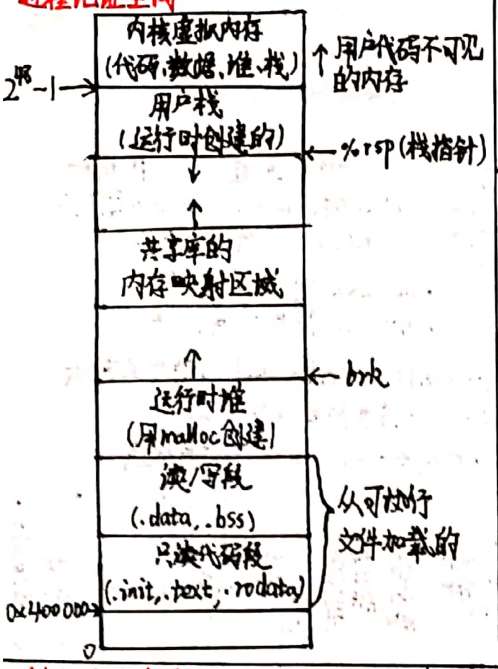
进程上下文 ①保存 ②恢复 ③控制传递

内核重新启动一个被抢占的进程所需的状态,包括通用目的寄存器、浮点寄存器、程序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构,如页表、进程表和文件表.

顺序架构指令(非控制转移)(cmov指令)

阶段	OPq rA, rB	rrmovq rA, rB	irmovq V, rB	rmmovq rA, D(rB)	mrmovq D(rB), rA	pushq rA	popq rA
取指	icode: ifun←M1[PC] rA: rB←M1[PC+1] valP←PC+2	~←M1[PC] ~←M1[PC+1] ~←PC+2	~←M1[PC] ~←M1[PC+1] valC←M8[PC+2] valP←PC+10	~ ~ ~ ~ 同左	同左	icode: ifun←M1[PC] rA: rB←M1[PC+1] valP←PC+2	同左
译码	valA←R[rA] valB←R[rB]	valA←R[rA]		valA←R[rA] valB←R[rB]	valB←R[rB]	valA←R[rA] valB←R[%rsp]	valA←R[%rsp] valB←R[%rsp]
执行	valE←valB OP valA Set CC	valE←0+valA	valE←0+valC	valE←valB+valC	valE←valB+valC	valE←valB+(-8)	valE←valB+8
访存				M8[valE]←valA	valM←M8[valE]	M8[valE]←valA	valM←M8[valA]
写回	R[rB]←valE	R[rB]←valE	R[rB]←valE		R[rA]←valM	R[%rsp]←valE	R[%rsp]←valE R[rA]←valM
更新PC	PC←valP	PC←valP	PC←valP	PC←valP	PC←valP	PC←valP	PC←valP

进程地址空间



异常的类别

中断	来自I/O设备的信号	异步	总是下一条
陷阱	有意的异常	同步	总是下一条
故障	潜在可恢复的错误	同步	可能当前
终止	不可恢复的错误	同步	不会返回

HCL

```
word Min3= [
    A<=B && A<=C : A;
    B<=C         : B;
    1            : C;
];
```

```
bool s1 = code in {2,3};
word srcA = [
    icode in {IRRMOVQ, ...}: rA;
    icode in {IPOPQ, ~}: RRSP;
    1: RNONE;
];
```

处理器体系结构

时钟寄存器：单个位或字(程序计数器PC、条件代码CC和程序状态Stat)

随机访问寄存器：多个字(内存、寄存器文件)

·流水线：流水线化的系统，待执行的任务被划分成若干个独立的阶段，将处理器的硬件也组织成若干个单元，让各个独立的任务阶段在不同的硬件单元上一次执行，从而使多个任务并行操作。如x86-64将指令执行分为取指、译码、执行、访存、写回五个阶段，通过在每个阶段插入流水线寄存器，利用时钟信号控制流水线的时序和操作，理想情况下可实现5条指令的同时运行。

·流水线的局限性：①不一致的划分：运行时钟速率是由最慢的阶段的延迟限制的.

②流水线过深，收益反而下降：寄存器更新延迟所占比例

·流水线冒险：①数据相关 ②控制相关

例：#prog1

```
irmovq $10, %rdx
irmovq $3, %rax
nop
nop
nop
addq %rdx, %rax
halt
```

3×nop 无数据冒险

2×nop 与rax错误

1×nop 两个内错

0×nop 两个内错

数据冒险

指令使用寄存器R为目的，暂时之后使用寄存器R为源，处理方法有暂停：通过在执行阶段插入气泡，使得当前指令执行暂停在译码阶段；数据转发：增加valM/valE的旁路路径，直接送到译码阶段；加载/使用数据冒险：指令暂停在取指和译码阶段，在执行阶段插入气泡.

控制冒险

分支预测错误：在条件为真的地址target处的两条指令分别插入1个bubble.

ret：在ret后插入3个bubble.

2×nop：valB ← W_valE = 3

1×nop：valA ← W_valE = 10

valB ← M_valE = 3

0×nop：valA ← M_valE = 10

valB ← e_valE = 3

优化程序性能

·妨碍优化的因素：

①内存别名使用 ②函数调用

例：循环展开优化 前置和函数：

```
p[0] = a[0];
for (i=1; i<n-1; i+=2)
{
    float mid = p[i-1] + a[i];
    p[i] = mid;
    p[i+1] = mid + a[i+1];
}
if (i<n)
    p[i] = p[i-1] + a[i];
```

优化示例：

```
long i;
long length = vec_length(v);   代码移动
long limit = length - 1;
data_t *data = get_vec_start(v);
data_t acc0 = IDENT;   累积变量
data_t acc1 = IDENT;
for (i=0; i<limit; i+=2)
{
    acc0 = acc0 OP data[i];     2×2循环
    acc1 = acc1 OP data[i+1];   展开
}
```

```
for(; i<length; i++)
{
    acc0 = acc0 OP data[i];
}
*dest = acc0 OP acc1;
```

消除内存引用.

灌满流水线，实现最高性能，$k \geq 10$.

·并行度超出可用寄存器数量 —— 寄存器溢出.

·分支预测和预测错误处罚

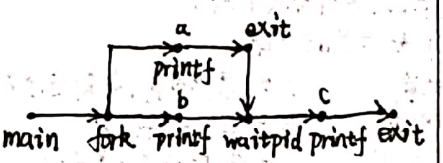
·书写适合用条件传送的代码

(使用 ? : 形式).

·加载单元的延迟(两个加载单元，k个值则CPE不低于k/2)

·写/读相关

进程图规范



针对CPU优化：

指令级并行——循环展开

针对Cache：

矩阵分块.

顺序实现指令(控制转物)

阶段	jxx Dest	call Dest	ret
取指	icode: ifun ← M1[PC] valC ← M8[PC+1] valP ← PC+9	同左	icode: ifun ← M1[PC] valP ← PC+1
译码		valB ← R[%rsp]	valA ← R[%rsp] valB ← R[%rsp]
执行	Cnd ← Cond(CC, ifun)	valE ← valB + (-8)	valE ← valB + 8
访存		M8[valE] ← valP	valM ← M8[valA]
写回		R[%rsp] ← valE	R[%rsp] ← valE
更新PC	PC ← Cnd ? valC : valP	PC ← valC	PC ← valM

cmovXX 指令(书上与课件中不一样)

```
cmovXX rA, rB
icode: ifun ← M1[PC]
rA : rB ← M1[PC+1]
~~valC~~
valP ← PC+2
valA ← R[rA]
valB ← 0          valB
valE ← 0 + valA
Cnd ← Cond(CC, ifun)   If !Cond(CC, ifun) rB ← 0xF
if (Cnd)               R[rB] ← valE
    R[rB] ← valE
PC ← valP
```

存储器层次结构

随机访问存储器：SRAM、DRAM

SRAM：静态、双稳态、6晶体管、1×访问时间、持续、不敏感、1000×花费、高速缓存存储器

DRAM：动态、1晶体管、10×访问时间、不持续、敏感、1×花费、主存，帧缓冲区

DRAM↔内存控制器：行地址 i RAS ——→复制到一个内部行缓冲区；列地址 j CAS ——→从缓冲区中复制出超单元。共享相同DRAM地址引脚。降低了引脚数量但必须分两步发送地址，增加了访问时间。

快页模式DRAM（行缓冲区直接获得）；扩展数据输出DRAM（各CAS信号时间上是紧）；同步DRAM（快）；双倍数据速率同步DRAM；视频RAM

- 总线：系统总线，内存总线
  CPU↔I/O桥接器　I/O桥接器↔主存
- 读事务：从主存传送数据到CPU
- 写事务：从CPU传送数据到主存

读事务：①CPU将地址放在内存总线上 ②主存从总线读出地址，取出字，将字放在总线上 ③CPU从总线读出字，并将它复制到寄存器中

写事务类似三步。

- 磁盘：由盘片构成，每个盘片包含两面，每面由一组磁道同心圆构成，每个磁道划分成一组扇区，扇区之间由一些间隙隔开。

容量由记录密度、磁道密度和面密度（①×②）决定。现代磁盘多区记录技术，每个区——一组连续柱面，每个柱面每个磁道都有相同数量的扇区，扇区数量由区最里面的磁道确定。

$$\text{磁盘容量}=\frac{\text{字节数}}{\text{扇区}}\times\frac{\text{平均扇区数}}{\text{磁道}}\times\frac{\text{磁道数}}{\text{表面}}\times\frac{\text{表面数}}{\text{盘片}}\times\frac{\text{盘片数}}{\text{磁盘}}$$

- 读写头冲撞
- 逻辑磁盘块　逻辑块号 —控制器固件，块选表查找→ （盘面，磁道，扇区）

"格式化容量"与"最大容量"

- 读磁盘
  ①CPU通过将命令、逻辑块号和目的内存地址写到与磁盘相关联的内存映射地址，发起一个磁盘读操作；
  ②磁盘控制器读扇区，并执行到主存的DMA传送
  ③完成后，用中断通知CPU
- DRAM和磁盘的性能与CPU性能差距越来越大

访问时间 = 寻道时间 + 旋转时间 + 传送时间

$$\frac{1}{2}T_{\max\text{ rotation}}=\frac{1}{RPM}\times\frac{60s}{1min}\times\frac{1}{2}\qquad \frac{1}{RPM}\times\frac{\text{磁道}}{\text{平均扇区数}}\times\frac{60s}{1min}$$

- 存储器层次结构
  L0-L4 ✓
  L5：本地二级存储（本地磁盘）
  L6：远程二级存储（分布式文件系统、Web服务器）

中心思想：对于每个k，位于k层的更快更小的存储设备作为k+1层的更大更慢存储设备的缓存。

- 缓存不命中的种类：冷不命中、冲突不命中和容量不命中。
- 高速缓存(S,E,B,m)：①组选择 ②行匹配 ③字抽取
- 高速缓存性能影响：
  ①大小，命中率↔命中时间　②块大小：空间局部性↔时间局部性　③相联度：不命中处罚↔命中时间　④写策略
- 重新排列循环以提高空间局部性：矩阵乘法 A[i][k] B[k][j] C[i][j]　ijk, jij > jik, ijk > kij, jki
- 分块矩阵乘法　+=I　总不命中率 $n^3/(4T)$　ijk　il jl kl　jl=n+jl　jl=n+kl　kl=n+jl　$3T^2<C$

链接

使分离编译成为可能，是将各种代码和数据片段收集并组合成为一个单一文件的过程，这个文件可被加载（复制）到内存并执行。

模块化　效率　步骤：①符号解析　②重定位

- 符号解析：链接器将每个符号引用与一个确定的符号定义关联起来。
- 重定位：把每个符号定义与一个内存位置关联起来，重定位这些节，修改所有对这些符号的引用，使得它们指向这个内存位置。使用汇编器产生的重定位条目的详细指令不加甄别地执行重定位。

目标文件：可重定位目标文件　可执行目标文件

形式：共享目标文件

[扩展名字——ELF二进制文件]

- 典型的ELF可重定位目标文件：ELF头，.text，.rodata，.data，.bss，.symtab，.rel.text，.rel.data，.debug，.line，.strtab，节头部表

.data节：已初始化的全局和静态C变量　.bss节：未初始化/初始化为0的全局和静态变量，仅有节头，节本身不占用磁盘空间

.symtab节：符号表，函数和全局变量的信息　.rel.text节：.text节的可重定位信息　.rel.data　.debug：调试符号表，符号调试信息

.line：原始C源程序与.text节的行号-机器指令的映射　.strtab：一个字符串表，包括.symtab、.debug节中的符号表、节头部表的节名字，以null结尾。

- 符号：①由模块定义并能被其他模块引用的全局符号（对应非静态C函数和全局变量）　③只被模块定义和引用的局部（static C函数和全局变量）
  ②由其他模块定义并被模块引用的全局符号（外部符号，对应其他模块中定义的）
- 局部符号：~非静态C变量——栈　~静态~——.bss或.data（未初始化全局）　数据目标
- 三个伪节：ABS——不该被重定位的符号　UNDEF——未定义的符号，本模块引用但却在其他地方定义　COMMON——未被分配位置的未初始化
- 符号处理规则：函数和已初始化的全局是强符号，未初始化全局是弱符号　-Werror
  ①不允许多个同名强符号　②若有一个强符号和多个弱符号同名，选择强　③如果多个弱~同名，从中任意选择一个　GCC -fno-common
- 静态库：编译时将所有相关的目标模块打包成为一个单独的文件用作链接器的输入。这个文件称为静态库。使用时，通过把相关函数编译为独立模块，然后封装成一个单独的静态库文件，应用程序可以通过在命令行上指定单独的文件名来使用这些在库中定义的函数。链接器只需复制被程序引用的目标模块，减少了可执行文件在磁盘和内存的大小；同时，应用程序员只需包含较少的库文件名字。
- 共享库：一个.so目标模块，在运行和加载时，由动态链接器程序加载到任意的内存地址，并和一个内存中程序动态完全连接为一个可执行程序。使用它可节省内存与磁盘空间，方便软件更新升级。如标准C库libc.so。

加载时动态链接：应用程序第一次加载和运行时，通过ld-linux.so动态链接器重定位动态库的代码和数据到某个内存段，再重定位当前应用程序中对共享库定义的符号的引用，然后将控制传递给应用程序。

运行时动态链接：在程序执行过程中，通过dlopen/dlsym加载和链接共享库，实现符号重定位，通过dlclose卸载动态库。

- 库打桩机制：目标函数→包装函数

编译时-I，链接时　运行时。

·典型ELF可执行文件 只读内存段(代码段): ELF头、段头表、.init、.text、.rodata 读/写内存段(数据段): .data, .bss
将连续的文件节映射到运行时内存段 ←→ 不加载到内存的 .symtab, .debug, .line, .strtab
符号表和调试信号; 节头部表(描述目标文件的节)

·异常控制流 硬件层: 硬件检测到的事件会触发控制突然转移到异常处理程序 操作系统层: 上下文切换 应用层: 信号、非本地跳转
异常表起始地址放在一个叫做异常表基址寄存器的特殊CPU寄存器里. ·进程: 一个正在运行的程序的实例
两个关键抽象: 逻辑控制流、私有地址空间
(上下文切换) (虚拟内存)
[PC值序列]
·并发流: 一个逻辑流的执行在时间上与另一个流重叠 [并行流是真子集]
·并发: 多个流并发地执行 ·多任务: 一个进程与其他进程轮流运行.
·时间片: 一个进程执行它的控制流的一部分的每一个时间段.
·错误报告函数与错误包装函数.
[init进程 PID=1
所有进程的祖先]
·进程终止原因: ①收到一个信号,这个信号的默认行为是终止进程 ②从主程序返回 ③调用exit函数. WNOHANG 立即返回
·fork(): 调用一次返回两次,并发执行,相同但独立的地址空间,共享文件. WUNTRACED 已终止/被停止
·getenv()在环境数组中查找"name=value"返回指针或NULL. WCONTINUED SIGCONT/终止
setenv(), unsetenv(). 替换或增加,删除.
·shell的主要原理及过程:
Linux系统中,shell是一个交互型应用级程序,代表用户运行其他程序.其基本功能是解释并运行用户的指令,重复以下处理过程:
①终端进程读取用户由键盘输入的命令行; ②分析命令行字符串,获取命令行参数,并构造传给execve的argv向量;
③检查第一个命令行参数是否是一个内置的shell命令; ④如果不是shell内部命令,调用fork()创建新进程;
⑤在子进程中,用②获取的参数调用execve()执行指定程序; ⑥如果用户没要求后台运行(&),则shell使用waitpid等待作业终止后返回;
⑦如果用户要求后台运行,则shell直接返回.
·发送信号: 内核通过更新目的进程上下文中的某个状态,发送一个信号给目的进程 1)系统事件 2)进程调用kill函数 待处理位向量 / 阻塞位向量
·接收信号: 目的进程被内核强迫以某种方式对发送来的信号做出反应 ①忽略 ②终止 ③信号处理程序捕获
·隐式阻塞机制(与当前处理相同)、显式阻塞机制 sigprocmask sigaddset sigdelset … SIG_UNBLOCK SIG_SETMASK
sigaction 指定信号处理语义 sigsuspend暂时用mask替换当前阻塞集合,挂起进程
·非本地跳转: 强大但危险的用户级机制,将控制转移至任意位置,不遵守调用/返回规则、错误恢复、信号处理 env envbuf
重要应用: 允许从一个深层嵌套的函数调用中立即返回. 只能跳到被调用但尚未完成的函数环境中.

·虚拟内存 三个重要能力: ①将主存看成是一个存储在磁盘上的地址空间的高速缓存,在主存中只保存活动区域,并根据需要在磁盘和主存之间来回传送数据,通过这种方式,它高效地使用了主存; ②它为每个进程提供了一致的地址空间,从而简化了内存管理; ③它保护了每个进程的地址空间不被其他进程破坏. ·(线性)地址空间: 非负整数(连续)
基本思想,把每个数据对象看成多个独立的地址,其中每个地址都选自一个不同的地址空间.
·虚拟内存是存放在磁盘上,有N个连续字节的数组,被缓存在物理内存中. ·三种状态: 未分配、缓存的、未缓存的.
·DRAM缓存组织结构,完全由巨大的不命中开销驱动,4KB~2MB,全相联,更复杂精密的替换算法,总是写回.
·页表: 一个页表条目的数组,虚拟页→物理页. ·现代系统均使用按需页面调度,当不命中发生时才换出页面.
·VM 简化链接: 每个程序使用相似的虚拟地址空间,代码、数据和堆都使用相同的起始地址;
简化加载: 使得容易向内存中加载可执行文件和共享对象文件,按照需要自动地调入页面;
简化共享: 不同的虚拟页面映射到相同的物理页面,便于进程间共享代码和数据.
简化内存分配: 物理页面可以随机地分散在物理内存中,每个虚拟内存页面都要被映射到一个物理页面,~
·PTE上扩展许可位 SUP READ WRITE 来控制对虚拟页面内容的访问.
·页表基址寄存器(PTBR/CR3): 存储进程页表物理地址.
·地址翻译过程: ①处理器生成一个虚拟地址,并将其传送给MMU; ②MMU生成PTE地址,并从高速缓存/主存中请求得到PTE;
③~/~向MMU返回PTE; ④MMU将物理地址传送给高速缓存/主存; ⑤ ~/~返回所请求的数据字至处理器.
若缺页, ⑥PTE有效位为0,因此MMU触发缺页异常; ⑦缺页处理程序确定物理内存中的牺牲页(若页面修改则换出至磁盘,写回)
⑧缺页处理程序调入新的页面,并更新内存中的PTE; ⑨缺页处~返回到当原来进程,再次执行导致缺页的指令.
·TLB: 高相联度、虚拟页号→物理页号,很少不命中.
·多级页表: 减少内存要求 ①如果一级页表中的一个PTE是空的,那么相应的二级页表根本不会存在; ②只有一级页表才需要总是在主存中,虚拟内存系统可以在需要时创建、页面调入或调出二级页表,减少了主存的压力,只有最经常使用的二级页表才需要缓存在主存中.
·Core i7 48位虚拟,52位物理/32位虚拟,物理 ·pgd 指向第一级页表的基址 mmap: 指向一个vm_area_structs链表
·普通文件、匿名文件(请求二进制零的页) 交换文件: 限制当前运行着的进程能够分配的虚拟页面总数.
·私有写时复制,写→保护故障→物理内存创建新副本,更新页表条目,恢复可写权限.
·fork(): 为新进程创建虚拟内存, mm_struct, vm_area_struct和页表的原样副本,页面标记为只读,区域结构为私有写时复制
·execve(): 删除已存在的用户区域,映射私有区域、映射共享区域、设置程序计数器指向代码区域入口点. ·mmap(): 创建新虚拟区域(用户级) 并映射对象.
·显式/隐式分配器(垃圾收集) 应用到程序实际运行时才知道某些数据结构的大小.
处理任意请求序列、立即响应请求、只使用堆、对齐块、不修改已分配块. 最大化吞吐率、内存利用率
·内部碎片: 有效载荷小于块大小. ·外部碎片: 合计满足,没有独立空闲块满足请求. 序言块 8/1 8/1 结尾块 0/1
·垃圾收集: 有向可达图 根节点+堆节点. 根: 寄存器、栈里的变量或虚拟内存读写数据区域内的全局变量.
维护可达图的某种表示,并通过释放不可达节点且将它们返回给空闲链表,来定期地回收它们.

·系统级I/O 设备映射为文件 普通文件、目录、套接字 dup2(int oldfd, int newfd) v-node表 所有进程共享