

计算机系统的设计/思考方式

系统的层次化设计思想

- 软件/编译/硬件
- Memory hierarchy（存储层次结构）
- 指令集体系结构（ISA） / 微体系结构（组成原理课的主要内容） / 电路实现

物理资源的复用

- 虚存机制以及进程上下文切换（通过异常实现快速切换）等形成的面向用户层任务的假象
- 通过系统调用（UNIX IO）实现对于物理资源（文件资源）的访问

系统运行依赖于处理器执行机器指令

- 在异常处理方面需要处理器支持（不是完全依靠指令执行） / 快表（TLB）是硬件实现的

重视实践

- 课堂上多数内容是在系统上验证的
- 编译器版本、语言规范一直在更新

第一部分——基本数据类型 / C程序在硬件（机器）层面的表示

要求：掌握程序的语义等价转换（C与汇编语言），包括：基本计算、控制流（顺序执行，以及if...else等）、典型循环代码、函数调用与返回、简单数据类型与数值/结构等复合数据类型的机器层面表示与访问等。能够对照C代码及其对应的汇编代码，理解语义，包括（但不限于）补全相关代码

要求：理解并掌握指令/函数/数据的“定位”机制。能够对照C代码及其对应的汇编代码，针对以下各种类型的变量、函数，以及函数调用相关的参数/返回值等，理解其定位机制，包括（但不限于）补全相关代码

➤ 相对地址定位的要求：理解相对于PC(%rip)定位的基本依据（不管对象文件被装载到进程的哪个地址，代码段中的任一给定指令与数据段（包括只读数据段）中的任一给定位置之间的“距离”是一个常量），能够理解并掌握简单的相对地址定位机制（不超过课上讲解的难度）

	编译	链接（静态）	说明
局部变量(非静态)	√		存于栈内 or 寄存器 or 被优化掉
全局变量		√	各个模块(.o文件)的数据段、代码段合并后，方能确定地址
函数参数	√		通过指定的寄存器与栈传递
函数返回值	√		%rdx + %rax
跳转指令（直接）	√		相对于PC的offset
函数返回地址	√		存于栈顶
静态函数入口地址	√	√（与编译优化开关有关 -Og）	相对于PC的offset
全局函数入口地址		√	各个模块(.o文件)的数据段、代码段合并后，方能确定地址

第1讲——计算机系统概论

基本概念（了解）

- 计算机体系结构（狭义）
- 程序、编译、链接等的基本概念与作用
- 冯·诺依曼结构的基本特点与构成：
 - ① 程序存储执行（或者，以指令为中心 / 存算分离）
 - ② 计算机由五个部分组成（运算器、控制器、存储器、输入设备、输出设备）
- 存储程序工作方式：
 - ① 程序包含着指令序列
 - ② 程序存储在计算机中
 - ③ 在程序执行的过程中，处理器重复从存储器中取指令然后解释执行这一指令
- 程序基本执行过程：
 - 将程序载入内存、将处理器的取指令指针指向程序开始处、运行程序

机器语言程序（汇编程序）



第2.1讲——整数

数据大小表示单位 (了解)

- Bit(位): 计算机中最小的数据表示单位
- 字节: 1字节 = 8 bit
- 字: 计算机字长的概念、32位机/64位机的字长概念

机器字在内存中的组织 (掌握)

- 机器字中第一个字节的地址,即为字的地址
- 大端/小端方式

无符号整数 (掌握)

- 无符号整数的应用场景, 无符号整数的二进制表示, 表示范围;
- (了解) C语言编程时何时用unsigned int会比较好, 容易出现什么错误。

有符号整数 (掌握)

- 补码的二进制表示, 表示范围, 以及与无符号整数的关系;

原理 $x + (-x) = 0$ $-x = 0000 \dots 0000 - x$
 $= (1111 \dots 1111 + 1) - x = (1111 \dots 1111 - x) + 1$
(即按位取反 + 1)

- N-bit二进制位串的补码表示范围
- 补码表示的优点: 一个0、直接运算; 在计算机里都用此表示

给出二进制位串, 写出对应无/带符号整数的值以及能进行基本运算 (二进制或十进制;掌握)



C程序中的逻辑运算、位操作 (要看懂)

第2.3讲——浮点数

科学计数法与浮点数表示（掌握）

- 单精度浮点数的表示、双精度浮点数的表示、以及符合754标准的任意长度浮点数的表示
 - 三类：规格化、非规格化、其它
- 最大/最小的规格化的浮点数等（即各类极值）
- 如何将一个实数转换成浮点数（注意舍入方式）

浮点数表示的范围和精度（了解）

- 任意相邻的两个浮点数之间的距离是相等的吗？
- 尾数的位数、阶码的位数与浮点数的表示范围和精度之间的关系？
- 浮点数运算中也满足整数的结合定律，即 $(a+b) + c = a + (b+c)$ 吗？
- C语言中的浮点数类型以及与整数之间的转换

第3.1讲——基本指令

汇编程序员眼中的体系结构（了解处理器状态、以及程序的执行过程）

x86-64的通用寄存器 / x86-32的通用寄存器（掌握）

各类基本指令（掌握）

- mov指令、地址计算指令lea、各类算逻指令等
- 操作数类型及其限制
- 地址表达式的各种形式
- 利用地址表达式做计算



第3.2讲——控制流

程序的指令执行顺序（或者序列）被称为处理器控制流

- 两种改变控制流的方式：jump/branch 以及 call/return

条件码与设置（掌握）

- 通过执行相关指令设置CPU条件码CF、OF、ZF、SF
- 影响条件码的指令：算术运算类指令；位操作与逻辑运算类指令；比较指令、测试指令
- C语言条件表达式的汇编级实现，就是上述算术运算、比较、测试、读取条件码（setX）等指令的组合
- 与0，1等的比较，一般汇编语言转换成test/and + js/jns等指令
- 条件跳转指令、set指令、条件移动指令以及这些指令的后缀含义（包括区分带符号数/无符号数等）
- 条件移动指令的微架构基础（了解）

循环结构的汇编级实现（掌握）

- for循环： 单分支
- while 循环： 单分支
- do while 循环： 单分支

switch分支的跳转表实现（掌握）

- 跳转表
- 采用间接寻址方式，jmp *【地址表达式】

第4讲——函数调用（运行栈）

C函数调用机制（掌握）

- 传递控制
- 传递数据
- 栈内存分配与释放

栈访问指令、调用指令和返回指令（掌握）

- x86-64的入栈指令pushq
- x86-64的出栈指令popq
- call / ret指令

函数调用的汇编代码（掌握）

- 展示函数调用中如何传递控制的：执行call之后的PC和栈顶内容，以及执行ret之后的
- X86-64中的参数传递大部分是通过寄存器实现的
 - 传递函数参数的寄存器有哪些？寄存器对应传递参数的顺序？超过6个的参数如何传递？通过栈传递参数时的规则（参数顺序）
- 对照实际C程序代码及其反汇编指令，解释参数传递时寄存器的使用情况
- 函数返回值如何传递（%rdx+%rax）

栈帧结构（掌握）

- 栈帧的概念，栈帧中存储的信息可以有哪些，如何管理（分配与释放）
- 函数调用过程中栈帧的变化过程
- 栈帧的布局
- 寄存器的保存约定（了解）

非静态局部变量的存储（掌握）

- 栈上的局部变量存储。
- 寄存器中的局部变量存储
- 被编译优化掉

递归过程与一些其他示例（了解）

基于RIP的相对寻址（掌握简单示例）

- 类似于课上的示例，如`incq %rcount(%rip)`、`movl $1, a(%rip)`之类

x86-32的函数调用规范等（掌握）

- X86-64是向下兼容的

静态函数入口地址定位（掌握）

- Call指令：相对于PC寻址

第5讲——复合数据结构

指针的汇编级实现（掌握）

- 指针的汇编级本质：地址；就是一个数，只不过其值是一个地址而已
- 指针的算术运算： $p++$ ； $q=p+1$ ；汇编级实现为加上 $\text{sizeof}(*p)$ ，即 p 指向的数据的类型占用的空间
- $*p$ 操作：取内容，寻址方式 $D(Rb, Ri, S)$
- $P \rightarrow$ 操作：访问结构体成员，汇编级为 $*(p + \text{在结构体内的偏移})$

数组的汇编级实现（掌握）

- 数组的汇编级本质：地址。是一组数据在内存的起始地址，即指针
- 数组的名字，就是该指针
- 数组的元素下标，表示相对首地址的偏移
- $A[i] = *(A+i)$ 在循环结构中， i 是变化的。对应的汇编级实现为 $A(, i, \text{size})$
- 数据作为全局变量、局部变量、参数、返回值时的具体实现

结构体的汇编级实现与对齐（掌握）

- 结构体名字的本质：地址；一批成员在内存的起始地址
- 结构体成员在内存的存放布局：数据对齐三要素！
- 结构体成员访问`struct.item`： `* (struct地址 + item偏移)`
- 结构体指针及成员访问`ps->item`： `* (ps + item偏移)`
- 结构体数组及成员访问`as[i]->item`： `*(as + i*size + item偏移)`
- 结构体数据作为全局变量、局部变量、参数、返回值时的具体实现

联合(掌握)

- 联合体名字的本质：地址。成员在内存的起始地址
 - 联合体的成员的存储空间是重叠的；联合体的size为其中最大值
- 联合体成员在内存的存放：共享内存（隐含强制类型转换）
- 联合体成员访问`union.item`： `(item 类型) (union地址)`
- 联合体数组及成员访问`au[i]->item`： `(item 类型) (au+i*size)`

第6.1讲——程序链接

汇编的作用：汇编器（Assembler）将编译过程输出的汇编代码转换为二进制机器代码，生成一个包含机器代码的可重定位目标文件（relocatable object file）

链接的作用：链接器（Linker）将汇编器输出的若干可重定位目标文件（可能包括若干标准库函数目标模块）组合起来，生成一个可执行目标文件（executable object file）。

链接的本质和基本过程（掌握）：理解程序链接中涉及的全局/外部/本地不同符号类型及其如何参与符号解析；理解全局符号的强、弱特性与多重符号处理方式；理解符号解析的目的与过程；理解重定位的概念、类型与过程，理解静态库的生成方法及其如何参与符号的解析与链接。

➤ 程序中的符号、模块内和模块间符号引用的基本概念

- 解释全局符号的强、弱特性与多重符号处理方式

- 全局符号定义的强/弱属性划分：针对不同目标模块中可能包含同名的全局符号定义（即多重定义），将其按是否初始化划分为强符号和弱符号——定义时具有初始值的符号称为强符号，没有初始值的称为弱符号。

- 多重定义的全局符号在解析过程中的处理规则：将所有模块中针对该符号名的符号引用绑定到唯一的强符号定义上（不允许同时存在多个强符号定义）。如只存在（多个）弱符号定义，则将符号引用绑定到其中任一但唯一的符号定义上

- 链接的本质是要“合并”重定位目标文件。要能正确“合并”，必须解决**符号解析**和**重定位**两个问题
- 链接器对可重定位目标文件中的符号引用进行解析，合并所有可重定位目标文件中相同类型的section，确定符号的运行时地址，并对符号引用进行重定位，具体如下——
 - （1）符号解析的目的：将各目标模块中对一符号的引用与某个模块中该符号的定义关联起来。
 - （2）符号解析的过程：链接器扫描所有输入模块中的符号表，针对其中每一尚未完成绑定的符号引用，搜索和确定其定义所在的模块

	编译	链接 (静态)	说明
局部变量(非静态)	√		存于栈内 or 寄存器 or 被优化掉
全局变量		√	各个模块(.o文件)的数据段、代码段合并后, 方能确定地址
函数参数	√		通过指定的寄存器与栈传递
函数返回值	√		%rdx + %rax
跳转指令 (直接)	√		相对于PC的offset
函数返回地址	√		存于栈顶
静态函数入口地址	√	√ (与编译优化开关有关 -Og)	相对于PC的offset
全局函数入口地址		√	各个模块(.o文件)的数据段、代码段合并后, 方能确定地址

上表中静态链接采用的是绝对地址定位的方式, 即链接器填入的是符号的绝对地址 (掌握)

后面我们学习了“**共享库中的全局变量寻址**”, 知道共享库被不同进程装载的时候, 其绝对地址是不一样的, 这就给绝对地址定位的方式带来了难度

- 一种解决方式是“相对定位”方式, 其依据的事实是: 不管对象文件被装载到进程的哪个地址, 代码段中的任一给定指令与数据段 (包括只读数据段) 中的任一给定位置之间的“距离”是一个常量, 据此编译器生成与绝对地址无关 (Position Independent Code) 代码

第6.2讲——内存布局与缓冲区溢出

Linux内存布局（了解）

- x86-64 Linux内存布局，显示代码、数据、栈和堆的所在位置
- 通过实际内存分配示例程序，显示出程序运行后各变量的地址，分析其位于内存中的位置

缓冲区溢出（掌握）

- 在机器级代码层面解释访问越界引起的漏洞
- 能够解释如何利用缓冲区溢出进行攻击
- 能够运用所学知识构造缓冲区溢出攻击代码
- 理解缓冲区溢出攻击的基本防范措施（了解）
 - 代码中避免溢出漏洞
 - 栈随机化
 - 栈破坏检测(金丝雀机制)
 - 限制可执行代码区域

第二部分——多任务视角下的系统

第一部分是从“单任务”的视角来“看”程序在机器层面的表示，即（看上去）一个程序占据了整个处理器及完整的内存空间；但是实际情况要远为复杂

计算机系统（包括操作系统与处理器）采用“虚存”、“异常”等关键机制，在有限物理内存前提下设计出连续的、进程间相互独立的虚拟内存，以及通过高速的任务切换营造出单进程占据整个处理器的“假象”

- 这背后涉及的系统设计思想是“物理资源的虚拟化复用”：时分复用（任务切换）、空分复用（虚存）；更进一步，对于其它硬件资源的使用（如文件IO），也是通过系统进行了抽象

具体要求如下所示——

第7讲——虚存

虚拟存储概述（掌握）

- 虚存的基本概念与作用; 虚拟地址和物理地址的概念
- 进程的虚拟地址空间布局
- 理解如何由可执行文件构建虚拟地址空间

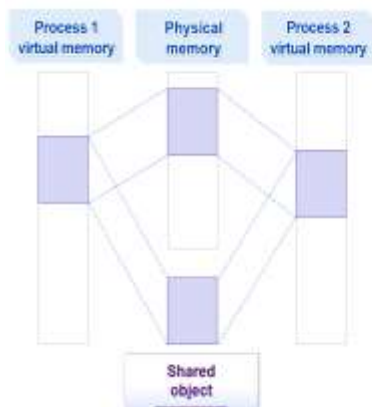
虚拟存储实现（掌握）

- 页式虚拟存储的（逻辑）实现，包括页表和页表项（页的属性）、虚实地址转换过程、TLB的概念与作用；
- 处理器访存的过程，包含对TLB、页表、Cache和主存的访问；TLB缺失或缺页情况下对应的处理；
- Address Translation Example（虚实地址转换、TLB查找、页表查找等）

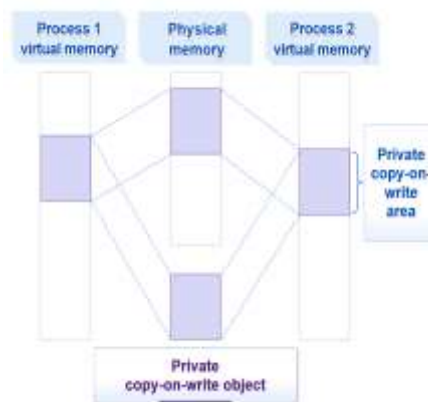
内存映射（了解）

- 内存映射的概念
 - 理解VM areas initialized by associating them with disk objects
 - 匿名文件（anonymous）
 - 何为“脏页”

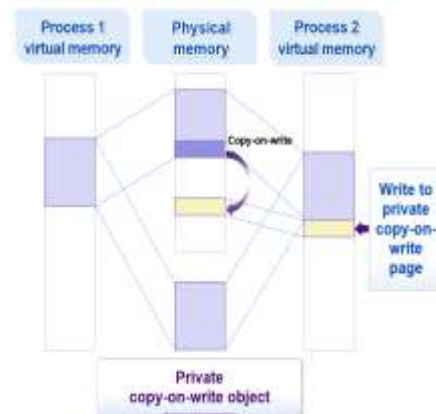
- vm_area_struct这一结构的基本作用、与虚存内各个区域（正文段、数据段等）的关系
- 不同情况下的缺页处理（page fault）
- 磁盘文件、进程的虚存各区域以及虚存相关的内核数据结构（主要是页表与vm_area_struct的关系）
- demand paging机制
- （通过虚存的）进程间对象共享机制
- COW机制



- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.



- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only



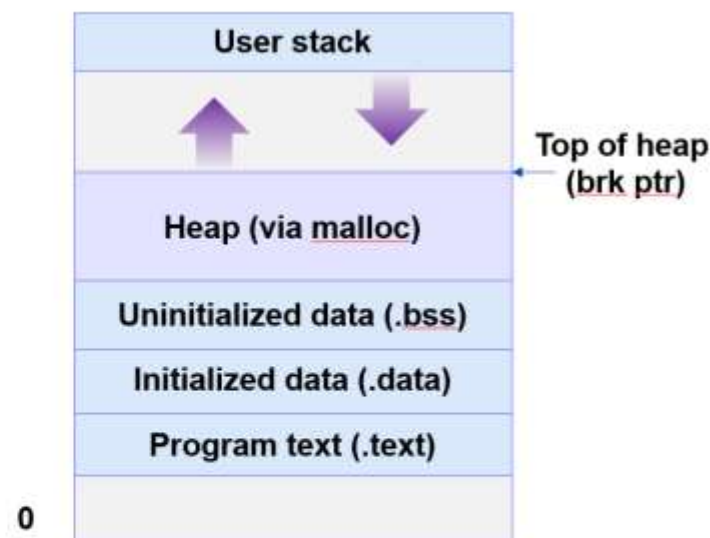
- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying *deferred as long as possible!*

第8讲——内存分配

堆、堆顶、C库提供的内存分配函数与系统调用 (brk) 的关系 (了解)

内存分配的内部碎片、外部碎片的基本概念 (了解)

- Programmers use **dynamic memory allocators** (such as *malloc*) to acquire VM at run time.
 - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as the **heap (堆)**.



Linux系统提供了名为brk的系统调用，用于设定堆顶，有兴趣同学可以搜索下Linux system call: [brk](#)

第9.1讲——ECF

异常和中断的基本概念（了解）

- 程序执行过程中CPU会遇到一些特殊情况，使正在执行的程序被“打断”
- CPU响应、处理“非预期事件”的手段
- 同步异常（内部） / 异步异常（外部）
 - 了解异常处理机制需要处理器支持（不是完全依靠指令执行）
- 异常处理向量的概念

同步异常的分类（了解）

- 故障 (fault)：程序“无意”引起，可能可以被“修复”
- 陷阱(trap)：“有意”的异常，如系统调用
- 终止(abort)：不可恢复的致命错误导致的结果
 - 举例：在虚存中，出现的缺页，属于故障、陷阱还是终止？指令执行过程中，碰到了非法指令，属于故障、陷阱还是终止（属于trap）？ECC内存发生错误呢？

进程概念及其它（掌握）

- 进程：程序的一个运行实例
- 进程的两个关键抽象：（逻辑）控制流，即“看上去”拥有一个独立的处理器；虚存，即“看上去”拥有独立的连续内存空间
- 进程的逻辑控制流
 - 程序指令执行序列，即处理器重复从存储器中取指令然后解释执行指令这一周而复始的过程；这一过程涉及处理器状态（寄存器堆、PC、条件码等）、虚存状态，以及它们的修改。
- 进程的上下文切换
 - 系统使用上下文切换来实现多任务；异常处理是切换的时机
 - 上下文包含的内容：虚存状态、处理器状态，以及操作系统内核维护的进程数据结构
 - 上下文高速切换使得进程并发可行
- fork函数
 - 进程的创建:clone父进程
 - 一次调用，二次返回：区分父子进程
 - 与COW/demand paging的联系
 - fork的各类代码示例的分析

➤ execve函数

- 装载并运行特定程序
- 流程（了解）
- 与fork函数的使用关系

➤ exit函数

- 终止当前进程/返回进程的退出状态值
- atexit函数
- 僵尸进程的概念及其产生的原因

➤ wait系列函数

- 等待子进程退出，可以获得进程的退出状态值；不产生僵尸进程
- wait/fork的各类代码示例的分析

第9.2讲——信号处理

信号概念（了解）

- 信号：高层次的软件形式的异常控制流，用于通知进程发生了某种类型的事件
 - Linux中有30种不同类型的信号（主要的）

信号处理过程（了解）

- 信号的产生
- 信号的接收
- 信号的发送
- 信号的处理

信号的发送和接收（掌握）

- 信号的发送：/bin/kill程序、键盘发送信号、kill函数发送信号、alarm函数发送信号；异常处理发送信号
- 信号的接收：进程的信号接收默认行为，通过signal函数修改默认行为；有些信号，如SIGSTOP、SIGKILL不能修改默认行为。

信号处理函数 (signal handler; 掌握)

- signal handler 与进程的主control flow拥有不同的栈与处理器上下文（也可称为处理器状态或者control flow）
- 多个同样信号抵达时，处理信号的问题。
- 异步信号安全函数的概念 (Async-Signal-Safety)

非本地跳转处理(setjmp / longjmp; 掌握)

- 函数语义
 - setjmp一次调用/两次返回，如何区分
- 其局限性

各类上述函数以及wait/fork/kill等函数组成的各类代码示例的分析 (掌握)

第10讲——IO处理

CSAPP章节 10.1——10.11 （10.5不要求）

UNIX系统下的文件（了解）

- 文件的基本概念
- 为何所有I/O操作都可以通过读写文件实现(因为IO设备都被视作文件)
- 文件的分类
 - 常规文件/目录
 - 字符设备，典型的字符输入设备和输出设备
 - 块设备，典型的块设备
- 文件的创建和打开、文件的读/写、文件的定位和关闭。

UNIX IO（系统级IO）函数（掌握）

- CREATE系统调用
- OPEN系统调用
- READ系统调用
- WRITE系统调用
- LSEEK系统调用
- STAT/FSTAT系统调用
 - 文件元数据（了解）
- CLOSE系统调用
- 上述函数的语义、以及相关示例代码的分析

UNIX如何表示打开的文件（掌握）

- 三张表（file descriptor table、open file table、v-node table）：每张表的含义以及是否进程间共享
- 单个文件被多次打开
 - 三张表间的关系
- 父子进程通过fork共享打开的文件
 - 三张表间的关系
- IO重定向
 - dup
 - dup2
 - 三张表间的关系

C标准I/O库函数（Standard I/O Functions）（掌握）

- 包裹了UNIX IO实现了带缓冲I/O
 - 通过fflush调用显式地flush到输出文件
- 标准输入、标准输出和标准错误的定义
- Unix I/O、Standard I/O的优缺点与适用场合（了解）

各类上述函数以及wait/fork/kill等函数组成的各类代码示例的分析（掌握）

第11讲——线程、线程同步以及多线程编程基础

CSAPP 12.3 (socket相关不要求)、
章节 12.4、12.5 (12.5.5
不要求)、12.6、12.7.4、
12.7.5

线程及其与进程的区别 (掌握)

- 回顾进程：进程占有的系统资源：进程上下文（处理器状态+内核上下文）与虚存空间内存段（code, data, stack, heap, ...）等等
- 线程与进程的区别，线程的优势在哪里？
 - 并行执行的本质是要构造多control flow，而后者最必要的资源是Stack、处理器状态
 - 一个进程中可以有多个线程（control flow）：一个线程对应一个指令流，因此一个进程内的多线程可以共享进程中的其他内存段、Kernel context来节省资源
 - 进程内线程切换的代价 vs. 进程切换的代价(了解)
 - 协程与线程的区别：协程相当于用户态线程，一个线程内的协程切换是用户态程序内部的事情，在操作系统看起来是一个线程的连续工作，没有切换开销（了解）

Pthreads多线程编程接口（掌握）

- 线程创建和回收：pthread_create和pthread_join
 - 介绍线程joinable和detached模式的区别
- 查看线程ID：pthread_self
- 结束线程：pthread_cancel和pthread_exit
- 简单的示例代码分析

线程执行模型/内存模型（掌握）

- 线程抽象的并行模型是“多指令流+共享内存”
- 多线程中的共享变量
 - 进程内多线程共享变量的范围：全部变量，不限于全局变量/静态变量，因为在同一个进程空间内，有内存地址即可访问
 - ◆ 多线程访问共享变量的代码示例
- 概念上的内存模型与实际的共享内存可能带来微妙的、难以重现的错误
- 失败的实例分析

互斥与基于信号量的线程同步（掌握）

- 为什么共享变量并发访问可能导致出错
 - 进程图的基本概念（了解）
- 线程同步的基本概念
 - 解决上述错误的方式是进行线程同步
 - 线程同步是指线程之间互相协调，通过互斥、等待、唤醒等操作，使多线程按照正确的顺序或符合约束条件地执行，例如避免线程之间同时操作同一个资源，或保证线程某段代码在依赖条件达成后才开始运行

- 线程同步的具体方式
 - 临界区/资源互斥：对同一资源访问串行访问，具体可以使用信号量等
 - 资源并发访问数控制：资源最多允许N个线程并发访问，临界区和互斥可以看作N=1的特例，具体可以使用信号量（Semaphore）
 - 信号量的基本概念
 - ◆ 信号量简介，P、V操作介绍
 - ◆ 信号量POSIX接口：sem_init, sem_wait (P), sem_post (V)

- 使用信号量实现互斥和共享资源调度

- ◆信号量实现互斥的程序示例（cnt程序中保护临界区，相当于互斥锁作用）

- ◆使用信号量的cnt程序对应的进程图变化（更直观理解线程同步的作用；了解）

- 使用信号量实现线程调度

- ◆理解信号量的调度作用：拿到“令牌”（通过P操作）的线程可以工作，否则等待；释放令牌（V操作）不一定是在同一个线程中，可以是其他线程、其他条件，因此信号量可以实现线程执行顺序的调度；

- ◆生产者消费者模型等代码实例分析（两个信号量分别控制slots和items）

- 其它（了解）

- 竞争（race）：程序的正确性依赖于某种顺序

- 死锁及避免