操作系统 Operating System Concepts

> **Note**
> LAB 0-7（5+15+15+15+20+20+10+10=110）

1. 概述 Overview

1.1 操作系统功能&定义

从计算机角度，**操作系统**是个程序，管理电脑硬件

- resource allocator：分配资源（cpu、内存、I/O设备——硬件hardware 提供资源）
- control program：防止资源滥用

定义： The operating system is the one program running at all times on the computer——namely kernel
操作系统是一直运行在计算机上的程序——通常称为内核

multi-user OS: Linux ubuntu

1.2 计算机系统的运行

计算机= CPU + 多个设备控制器（I/O devices），通过公用总线相连，该总线提供了共享内存的访问

CPU和I/O设备可以并发执行（execute concurrently），并竞争访问内存，需要内存控制器协调访问内存

每个设备控制器 device controller

- 控制特定的设备（磁盘驱动器、音频、视频显示..）
- 有本地缓冲区（local buffer），存储I/O
- 经系统总线system bus触发中断interrupt告知CPU已完成

引导程序bootstrap program在开机power-up/reboot时加载

- 到ROM/EPROM上（Read-only Memory是firmware固件）
- 初始化系统
- bootstrap定位操作系统内核并加载到内存、开始执行

1.2.1 中断 interrupt

中断是

ISR中断服务例程interrupt service routine

中断向量interrupt vector：存储中断*例程*的地址 the addresses of all the service routines

- 寄存器（Interrupt architecture）储存被中断*指令*的地址

有中断在处理时会停止其他未到的中断
Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt.

trap ： 软件的中断，经常由错误或用户请求（又称 系统调用system call）导致

操作系统是由中断驱动的
An operating system is interrupt-driven

中断处理——一种软件例程处理

1. 恢复CPU状态——存寄存器&程序计数器（program counter）
    - 计数器：存储下一条要执行的指令的位置
2. 确定中断类型：polling by a generic routine or vectored interrupt system

两种I/O处理方式：

1. 要等待，处理完传回——同步 synchronous
2. 接收到（没做完）就传回——异步 asynchronous

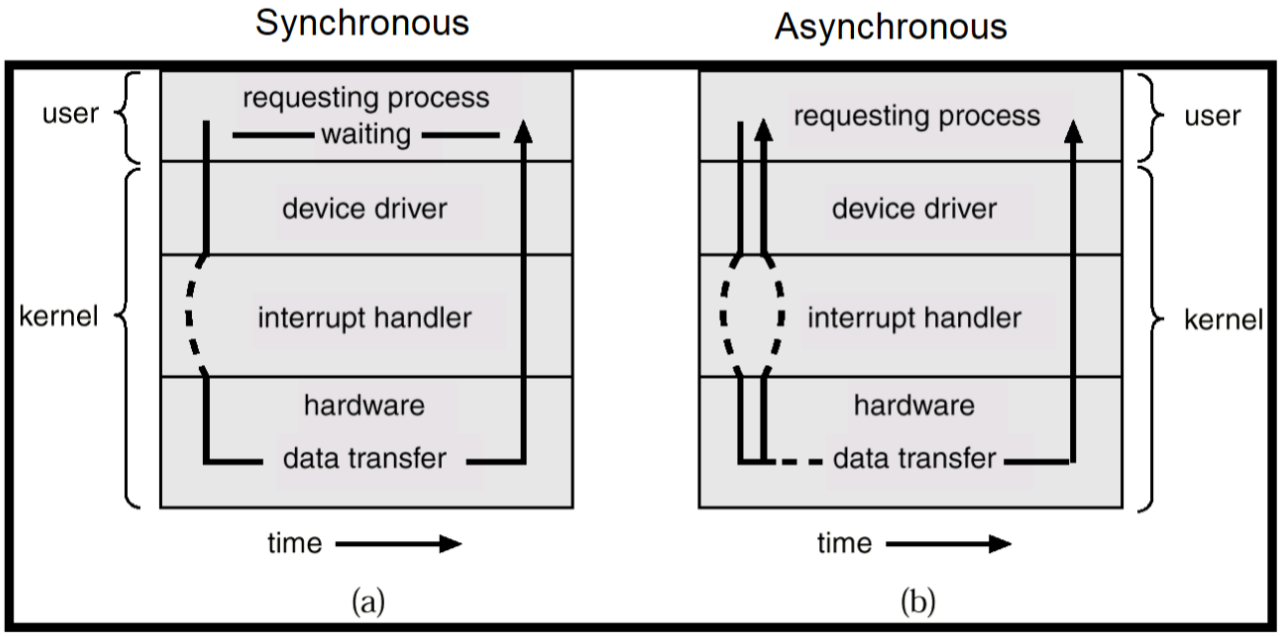


Figure 1

DMA：直接内存访问

Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds.
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.
- Only one interrupt is generated per block, rather than the one interrupt per byte.

Figure 2

每块只产生一个中断

1.3 存储

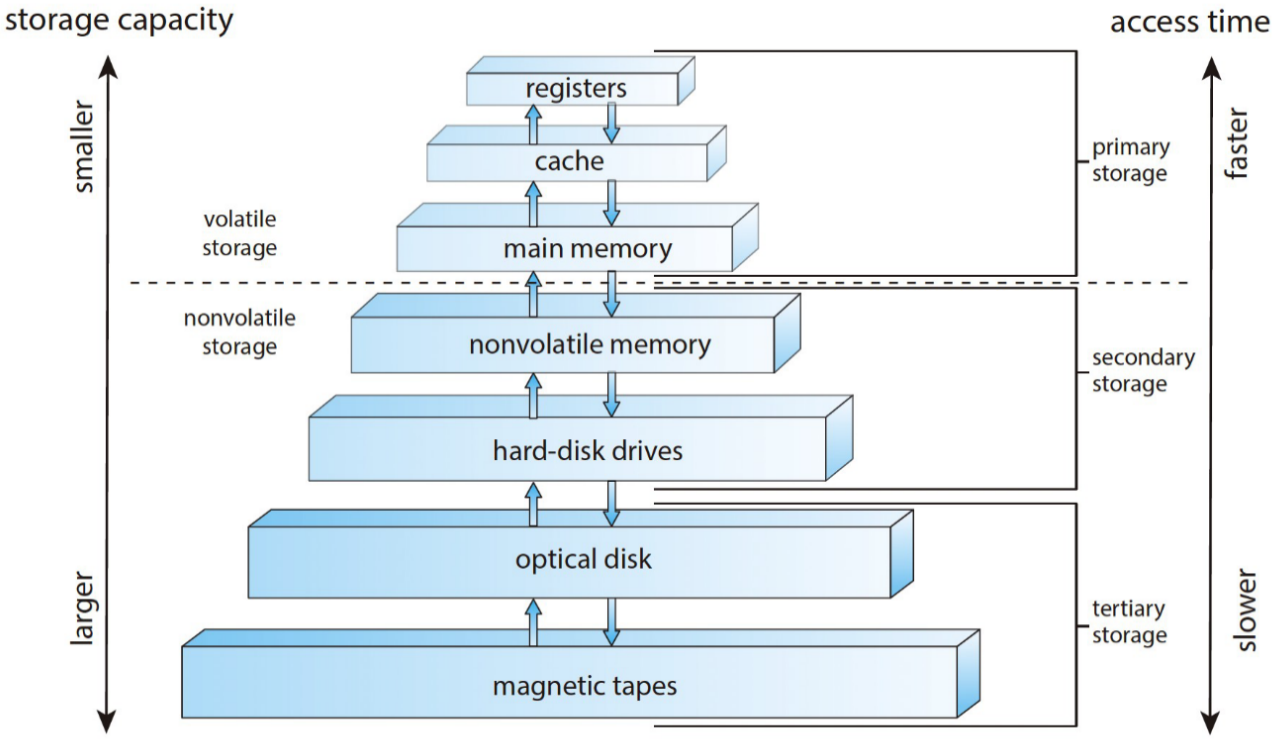


Figure 3

速度、花销、易失性

一级二级三级是按容量分的

高速缓存cache——更快的数据访问机制，主存可以认为是二级存储的最后一个cache

1.4 计算机体系结构

多处理器 multiprocessing

- 增加吞吐量
- 规模经济
- 增加可靠性

对称多处理器 SMP symmetric multiprocessing：所有处理器对等、没有主从关系，共享内存

多核 multicore

- 单片通信比芯片间通信快
- 多核电源消耗低

非均匀内存访问 Non-uniform memory access(NUMA architecture)

- CPU通过一个共享系统连接
- 随着处理器的增加而更有效地扩展
- 跨互连的远程内存速度较慢
- 操作系统需要仔细的 CPU 调度和内存管理

1.5 操作系统结构

Multiprogramming 多道程序设计 needed for efficiency (CPU utilization)

- 组织各项任务让CPU总有工作做
- 加载多项任务到内存
- 通过 job scheduling 安排选一个任务做
- 定义：在一个cpu上并发运行多个进程

Timesharing (multitasking) is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating interactive computing (interactivity)

- 响应速度足够快
- 多任务（进程 process）并发在CPU上执行 -> CPU调度scheduling
- 如果一个进程不能全放入内存memory -> 换入换出swapping（将没用的进程挪出）
- 虚存virtual memory：一种技术允许CPU执行不完全在内存的进程

1.6 操作系统执行 OS operations

> **Important**
> Interrupt driven by hardware
> Software error or request creates exception or trap
>
> - Division by zero, request for operating system service
>
> Other process problems include infinite loop, processes modifying each other or the operating system

双模式dual-mode 操作允许操作系统保护自身和其他系统组件

- 用户模式和内核模式 user mode & kernel mode
- 硬件提供的模式位 mode bit
  - 提供区分系统何时运行用户代码或内核代码的能力
  - 一些指令被指定为特权privileged，只能在内核模式下执行
  - 系统调用system call将模式更改为 kernel 模式，再通过系统调用回来将其重置为 user 模式

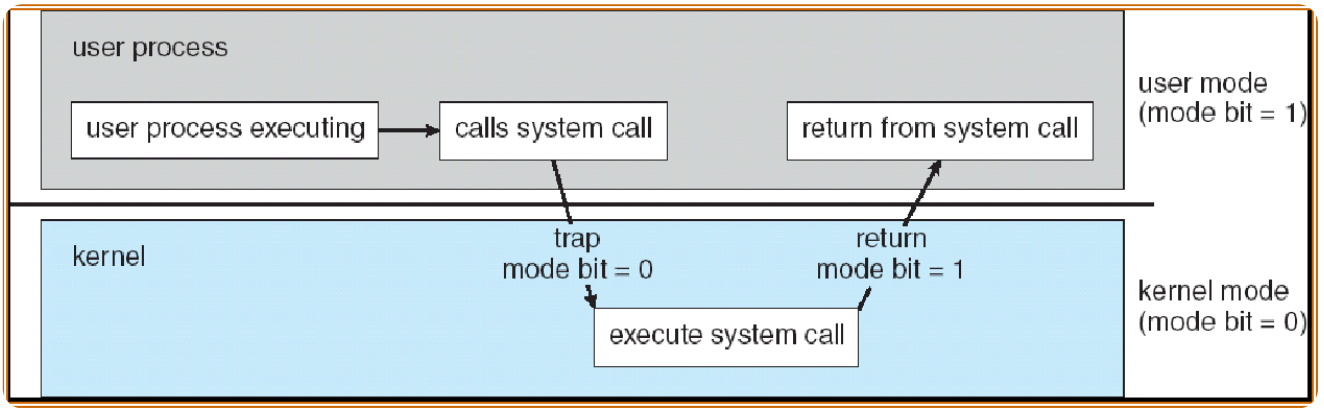


Figure 4

Timer定时器：防止无限循环/进程占用资源

- 在一段时间后中断进程 set interrupt
- 操作系统将计数器counter `-1`，**到0产生中断**
- 在安排流程之前进行设置，以重新获得控制权或终止超过分配时间的程序

1.6.1 Process management 进程管理

程序是被动实体，进程是主动实体 active entity

- 需要资源去执行
- 进程终止时将资源还给操作系统
- **多线程进程Multi-threaded process** 每个线程都有一个计数器
- **多路复用multiplexing**->实现并发concurrency

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes 暂停/恢复进程
- Providing mechanisms for process synchronization 同步
- Providing mechanisms for process communication 交流
- Providing mechanisms for deadlock handling 死锁

Figure 5

1.6.2 Memory management内存管理

- All data must be in memory before and after processing
- All instructions must be in memory in order to execute
- Memory management determines what is in memory when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

Figure 6

1.6.3 File management 文件管理

文件管理：创建、删除、加密

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit - **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories 目录
  - Access control on most systems to determine who can access what 权限控制
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and dirs
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

Figure 7

1.6.4 Storage management 存储管理

Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage needs not be fast
  - Tertiary storage includes optical storage, magnetic tape
  - Still must be managed
  - Varies between WORM (write-once, read-many-times) and RW (read-write)

Figure 8

I/O Subsystem 子系统

- One purpose of OS is to hide peculiarities of hardware devices from the user – *ease of usage & programming*
- I/O subsystem responsible for
  - Memory management of I/O including buffering 缓冲 (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling 假脱机处理 (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices 驱动程序

Figure 9

1.7 总结 OS purposes

- Abstraction 抽象： a way to hide complexity
- Multiplex 多路复用（空间、时间）
- Isolation 隔离（用户/核模式）
- Sharing 共享（用户、进程 资源共享->并发）
- Security 安全（特权指令
- Performance 性能
- Range of uses

2. 结构 Operating-System Structures

2.1 操作系统服务

操作系统提供的服务有：

- UI(user interface)
- CLI(command line)
  - 优点：对重复性工作高效
- GUI (Graphics User Interface)

2.2 System call 系统调用

2.2.1 系统调用实现

API（Application program interface)：被封装过high-level的系统调用

- Mostly accessed by programs via a high-level API rather than direct system call use 主要通过高级的API接触程序而不是直接的系统调用

每个系统调用有个编号，通过系统调用表 table

系统调用发生时，程序控制权将交给内核来完成服务，服务完成后，控制权再还给程序（回到用户状态

系统调用是用户应用（软件）和硬件之间沟通的桥梁——保证了安全和有效的沟通 secure & efficient

2.2.2 参数传递

最简单：寄存器保存

间接方式：block、table、memory，把内存中buffer地址告诉寄存器（Linux & Solaris）

其他：栈stack，不一定支持多用户

- block和stack方法不限制参数的数量或长度

2.2.3 系统调用分类

Process control 进程管理
File management 文件管理
Device management 设备管理
Information maintenance (e.g. time,date)信息维护
Communications 交流
Protection 保护

	Windows	Unix
Process control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communications	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Figure 10

2.3 系统程序

分类： File manipulation、 Status information、 File modification、 Programming language support、Program loading and execution、Communications、Application programs

.....debuggers

不是系统程序：web browsers、word processors（用户程序）

2.4 os设计和实现

os设计和实现不是一个解决问题的过程（不是具体的问题而是创造性过程）

1.由目标和指标出发：用户目标、系统目标

2.被硬件和系统类型影响

Policy： What to do? 策略（确定具体做什么事，eg允许某些用户访问某些文件）
Mechanism：How to do it? 机制（定义做事方式）

内存管理、cpu调度是机制？（算法是机制?

配置文件写的是策略（policy）

Provide mechanism rather than policy. In particular, place user interface policy in the clients' hands.

2.5 操作系统结构

2.5.1 Simple Structure

MS-DOS: provide the most functionality in the least space

- 不分模块，界面和多级功能间没有很好的分开

2.5.2 UNIX

UNIX – 受硬件功能限制，原始 UNIX 操作系统结构有限。UNIX 操作系统由两个可分离的部分组成

- 系统程序
- 内核
  - 包括系统调用接口以下和物理硬件以上的所有内容
  - 提供文件系统、CPU 调度、内存管理和其他操作系统功能；一个级别的大量功能

2.5.3 Monolithic System Structure 宏内核架构

优点:

-

缺点:

-

UNIX System Structure 宏内核

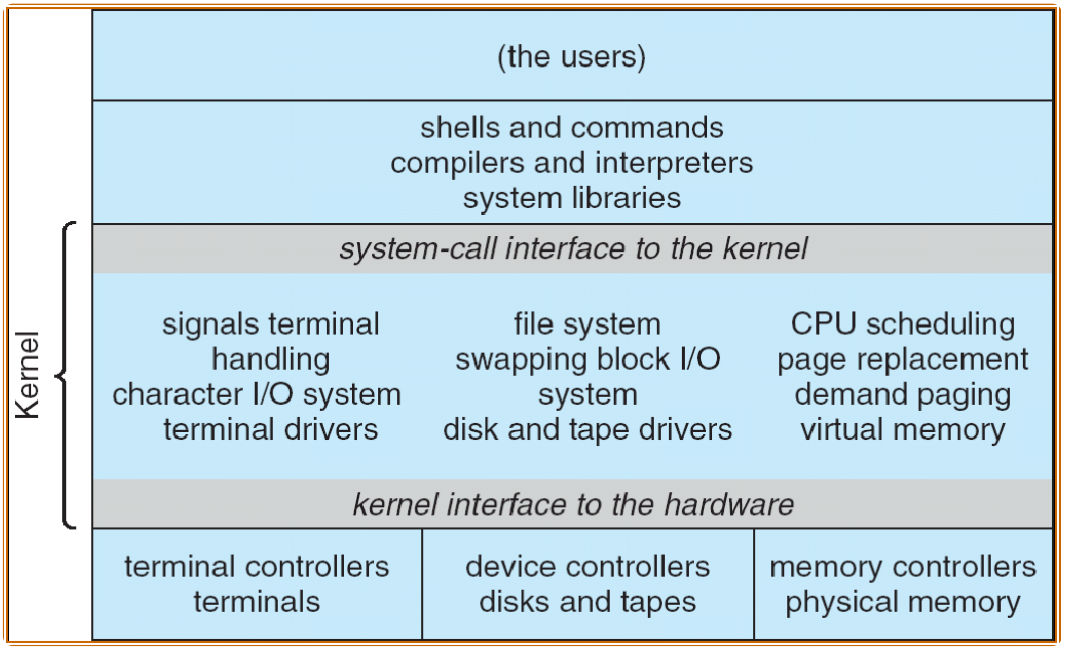


Figure 11

2.5.4 Microkernel System Structure 微内核

对应宏内核

将尽可能多的内容从内核移至“用户”空间，提供一小部分服务例如进程调度

用户模块之间使用消息传递进行通信 message passing

优点：

- 更易于扩展 extend 微内核
- 更易于将操作系统移植 port 到新架构（flexibility）
- 更可靠（内核模式下运行的代码更少
- 更安全

缺点：

- 用户空间与内核空间通信的性能开销performance低（宏内核更高效）
- 系统服务、驱动程序的复杂性
- 上下文切换不太有效

2.5.5 Hybrid Structure——Darwin

混合模式：By ??

Windows, MacOS, iOS——混合内核

2.5.6 Layered Approach 分层方法

操作系统分为多个层（级别），每层都建立在较低层之上。**最底层（第0层）是硬件；最高层（第N层）是用户界面。**
通过模块化 modularity，可以选择各层，以便每层仅使用较低层的功能（操作）和服务

逻辑分层：

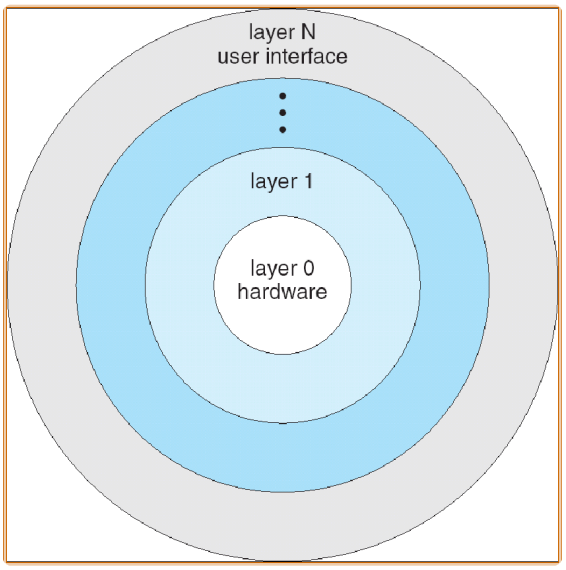


Figure 12

逐层调用（不能跨层调用）、逐层封装——层之间通讯的效率问题

优点：

缺点：require more overhead for iter-layer communication

2.5.7 Modules 模块化

和分层相似但更灵活、可拓展性 scal，但有模块兼容性问题

- 使用目标导向的方法
- 每个模块再需要时可以加载到内核中（loadable）
- 每个模块核心组件隔离
- 模块相互调用（通过已知的接口）而不是消息传递

LKMs(loadable kernel modules): extend the functionality of the kernel dynamically

优点：

缺点：

2.5.8 Other Structures

Exokernel 外核 (1994)：高度简化kernel，只负责资源分配，提供了低级的硬件操作，必须通过定制library供应用使用
高性能，但定制化library难度大，兼容性差

Unikernel: statically linked with the OS code needed.Good for cloud service, APP boots in tens of ms.

2.6 虚拟机 Virtual Machines

A virtual machine takes the layered approach to its logical conclusion. It treats *hardware and the operating system kernel* as though they were all **hardware** 都看作硬件

A virtual machine provides an interface identical to the underlying bare hardware

The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory

2.6.1 HyperVisor

TYPE1 Bare-Metal 裸金属 architecture

- 更高的性能、和硬件直接接触、安全性更高

TYPE2 Hosted 宿主 需要hostOS，eg VMware、Oracle

- 应用简单、和现有OS兼容性高、适合测试和开发环境

选2不选1的常见原因：easier integration with existing host OS

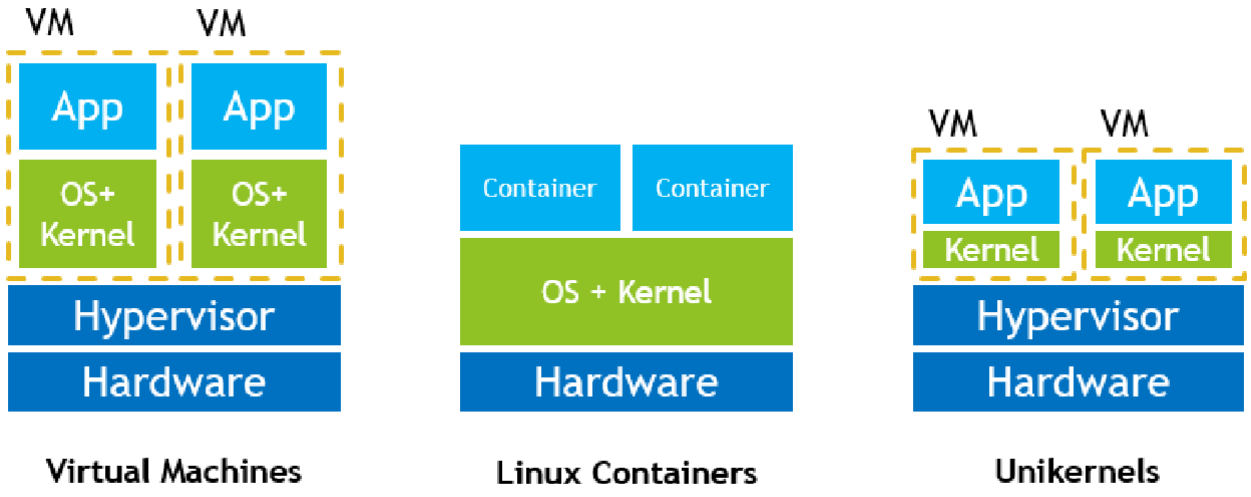


Figure 13

三种技术：

VM：更好的隔离性、灵活性，有资源消耗（复杂）的问题

LC：更加轻量级，不需要加载内核os（docker）

Uk：所有东西都打包，只能允许一个app运行

2.7 OS生成

2.8 系统启动 System Boot

3. 进程 Processes

3.1 进程概念

An operating system executes a variety of programs:

- Batch system – a batch is a sequence of jobs
- Time-sharing systems – user programs or tasks
  - Multitasking
  - Less turnaround, less CPU idle, user interaction

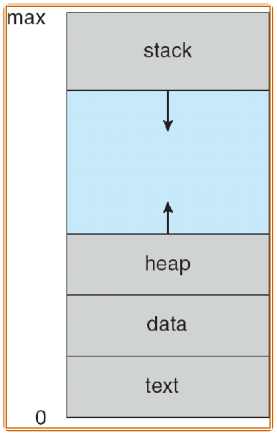
> **Note**
> Textbook uses the terms *job* and *process* almost interchangeably

3.1.1 定义

进程是运行中的程序，按照指令流顺序进行

一个进程包含

- text section (code)
- data section (global vars)
- stack (function parameters, local vars, return addresses)
- heap (dynamically allocated memory)
- program counter



stack和heap相对而生：灵活应用内存空间

Figure 14

3.1.2 Process state

- new
- running
- ready：等待被分配给一个处理器 processor（已经被加载进内存memory
- waiting/block
- terminated：结束进程

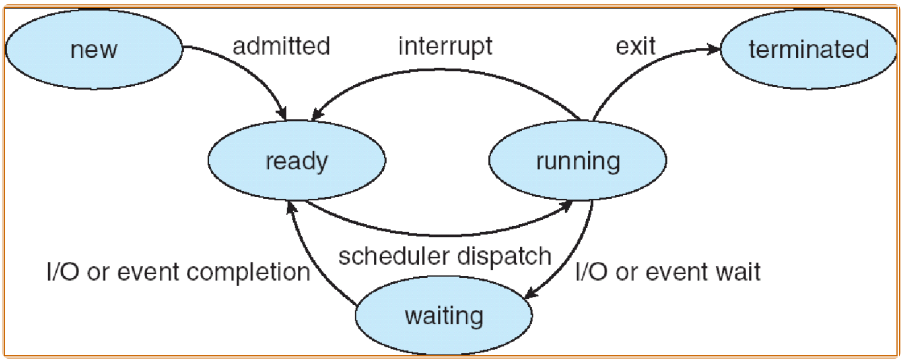


Figure 15

3.1.3 Process control block(PCB)

- OS数据结构，用来跟踪进程和相关资源
- 记录 进程状态、PC、寄存器值、CPU调度、内存管理信息、accounting信息、I/O信息
- Process ID 是进程的独特编号
- PCB中的PC是记录程序目前正在运行的位置

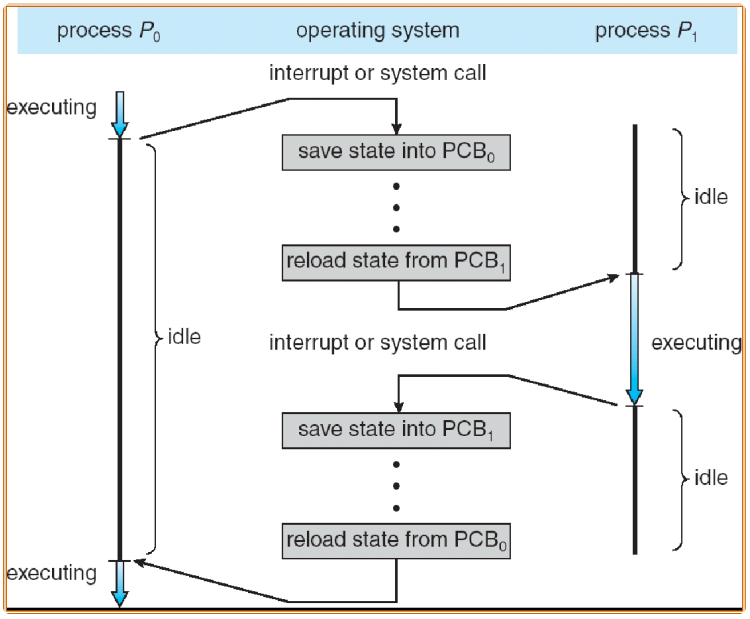


Figure 16

Context Switch上下文切换的overhead

> 分配设备不需要创建新进程
> 用户登录成功、启动程序执行需要创建新进程

3.2 进程调度 scheduling

3.2.1 Process Scheduling Queues

CPU Switch From Process to Process

Job queue – set of all processes in the system（包括ready和正在执行的

Ready queue – set of all processes residing in *main memory*, ready and waiting to execute（等待CPU

Device queues – set of processes waiting for an I/O device 有很多设备（申请使用device——通过系统调用

Processes migrate among the various queues

> **Important**
> 一个PCB能不能在两个队列（device&ready）里同时排队？- 不可能

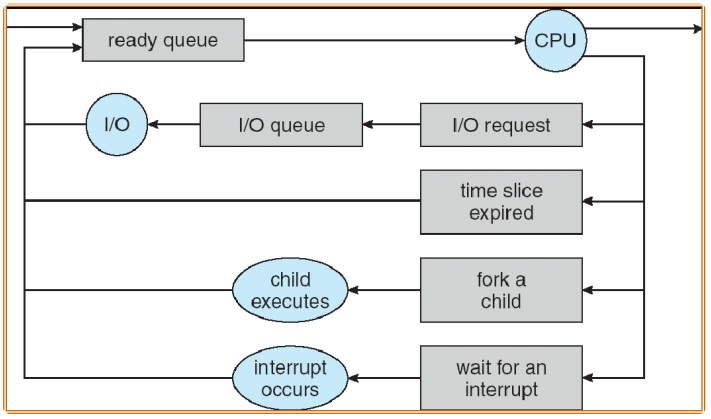


Figure 17

> **Note**
> 我们是把PCB放到进程队列里；
> CPU有四个核可以有4个队列，也可以只有1个队列

3.2.2 Schedulers 调度器

调度器是程序的一部分，A scheduler is a piece of program

Long-term scheduler (or job scheduler) – selects which processes should be brought into memory (the ready queue)

长期调度控制多道程序设计(并发) degree of multiprogramming

Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

- The long-term scheduler is invoked very infrequently (seconds, minutes) => (may be slow)
- The short-term scheduler is invoked very frequently (milliseconds) => (must be fast)

Medium Term Scheduling：进程在内存和磁盘间交换 swap in and out

> **Warning**
> UNIX and Windows do not use long-term scheduling

交换运行程序到磁盘（....?

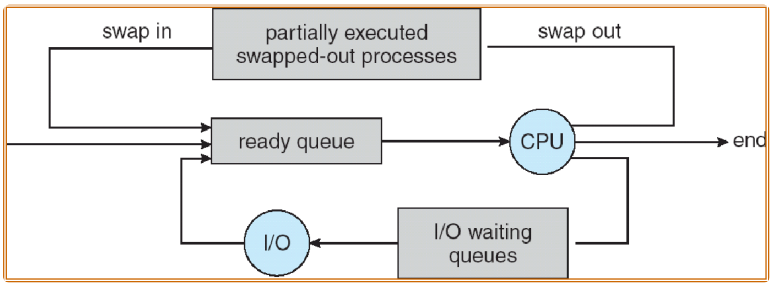


Figure 18

Processes can be described as either:

- **I/O-bound process** I/O绑定进程 – spends more time doing I/O than computations, many short CPU bursts，eg：下载
- **CPU-bound process** CPU绑定进程 – spends more time doing computations; few very long CPU bursts，eg：科学计算

3.2.3 Context Switch 上下文切换

当CPU切换到另一个进程时，系统必须保存旧进程的状态，并为新进程加载保存的状态

上下文切换的时间是开销 overhead；系统在切换时没有任何有用的工作，通常需要几毫秒

时间取决于硬件支持；在SPARC架构中，提供了寄存器组

- 上下文切换时PCB不必记录进程优先级，优先级是内核kernel判断的

3.3 进程操作

3.3.1 Process Creation

父进程创造子进程 fork

- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child *has a program loaded into it*
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a fork to replace the process's memory space with a new program

> 父进程和子进程可以并发执行
> 父进程和子进程有各自的空间，不共享虚拟地址
> 父进程和子进程有不同的进程控制块（PCB）-> 隔离

父进程和子进程不能同时使用

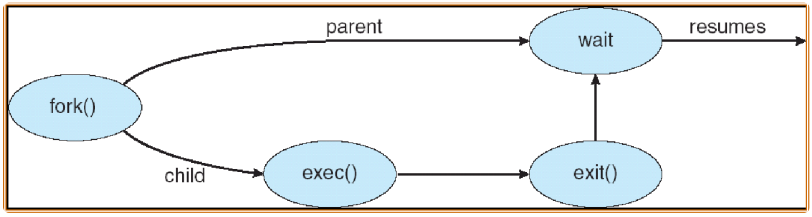


Figure 19

> **Note**
> 子进程可以**继承**父进程的内容；
> 需要重置 pid，cpu time，优先级，堆栈
> list of open files？？

OS不是真的copy，显式copy

> **Caution**
> 创造进程的新线程不是通过fork

3.3.2 Process Termination

3.4 共同协作 cooperating

3.4.1 Cooperating Processes

Independent process 独立进程 cannot affect or be affected by the execution of another process

Cooperating process can affect or be affected by the execution of another process

Advantages of process cooperation:

- Information sharing
- Computation speed-up (Multiple CPUs)
- Modularity
- Convenience

3.4.2 Producer-consumer

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* 无界 places no practical limit on the size of the buffer. Consumer has to wait if no new item.
  - *bounded-buffer* 有界 assumes that there is a fixed buffer size. Producer must wait if buffer full.

Figure 20

主要目的是在生产者线程和消费者线程之间进行数据的**同步**操作

```
//Shared data
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;


//Bounded-Buffer – Insert() Method
//Producer pseudo-code:
while (true) {
    Produce an item;
    while (((in + 1) % BUFFER_SIZE   == out); /* do nothing -- no free
buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}

//Bounded Buffer – Remove() Method
//Consumer pseudo-code:
while (true) {
    while (in == out); //do nothing, nothing to consume

    Remove an item from the buffer;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    return item;
}
```

Fence 1

How many items?

- (in-out) % BUFFER_SIZE
- 或者加一个变量count

3.5 进程间通信

3.5.1 Interprocess Communication (IPC)

Two models for IPC: **message passing**消息队列 and **shared memory**共享内存

通信模型：

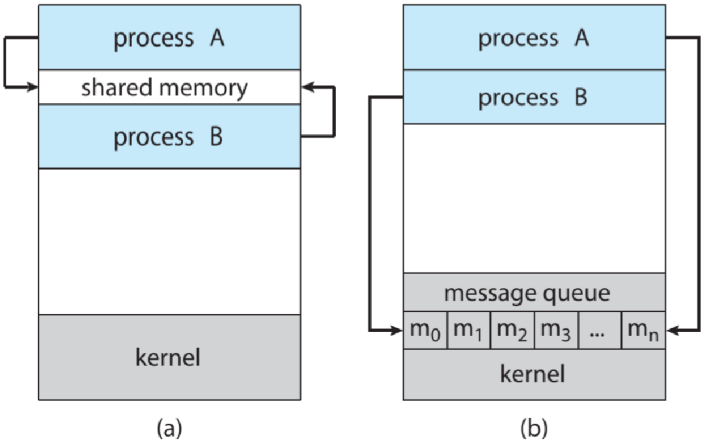


Figure 21

共享内存的主要优点：数据一致性

In IPC, minimize shared resources can reduce conflicts

3.5.2 Direct Communication

explictly显式的

A link is associated with exactly one pair of communicating processes

Between each pair there exists exactly one link

3.5.3 Indirect Communication

mailbox(ports)——both synchronous and asynchronous communication

A link may be associated with many processes

Each pair of processes may share several communication links

3.5.4 Synchronization 同步性

send 是一个系统调用

Blocking is considered **synchronous**

- Blocking send has the sender blocked until the message is received
- Blocking receive has the receiver block until a message is available

Non-blocking is considered **asynchronous**

- Non-blocking send has the sender send the message and continue
- Non-blocking receive has the receiver receive a valid message or null

3.6 Communication in Client-Server Systems

3.6.1 Sockets

4. 线程 Threads

4.1 overview

程序内部隔离和调度

4.1.1 Single and Multithreaded Processes

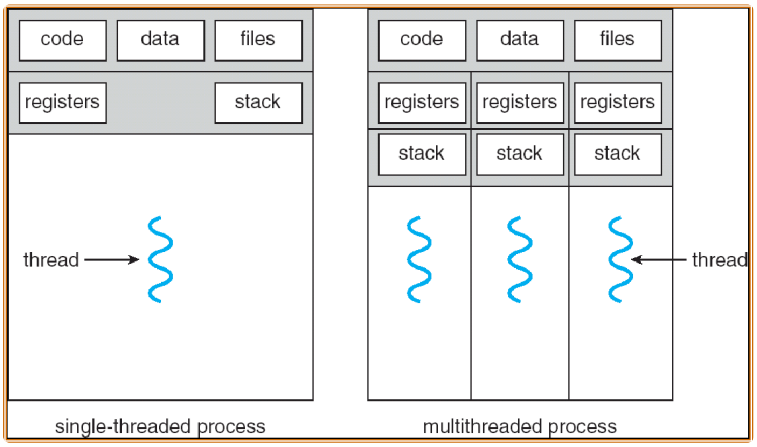


Figure 22

cpu调度的最小（基本）单位：线程 thread

资源分配的最小（基本）单位：进程 process

4.1.1.1 进程和线程的区别

	进程	线程
调度	不再是调度的最小单位（引入内核线程后）	是调度的最小单位
拥有资源	进程都是资源分配的基本单位	线程只拥有少量资源
并发性	多个进程间可以并发执行	一个线程的多个线程也能并发执行
独立性	只共享全局变量	只有少数资源不能共享，例如线程的栈区
系统开销	通信、进程切换开销大	通信、切换开销小
其他	进程间相互不影响	用户级线程的阻塞会影响整个进程

Figure 23

4.1.2 Benfits

Responsiveness：interactive applications

Resource Sharing：memory for code and data can be shared.

Economy：creating processes are more expensive.

Utilization of MP Architectures 多处理器架构：multi-threading increases concurrency 可拓展性

4.1.3 User Threads

Thread management done by user-level threads library

Three primary thread libraries: POSIX Pthreads (can also be provided as system library)，Win32 threads， Java threads —— 都可能会产生？？？

一个用户级线程只能映射到一个内核级线程

对于用户级线程，内核并不知情

4.1.4 Kernel Threads

Almost all contemporary OS implements kernel threads.

Kernel level threads：支持调度，便于线程切换，避免阻塞；执行的载体

User level threads：；逻辑的载体

内核级线程所需要的资源是以进程为单位申请的

用户级线程的切换通过线程库，不需要内核的支持，即线程库为用户级线程建立一个线程控制块
内核级（系统级）线程的调度由操作系统完成

用户级线程间的切换比内核级切换效率高
用户级线程可以在不支持内核级线程的操作系统上实现

Note
多线程的特长：提高并发性，例如矩阵乘法、web服务器响应HTTP
多线程不共享栈指针
同一进程中的各个线程有相同的地址单位

4.2 Multithreading Models

4.2.1 Many-to-one Model

一个用户级线程卡住了，整个进程就卡住了

thread management is efficient, but will block if making system call, kernel can schedule only one thread at a time

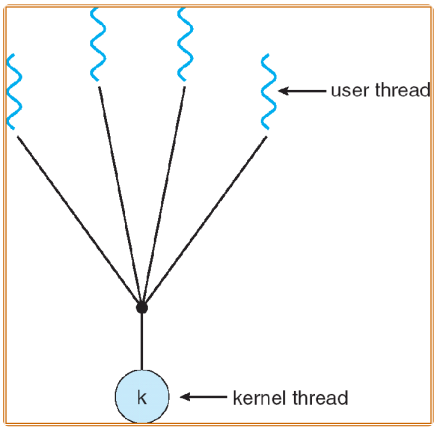


Figure 24

Q：某个分时系统采用**多对一**线程模型。内存中有10个进程并发运行，其中9个进程中只有一个线程，另外一个进程A拥有11个线程。则A获得的CPU时间占总时间的 1/10

4.2.2 One-to-one

Each user-level thread maps to kernel thread

more concurrency, but creating thread is expensive

能让线程并行，每个用户级线程都有一个内核级线程

> Q：某个分时系统采用**一对一**线程模型。内存中有10个进程并发运行，其中9个进程中只有一个线程，另外一个进程A拥有11个线程。则A获得的CPU时间占总时间的 11/20

4.2.3 Many-to-Many Model

flexible，LWP ID kernel根据情况看调度

Allows many user level threads to be mapped to many kernel threads

Allows the operating system to create a sufficient number of kernel threads

4.2.4 Two-level Model

Similar to M:M, except that it allows a user thread to be bound to kernel thread

4.3 Threading Issues

Semantics语义 of fork() and exec()

> Does fork() duplicate only the calling thread or all threads?
>
> - Some unix systems have two versions of fork(), one that **duplicates all threads** and another that **duplicates the thread** that invokes fork().
> - Exec() will replace the entire process.

4.3.1 Thread Cancellation

Asynchronous cancellation异步 terminates the target thread immediately

Deferred cancellation 延后 allows the target thread to periodically check via a flag if it should be cancelled

4.3.2 Signal Handling

A **signal handler** is used to process signals, either synchronous or asynchronous:

- Signal is generated by particular event
- Signal is delivered to a process
- Signal is handled

4.3.3 Thread Pools 线程池

Create a number of threads in a pool where they await work

Advantages

- Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool

Thread-Local Storage（TLS）：Allows each thread to have its own copy of data

4.3.4 Scheduler Activations 调度激活

LWP is a virtual processor attached to kernel thread

Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library

Upcalls are handled by the thread library with an **upcall handler**.

This communication allows an application to maintain the correct number of kernel threads

When an application thread is about to block, an upcall is triggered.

5. 调度 CPU Scheduling

5.1 Basic concepts

scheduler dispatch

调度资源？CPU

调度目标？Processes/threads

Goal: 在multiprogramming下CPU使用率最大化

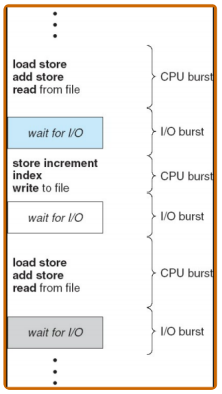


Figure 25

CPU-I/O Burst Cycle：Process execution consists of a cycle of CPU execution and I/O wait

进程实际运行时CPU占用时间少，I/O多

大部分CPU burst时间非常短

5.1.1 CPU Scheduler

CPU scheduling decisions may take place when a process:

1. When a process switches from the running state to the waiting state *(e.g. I/O request or an invocation of wait())*
2. When a process switches from the running state to the ready state *(e.g., when an interrupt occurs)*
3. When a process switches from the waiting state to the ready state *(e.g., at completion of I/O)*
4. When a process terminates

Figure 26

nonpreemptive 非抢占式调度 1 & 4

preemptive ✔ 2 & 3

5.1.2 Dispacher

dispacher latency：P1停止运行到P1运行

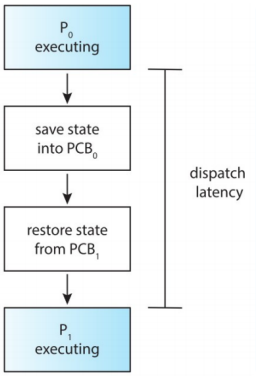


Figure 27

5.2 scheduling criteria

CPU utilization CPU利用率： **CPU 使用率= (1 - 空闲态运行时间/总运行时间) * 100%**

Throughtput 吞吐率： **进程数/总执行时间**

Turnaround time 周转时间 进程提交到结束

waiting time 等待时间

response time 响应时间

- Max ■ CPU utilization (CPU 利用率) – keep the CPU as busy as possible
- Max ■ Throughput (吞吐率)– # of processes that complete their execution per time unit
- Min ■ Turnaround time (周转时间)– amount of time to execute a particular process, include what?
- Min ■ Waiting time (等待时间)– amount of time a process has been waiting in the ready queue
- Min ■ Response time (响应时间)– amount of time it takes from when a request was submitted until the *first* response is produced, **not** output (for time-sharing environment)

Not to confuse with 'waiting' state

Figure 28

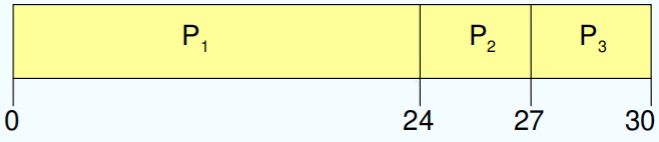
5.3 调度算法 scheduling algorithms

5.3.1 FCFS(First-Come, First-Served)

先来先服务算法

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Turnaround time: P_1 = 24; P_2 = 27; P_3 = 30
- Throughput: 3/30

Figure 29

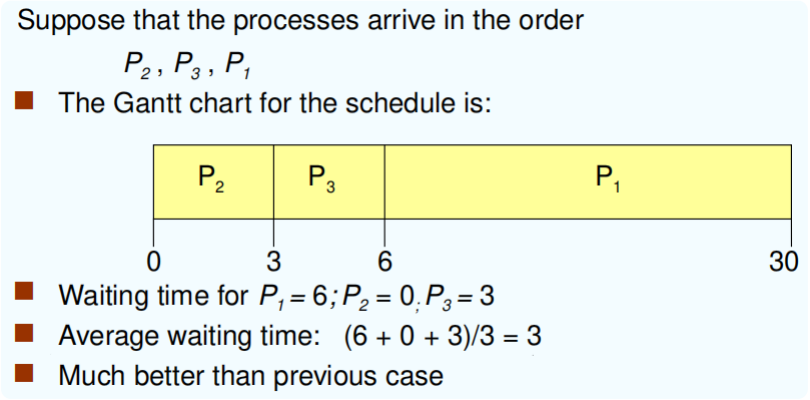


Figure 30

> **Important**
> Arrival order makes a difference!
> 到达顺序很重要

- FCFS是**非抢占式**算法
- FCFS简单、公平
- **Convoy effect**（护航效应）：short process behind long process, the average waiting time may be longer, leading to I/O devices & CPU being idle

有利于长作业，不利于短作业；有利于CPU繁忙型，不利于I/O繁忙型

> 短作业位于长作业后时调度时间要很长；I/O繁忙会有很多waiting，不断排队到ready queue队尾，如果排到长作业后面就要很长时间

5.3.2 SJF(Shortest-Job-First)

短作业优先算法

- SJF is a priority scheduling where priority is the predicted next CPU burst time
- SJF既可以是抢占式也可以是非抢占式
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the **Shortest-Remaining-TimeFirst (SRTF)**最短剩余时间调度算法
- SJF is **optimal** – gives **minimum average waiting time** for a given set of processes
- Which is better? Preemptive? Nonpreemptive? 不确定，要根据到达时间和cpu burst确定
- Starvation 饥饿问题，优先级低的进程一直无法运行，短作业都会发生
- Not good for long process 不适合长进程，因为优先调度cpu burst最短的进程

5.3.2.1 nonpreemptive

Figure 31

5.3.2.2 preemptive

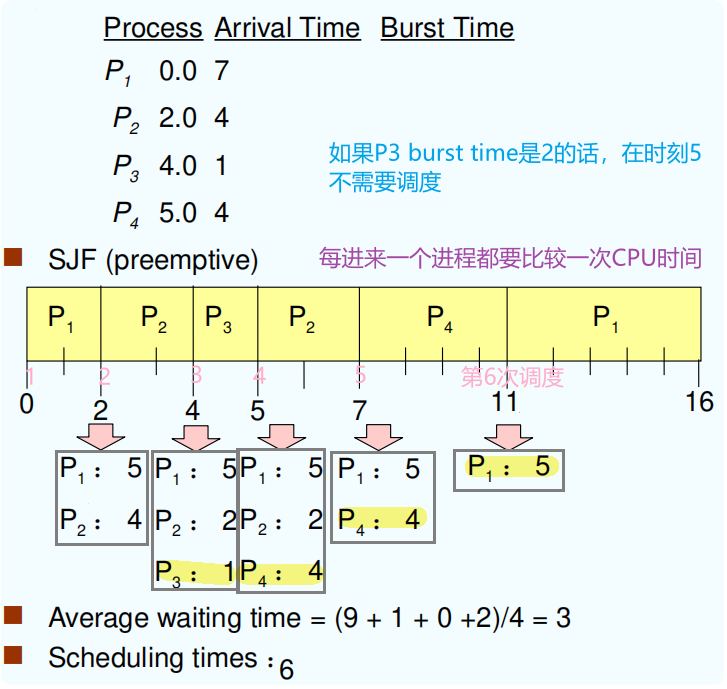


Figure 32

5.3.3 Priority Scheduling

优先级调度算法

The CPU is allocated to the process with the highest priority (smallest integer means highest priority)

- Preemptive
- nonpreemptive

Static Priority: determine when processes is created; do not change 静态优先

Problem： **Starvation** – low priority processes may never execute

Solution： **Aging** （老化）– as time progresses increase the priority of the process——**Dynamic Priority** 提高优先级

5.3.4 Highest Response Ratio Next （HRRN)

高响应比优先算法

- 非抢占式
- HRRN is a **compromise** 折中 between FCFS and SJF
- Computing response ratio requires time
- improve the responsiveness of the system 提高系统响应性

Response Ratio （响应比） = $\frac{\text{cpu burst}+\text{waiting time}}{\text{cpu burst}}$ （等待时间和执行时间都要考虑

Take Response Ratio as priority

Larger Response Ratio, higher priority

5.3.4.1 HRRN 算法的工作原理

1. **初始化**： 记录每个进程的到达时间和所需的服务时间。
2. **计算响应比**： 对于每个就绪队列中的进程，计算其响应比。
3. **选择最高响应比的进程**：
    1. 选择具有最高响应比的进程执行。
    2. 如果有多个进程具有相同的最高响应比，则可以按照其他规则（如先到先服务）来选择。
4. **更新等待时间和响应比**：
    1. 每次调度后，更新所有就绪队列中进程的等待时间。
    2. 重新计算所有进程的响应比

5.3.5 Round Robin (RR)

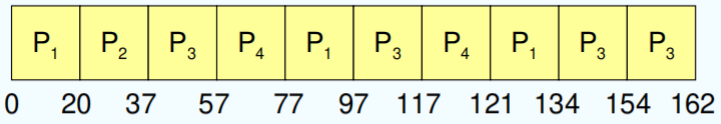
时间片轮转算法

- Origin from signature method
- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue. 提高系统交互性
- Application: Time-sharing system, Multi-tasking system
- 当前进程的时间片用完后，该进程的状态由执行态变成就绪态

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- What are the waiting times of RR?
- Typically, higher average turnaround than SJF, but better *response*

CPU utilization and Interactivity may not be satisfied at the same time

Figure 33

- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large ⇒ FCFS
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high
- Application: Time-sharing system, Multi-tasking system

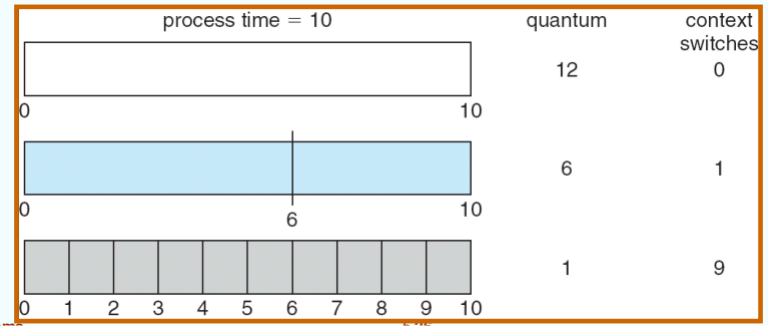


Figure 34

5.3.6 Multilevel Queue

多层队列算法

Ready queue is partitioned into separate queues:

- foreground (interactive)
- background (batch)

Each queue has its **own scheduling algorithm**, for example

- foreground – RR

- background – FCFS 计算密集

Scheduling must be done between the queues

- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR， 20% to background in FCFS

5.3.7 Multiple Feedback Queue

多级反馈队列

- 多个就绪队列，优先级逐一降低，按照队列优先级调度
- 设置多个就绪队列，优先级从第一级依次降低
- 优先级高的队列，进程时间片越短
- 每个队列都采用 FCFS ，若在该时间片完成，则撤离系统，未完成，转入下一级级队列
- 按队列优先级调度，仅当上一级为空时，才运行下一级

进程进来，先到优先级最高的队列中，时间片内完成则撤离；否则，到下一级队列排队

5.3.8 Multiple-processor scheduling(了解)

多处理器调度

Load balancing 负载均衡

Symmetric multiprocessing(SMP) 对称处理器：每个处理器都是自我调度，多个处理器可以运行、更新一个相同的数据结构

Asymmetric multiprocessing 非对称处理器：只有一个处理器可以接触系统数据结构（调度），减轻数据共享，其他处理器只执行用户代码

- 当调度的处理器坏了，整个系统就运行不下去了

5.3.9 Real-time scheduling(了解)

实时调度

实时系统一定要在规定时间内完成

Hard real-time systems 硬实时系统：ddl前没运行完有严重后果

Soft real-time systems 软实时系统：可以在ddl前没运行完（eg，腾讯会议

- Earliest ddl First 最早截止时间优先

- Least Laxity First 最低松弛度优先（不怎么用）
  - A的松弛度=A必须完成的时间-A需要运行的时间-当前时间
- Rate Monotonic scheduling 速率单调调度
  - 基于任务的周期（一个进程多久执行一次）来分配优先级，周期越短任务优先级越高

5.3.10 Thread scheduling(了解)

线程调度

Local scheduling 用户级

Global scheduling 内核级

> **Note**
> 批处理 FCFS
> 分时系统
> 实时系统

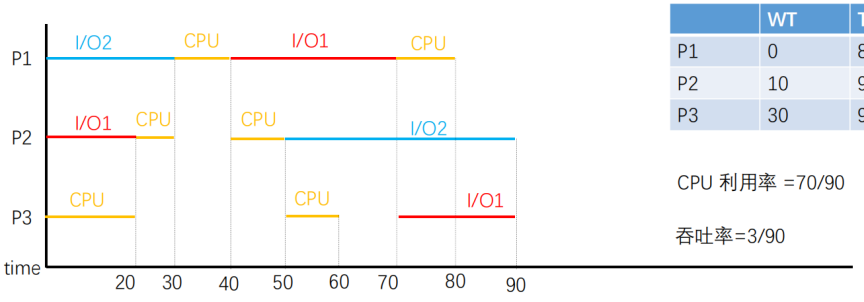
5.4 Q

- 某多道程序系统配有一台处理器和两台外设 101、102，现有3个优先级由高到低的进程P1、P2 和 P3 已经在ready queue中，它们使用资源的先后顺序和占用时间分别是

  P1: I/O2(30ms)； CPU(10ms)； I/O1(30ms)； CPU(10ms)

  P2: I/O1(20ms)； CPU(20ms)； I/O2(40ms)

  P3: CPU(30ms)； I/O1(20ms)
- 处理器调度采用可抢占式的优先数算法，忽略其他辅助操作时间，回答下列问题:



	WT	TT
P1	0	80
P2	10	90
P3	30	90

CPU 利用率 =70/90

吞吐率=3/90

Figure 35

外设是不可抢占的
waiting time是进程等待CPU资源的时间!

某进程调度程序采用基于优先数（priority）的调度策略，即选择优先数最小的进程运行，进程创建时由用户指定一个nice作为静态优先数。为了动态调整优先数，引入运行时间cpuTime和等待时间waitTime，初值均为0。进程处于执行态时，cpuTime定时加1，且waitTime置0；进程处于就绪态时，cpuTime置0，waitTime定时加1。请回答下列问题。

（1）若调度程序只将nice的值作为进程的优先数，即priority=nice，则可能会出现饥饿现象，为什么？

当队列里有个优先级低的进程。后面进来的进程都比它高，它一直在ready queue里排队，得不到运行

（2）使用nice、cpuTime和waitTime设计一种动态优先数计算方法，以避免产生饥饿现象，并说明waitTime的作用。

priority = nice + CPU time - wait time 或 priority = (nice + CPU time) / wait time
前者更好，加减法更快

Figure 36

6. 进程同步 Process synchronization

> **Note**
> 基本概念：syn，race，critical，four requirements（忙则等待、、、）
> 软件实现方法：单标志，双标志、、、
> 硬件实现方法：原理、关中断、Test、Swap、CAS、mutex lock
> 信号量机制：信号量基本概念（形式、类型、用法、实现、缺点）三个经典问题（有界缓冲区、读写者、哲学家）
> 管程：基本概念（形式、特点、条件变量）、应用

6.1 背景 background

共享数据的并发访问可能导致数据不一致性

Producer-consumer

- Employing an integer count to keep track of the number of full buffers.
- Initially, count is set to 0.
- It is incremented by the producer after it produces a new buffer
- It is decremented by the consumer after it consumes a buffer.
- Producer and consumer perform concurrently.

Producer

```
while (true) {
/* produce an item and put in
   nextProduced */
      while (count == BUFFER_SIZE)
            ; // do nothing
      buffer [in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
}
```

Consumer

```
while (true) {
    while (count == 0)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /* consume the item in nextConsumed
}
```

Figure 37

count++和count--两步有可能出错

6.1.1 Race condition 竞态条件

出错的example：

- count++ could be implemented as
  register1 = count
  register1 = register1 + 1
  count = register1
- count-- could be implemented as
  register2 = count
  register2 = register2 - 1
  count = register2
- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register1 {count = 6 }
  S5: consumer execute count = register2 {count = 4}
- Consider this execution interleaving with "count = 5" initially:
  S0: producer execute register1 = count {register1 = 5}
  S1: producer execute register1 = register1 + 1 {register1 = 6}
  S2: consumer execute register2 = count {register2 = 5}
  S3: consumer execute register2 = register2 - 1 {register2 = 4}
  S4: producer execute count = register2 {count = 4 }
  S5: consumer execute count = register1 {count = 6}

Figure 38

出错的原因：抢占式调度，多个进程对shared data进行操作

Race condition 定义：a memory location is accessed concurrently, and at least one access is a write

> **对于访问共享的内核数据，非抢占式的内核是否受竞态条件的影响?**
>
> - 可能会受影响，当多处理器对shared data进行操作

6.2 临界区问题 critical-section problem

What operations/processes may have critical problems in OS kernel?

- cpu
- 用户态：一个进程的多个线程（满足shared data和并发执行）

临界资源：多进程或多进程中被共享的资源(shared data)且一次只允许一个进程使用

临界区：程序中一个访问公共资源的程序片段，每个进程中访问临界资源的那段代码被称为临界区

> **Note**
> **以下是临界资源吗?**
>
> - 全局共享变量（是），局部变量（不是），只读数据（不是），CPU（不是）

对一个进程，可能存在多个临界区；临界区可以合并（depends）

> **Caution**
> 共享资源才需要互斥

6.2.1 解决方案 solution

6.2.1.1 Mutual Exclusion（互斥/忙则等待）

如果进程 P_i 正在其临界区中执行，则其他进程不能在其临界区中执行

6.2.1.2 Progress（空闲让进）

如果没有任何进程在其临界区中执行，并且存在一些希望进入其临界区的进程，则不能无限期地推迟选择下一个将进入临界区的进程（即允许一个请求进入临界区的进程立即进入临界区）

6.2.1.3 Bounded waiting（有限等待）

在一个进程发出进入其临界区的请求之后，在该请求被批准之前，必须对允许其他进程进入其临界区的**次数**进行限制

- 假设每个进程以非零速度执行
- 没有关于*N*个进程相对速度的假设

6.2.1.4 让权等待

（原则上应该遵守，但非必须）当进程不能进入临界区时，应立即释放处理器，防止进程忙则等待。

6.3 同步机制

6.3.1 软件方法

6.3.1.1 单标志法

- 设置公共整型变量 turn，指示允许进入临界区的进程编号
  turn = i，允许 Pi 进入临界区
- 进程退出临界区时交给另一个进程
  turn = j

Figure 39

i和j交替执行

- 满足Mutual Exclusion 和 bounded waiting
- 不满足progess

6.3.1.2 双标志后检查法

- 设置布尔型数组 flag[2]，用来标记各进程进入临界区的意愿
  flag[i]=true 表示进程 Pi 想进入
- 先表达自己进入临界区意愿
- 再轮询对方是否想进入，确定对方不想进入后再进入
- 访问结束退出后设置 flag[i]=false，表示不想进入，允许对方进入

Figure 40

- 满足Mutual Exclusion
- 不满足Progress（存在CPU调度的一种情况，两个标志都为TRUE后一直循环下去）
- 如果没有死循环是bounded waiting，但可能有所以总体不是

6.3.1.3 双标志先检查法

- 设置布尔型数组 flag[2]，用来标记各进程进入临界区的意愿
  flag[i]=true 表示进程 Pi 想进入
- 进程进入临界区前先轮询对方是否想进入
- 确定对方不想进入后再进入
- 访问结束退出后设置 flag[i]=false，表示不想进入，允许对方进入

Figure 41

和前面两种算法相比，先while再设flag值：

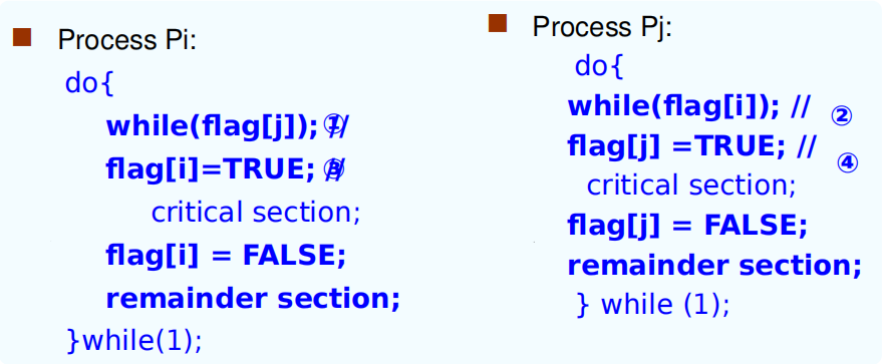


Figure 42

- **不满足Mutual Exclusion**（存在CPU调度的一种情况，两个标志都为TRUE，并进入临界区
- 满足Progres

6.3.1.4 Peterson's Solution

双进程解决方案
假设 LOAD 和 STORE 指令是原子的；即不能被中断。
两个进程共享两个变量：`int turn;Boolean flag[2];`
变量`turn`表示轮到谁进入临界区。
flag 数组用于指示进程是否已准备好进入临界区。`flag[i] = true`表示进程 P_i 已准备好

基本思想：

- 结合单标志法和双标志后检查法，首先表达自身意愿(flag[]=true)之后设置自身要进入(turn=0/1)；
- 若双方互相确定对方都想进入时，turn 只能等于一个值，因此会谦让对方进入
- 若一方不想进入，则其 flag[i]=false，对方可直接进入

```
//Process Pi
while (true) {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
    CRITICAL SECTION
        flag[i] = FALSE;
    REMAINDER SECTION
}

//Process Pj
while (true) {
    flag[j] = TRUE;
    turn = i;
    while ( flag[i] && turn == i);
    CRITICAL SECTION
        flag[j] = FALSE;
    REMAINDER SECTION
```

```
19  }
```

Fence 2

- 满足mutual exception 和 progress，不会
- 满足bounded waiting，bound是1

> **Caution**
> There are no guarantees that Peterson's solution works correctly on modern computer architectures.
> Reason：计算机编译优化 -> 代码的乱序执行【先load（读），再store（赋值）】
> Solution：Memory barrier (内存栅栏) -- 插入这条指令后就不会乱序执行了

6.3.1.5 Bakery Algorithm (面包房算法) Lamport

Multiple-Process Solutions： for **n** processes

1. 任何时间，最多只能有一个进程进入 critical section ；
2. 每个进程最终都会进入 critical section ；
3. 每个进程都能停在 noncritical section ；
4. 不能对进程的速度做任何假设。

基本思想（排队取号）：

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section. 最小的号进入临界区
- If processes Pi and Pj receive the same number, if i < j, then Pi is served first; else Pj is served first. 小的号先服务
- The numbering scheme always generates numbers in increasing order of enumeration; e.g., 1, 2, 3, 3, 3, 3, 4, 5. 以递增顺序（非递减）产生号

- boolean choosing[n]: 表示进程是否在取号；初始 false 。
- int number[n]: 记录每个进程取到的号码；初始 0 。
- （ a， b ）＜（ c， d ）: (1) a ＜ c, or (2) a==c 且 b ＜ d

```
//Multiple-Process Solutions
do{
    choosing[i] = true;
    number[i] = max{number[0], number[1], …, number[n-1]}+1; // 选号码
    choosing[i] = false;
    for(j = 0; j < n; j++){
        while (choosing[j]);
        while ((number[j] != 0) && (number[j], j) < (number[i], i));
    };
    CRITICAL SECTION
        number[i] = 0;
    REMAINDER SECTION
} while(1);
```

Fence 3

- 满足mutual exception 和 progress，
- 满足bounded waiting，bound为 n-1

6.3.2 硬件方法

Synchronization Hardware

Modern machines provide special atomic hardware **instructions**

-> Atomic = non-interruptable (原子操作，不能中断；后三种方法都用了这一思想)

优点

- 适用于任意数目的进程，在单处理器或多处理器上
- 简单，容易验证其正确性
- 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量

缺点

- 耗费 CPU 时间，不能实现“让权等待”
- 可能不满足有限等待：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
- 可能死锁

6.3.2.1 关中断法 Disable interrupts(中断屏蔽法)

思想：进入临界区前直接屏蔽中断，保证临界区资源顺利使用；使用完毕，打开中断

```
while (true) {

    Disable Interrupts;
    Critical section;
    Enable Interrupts;
    Remainder section;

}
```

Fence 4

6.3.2.1.1 缺点

- **可能影响系统效率**：滥用关中断会严重影响 CPU 执行效率，其锁住CPU可能导致原本一些短时间即可完成的需要等待开中断，影响cpu并发执行，cpu利用率下降
- **不适用于多 CPU 系统**：中断屏蔽法适用于单 CPU 系统，在多 CPU 系统中**无法有效同步**各个 CPU 的操作。
- **安全性问题**：滥用关中断权力可能导致严重后果，例如在关闭中断期间，一些重要的中断请求可能被错过，影响系统的稳定性和可靠性。

6.3.2.2 TestAndSet Lock Instruction（TSL）

Test and modify the content of a word atomically.

Shared boolean variable lock, initialized to false.

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
while (true) {
    while ( TestAndSet (&lock ));/* do nothing*/
    // critical section
    lock = FALSE;
    // remainder section
};
```

Fence 5

- 满足mutual exclusion, progress
- 不满足bounded waiting（进入临界区靠运气），不满足让权等待
- 等待进入临界区的进程不会主动放弃CPU

6.3.2.3 Swap Instruction

思想

1. 对每个临界资源，swap设置一个全局 `bool` 变量 `lock` (初值为false)，每个进程设置局部变量 `key` (初值为 true)
2. 进程调用 `swap()` 指令访问临界区，会交换key和lock的值，实现上锁，进入访问
3. 退出时把 `lock` 重置为 `false`

```
while (true) {
    key = TRUE;
    while (key = =TRUE)
        Swap(&lock, &key) ;

    // critical section
    lock = FALSE;
    // remainder section
}

void Swap(boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Fence 6

- 满足mutual exclusion, progress
- 不满足bounded waiting（

6.3.2.4 The compare_and_swap (CAS) Instruction

思想

1. Executed atomically
2. Returns the original value of passed parameter **value**
3. Set the variable **value** the value of the passed parameter **new_value** but only if `*value == expected` is true. That is, the swap takes place only under this condition.

```
//Shared integer lock initialized to 0;
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0) ; /* do nothing */
    /* critical section */
    lock = 0;
```

```
    /* remainder section */
}
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

Fence 7

- 满足mutual exclusion, progress
- 不满足bounded waiting（

解决办法：排队

6.3.2.5 Bounded-waiting with compare-and-swap

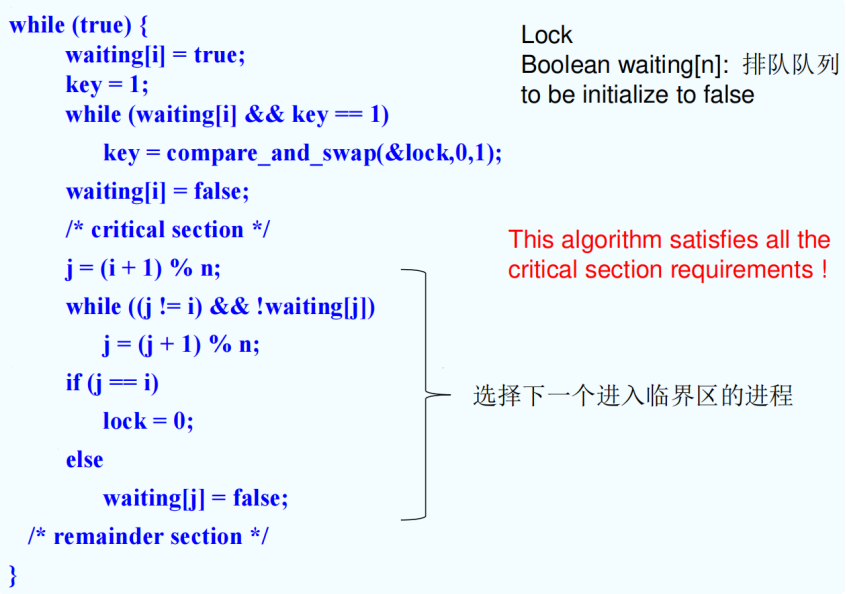


Figure 43

- 满足mutual exclusion, progress和bounded waiting

> **Caution**
> 以上对TS和Swap指令的描述仅为功能描述，它们由硬件逻辑实现，不会被中断

6.3.3 互斥锁 Mutex locks

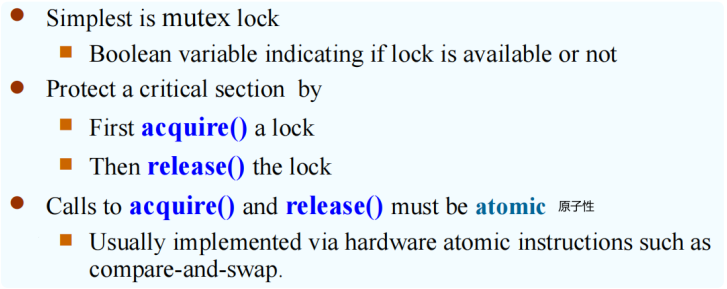


Figure 44

But this solution requires **busy waiting （不停空循环）**

This lock therefore called a **spinlock （也叫 自旋锁）**

- 主要采用硬件机制来实现
- 缺点：忙等待

6.3.4 信号量方法 Semaphores

用来解决同步和互斥问题

Two indivisible operations modify S:

- `wait()` and `signal()`, originally called `P()` and `V()`
- Proberen(测试)，Verhogen(增加)
- 只能在wait和signal中对信号量进行操作
- 低级的进程通信原语

Can only be accessed via two indivisible (atomic) operations:

```
wait (S) { //资源进入临界区
    while S <= 0; // no-op，S <= 0：有进程在临界区
    S--;
}
signal (S) { //资源退出临界区，空出资源给其他进程
    S++;
}
```

Fence 8

Can be implemented without busy waiting --> 实现让权等待

> **Important**

- 通常用信号量表示资源或临界区
- 信号量的物理含义
- S.value >0 表示有 S.value 个资源可用；
- S.value=0 表示无资源可用或表示不允许进程再进人临界区；
- S.value<0 则 |S.value| 表示在等待队列中进程的个数或表示等待进入临界区的进程个数。

- wait 、 signal 操作必须成对出现，有一个 wait 操作就一定有一个 signal 操作一般情况下：当为互斥操作时，它们同处于同一进程；当为同步操作时，则不在同一进程中出现。
- 如果两个 wait 操作相邻，那么它们的顺序至关重要，而两个相邻的 signa 操作的顺序无关紧要。一个同步 wait 操作与一个互斥 wait 操作在一起时，同步 wait 操作在互斥 wait 操作前。互斥wait在前可能会导致死锁

Figure 45

S.value=0 已经有一个进程在临界区

信号量种类：

- Counting semaphore （计数型） - integer value can range over an unrestricted domain
- Binary semaphore （二进制型） - integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore S;  // initialized to 1
wait (S);
Critical Section
signal (S);
Remainder Section
```

Figure 46

👆互斥访问

同步操作：

- P1 has a statement S1, P2 has S2
- Statement S1 to be executed before S2

P1 S1;
Signal(S);

Question: What's the initial value of S?
S=0

P2 Wait(S);
S2;

同步操作在两个进程中

Figure 47

Question：有四个房间，四个进程访问

Provide synchronized access to:

- Four rooms, four identical keys.
  How many semaphore?

  1 is enough　4个也可以，但会有资源浪费（可能导致效率低

- Four rooms, each with a unique key (four different keys)
  How many semaphore?

  4

  What if using 1 semaphore? (初始值为4)

  导致多个房间之间的访问冲突，无法保证互斥性

Figure 48

6.3.4.1 实现 Semaphore Implementation

6.3.4.1.1 Busy waiting

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Figure 49

Example1: Semaphore Implementation (Busy Waiting)

```
100   struct semaphore {
101     struct spinlock lock;
102     int count;
103   };
104
105   void
106   V(struct semaphore *s)
107   {
108       acquire(&s->lock);
109       s->count += 1;
110       release(&s->lock);
111   }
112
113   void
114   P(struct semaphore *s)
115   {
116       while(s->count == 0)
117         ;
118       acquire(&s->lock);
119       s->count -= 1;
120       release(&s->lock);
121   }
```

To allow one producer and one consumer, running P() and V() on different CPUs

即V()很少运行

If producer rarely acts, consumer spins on line 116. Expensive!!

Figure 50

6.3.4.1.2 no Busy waiting 非忙等

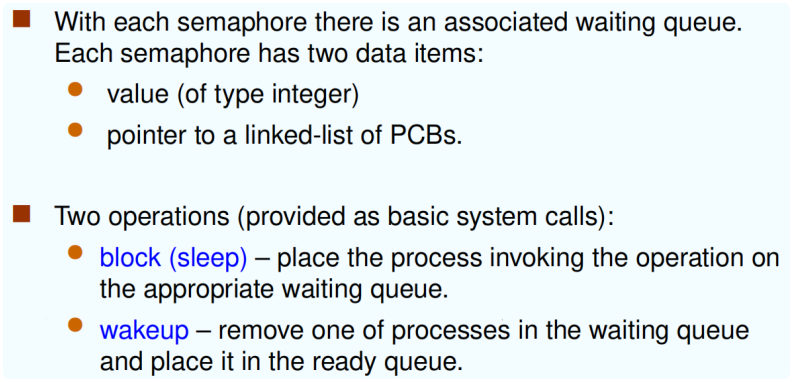


Figure 51

```
/*Implementation of wait*/
wait (S){ //取信号量操作
    value--;
    if (value < 0) {
        // add this process to waiting queue
        block(); // 插到S的waiting queue中
    }
}

/*Implementation of signal*/
Signal (S){
    value++;
    if (value <= 0) {
        // remove a process P from the waiting queue
        wakeup(P); // 把一个进程唤醒，移出waiting queue，到临界区执行
    }
}
```

Fence 9

- S的取值可以是负的了（相比原先的wait和signal），S取负表示当前队列排队进程的个数

Example2: *sleep* and *wakeup* operations

```
200    void
201    V(struct semaphore *s)
202    {
203        acquire(&s->lock);
204        s->count += 1;
205        wakeup(s);
206        release(&s->lock);
207    }
208
209    void
210    P(struct semaphore *s)
211    {
212      while(s->count == 0)
213        sleep(s);
214      acquire(&s->lock);
215      s->count -= 1;
216      release(&s->lock);
217    }
```

- Sleep on wait queue `s`
- Overhead reduced by giving up CPU with sleep(block) operation.
- Unfortunately suffers from the *lost wake-up* problem!
  轮到进程wakeup了但它还没sleep
- P is running between 212-213, V increments count and wakeup(). P then sleeps, waiting for a wakeup() call that has happened already.

运行到212-213调用了wakeup，后面213又sleep，最后丢失wakeup

Figure 52

Moving things under protection of spinlock?

```
300    void
301    V(struct semaphore *s)
302    {
303      acquire(&s->lock);
304      s->count += 1;
305      wakeup(s);
306      release(&s->lock);
307    }
308
309    void
310    P(struct semaphore *s)
311    {
312      acquire(&s->lock);
313      while(s->count == 0)
314        sleep(s);
315      s->count -= 1;
316      release(&s->lock);
317    }
```

- Lost wakeup prevented, but
  Deadlocks!!! Sleep forever
- Solution?

Figure 53

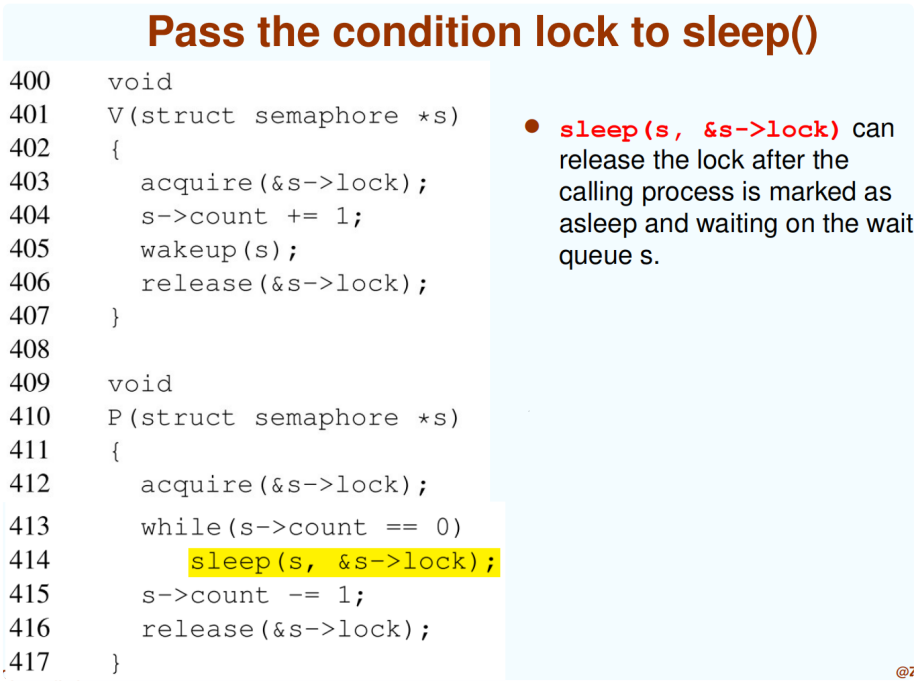


Figure 54

> ⚠ **Warning**
> 进程在标黄处sleep后，再次被唤醒时会从头开始执行!

Busy waiting? No busy waiting?

- No busy waiting

What if one must choose busy waiting?

- 电脑要多CPU
- 上下文切换时间 > busy waiting，选择no busy waiting；<的话两者都行

6.3.4.2 Deadlock and Starvation

Deadlock 死锁 – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Starvation 饥饿 – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

6.3.4.3 Classical Problems of Synchronization

6.3.4.3.1 Bounded-Buffer Problem

N buffers, each can hold one item

- Semaphore **mutex** initialized to the value **1** 互斥信号量1个
- Semaphore **full** initialized to the value 0, counting full items
- Semaphore **empty** initialized to the value **N**, counting empty items

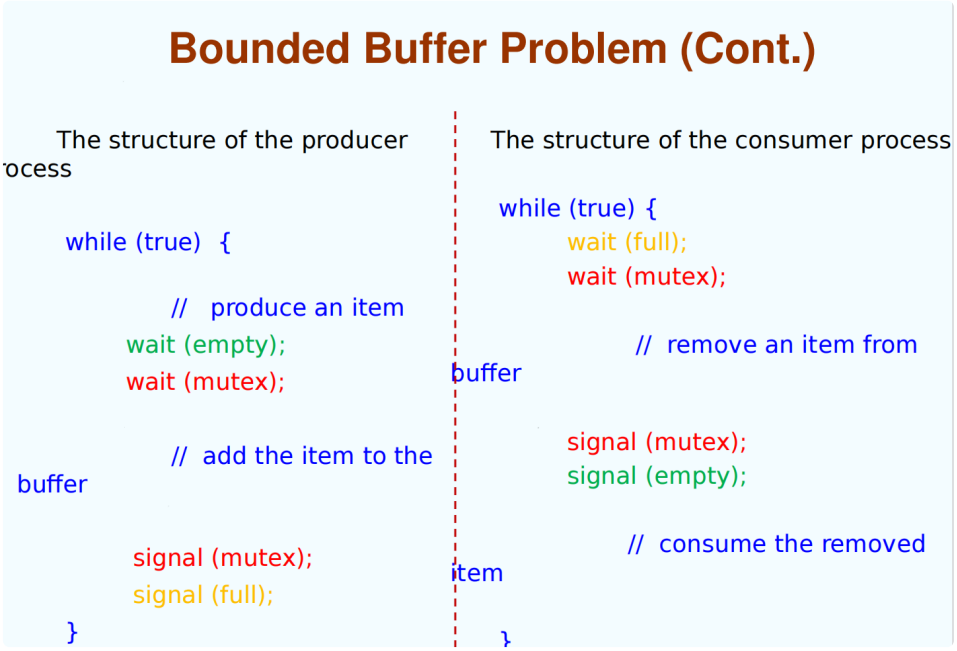


Figure 55

6.3.4.3.2 Readers and Writers Problem

A data set is shared among a number of concurrent processes

- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write.

Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time. (读者优先)

Shared Data

- Data set
- Semaphore **mutex** initialized to 1, to ensure mutual exclusion when readcount is updated.
- Semaphore **wrt** initialized to 1.
- Integer **readcount** initialized to 0 读者的数量

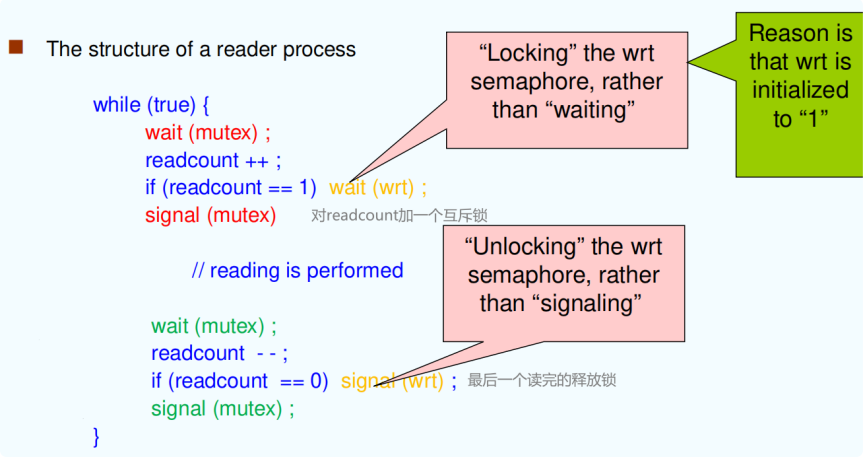


Figure 56

6.3.4.3.3 Dining-Philosophers Problem

哲学家就餐问题

Shared data

- Bowl of rice (data set)
- Semaphore `chopstick [5]` initialized to 1

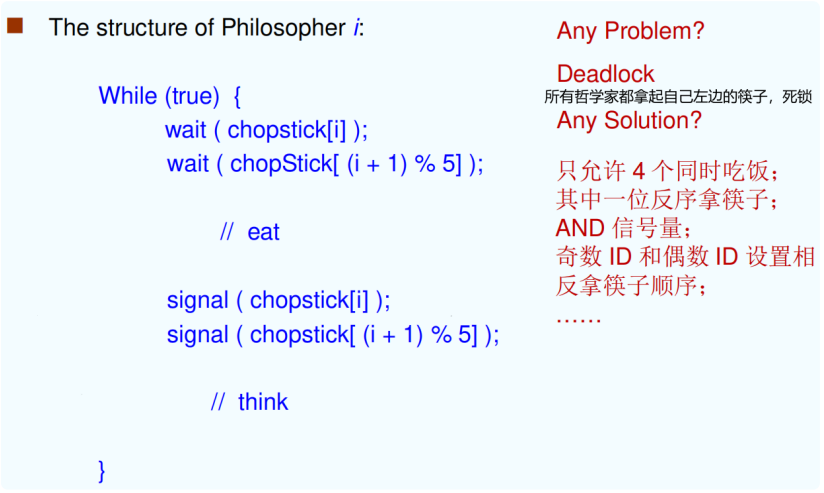


Figure 57

如果只设置一个筷子的信号量，设置为5，有什么问题？一个筷子可能被拿两次，违反互斥性

6.3.5 管程方法

实现互斥和同步

6.3.5.1 Monitor

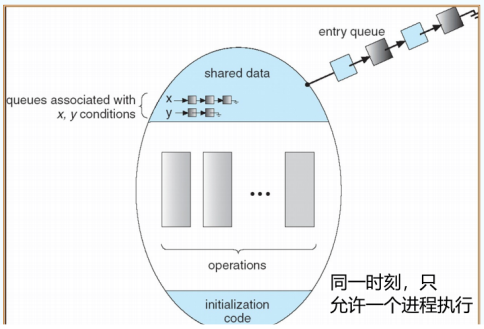
A high-level abstraction that provides a convenient and effective mechanism for process synchronization.

Only one process may be active within the monitor at a time. (hint: the other processes may be sleeping within the monitor)

```
//管程变量只有内部函数可以访问
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (…) { …. }
    …
        procedure Pn (…) {……}
    Initialization code ( ….) { … }
    …
}
```

Fence 10

- condition x, y; 条件变量
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()



函数挂起

Figure 58

x.wait()阻塞该进程并将他插入到x序列

6.3.5.2 管程方法解决哲学家就餐问题

```
monitor DP
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
        condition self [5];
        //philosopher i can delay herself when unable to get chopsticks
        void pickup (int i) { // 拿筷子
            state[i] = HUNGRY;
            test(i); // 看看左边右边有没有人在eating
            if (state[i] != EATING) // 自己hungry
                self[i].wait; // 筷子还不可用，等待
        }
        void putdown (int i) {
            state[i] = THINKING;
            // test left and right neighbors
```

```
15                  test((i + 4) % 5); // 如若满足则唤醒进行吃饭
16                  test((i + 1) % 5);
17              }
18              void test (int i) {
19                  if ( (state[(i + 4) % 5] != EATING) &&
20                      (state[i] == HUNGRY) &&
21                      (state[(i + 1) % 5] != EATING) ) {
22                      state[i] = EATING ;
23                      self[i].signal() ; //执行时不在signal中等待，这句signal没有
    用
24                  } }
25              initialization_code() {
26                  for (int i = 0; i < 5; i++)
27                      state[i] = THINKING;
28              } }
```

Fence 11

- When the left and right philosophers, **self[(i+4)%5]** and **self[(i+1)%5]** continue to eat, **self[i]** may *starve*

6.3.6 例子

6.3.6.1 Pthreads

It provides:

- mutex locks
- condition variables

Non-portable extensions include:

- read-write locks
- spin locks

Using `pthread_cond_wait()` & `pthread_cond_signal()`

6.4 题目

1.

7. 死锁 Deadlock

> **Note**
> 基本概念：死锁概念，4个必要条件，资源分配图，cycle &deadlock
> solutions：3种策略
> 死锁预防（如何破坏四个必要条件)
> 死锁避免（安全状态，安全状态与死锁，单实例算法，多实例算法——银行家算法)
> 检测（单实例算法——wait for graph，多实例算法)
> 恢复

7.1 The Deadlock Problem

死锁：多个进程因竞争共享资源而造成相互等待的一种僵局，使得各个进程都被阻塞，若无外

力作用，这些进程都将永远不能再向前推进

7.1.1 产生死锁的四个必要条件

Deadlock can arise if four conditions hold simultaneously. 同时

- **Mutual exclusion （互斥条件）：** only one process at a time can use a resource.
- **Hold and wait （请求并保持条件）：** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption （不剥夺条件）：** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait （循环等待 / 环路等待条件）：** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Figure 59

7.2 系统模型 System Model

resource type & resource instances

7.2.1 资源分配图 Resource-Allocation Graph

A set of vertices **顶点 V** and a set of edges **边 E**

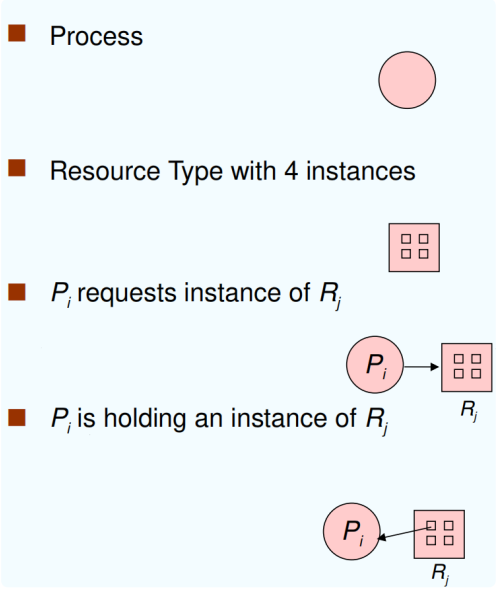


Figure 60

顶点表示资源R或进程P

边： **P->R 等待资源**；**R->P 可以使用资源（资源R已经被P占用）**

死锁一定有环，有环不一定死锁

If graph contains no cycles => no deadlock. 无环一定没有死锁

If graph contains a cycle =>

- if only one instance per resource type, then deadlock.
- if several instances per resource type, possibility of deadlock.

7.3 死锁处理方法 Methods for Handling Deadlocks

Ensure that the system will never enter a deadlock state. 让系统永远不进入死锁状态 ---- **Prevention 死锁预防**、 **Avoidance 死锁避免**

Allow the system to enter a deadlock state and then recover. 允许系统进入死锁状态，但可以恢复 ---- **Detection 死锁检测**、 **Recovery 死锁解除**

Ignore the problem and pretend that deadlocks never occur in the system. 忽略，假装不出现死锁 ---- **鸵鸟算法**

- 鸵鸟算法 is used by most operating systems, including UNIX、Linux、Windows.
  为什么选这个？用户运行多，解决代价少

7.3.1 Deadlock Prevention (预防)

破坏死锁产生的四个必要条件之一 Restrain the ways request can be made.

- 限制用户申请资源的顺序 -- 破坏循环等待（条件4）

1. **Prevent Mutual Exclusion 不互斥**： not required for sharable resources; must hold for nonsharable resources 不共享资源，实际中不太可行
2. **Prevent Hold and Wait 不请求等待** ： 一次分配好所有资源 must guarantee that whenever a process requests a resource, it does not hold any other resources.
   1. Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none (**release all current resources before requesting any additional ones**).
   2. **Low resource utilization**； **starvation possible**. (example: copy data from DVD drive to a disk file, sorts the file, then prints the results to a printer.)
3. **Prevent No Preemption 可剥夺**：变成非抢占式，实际中不太可行
4. **Prevent Circular Wait 不循环等待**： impose a total **ordering** of all resource types, and require that each process requests resources in an increasing order of enumeration.

以上方法在实际中都不太可行

7.3.2 Deadlock Avoidance (避免)

通过动态检测资源分配的安全性，确保系统不会进入不安全状态

- 为实现安全性，我们需要知道
  - 每个进程所需资源max数量 each process declares the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

7.3.2.1 safe state 安全状态

<P1 , P2 , …, Pn> 称为安全序列。如果系统不存在安全序列，则称系统处于不安全状态

对于进程序列中的每一个进程Pi，当前系统已经分配了一些资源，还剩下一些资源。如果**Pi前面的资源之和+系统剩下的资源 能够满足Pi执行完毕**，则这个序列是个安全状态。

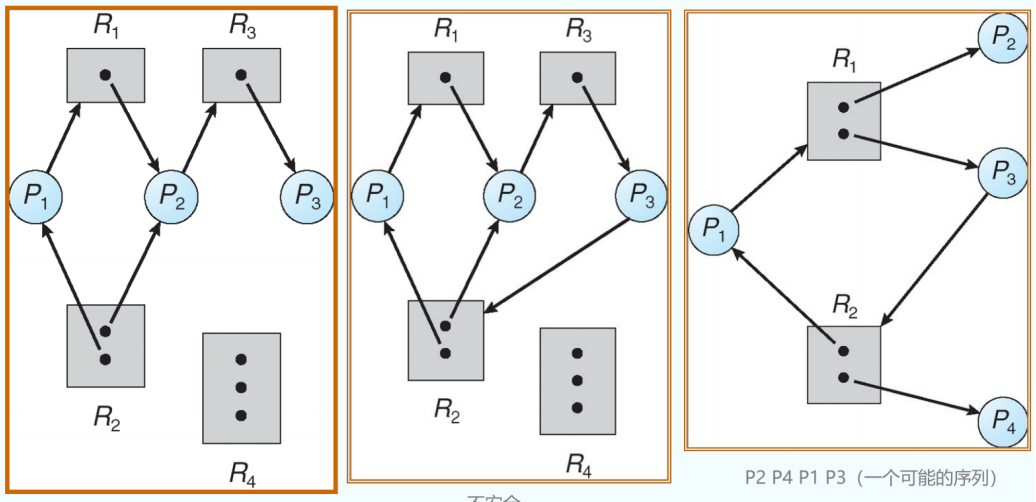


Figure 61

> **Note**
> If a system is in safe state => no deadlocks. 安全状态一定无死锁
> If a system is in unsafe state => possibility of deadlock. 不安全可能有死锁
> Avoidance => ensure that a system will never enter an unsafe state.

进程	最大需求	已分配	还需请求	可用	
P1	10	5	5	3	Safe state? P2 、 P1 、 P3
P2	4	2	2		
P3	9	2	7		

在上面的状态上 P3 申请 1 个资源

进程	最大需求	已分配	还需请求	可用	
P1	10	5	5	2	Safe state? Deadlock?
P2	4	2	2		不安全，可能有死锁
P3	9	3	6		

进程	最大需求	已分配	还需请求	可用	
P1	13	5	8	2	Safe state? Deadlock?
P2	4	2	2		不安全，不是死锁，资源不够
P3	6	3	3		

Figure 62

7.3.2.2 Avoidance algorithms

Single instance of a resource type. Use a resourceallocation graph

Multiple instances of a resource type. Use the banker's algorithm

7.3.2.2.1 Resource-Allocation Graph Algorithm 资源分配图算法

Claim edge Pi -> Rj indicated that process Pi may request resource Rj

有三种边：claim edge、request edge 和 assignment edge

?

7.3.2.2.2 Banker's Algorithm 银行家算法 🔥

设银行家有 10 万贷款， P 、 Q 、 R 分别需要 8 、 3 、 9 万元搞项目，P 已申请到了 4 万，

(1) Q 要申请 2 万，如果满足 Q 的申请，有安全序列 <P, Q, R>/<Q, P, R> 。P申请4w，Q申请2w，还剩4w；先给P的话，P有了8w再还贷，给Q1w，再R

(2) R 要申请 4 万，如果满足 R 的申请，显然不存在安全序列。
R申请4w，P申请了4w，还剩2w，现在谁都不能申请满钱

Figure 63

数据结构

Let n = number of processes, and m = number of resources types.

- ***Available***: Vector of length m. If available [j] = k, there are k instances of resource type R_j available.
- ***Max***: n x m matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j.
- ***Allocation***: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j.
- ***Need***: n x m matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task.

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j].$$

Figure 64

Example

- The content of the matrix *Need* is defined to be *Max* – *Allocation*.

	Allocation	*Need*	*Work*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	7 5 5
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety criteria.

Figure 65

Example: P_1 Requests (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.

Figure 66

7.3.2.3 Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:
   Work = *Available*
   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.
2. Find an *i* such that both:
   (a) *Finish* [*i*] = *false* 没完成
   (b) $Need_i \le Work$ 所需求资源量<系统所能给的
   If no such *i* exists, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true* 完成，资源还给系统
   go to step 2.
4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state.
   做完了，或者做不下去了

Figure 67

7.3.3 Deadlock Detection (检测)

本质：safety 算法，全部满足就没有死锁

7.3.3.1 单实例 Single Instance of Each Resource Type

检查wait for graph有没有环

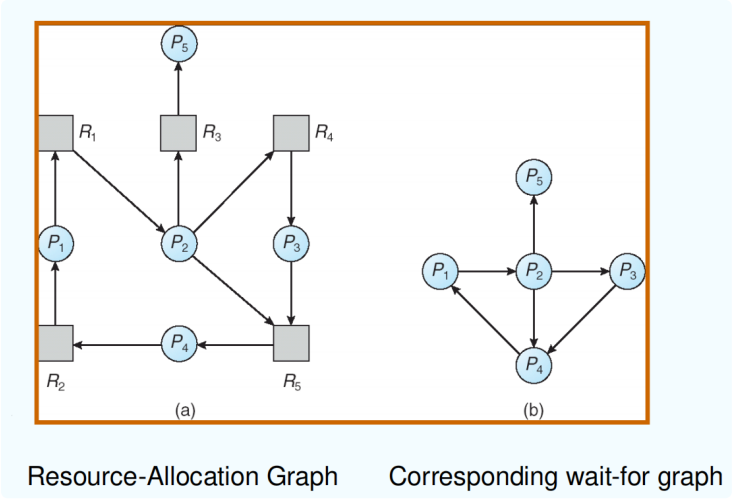


Figure 68

7.3.3.2 多实例 Several Instances of a Resource Type

多实例，调用safety算法

时间复杂度：$O(m \times n^2)$

7.3.3.2.1 Completely Reducible Graph 可完全化简图

能消去图中所有边，能则称为可完全化简图

找出一个既不阻塞又非独立的进程结点 Pi

在顺利的情况下，分配给其资源让其完成，消去所有边变成孤立点

循环上述两步操作，直至消去所有边，代表无死锁。

7.4 Recovery from Deadlock

资源剥夺法：把部分进程挂起，剥夺其资源

撤销进程法：撤销部分进程，释放资源

进程回退法：让一个进程或多个进程回退到避免死锁的地步，释放中间资源

依据：进程的优先级、已执行时间、剩余时间、已用资源、交互还是批处理等

7.4.1 Resource Preemption

 Selecting a victim – minimize cost.
 Rollback – return to some safe state, restart process for that state.
 Starvation – same process may always be picked as victim, include number of rollback in cost factor

8. 主存管理 Memory Management

8.1 Background

电磁感应

Semiconductor ROM/RAM

DRAM: 动态随机存取存储器

SRAM: 静态随机存取存储器

ROM: 只读存储器

“Memory Wall”

内存墙 Memory is much slower than processors, and consumes more energy

Emerging NVM Technologies

NVMs：非挥发性存储器，非易失性存储器

- non-volatile; low idle power; no refreshes; high write overheads; etc.

Phase-Change Memory(PCM)： 相变存储器

- Intel/Micron 3D Xpoint； Intel Optane DC Persistent Memory / DC SSD

ReRAM/RRAM： 电阻式存储器

- Arbitrary programmed cell resistance (“memristor”).
- First invented by HP Labs, now produced by many companies (in early stage).

Program must **be brought (from disk) into memory** and placed within a process for it to be run

only storage CPU can access Main memory and registers directly

Register access in one CPU clock (or less)

Main memory can take many cycles

Cache sits between main memory and CPU registers

8.1.1 Cache Hierarchy

L1离寄存器更近，hit time更小

	L1 Cache	L2 Cache	Main memory
Block size	16--32 bytes	32--64 bytes	4--16 KB
Size	16--64 KB	256KB—8MB	
Hit time	1 Clock Cycle	1—4 Clock Cycles	10—40 Clock Cycles
Backing Store	L2 Cache	Main memory	Disk
Block replacement	Random	Random	Replacement strategies
Miss penalty	4-20 clock cycles	40-200 clock cycles	~6M clock cycles

Figure 69

8.1.2 Multistep Processing of a User Program

运行程序过程：complier -- linker -- loader -- run

- **Symbolic Address 符号地址**: Addresses in the source program are generally symbolic (such as the variable count 函数名/变量名).
- **Relocatable Addresses 可重定位地址** : A compiler typically binds these symbolic addresses to relocatable addresses (such as “14 bytes from the beginning of this module”).
- **Absolute Addresses 绝对地址** : The linker or loader binds the relocatable addresses to absolute addresses (such as 74014)

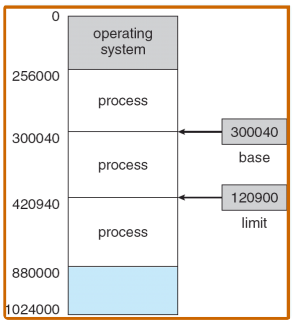
8.1.3 Binding of Instructions and Data to Memory

地址绑定

Address binding (Mapping From one address space to another) of instructions and data to memory addresses can happen at three different stages

- **Compile time 编译时刻** : If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
- **Load time 装入时刻** : Must generate relocatable codeif memory location is not known at compile time
- **Execution time 执行时刻** : Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers 两个寄存器用来快速地址映射和绑定)

- Base register 基址寄存器 , limit register 限长寄存器



- Linux, Windows 系统在**执行时刻**进行地址绑定

8.1.4 Logical vs. Physical Address Space

The concept of a logical address space that is bound to a separate physical address space is central to proper memory management

- Logical address 逻辑地址 – generated by the CPU; also referred to as virtual address
- Physical address 物理地址 – address seen by the memory unit

Logical and physical addresses are the **same** in **compile-time and load-time** address-binding schemes

Logical (virtual) and physical addresses **differ** in **execution time** address-binding scheme

8.1.5 Memory-Management Unit (MMU)

Hardware device that maps virtual to physical address 硬件 实现地址转换

The user program deals with logical addresses; it never sees the real physical addresses

8.1.6 Dynamic Loading

动态装入：程序用到了再装进内存

- Better memory-space utilization; unused routine is never loaded
- 执行时再绑定

静态转入：把程序全部装入内存

8.1.7 Dynamic Linking

动态链接

Linking postponed until execution time

- Small piece of code, **stub 桩**, used to locate the appropriate memory-resident library routine

- Stub replaces itself with the address of the routine, and executes the routine

Dynamic linking is particularly useful for libraries 共享库里的函数只用存一次

- Saves main memory space
- Reduces size of exe image file
- Relinking of new library not needed

8.1.8 总结

程序的链接与装入

- 编译
  - 从高级语言到目标模块的过程（实际是预处理、编译、汇编三个阶段的统称）
  - 本质是一些机器可以“看懂”的 0/1 指令和数据文件
- 链接
  - 把编译后的目标模块与所需库函数链接在一起形成一个整体
  - 静态链接、装入时动态链接、运行时动态链接
- 装入
  - 将虚拟地址映射为内存实际的物理地址
  - 绝对装入、静态重定位（可重定位装入）、动态重定位（动态运行时装入）

8.2 连续分配 Contiguous Memory Allocation

Relocation register 可重定位寄存器 contains value of smallest physical address
Limit register contains range of logical addresses – each logical address must be less than the limit register
MMU maps logical address dynamically

多分区分配 Multi

- 固定分区 fixed partitioning：如果内存大程序小，浪费资源
- 动态分区 dynamic partition
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it

- Operating system maintains information about: a) allocated partitions b) free partitions (hole)

8.2.1 动态分配的算法 Dynamic storage-allocation problem

FF：Allocate the first hole that is big enough 按顺序第一个放得下的洞

NF：下一个放的下的洞

BF： Allocate the smallest hole that is big enough; must search entire list, unless ordered by size 最小能放得下的洞，会产生一些小碎片（tiny leftover holes）

WF： Allocate the largest hole; must also search entire list 最大的洞，会产生一些大碎片（large leftover holes）

> **Note**
> First-fit and best-fit better than worst-fit in terms of speed and storage utilization

8.2.2 Fragmentation

External Fragmentation 外部碎片 – total memory space exists to satisfy a request, but it is not contiguous
Internal Fragmentation 内部碎片 (refer to the textbook p287) – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Reduce external fragmentation by compaction/defragmentation 通过压缩减少外部碎片

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only if* relocation is dynamic, and is done at execution time
- I/O problem I/O操作时也不允许进行压缩
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers
  - Another solution to external frag. is non-contiguous allocation

8.3 分页 Paging

Logical address space of a process can be noncontiguous 逻辑地址可以不连续; process is allocated physical memory whenever the latter is available

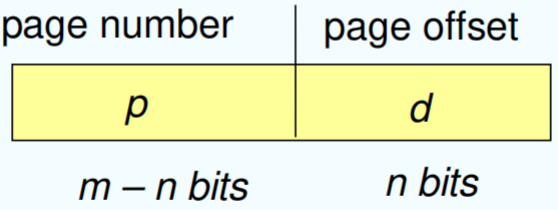
Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
Divide logical memory into blocks of same size called **pages**

To run a program of size n pages, need to find n free frames and load program

8.3.1 Address Translation Scheme

Address generated by CPU is divided into:

- **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



- For given logical address space 2^m *and page size* 2^n

Figure 70

页号、页面偏移

地址转换

Address Translation

页面大小为 4KB ， 虚地址 2362H 、 1565H 的物理地址分别是多少

页号	页框号
0	101H
1	102H
2	254H

Figure 71

为了将虚地址转换为物理地址，需要结合虚地址的页号和页内偏移，以及页表中页号与页框号的映射关系。以下是计算步骤：

已知条件

1. **页面大小** = $4\,\mathrm{KB} = 2^{12} = 4096\,\mathrm{bytes}$，页号用高 20 位，页内偏移用低 12 位表示。
2. 虚地址：
   - 2362H=9026102362H = 9026_{10}
   - 1565H=5477101565H = 5477_{10}
3. 页表：
   - 页号 0 → 页框号 101H
   - 页号 1 → 页框号 102H
   - 页号 2 → 页框号 254H

转换步骤

1. 计算虚地址的页号和页内偏移

$$\text{页号} = \left\lfloor \frac{\text{虚地址}}{\text{页面大小}} \right\rfloor, \quad \text{页内偏移} = \text{虚地址} \mod \text{页面大小}$$

- **2362H**： $\text{页号} = \left\lfloor \frac{9026}{4096} \right\rfloor = 2, \quad \text{页内偏移} = 9026 \mod 4096 = 1834$
- **1565H**： $\text{页号} = \left\lfloor \frac{5477}{4096} \right\rfloor = 1$, 页内偏移 $= 5477 \mod 4096 = 1381$

2. 根据页表找到对应页框号

- **页号 2** → 页框号 254H
- **页号 1** → 页框号 102H

页框号左移 12 位（乘以 4096），再加上页内偏移，得到物理地址。

3. 计算物理地址

- **2362H（页号 2，页框号 254H）**：物理地址 = 页框号 × 4096 + 页内偏移，
  $物理地址 = 254H \times 4096 + 1834 = 254000H + 072A = 25472AH$
- **1565H（页号 1，页框号 102H）**：
  $物理地址 = 102H \times 4096 + 1381 = 102000H + 0565 = 1020565H$

结果

1. 虚地址 2362H2362H → 物理地址 25472AH25472AH
2. 虚地址 1565H1565H → 物理地址 1020565H1020565H

8.3.2 Page

TablePage table is kept in main memory
Page-table base register (PTBR) points to the page table
Page-table length register (PTLR) indicates size of the page table
In this scheme every data/instruction access requires **two** memory accesses. One for the page table and one for the data/instruction.

> **Note**
> Pagetable放在内存里，访问逻辑地址要访问两次内存

The two-memory-access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (**TLB**s 转换旁视缓冲，一称快表)

8.3.3 Paging Hardware With TLB

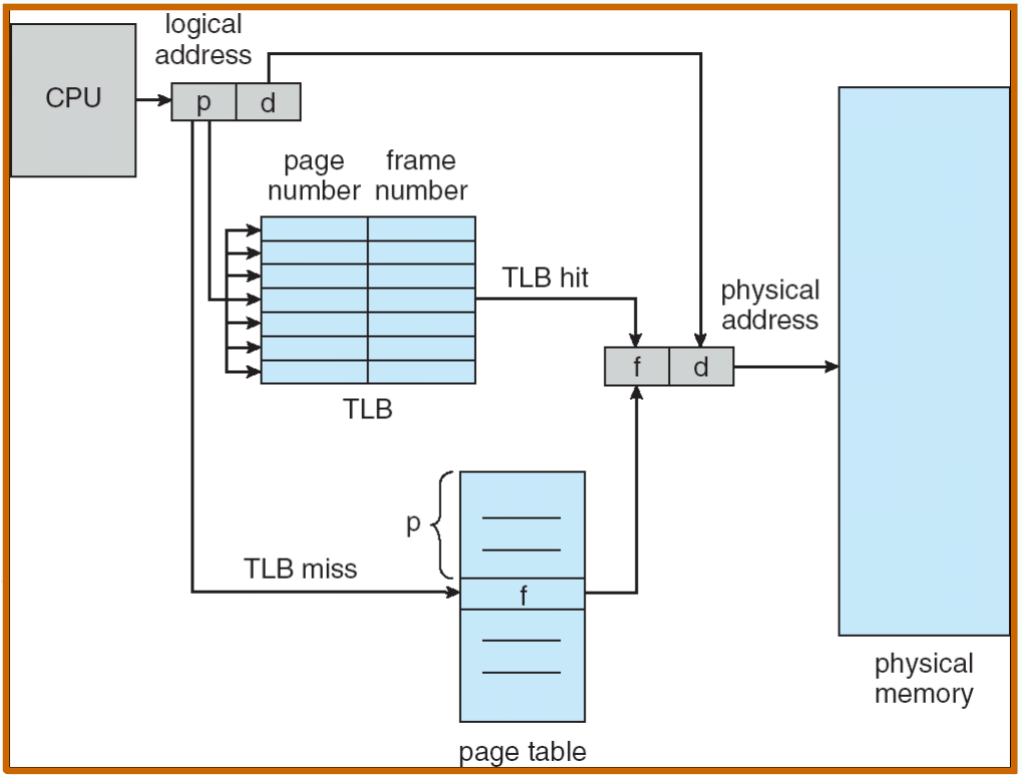


Figure 72

8.3.4 Effective Access Time

$$EAT = (1 + \varepsilon) \times \alpha + (2 + \varepsilon) \times (1–\alpha) = 2 + \varepsilon–\alpha$$

Associative Lookup = ε time unit

Assume memory cycle time is 1 microsecond

Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers

Hit ratio = α

例子:

α = 80%, ε = 2ns, memory access time = 100 ns
EAT=(100+2)*0.8+(100+100+2)*(1-0.8)=122 ns

8.3.5 Memory Protection in Paged Scheme

Valid-invalid bit attached to each entry in the page table

8.3.6 Shared Pages

Shared code： One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems); Shared code must appear in same location in the logical address space of all processes 逻辑地址一样在TLB只用存一次

8.4 Structure of the Page Table

8.4.1 Hierarchical Paging 分级页表

两级页表

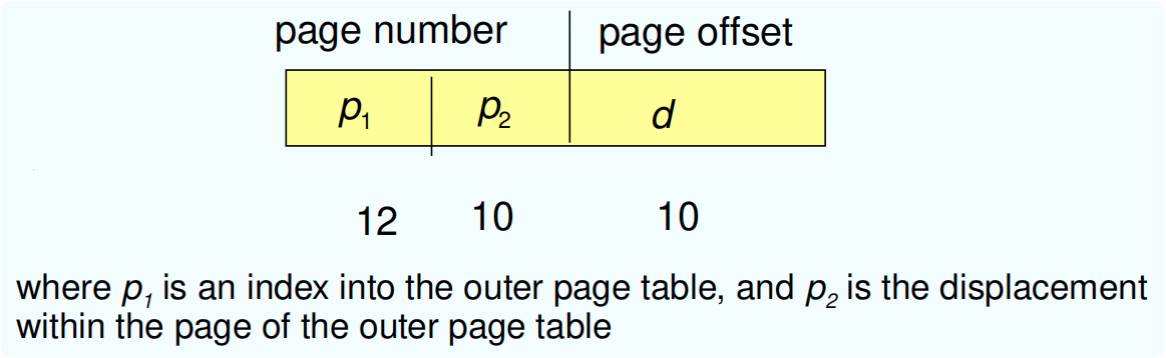


Figure 73

p1->outer page table -- p2 -> page of page table -- d -> real physical address

64=42+10+12, 32=12+10+10

> **Note**
> **Page-table base register (PTBR)** ?
> **What are the benefits of the hierarchical page**? 节省内存（pagetable的内存空间)

8.4.2 Hashed Page Tables

Variation for 64-bit addresses is the **clustered page table**

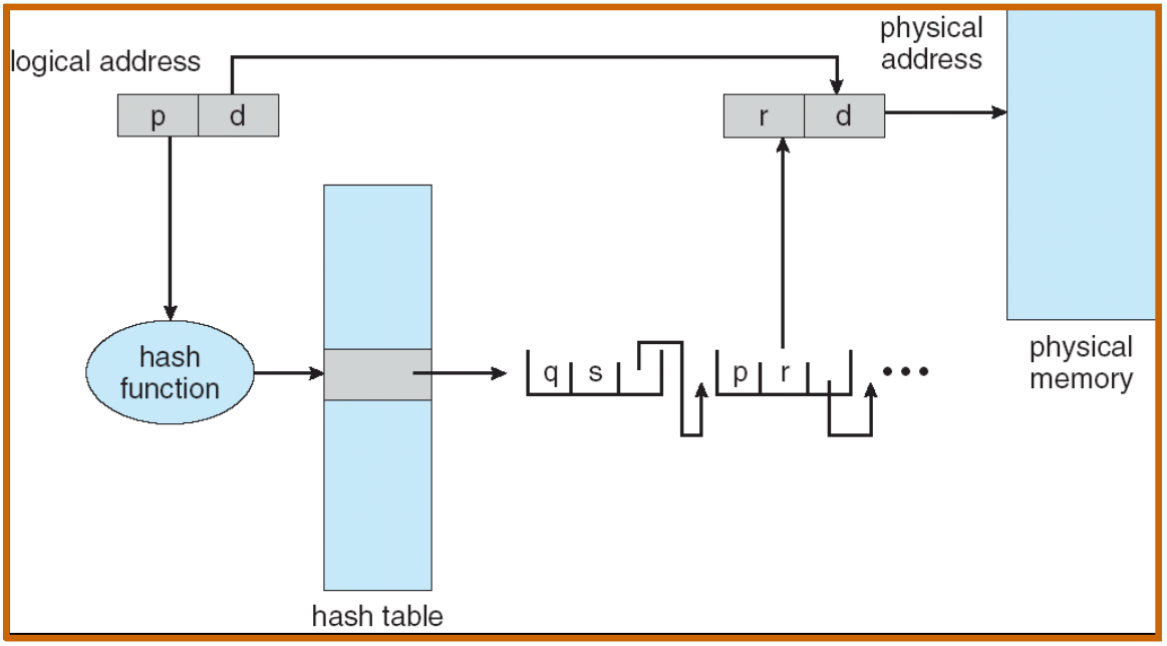


Figure 74

哈希优点：快，hashtable比pagetable小

8.4.3 倒排页表 Inverted Page Tables

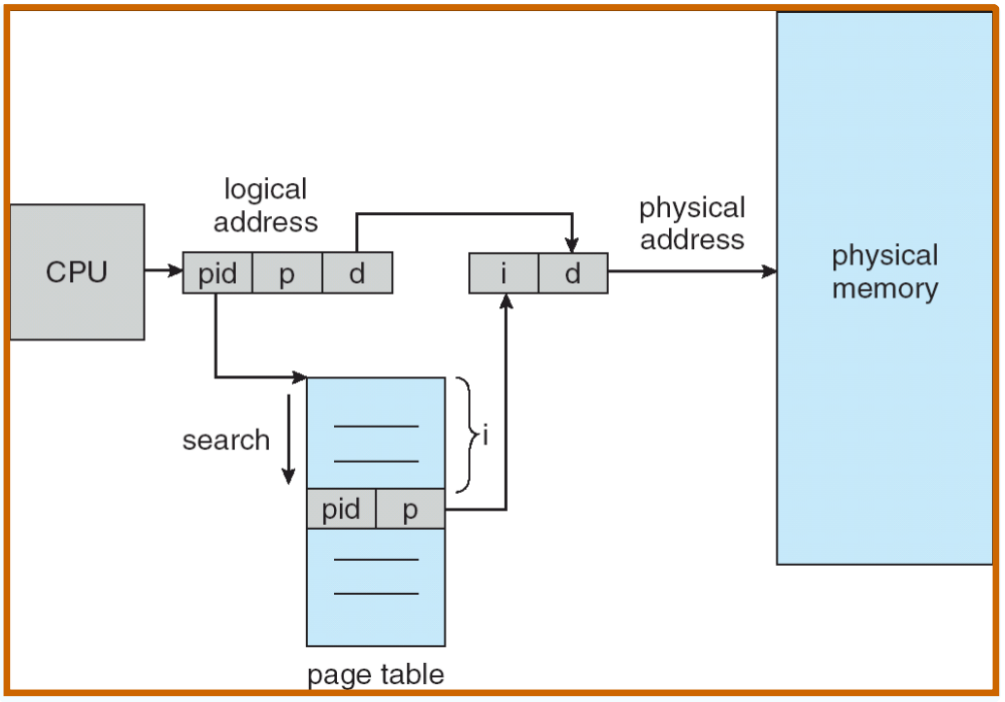


Figure 75

缺点：慢；好处：只有一个pagetable

Search is slow, so put page table entries into a hash table. TLB can be used to speed up hash-table reference.

8.5 Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution 进程可以暂时从内存交换到备用存储器，然后再返回到内存中继续执行

Backing store 交换区 – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images 足够大的快速磁盘，可以容纳所有用户的所有内存映像的副本；必须提供对这些内存映像的直接访问

- **Roll out, roll in** – swapping variant used for *priority-based* scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Figure 76

8.6 分割 Segmentation

Memory-management scheme that supports user view of memory

Segment table – maps two-dimensional physical addresses; each table entry has:

- **base** – contains the starting physical address where the segments reside in memory
- **limit** – specifies the length of the segment

Segment-table base register (STBR) points to the segment table's location in memory 段表中每个块对应的起始物理地址

Segment-table length register (STLR) indicates number of segments used by a program

- segment number **s** is legal if **s** < **STLR**

使用动态内存分配

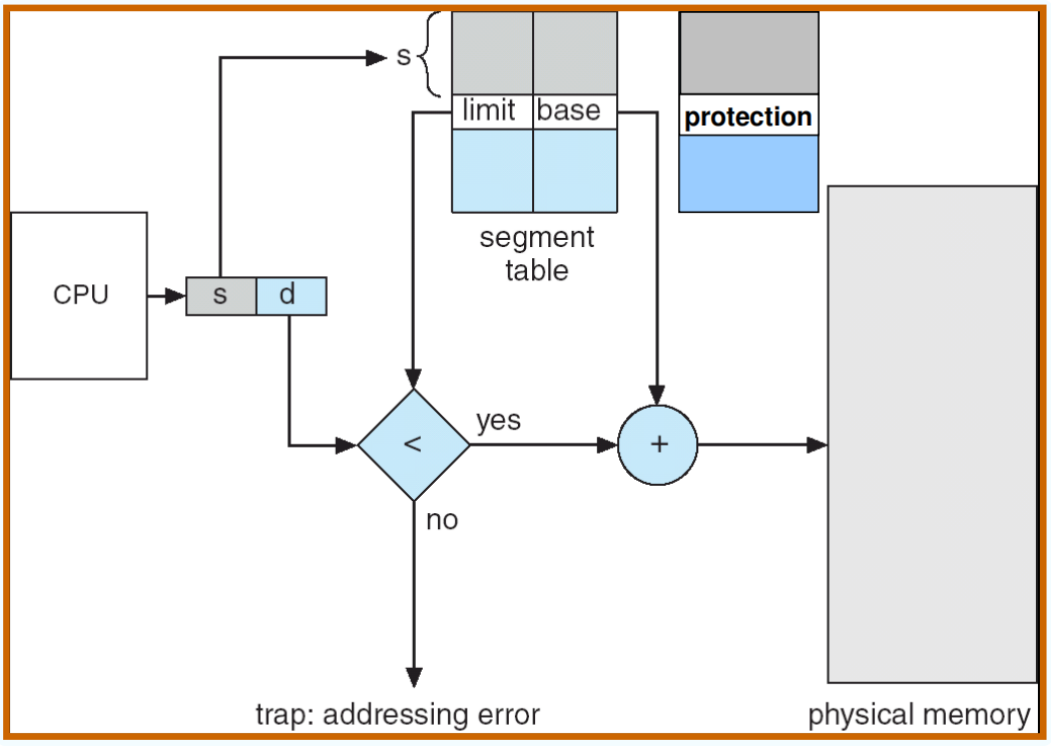


Figure 77

非连续分配

8.7 Example: The Intel Pentium

奔腾处理器

Supports both segmentation and segmentation with paging 段页混合，先分段再分页

linear address: 32 offset

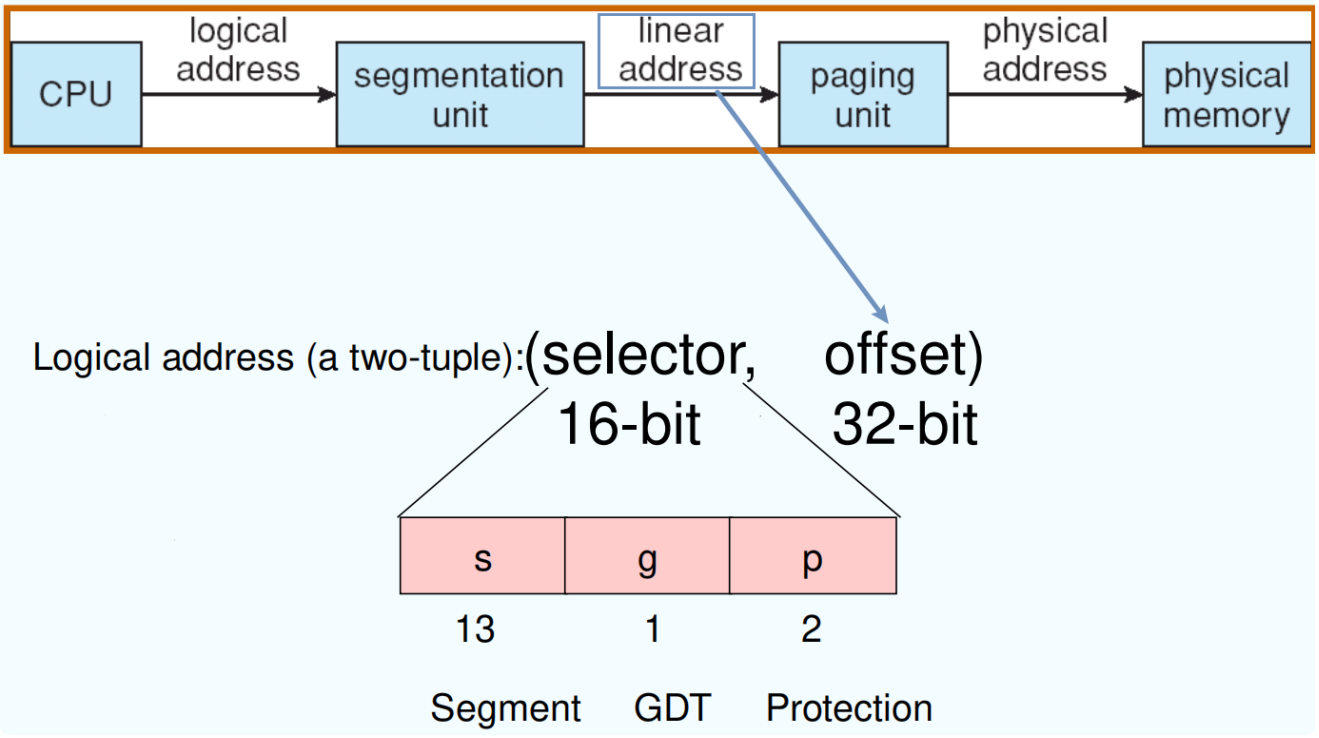


Figure 78

8.7.1 Intel Pentium Segmentation

根据段号找页表基址，根据分页的va（页框号）找到页号，两者拼接

Local Descriptor Table contains entries for the segments local to each program itself;
Global Descriptor Table contains entries of the system (OS).

9. Virtual Memory

9.1 Background

Virtual memory – separation of user logical memory from physical memory 虚拟内存不是物理对象，而是指内核提供的用于管理物理内存和虚拟地址的抽象和机制的集合

- Only **part** of the program needs to be in memory for execution
- Logical address space can therefore be much **larger** than physical address space
- Allows address spaces to be **shared** by several processes
- Allows for more efficient process **creation**

Virtual memory can be implemented via 虚拟内存的实现:

- Demand paging 请求式分页
- Demand segmentation 请求式分段

9.1.1 principle of locality

局部性原理 (principle of locality): 指程序在执行过程中的一个较短时期所执行的指令地址和指令的操作数地址，分别局限于一定区域。

- 时间局部性 : 一条指令的一次执行和下次执行，一个数据的一次访问和下次访问都集中在一个较短时期内 ;
- 空间局部性 : 当前指令和邻近的几条指令，当前访问的数据和邻近的数据都集中在一个较小区域内。
- 虚拟存储器是具有请求调入功能和置换功能，仅把进程的一部分装入内存便可运行进程的存储管理系统，它能从逻辑上对内存容量进行扩充的一种虚拟的存储管理系统。

虚拟内存其他优点:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space 通过将共享对象映射到虚拟地址空间，系统库可由多个进程共享
- Shared memory is enabled 共享内存
- Pages can be shared during process creation (speeds up creation) 可在进程创建期间共享页面（加快创建速度）

9.1.2 Process Creation

Virtual memory allows other benefits during process creation:

- **Copy-on-Write(COW) 写时复制**： CoW 的主要目的是减少内存使用和提高性能，通过延迟实际的内存
  复制，直到某个进程尝试修改内存内容时才进行复制（省时间和空间）
- **Memory-Mapped Files (later)**: 将文件内容映射到进程的地址空间的技术。通过内存映射文件，可以像访问内存一样访问文件内容，而无需显式地进行读写操作。这种技术在处理大文件、提高文件访问性能以及实现进程间通信等方面非常有用

9.2 Demand Paging

需要时放到内存

Bring a page into memory only when it is needed

- Less I/O needed
- Less memory needed

- Faster response
- More users
- Page is needed => reference to it
  - invalid reference => abort
  - not-in-memory => bring to memory
- Lazy swapper – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager

Transfer of a Paged Memory to Contiguous Disk Space 连续分配

9.2.1 Valid-Invalid Bit

With each page table entry a valid–invalid bit is associated (**v**: in-memory, **i**: not-in-memory)

- Initially valid–invalid bit is set to i on all entries
- During address translation, if valid–invalid bit in page table entry is **i** => **page fault** (a trap to the OS 缺页中断)

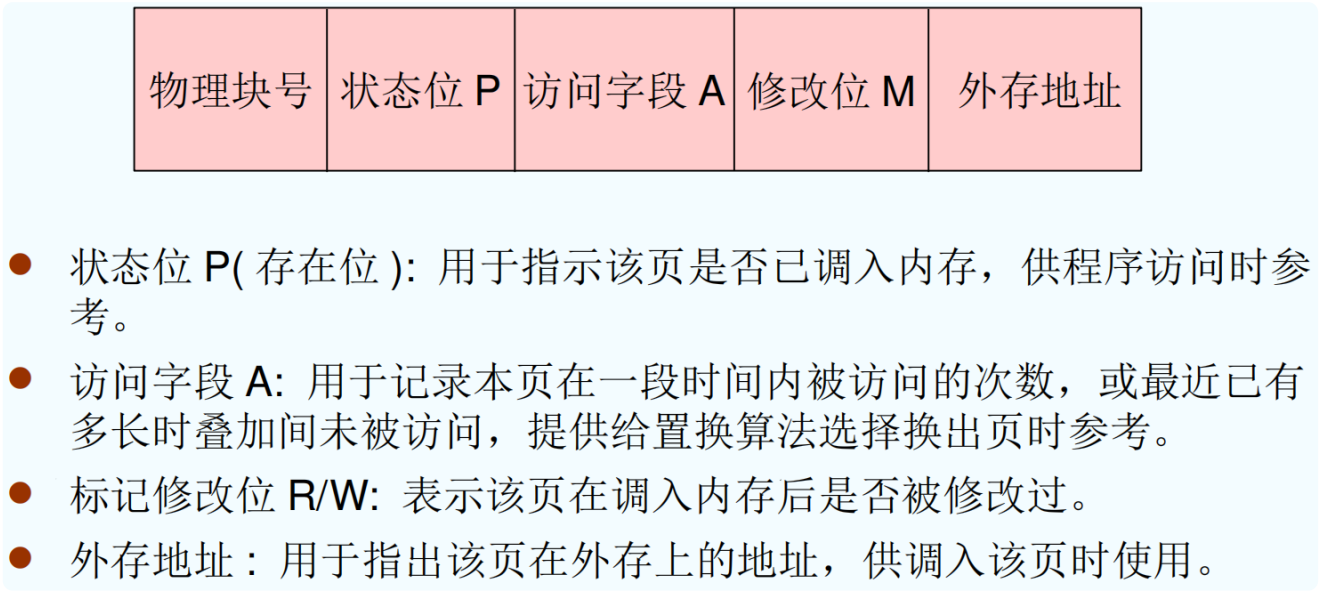


Figure 79

9.2.2 Page Fault

If there is a reference to a page, first reference to that page will trap to operating system: **page fault**

1. Operating system looks at another table (kept with PCB) to decide:
   - Invalid reference => abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame

4. Reset tables 磁盘内部表和页表
5. Set `validation bit = v`
6. Restart the instruction that caused the page fault
    1. 中断后恢复 block move
    2. auto increment/decrement location

What's the state of the process that has page fault?

9.2.3 Performance of Demand Paging

Page Fault Rate $0 \le p \le 1.0$
□ if p = 0 no page faults
□ if p = 1, every reference is a fault

Effective Access Time (EAT)
EAT = (1 – p) x memory access + p (page fault overhead + swap page out + swap page in + restart overhead)

例子:

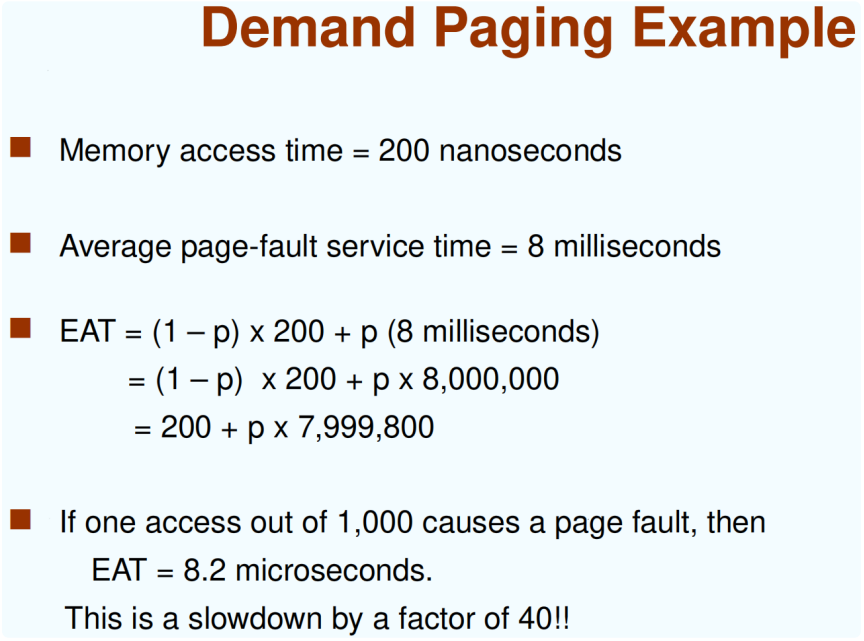


Figure 80

9.3 Copy-on-Write

9.4 Page Replacement

9.4.1 Page replacement

如果没有空闲帧 free frame：

Page replacement 页面替换 – find some page in memory, but not really in use, swap it out

- algorithm 页面替换算法
- performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

好处：

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit 脏位** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

实际上不是等到没有空闲帧的时候进行替换，操作系统会提前做

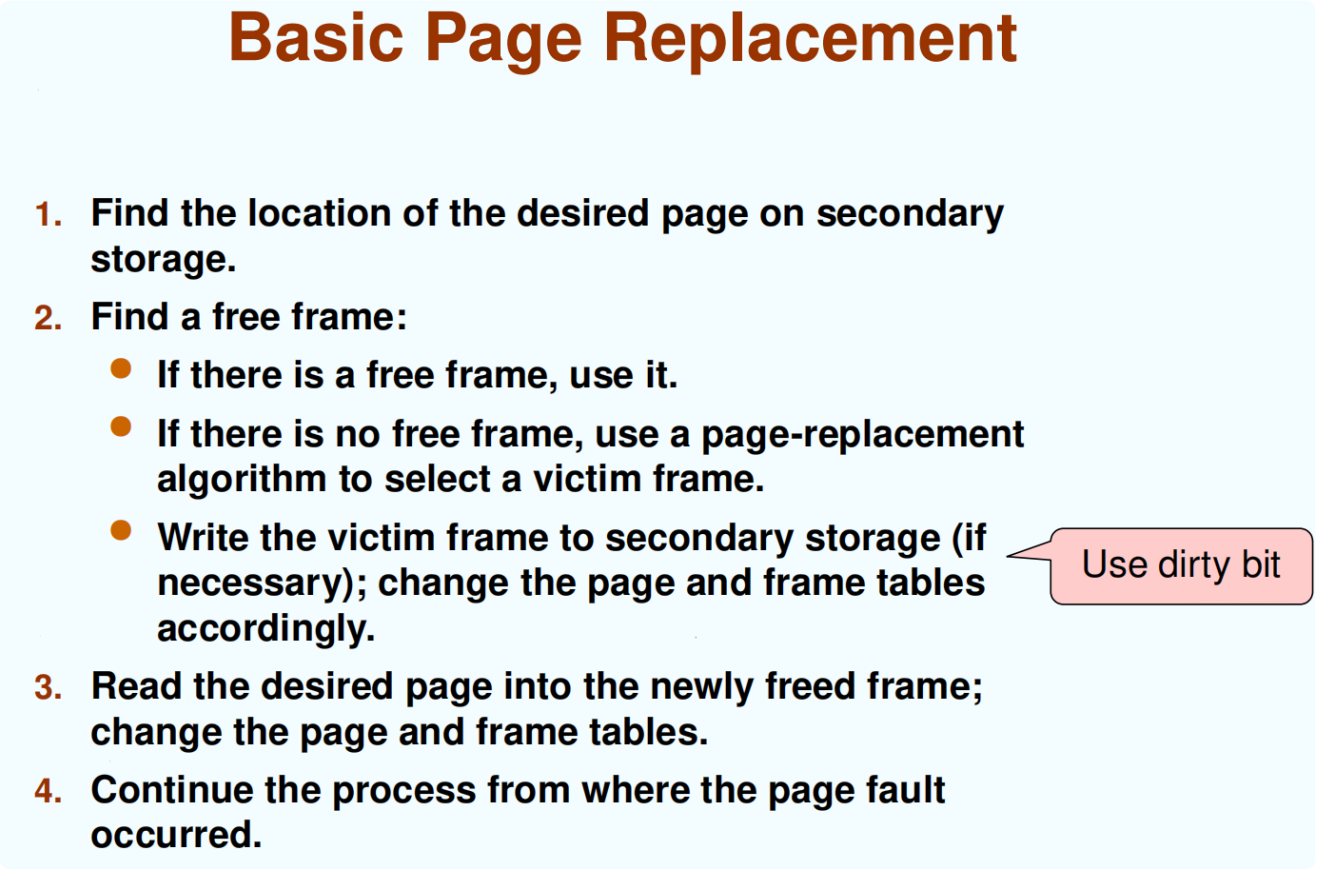


Figure 81

9.4.2 Page Replacement Algorithms

Want lowest page-fault rate 最低缺页率

输入：引用串 reference string；输出：缺页的次数 number of page faults on that string

> **Note**
> **内存无限大，缺页会无线趋近于零吗**？不会，趋于一个常数，缺页次数随着内存变大会变小

9.4.2.1 First-In-First-Out (FIFO) Algorithm

先进先出算法

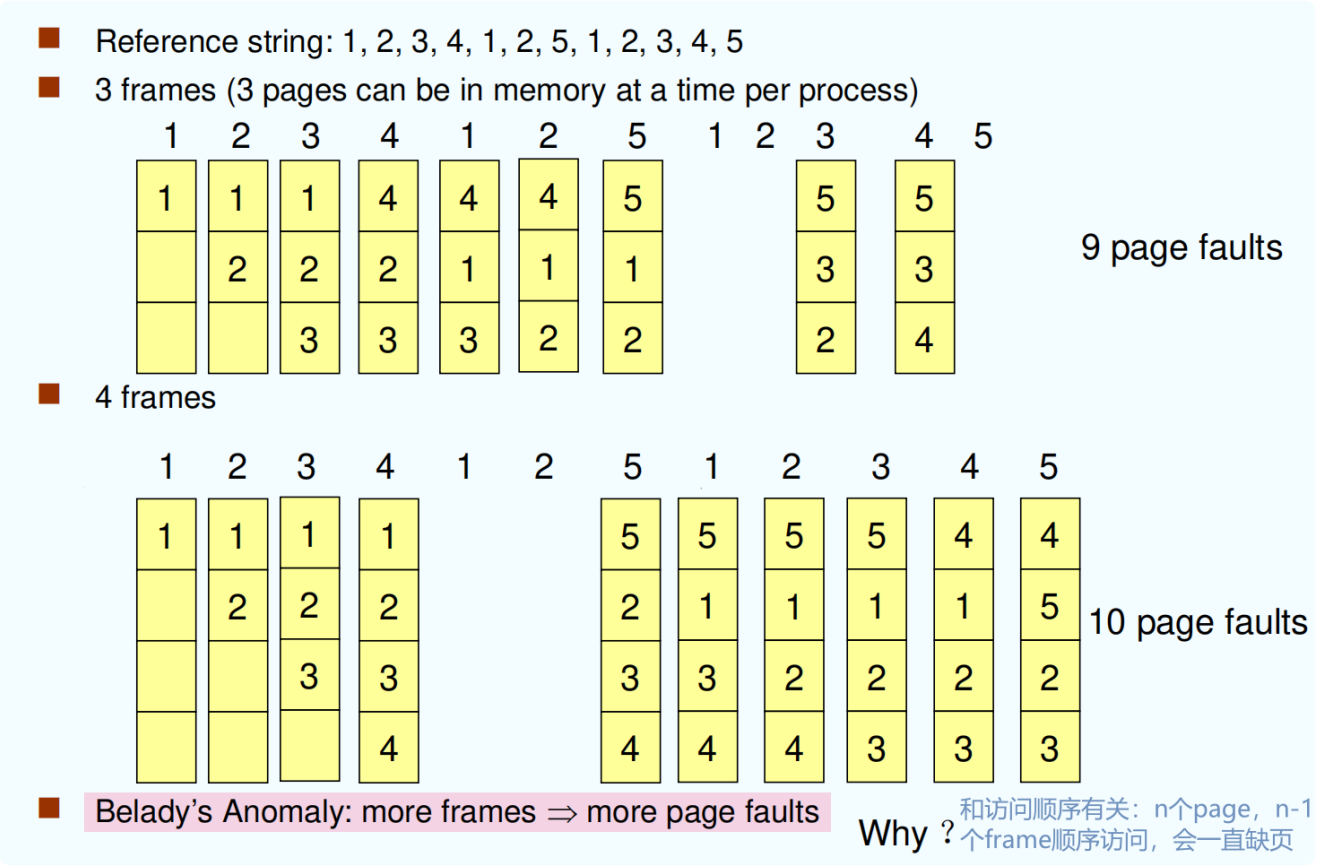


Figure 82

9.4.2.2 Optimal Algorithm

最佳置换算法 缺页次数最小 => optimal

替换将来最长不使用的page Replace page that will not be used for longest period of time

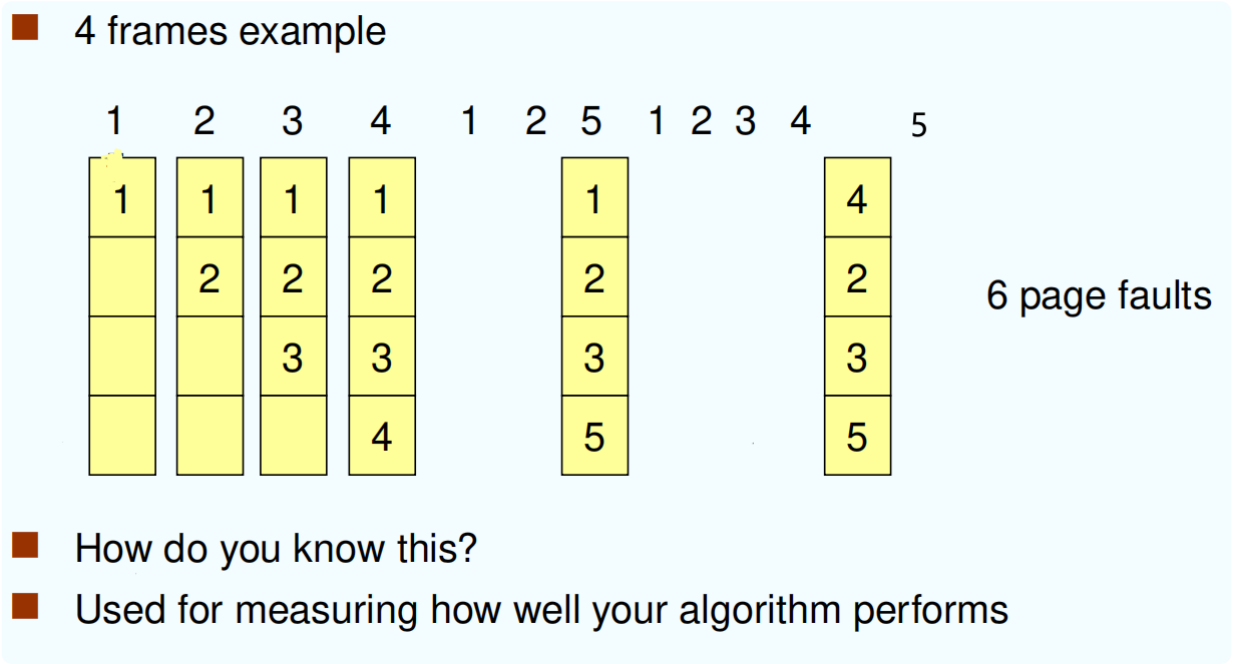


Figure 83

9.4.2.3 Least Recently Used (LRU) Algorithm

最近最久未使用置换算法：选择内存中最久没有引用的页面被置换。这是局部性原理的合理近似，性能接近最佳算法。但由于需要记录页面使用时间，硬件开销太大。

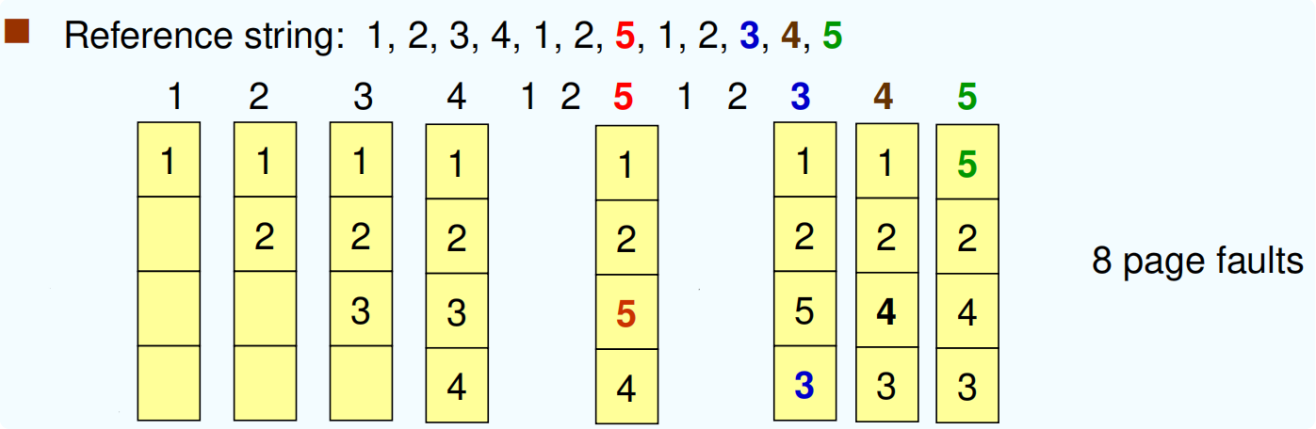


Figure 84

9.4.2.4 LRU Algorithm

以下三种方法都是通过硬件实现

Counter implementation：每次引用时钟+1

Stack implementation：设置一个特殊的栈，把被访问的页面移到栈顶，于是栈底的是最久未使用页面。

移位寄存器：被访问时左边最高位置 1 ，定期右移并且最高位补0 ，于是寄存器数值最小的是最久未使用页面。（AdditionalReference-Bits Algorithm 附加引用位算法）

9.4.2.5 LRU Approximation Algorithms

又叫 second chance algorithm，clock algorithm

reference bit

- 当访问到page，bit=1，周期内被访问过至少一次；bit=0，较长时间内没被访问过，替换出去

second chance

9.4.2.6 Enhanced second chance Algorithm

使用引用位和修改位 (reference bit, modified bit)，引用过或者修改过置为1

淘汰次序：

基本思想：

9.4.2.7 Counting-based Algorithms

Keep a counter of the number of references that have been made to each page

- **LFU Algorithm**: replaces page with smallest count 使用次数最少的页面置换出去
- **MFU Algorithm(most frequently used)**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used 使用次数最多的页面置换出去

9.4.2.8 Page Buffering Algorithm 页面缓冲算法

通过被置换页面的缓冲，有机会找回刚被置换的页面

被置换页面的选择和处理：用 FIFO 算法选择被置换页，把被置换的页面放入两个链表之一，如果页面未被修改，就将其归入到空闲页面链表的末尾，否则将其归入到已修改页面链表。

□ 需要调入新的页面时：将新页面内容读入到空闲页面链表的第一项所指的页面，然后将第一项删除。

□ 空闲页面和已修改页面，仍停留在内存中一段时间，如果这些页面被再次访问，这些页面还在内存中。
□ 当已修改页面达到一定数目后，再将它们一起调出到外存，然后将它们归入空闲页面链表。

实际中，Windows 、 Linux 页面置换算法是基于页面缓冲算法

9.5 Allocation of Frames

分配：Each process needs minimum number of pages - usually determined by computer architecture

9.5.1 Fixed Allocation 固定分配

Equal allocation 等分

Proportional allocation – Allocate according to the size of process 按进程比例分配

9.5.2 Priority Allocation 优先级分配

Use a proportional allocation scheme using priorities rather than size

If process Pi generates a page fault,
□ select for a replacement one of its frames
□ select for replacement a frame from a process with a lower priority number

9.5.3 Global vs. Local Allocation

Global replacement – process selects a replacement frame from the set of all frames; one process can take a frame from another

Local replacement – each process selects from only its own set of allocated frames

Problem with global replacement 问题: unpredictable page-fault rate. Cannot control its own page-fault rate. More common
Problem with local replacement: free frames are not available for others. – Low throughput

9.6 Thrashing 抖动、颠簸

If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 low CPU utilization
 Queuing at the paging device, the ready queue becomes empty
 operating system thinks that it needs to increase the degree of multiprogramming
 another process added to the system 循环

Thrashing： a process is busy swapping pages in and out 忙于页面替换，内存不够

Thrashing 解决方法 :

 增加物理内存
 优化页面置换算法
 在 cpu 调度中引入工作集算法
 动态调整进程的内存分配
 限制并发进程数
 内存压缩

9.6.1 Demand Paging and Thrashing

Why does demand paging work?
Locality model 局部性
 Process migrates from one locality to another
 Localities may overlap

Why does thrashing occur?
 size of locality > total memory size
 To limit the effect of thrashing: local replacement algo cannot steal frames from other processes. But queue in page device increases effective access time.
 To prevent thrashing: allocate memory to accommodate its locality

9.6.2 Working-Set Model

9.7 Memory-Mapped Files

9.8 Allocating Kernel Memory

9.9 Other Considerations

9.10 Operating-System Examples