operating system concept 9th edition

chapter 1 Introduction

1.1 What Operating Systems Do

- computer system four components:
    - hardware
        - provides the basic computing resources for the system
            - central processing unit (CPU)
            - memory
            - input/output (I/O) devices
    - operating system
        - controls the hardware and coordinates its use among the various application programs for the various users.
    - application programs
        - define the ways in which these resources are used to solve users' computing problems
    - users
- computer system another view
    - hardware
    - software
    - data

1.1 User View

- ease of use --- single-user
- performance
- resource utilization --- multiple users
    - how various hardware and software resources are shared
    - used efficiently
    - no individual user takes more than her fair share

1.1.2 System View

- resource allocator
    - CPU time, memory space, file-storage space, I/O devices......
- control program
    - prevent errors and improper use of the computer,especially the operation and control of I/O devices

1.1.3 Defining Operating Systems

what an operating system is?

The operating system is the one program running at all times on the computer—usually called the kernel

types of programs:

- kernel
- system programs **不是kernel**
- application programs
- （middleware： Mobile operating systems）

1.2 Computer-System Organization

1.3 Computer-System Architecture

1.4 Operating-System Structure

An operating system provides the **environment** within which programs are executed

- the ability to multiprogram
  - jobs are kept initially on the *disk* in the **job pool**
  - The set of jobs in *memory* can be a subset of the jobs kept in the job pool
  - picks and begins to execute one of the jobs in *memory*
- Time sharing/multitasking
  - extension of multiprogramming
  - the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running

1.5 Operating-System Operations

chapter 3 Process

3.1 Process Concept

3.1.1 The Process

a process is a program in execution

- text section - program code
- current activity - the value of the program counter and the contents of the processor’s registers

- stack - temporary (often call as 'local') data
- data section - global variables
- heap - memory that is dynamically allocated

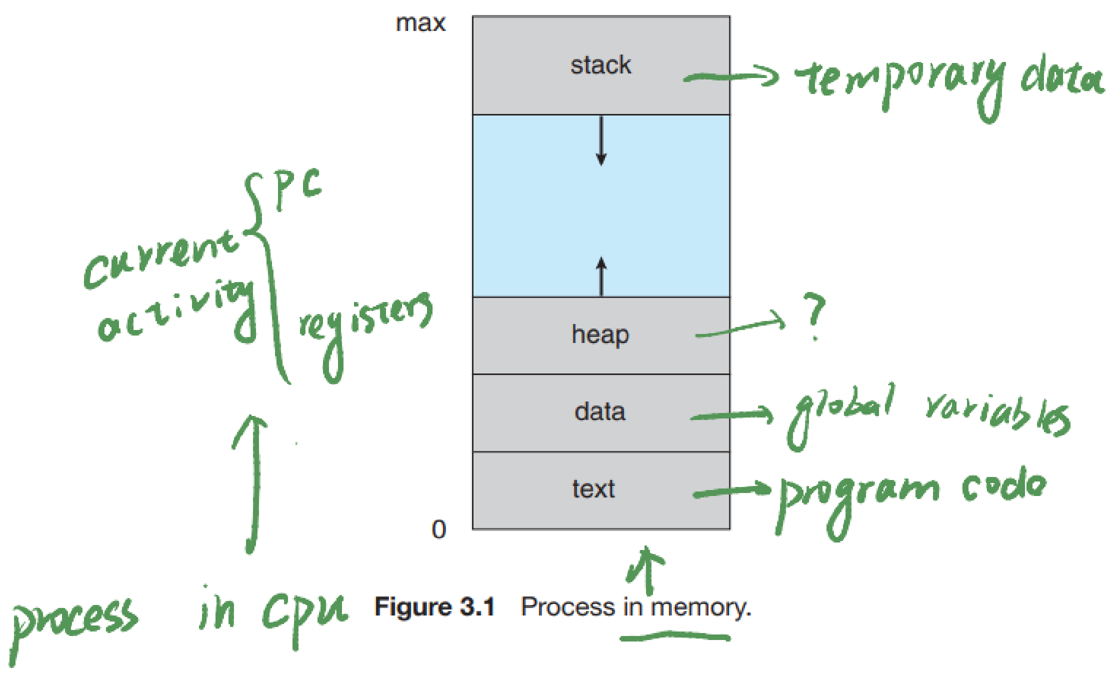


Figure 3.1 Process in memory.

A program becomes a process when an executable file is loaded into memory

a process itself can be an execution environment for other code

- Java Virtual Machine

3.1.2 Process State

- New
- Running - Instructions are being executed **in cpu**
- Waiting - The process is waiting for some event to occur
- Ready - The process is waiting to be assigned to a processor
- Terminated

Chapter 3 Processes

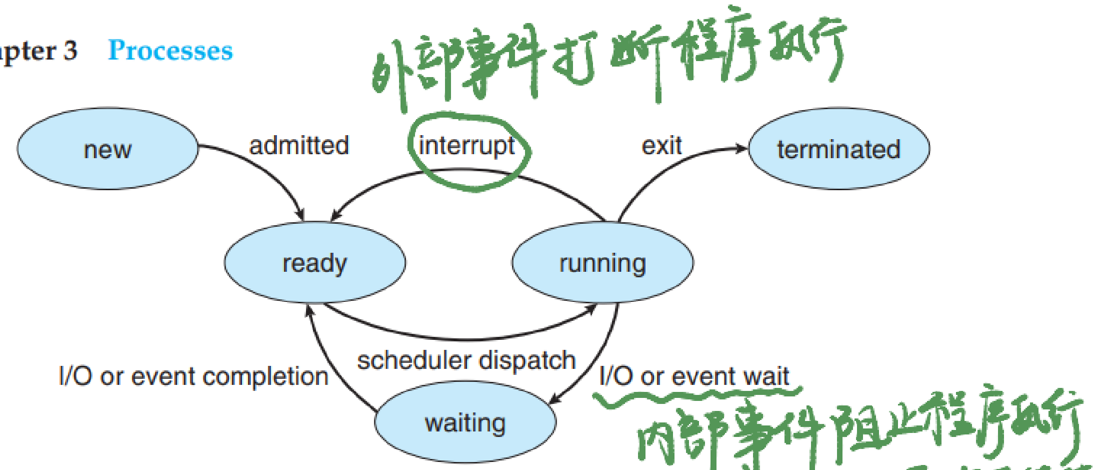


Figure 3.2 Diagram of process state.

3.1.3 Process Control Block

Each process is represented in the operating system by a process control block

一种数据结构，存储了一个进程的信息，一个该类型的变量用于代表一个进程

- Process state
- Program counter
- CPU registers
- CPU-scheduling information
  - This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- Memory-management information.
- Accounting information 记账信息 记录进程消耗
- I/O status information
  - the list of I/O devices allocated to the process, a list of open files, and so on

3.1.4 Threads

skip

3.2 Process Scheduling

The **objective of multiprogramming** is to have some process running at all times, to maximize CPU utilization.

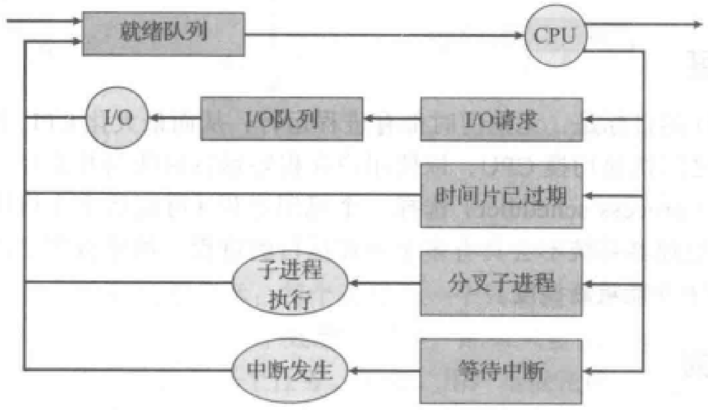
The **objective of time sharing** is to switch the CPU among processes so frequently that users can interact with each program while it is running.

3.2.1 Scheduling Queues

- job queue - all processes in the system
- ready queue - residing in main memory and are ready and waiting to execute
- device queue - processes waiting for a particular I/O device

队列图：

进程调度通常用队列图（queueing diagram）来表示，如图 3-6 所示。每个矩形框代表一个队列；这里具有两种队列：就绪队列和设备队列。圆圈表示服务队列的资源；箭头表示系统内的进程流向。



dispatch： 分派

3.2.2 Schedulers

批处理程序：

- long-term scheduler 长期调度程序 / job scheduler 作业调度程序
  - selects processes from process pool and loads them into memory for execution
  - controls the **degree** of multiprogramming 多道程序程度(the number of processes in memory)
  - decide which process should be selected for execution
    - I/O-bound process IO密集型进程
    - CPU-bound process CPU密集型进程
    - a good process mix of I/O-bound and CPU-bound processes
- short-term scheduler 短期调度程序/ CPU scheduler cpu调度程序
  - selects from among the processes that are ready to execute and allocates the CPU to one of them

分时系统 additionnal：

- medium-term scheduler 中期调度程序
  - remove a process from memory (and from active contention（主动竞争） for the CPU) and thus reduce the degree of multiprogramming

    - the process can be reintroduced into memory, and its execution can be continued where it left off
    - swapping 交换

3.2.3 Context Switch

- kernel saves the context of the old process in its PCB
- loads the saved context of the new process scheduled to run
- overhead - dependent on hardware support

3.3 Operations on Processes

3.3.1 Process Creation

fork()

- 从fork开始分裂成两个一摸一样（但是复制品，两者对数据的操作互不影响）的进程
- 进程树
- 有**共享内存段**在父进程和新分叉的子进程之间**共享**。为新创建的进程**复制堆栈。**

3.3.2 Process Termination

```
/* exit with status 1 */
exit(1);
```

```
pid_t pid;
int status;
pid = wait(&status);
```

3.4 Interprocess Communication

Types of **processes executing concurrently**:

- independent
    - it cannot affect or be affected by the other processes executing in the system
    - **not** share data with any other process
- cooperating
    - it can affect or be affected by the other processes executing in the system
    - shares data with other processes

reasons for providing an environment that allows process cooperation:

- Information sharing
- Computation speedup 计算加速
- Modularity 模块化
- Convenience

Inter**P**rocess **C**ommunication (IPC) mechanism:

- shared memory:
    - faster
    -
- message passing:
    - exchanging smaller amounts of data -- no conflicts
    - Message passing is also easier to implement in a **distributed system** than shared memory.
    - systems with several processing cores provides better performance
    -

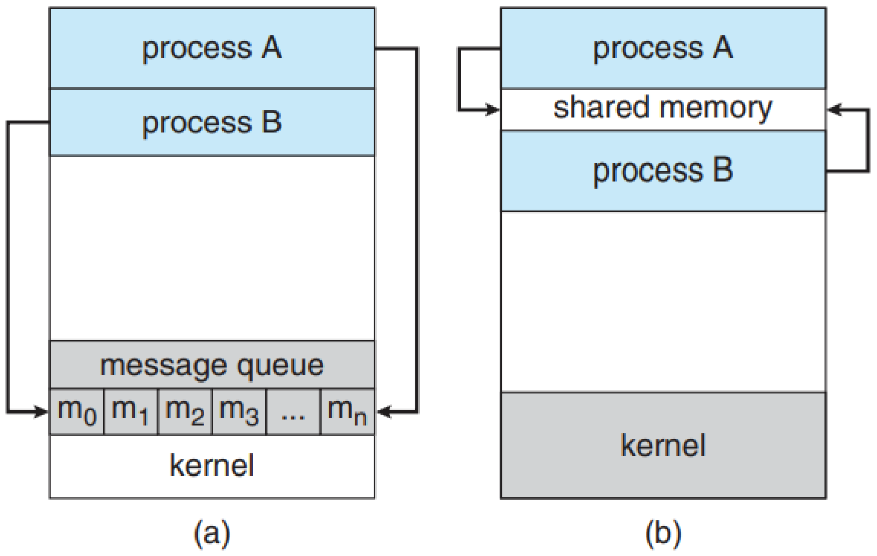


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

???producer-consumer problem coding???

chapter 4 Threads

4.1 Overview

A thread is a basic unit of **CPU utilization** CPU使用

- A traditional (or **heavyweight**) process has a single thread of control

4.1.1 Motivation

Process creation is time consuming and resource intensive

因为线程比进程小，所以创建线程通常比创建进程使用更少的资源。创建一个进程需要分配一个进程控制块(PCB)，这是一个相当大的数据结构。PCB包括内存映射、打开的文件列表和环境变量。分配和管理内存映射通常是最耗时的活动。创建用户或内核线程都需要分配一个小数据结构来保存**寄存器集、堆栈和优先级。**

4.1.2 Benefits

- Responsiveness 响应性
- Resource sharing 资源共享
  - 多线程进程的线程**共享堆内存和全局变量。**每个线程都有其独立的寄存器值集和独立的栈
- Economy 经济
- Scalability 可伸缩性 - multiprocessor architecture

4.2 Multicore Programming

- single computing core
  - interleaved 交错执行
  - one thread at a time
- multiple cores
  - the threads can run in parallel - assign a **separate thread** to each core

- parallelism 并行性
  - perform more than one task simultaneously
- concurrency 并发性
  - supports more than one task by allowing all the tasks to make progress
  - 内核线程之间的上下文切换通常需要保存被切换出去的线程的CPU寄存器的值，并恢复被调度的新线程的CPU寄存器

使用多个用户级线程的多线程解决方案能否在多处理器系统上比在单处理器系统上获得更好的性能?

由多个用户级线程组成的多线程系统不能在多处理器系统中同时使用不同的处理器。**操作系统只看到一个进程，不会将该进程的不同线程调度到不同的处理器上。**因此，在多处理器系统上执行多个用户级线程不会带来性能上的好处

4.2.1 Programming Challenges

- Identifying tasks 识别任务
    - This involves examining applications to find areas that can be divided into separate, concurrent tasks
- Balance
    - perform equal work of equal value
- Data splitting 数据分割
- Data dependency 数据依赖
- Testing and debugging 测试与调试

4.2.2 Types of Parallelism

- data parallelism
    - distributing **subsets** of the same data across multiple computing cores and performing the **same operation** on each core
- task parallelism
    - distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a **unique operation**
    - operating on the **same** data, or they may be operating on **different** data

4.3 Multithreading Models

- user threads
- kernel threads
- What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

1. 用户线程位于内核之上，它的管理无需内核支持，而内核线程由操作系统来直接支持与管理。
2. 在使用一对一模型或多对多模型的系统上，用户线程由线程库调度，而内核调度内核线程。

relationship between user threads and kernel threads

- many-to-one model
    - advantage：
        - **efficient** - Thread management is done by the thread library in user space
    - disadvantage：

        - entire process will **block** if a thread makes a blocking system call
        - unable to run in **parallel** on multicore systems
- one-to-one model
    - advantage：
        - allowing another thread to run when a thread makes a blocking system call
        - run in parallel on multiprocessors
    - disadvantage:
        - burden the performance - creating a user thread requires creating the corresponding kernel thread
- many-tomany model
    - advantage:
        - both of the two methods before
    - ---> two-level model

chapter 5 Process Synchronization

5.1 Backgroud

一个进程在另一个进程被调度前只能完成部分任务

5.2 The Critical-Section Problem

- 进入区
- 临界区
- 退出区
- 剩余区

解决临界区问题的三个要求：

- 互斥：同一个临界区不能有两个进程同时执行
- 进步：只有**不在剩余区**执行的进程才能**参与决定**下一个进入临界区的进程，并且该选择**不能无限推迟**
- 有限等待：从一个进程**提出**要进入临界区到请求**允许**为止，其他进程允许进入该临界区的**次数具有上限**

下列同步方案是如何实现这三种要求的？？

两种常用方法：

- 抢占式内核
- 非抢占式内核：不会造成竞争条件

5.3 Peterson解决方案

```
int turn;
boolean flag[2];


//Pi
do {
    flag[i] = true;//表示i要进入临界区
    turn = j;//表示j允许进入临界区，允许j进，看看j想进吗（如果断点后Pj想进，那就让他进）
    while(flag[j] && turn == j);//j要进入且j允许进入，那就等

    //cs...

    flag[i] = false;//表示i不想进入临界区


}while(true)
```

6.4 硬件同步

两个特殊函数，具有原子性

- test_and_set(&lock):
  - 函数定义：返回lock的值并设置为true
  - 进入区：lock原先为false，代表未上锁，能够进入等待区且上锁（**退出循环**）
  - 退出区：将lock设为false，代表解开锁
- compare_and_swap(&val,exp,new_val):
  - 函数定义：如果旧值等于期待值，将旧值改为新值，无论如何最后返回旧值
  - 进入区：当旧值为0时代表未上锁，能够进入等待区且上锁（**退出循环**）
  - 退出区：将lock设为0，代表解开锁

6.4 互斥锁

软件方式解决临界区问题

mutex lock(mutual exclusion)

- 布尔变量：每个锁都有一个available
- 进入区
  - acquire() 获取锁
  - 若锁是可用的，那么会进入关键区并让锁不可用
  - 若锁是不可用的，等待
- 退出区
  - release()释放锁

6.6 信号量

- 初始化 s = ?
  - 计数信号量，控制具有多个实例的某种资源，初值为**可用资源数量**
  - 二进制信号量 == 互斥锁，初始值一般为**1**
- wait(s) / P
  - 若s**小于等于**0，等待（减到零就不能再减了）
  - 若s大于0，进入临界区并自减
- signal(s) /V
  - s++

6.7 经典缓冲区问题

有界缓冲区（bounded-buffer）问题的信号量解决方法

多个进程对同一块有限的缓冲区进行写入和取出操作

- 初始化

```
int n;//有界缓冲区有n个资源
semaphore mutex = 1;
semaphore empty = n;//可写入（空）的资源数为n
semaphore full = 0;//可读取（满）的资源数为0
```

- 生产者进程:

```
do {
    wait(empty);//空余的资源-1，让生产者消费者互斥
    wait(mutex);//让生产者们也互斥

    //critical section....

    signal(mutex);//解放生产者们互斥锁
    signal(full);//满的资源数+1

    //reminder section...

}while(true);
```

- 消费者进程:

```
do{
    wait(full);//满的资源-1
    wait(mutex);//让生产者们也互斥

    //cs....

    signal(mutex);
    signal(empty);

    //rs....
}while(true);
```

读者-写者（reader-writer）问题

第一读者写者问题：对同一块内存区域，读者不应保持等待，除非写者获得权限

- 初始化

```
semaphore rw_mutex = 1;//读者写者互斥锁
int r_cnt = 0;//读者数量
semaphore r_mutex = 1;//读者之间对读者数量这一关键资源的互斥
```

- 读者进程

```
do{
    //记录读者数量 +1
    wait(r_mutex);
    if(r_cnt++ == 0) wait(rw_mutex);
    signal(r_mutex);

    //read

    //记录读者数量 -1
    wait(r_mutex);
    if(--r_cnt == 0) signal(rw_mutex);
    signal(r_mutex);

    //rs
}while(true);
```

- 写者进程

```
do{
    wait(rw_mutex);

    //write

    signal(rw_mutex);

}while(true);
```

哲学家就餐（dining-philosophers）问题

一个圆桌的哲学家，两两之间有一根筷子。哲学家一般在思考，会感觉饿，若饿并且能拿起附近的两根筷子时可以吃饭
采用下列信号量的方式解决问题会造成死锁和饥饿

- 初始化

```
semaphore chopstick[5] = {1};//假设有五个哲学家，有六根筷子，每一根筷子代表一把互斥锁
```

- 哲学家 i 的进程

```
do{
    wait(chopstick[i]);
    wait(chopstick[(i + 1)%5]);//等待他附近的两根筷子

    //eat

    signal(chopstick[(i + 1)%5]);
    signal(chopstick[i]);

    //think
}while(true);
```

? 5.8 管程

chapter 6 CPU Scheduling

6.1 Basic Concepts

6.1.1 CPU –I/O Burst Cycle 执行周期

- CPU burst CPU执行
- I/O burst IO执行

- An I/O-bound program typically has many short CPU bursts IO密集型程序
- A CPU-bound program might have a few long CPU bursts CPU密集型

6.1.2 CPU Scheduler 调度程序

short-term scheduler / CPU scheduler

- The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.
- The records in the queues are generally process control blocks (PCBs) of the processes.

6.1.3 Preemptive Scheduling

CPU-scheduling decisions may take place under the following four circumstances

1. When a process switches from the running state to the waiting state
2. When a process switches from the running state to the ready state
3. When a process switches from the waiting state to the ready state
4. When a process terminates

- When scheduling takes place only under circumstances 1 and 4（程序自然放弃）, we say that the scheduling scheme is **nonpreemptive**（非抢占的） or **cooperative**（协作）.
- Otherwise, it is preemptive（抢占的）
    - preemptive scheduling can result in race conditions when **data** are shared among several processes
    - affects the design of the operating-system **kernel**（kernel level race conditions）
    - the sections of code affected by interrupts must be guarded from simultaneous use 避免同时使用

6.1.4 Dispatcher 调度程序

functions：

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

dispatch latency 调度延迟

6.2 Scheduling Criteria 调度准则

衡量调度算法的方法：

- CPU utilization 使用率
- Throughput 吞吐量 - 一段时间内进程的完成数量
- Turnaround time 周转时间- 从进程提交到进程完成的时间
- Waiting time 就绪队列中等待时间之和
- Response time 响应时间 - 从提交请求到产生（开始而非输出）第一响应的时间

6.3 Scheduling Algorithms

CPU-scheduling algorithms：deciding which of the processes in the ready queue is to be allocated the CPU

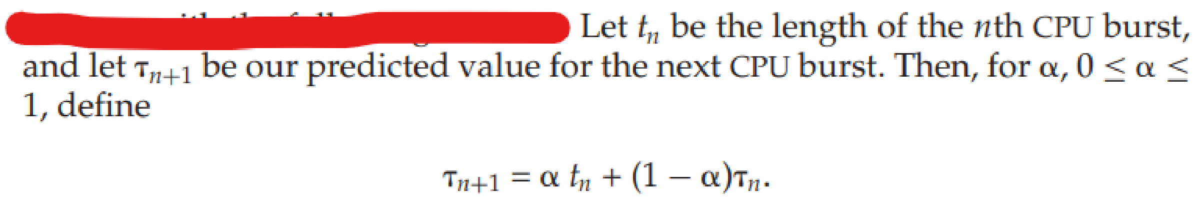
6.3.1 First-Come, First-Served Scheduling

- FIFO queue
- On the negative side, the average waiting time under the FCFS policy is often quite long
- convoy effect 护航效果
- nonpreemptive
- troublesome for time-sharing systems

6.3.2 Shortest-Job-First Scheduling

- associates with each process the length of the process’s **next CPU burst** -- shortest-next-CPU-burst
- minimum average waiting time
- The real difficulty with the SJF algorithm is knowing the length of the next CPU request
    - SJF scheduling is used frequently in **long-term scheduling**.
    - With **short-term scheduling**, there is no way to know the length of the next CPU burst - predictable:exponential average
- nonpreemptive - allow the currently running process to finish its CPU burst
- preemptive - **shortest-remaining-time-first**

The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts

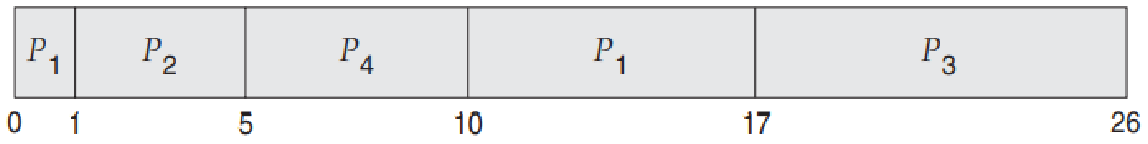


[redacted] Let t_n be the length of the nth CPU burst, and let τ_{n+1} be our predicted value for the next CPU burst. Then, for α, $0 \leq \alpha \leq 1$, define

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

calculate average wasting time：

- note the wasting time of p1:



[(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5milliseconds.

6.3.3 Priority Scheduling

The SJF algorithm is a special case of the general priority-scheduling algorithm.

the CPU is allocated to the process with the highest priority

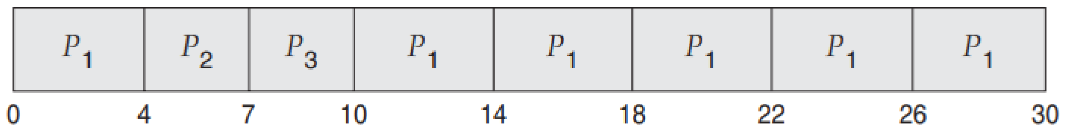
- Internally defined priorities / External priorities
- either preemptive or nonpreemptive
- problem： indefinite blocking,or starvation
    - solution:aging
        - increase the priority of a waiting process by 1 every 15 minutes

6.3.4 Round-Robin Scheduling

轮转调度

is designed especially for **time-sharing systems**

- is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
- A small unit of time - a time quantum 时间量 or time slice 时间片
- The ready queue is treated as a circular queue.The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.
- treat the ready queue as a FIFO queue of processes.New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.
    - The process may have a CPU burst of less than 1 time quantum：
        - the process itself will release the CPU
        - proceed to the next process in the ready queue
    - If the CPU burst of the currently running process is longer than 1 time quantum：
        - cause an interrupt to the operating system
        - context switch
        - the process will be put at the tail of the ready queue
        - select the next process in the ready queue.
- preemptive
- wasting time：



P1 waits for **6 milliseconds (10 - 4)** , P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds

performance：

- The performance of the RR algorithm depends heavily on the size of the time quantum
    - extremely large == FCFS policy
    - extremely small == a large number of context switches
- Turnaround time 周转时间 also depends on the size of the time quantum

6.3.5 Multilevel Queue Scheduling

多级队列调度

different groups:

- foreground (interactive) processes
- background (batch) processes

necessity：

- have different **response-time** requirements and so may have different scheduling needs
- foreground processes may have **priority** (externally defined) over background processes.

scheduling：

- within queue:has its own scheduling
- inter queue:
  - fixed-priority preemptive scheduling
  - time-slice among the queues

6.3.6 Multilevel Feedback Queue Scheduling

多级反馈队列调度

- allows a process to move between queues
  - a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents **starvation**.
- a multilevel feedback queue scheduler is defined by the following parameters:
  - The number of queues
  - The scheduling algorithm for each queue
  - The method used to determine when to upgrade a process to a higherpriority queue
  - The method used to determine when to demote a process to a lowerpriority queue
  - The method used to determine which queue a process will enter when that process needs service 进程需要服务时进入哪个队列

6.4 Thread Scheduling

6.4.1Contention Scope 竞争范围

- process contention scope (PCS) 进程竞争范围
  - among threads belonging to the same process
  - according to priority / preempt
- system-contention scope (SCS) 系统竞争范围
  - among all threads in the system

6.4.2 Pthread Scheduling

6.5 Multiple-Processor Scheduling

chapter 7 Deadlock

defination:

- a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes

7.1 System Model

types（classes）/ instances of resource

utilize a resource in only the following sequence

- Request
  - the requesting process must wait until it can acquire the resource
- Use
- Release

7.2 Deadlock Characterization

7.2.1 Necessary Conditions

- Mutual exclusion 互斥
- Hold and wait 占有并等待
- No preemption 非抢占
- Circular wait 循环等待

7.2.2 Resource-Allocation Graph

- request edge 申请边
- assignment edge 分配边
- if the graph contains **no** cycles, then **no** process in the system is deadlocked.
- If the graph does **contain** a cycle, then a deadlock **may** exist

if a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a **deadlocked state**

7.3 Methods for Handling Deadlocks

- deadlock prevention 死锁预防
- deadlock avoidance 死锁避免
- detect and recover from deadlocks 死锁检测与恢复

7.4 Deadlock Prevention 死锁预防

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock

7.4.1 Mutual Exclusion

cannot prevent deadlocks by denying the mutual-exclusion condition

7.4.2 Hold and Wait

whenever a process requests a resource, it does not hold any other resources

- hold and no wait:be allocated all its resources before process begins execution
  - system calls requesting resources for a process precede(先于) all other system calls.
- no hold and wait:a process requests resources only when it has none
  - Before it can request any additional resources, it must release all the resources that it is currently allocated.

two main disadvantages:

- resource utilization may be low
- starvation is possible

7.4.3 No Preemption

If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted ------ **implicitly released**

- If the resources are neither available nor held by a waiting process, the requesting process must wait
- This protocol is **often** applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It **cannot** generally be applied to such resources as mutex locks and semaphores.

7.4.4 Circular Wait

impose a total ordering of all resource types and to require that each process requests resources in an increasing/decreasing order of enumeration.

7.5 Deadlock Avoidance 死锁避免

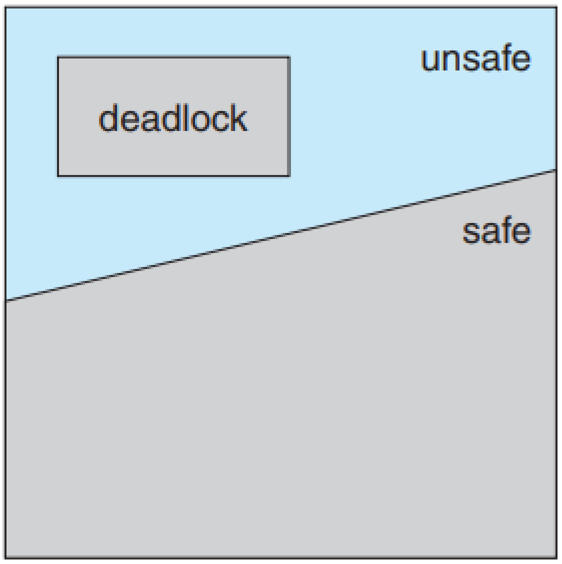
disadvantages of deadlock prevent:

- prevent deadlocks by limiting how requests can be made - low device utilization and reduced system throughput

- Deadlock Avoidance: require **additional information** about how resources are to be requested.With this knowledge of the complete sequence of requests and releases for each process, the system can **decide** for each request **whether or not the process should wait**.
- **deadlock-avoidance algorithm**: dynamically examines the **resource-allocation state** to ensure that a **circular-wait condition** can never exist
- state:the number of **available** and **allocated** resources and the maximum **demands** of the processes

7.5.1 Safe State

- safe state:if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock
- safe sequence 安全序列
    - 一个进程需要的资源可以由 可用资源 + 前一个进程终止释放的资源 满足（第一个进程只由现在可用资源即可满足，然后进行释放资源）
- deadlocked state : unsafe state
- **Not all unsafe states are deadlocks**, however . An unsafe state may lead to a deadlock



- avoidance algorithms:
  - ensure that the system will always remain in a safe state
  - Initially, the system is in a safe state.
  - Whenever a process **requests** a resource that is currently available, the system must decide **whether** the resource can be **allocated immediately** or whether the process must **wait**
    - The request is granted only if the allocation leaves the system in a safe state

与没采用死锁避免算法相比，使用后资源利用效率会更低

7.5.2 Resource-Allocation-Graph Algorithm

only one instance of each resource type：

- claim edge 需求边
  - A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the futur
  - When process Pi requests resource Rj , the claim edge Pi → Rj is converted to a request edge
  - when a resource Rj is released by Pi , the assignment edge Rj → Pi is reconverted to a claim edge Pi → Rj
- Now suppose that process Pi requests resource Rj . The request can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph

7.5.3 Banker's Algorithm 银行家算法

The resource-allocation-graph algorithm is not applicable to a resource allocation system with **multiple instances of each resource type**. The deadlockavoidance algorithm that we describe next is applicable to such a system but is **less efficient** than the resource-allocation graph scheme.

n is the number of processes in the system and m is the number of resource types

data structures encode the state of the resource-allocation system:

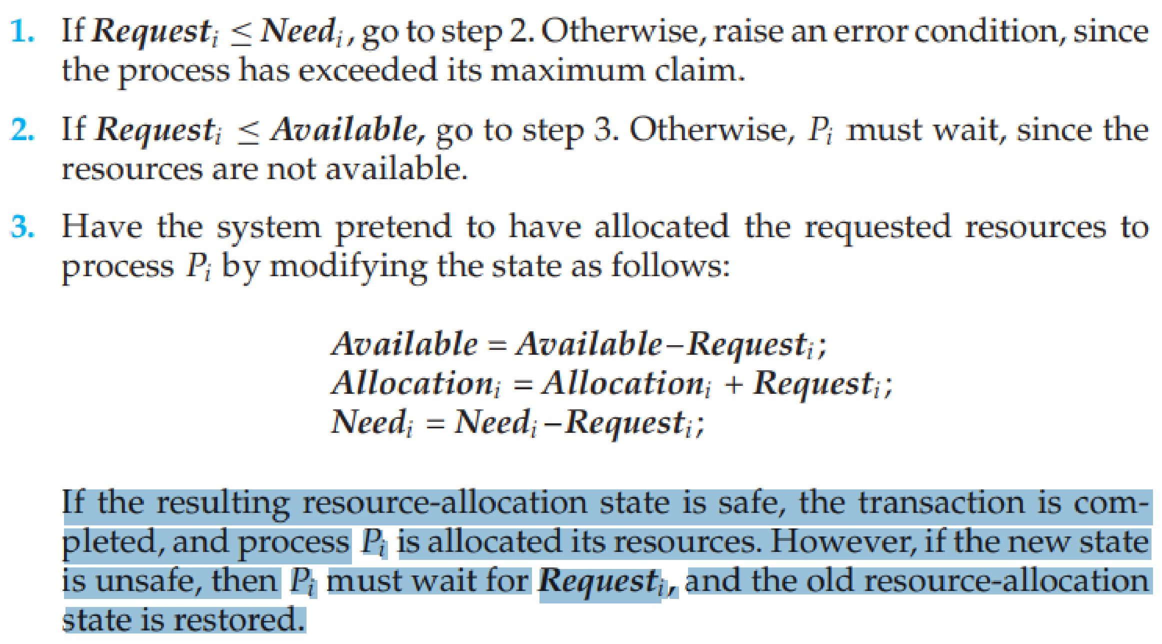
- Available
  - If Available[j] equals k, then k instances of resource type Rj are available.
- Max
  - If Max[i] [j] equals k, then process Pi may request at most k instances of resource type Rj .
- Allocation
  - If Allocation[i] [j] equals k, then process Pi is currently allocated k instances of resource type Rj .
- Need
  - If Need[i] [j] equals k, then process Pi may need k more instances of resource type Rj to complete its task
- Let Requesti be the request vector for process Pi . If Requesti [j] == k, then process Pi wants k instances of resource type Rj .

7.5.3.1 Safety Algorithm

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n − 1. 2.
2. 找到一个能够完成的未完成进程：
   1. Find an index i such that both
      1. Finish[i] == false
      2. Needi ≤ Work

      If no such i exists, go to step 4.
3. 将完成该进程所释放的资源加到目前可用资源数量的work中：
   1. Work = Work + Allocationi
   2. Finish[i] = true
   3. Go to step 2.
4. If Finish[i] == true for all i, then the system is in a safe state.

7.5.3.2 Resource-Request Algorithm



1. If $\textbf{\textit{Request}}_i \le \textbf{\textit{Need}}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $\textbf{\textit{Request}}_i \le \textbf{\textit{Available}}$, go to step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$\begin{aligned}&\textbf{\textit{Available}} = \textbf{\textit{Available}} - \textbf{\textit{Request}}_i;\\&\textbf{\textit{Allocation}}_i = \textbf{\textit{Allocation}}_i + \textbf{\textit{Request}}_i;\\&\textbf{\textit{Need}}_i = \textbf{\textit{Need}}_i - \textbf{\textit{Request}}_i;\end{aligned}$$

If the resulting resource-allocation state is safe, the transaction is completed, and process P_i is allocated its resources. However, if the new state is unsafe, then P_i must wait for $\textbf{\textit{Request}}_i$, and the old resource-allocation state is restored.

7.6 Deadlock Detection

detection-and-recovery scheme requires overhead

- run-time costs of maintaining the necessary information and executing the detection algorithm
- the potential losses inherent（固有） in recovering from a deadlock.

7.6.1 Single Instance of Each Resource Type

wait-for graph

- a variant（变体） of the resource-allocation graph
- an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs
- a deadlock exists in the system if and only if the wait-for graph **contains a cycle**

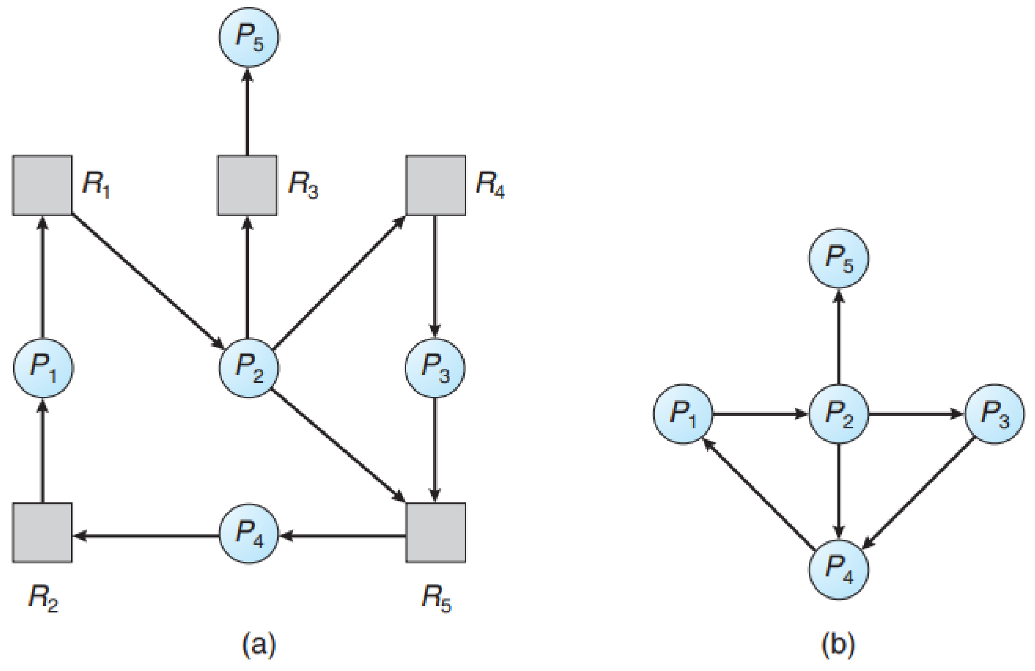


Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

7.6.2 Several Instances of a Resource Type

similar to the banker's algorithm

- Available
    - A vector of length m indicates the number of available resources of each type
- Allocation
    - An n × m matrix defines the number of resources of each type currently allocated to each process.
- Request
    - An n × m matrix indicates the current request of each process. If Request[i][j] equals k, then process Pi is requesting k more instances of resource type Rj

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available. For i = 0, 1, ..., n–1,Finish[i] = false（对finish 的初始化是我修改后的，认为原版的有错误：将request000的进程都设为true的话，在他给出的例子中，第二步无法进行）
2. Find an index i such that both
    1. Finish[i] == false
    2. **Request**i （非Needi）≤ Work
    3. If no such i exists, go to step 4.
3. Work = Work + Allocationi Finish[i] = true Go to step 2.
4. If Finish[i] == false for some i, 0 ≤ i < n, then the system is in a deadlocked state. Moreover, if Finish[i] == false, then process Pi is deadlocked.

7.6.3 Detection-Algorithm Usage

When should we invoke the detection algorithm

- we can invoke the deadlockdetection algorithm every time a request for allocation cannot be granted immediately
- invoke the algorithm at defined intervals

7.7 Recovery from Deadlock

- abort one or more processes to break the circular wait
- preempt some resources from one or more of the deadlocked processes.

7.7.1 Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
  - we should abort those processes whose termination will incur the minimum cost

7.7.2 Resource Preemption

three issues need to be addressed

- Selecting a victim 选择牺牲进程
- Rollback 回滚
- Starvation 饥饿

chapter 8 Main Memory

8.1 Backgroud

8.1.1 Basic Hardware

any instructions in execution, and any data being used
by the instructions, must be in one of these direct-access storage devices.

- main memory
- registers built into the processor(有些寄存器不归处理器管,如硬件寄存器)

speed of accessing

- cache - automatically speeds up memory access without any operating-system control.

correct operation/Protection of memory space - 不要看到不该看的东西：如不同的用户进程，用户进程和操作系统内核

This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

- base register- limit register：determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses/**only the operating system** can load the base and limit registers

8.1.2 Address Binding

不同形式的地址 两次绑定在是这三种形式的映射

- symbolic
- relocatable addresses
- absolute addresses

地址绑定的时机

- Compile time
  - source code -(compiler)->absolute code
  - starting location changes->recompile
  - identical logical and physical addresses
- Load time
  - source code -(compiler)->relocatable code
  - starting location changes->reload
  - identical logical and physical addresses
- Execution time : Most popular
  - differing logical and physical addresses.

8.1.3 Logical Versus Physical Address Space

logical address

- generated by the CPU - 存在内存中的
- =virtual address

physical address

- seen by the memory unit/loaded into the memory-address register of the memory - 向内存发送的

memory-management unit (MMU):mapping from virtual to physical addresses

8.1.4 Dynamic Loading

a routine is not loaded until it is calle

- useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines.

8.1.5 Dynamic Linking and Shared Libraries

linking is postponed until execution time

- save both disk space and main memory.
- stub - 一小段代码，指示如何定位适当的内存驻留库例程，或者如果例程尚不存在，如何加载库
    - replaces itself with the address of the routine and executes the routine.
    - the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
- easy to library updates

shared libraries:只有使用新库版本编译的程序才会受到其中包含的任何不兼容更改的影响。在安装新库之前链接的其他程序将继续使用旧的库

8.2 swapping

swapped temporarily out of memory to a backing store(通常是快速磁盘) and then brought back into memory for continued execution

交换使得所有进程的总物理地址空间可以超过系统的真实物理内存，从而提高系统的多道程序设计程度。

8.2.1 Standard Swapping

ready queue

- memory images are **on the backing store or in memory** and are ready to run

dispatcher

- checks to see whether the next process in the queue is in memory.
- not in memory and no free in memory : swap
- reloads registers and transfers control to
  the selected process.

context-switch time

- major part of the swap time is transfer time.
- know exactly how much memory a user process is using

problems of swapping process waiting I/O operation：

- never swap a process with pending I/O
- execute I/O operations only into **operating-system buffers** - double buffering.

8.3 Contigous Memory Allocation

In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process.

8.3.1 Memory Protection

relocation-register scheme

- relocation register
- limit register
- protect both the operating system and the other users' programs and data from being modified by this running process.
- allow the operating system's size to change dynamically

8.3.2 Memory Allocation

fixed-sized partitions

- the degree of multiprogramming is bound by the number of partitions.

variable-partition

- hole
- 可以有一个表记录内存的使用情况，如用始末地址的方式记录已被占用的空间，这与固定分区的用位来映射分区使用情况的表不一样

select a free hole from the set of available holes

- First fit 开始位置可以是开头或者结尾亦或者是上次查询的位置
- Best fit 选择满足条件的最小孔，最终生成最小的剩余孔
- Worst fit 每次选择最大的孔，最终生成最大的剩余孔，有时会有用

8.3.3 Fragmentation

external fragmentation

- there is enough total memory space to satisfy a request but the available spaces are not contiguous
- 50-percent rule:给定 N 个分配的块，另外 0.5 N 个块将因碎片而丢失。也就是说，有**三分之一**的内存可能无法使用
- solution : compaction 紧缩
    - it is possible only if relocation is dynamic and is done at execution time
    - expensive
- solution : permit the **physical**(课本原文是logical,可能有错误，因为下文也提到了两种方法都是允许进程由不连续的物理地址空间) address space of the processes to be noncontiguous
    - segmentation & paging

internal fragmentation

- the memory allocated to a process may be slightly larger than the requested memory.
- unused memory that is internal to a partition.

8.4 Segmentation

8.4.1 Basic Method

程序员的内存视角：将内存视为可变大小段的集合，段之间没有必要排序

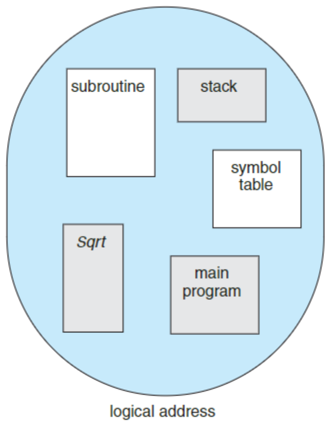


Figure 8.7 Programmer's view of a program.

logical address = a segment name(segment-number) and an offset

8.4.2 Segmentation Hardware

segment table:将二位的逻辑地址映射为一维的物理地址

- Each entry in the segment table has a segment base and a segment limit
- The segment number is used as an **index** to the segment table --> **base**
- The offset of the logical address must be between 0 and the segment limit.If it is not, we **trap** to the operating system (logical addressing attempt beyond end of segment)--> **offsets**
- paddr = base + offsets

8.5 Paging

分页的优势：

- paging avoids **external fragmentation** and the need for **compaction**, whereas segmentation does not.
- solves the problem of fitting memory chunks of varying sizes onto the backing store.
- paging lets us use physical memory that is larger than what can be addressed by the CPU's address pointer length
- possibility of sharing common code.

缺点：

- have some internal fragmentation.

8.5.1 Basic Method

frames 帧 & pages 页

- same size
- The backing store is divided into **fixed-sized blocks** that are the same size as the memory frames or clusters of multiple frames.
- the logical address space is now totally separate from the physical address space

Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset** (d)

- The page number is used as an **index** into a page table
- paddr = base address + page offset
- dynamic relocation

page sizes：

- 为什么页大小要是2的指数：
  - The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.If the size of the logical address space is 2 m , and a page size is 2n bytes, then the high-order m − n bits of a logical address designate the page number, and the n low-order bits designate the page offset.
- small page sizes：
  - less internal fragmentation
  - overhead is involved in each page-table entry
  - disk I/O is more efficient when the amount data being transferred is larger

frame table:

- one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

Paging increases the context-switch time:

- The operating system maintains a copy of the page table for each process
  - translate logical addresses to physical addresses
  - define the hardware page table

8.5.2 Hardware Support

hardware implementation of the page table

- a set of dedicated registers - 表条目的数量少
- page-table base register (**PTBR**) points to the page table
  - reducing context-switch time
  - 问题：内存访问慢：two memory accesses are needed to access a byte
- cache:translation look-aside buffer (**TLB**)
  - 查TLB的过程是cpu指令流水线的一部分-adding no performance penalty compared with a system that does not implement paging.
  - TLB miss
  - replacement
  - wired down:TLB entries for key kernel code are wired down.不被换出
  - address-space identifiers (ASIDs) : address-space protection for a process & allows the TLB to contain entries for several different processes simultaneously.
  - hit ratio 命中率

effective memory-access time：

- TLB命中的话，映射内存访问（ mapped-memory access ）只要一次引用时间（memory access time）
- TLB失效的话，引用内存需要两次引用时间

8.5.3 Protection

protection bits - in the page table

- One bit can define a page to be read–write or read-only ->expand. illegal attempts will be trapped to the operating system
- a valid–invalid bit -> illegal addresses are trapped by use of the valid–invalid bit

page-table length register (PTLR)

- In fact, many processes use only a small fraction of the address space available to them.It would be wasteful in these cases to create a page table with entries for every page in the address range

8.5.4 Shared Pages

sharable code must be reentrant.Reentrant code is non-self-modifying code

8.6 Structure of the page table

8.6.1 Hierarchical Paging

页表很大->divide the page table into smaller pieces

- two-level paging algorithm - page table itself is also page
    - page number = p1 & p2
    - p1 = index of outer page table
    - p2 = displacement of inner page table
    - = forward-mapped page table

8.6.2 Hashed Page Tables

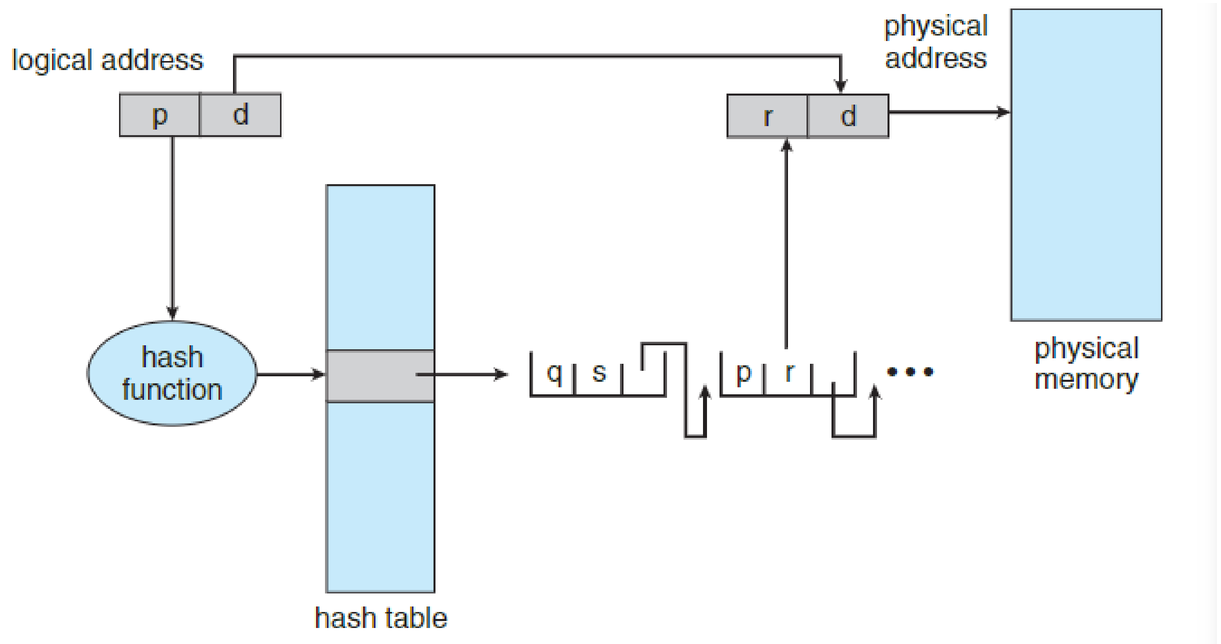


Figure 8.19 Hashed page table.

skip 8.6.3 Inverted Page Tables

普通分页的缺点：each page table may consist of millions of entries.These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

- has one entry for each real page (or frame) of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page
- only one page table is in the system, and it has only one entry for each page of physical memory. = 节约内存空间
- increases the amount of time needed to search the table
- have difficulty implementing shared memory.

chapter 9 Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory

- programs can be larger than physical memory.
- abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- share files easily and to implement shared memory
- provides an efficient mechanism for process creation

9.1 Background

in many cases, the entire program is not needed.

- handle unusual error conditions.
- Arrays, lists, and tables are often allocated more memory than they actually need.
- Certain options and features of a program may be used rarely.

执行一个部分在内存内的程序 优势：

- no longer be constrained by the amount of physical memory that is available
- more programs could be run at the same time
- Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.（在交替执行不同进程时，有可能不需要进行换出操作就可以进行，也有可能仅交换部分内存即可完成，比全部载入和换出节约了硬盘IO的时间）

virtual address space --> sparse address spaces 有孔

benefit:virtual memory allows files and memory to be shared by two or more processes through page sharing

- System libraries can be shared by several processe
- processes can share memory
- Pages can be shared during process creation

9.2 Demand Paging

请求调页：load pages only as they are needed -> lazy swapper/pager

9.2.1 Basic Concepts

- valid–invalid bit:distinguish between the pages that are in memory and the pages that are on the disk.
- memory resident - execution proceeds normally

- page fault:

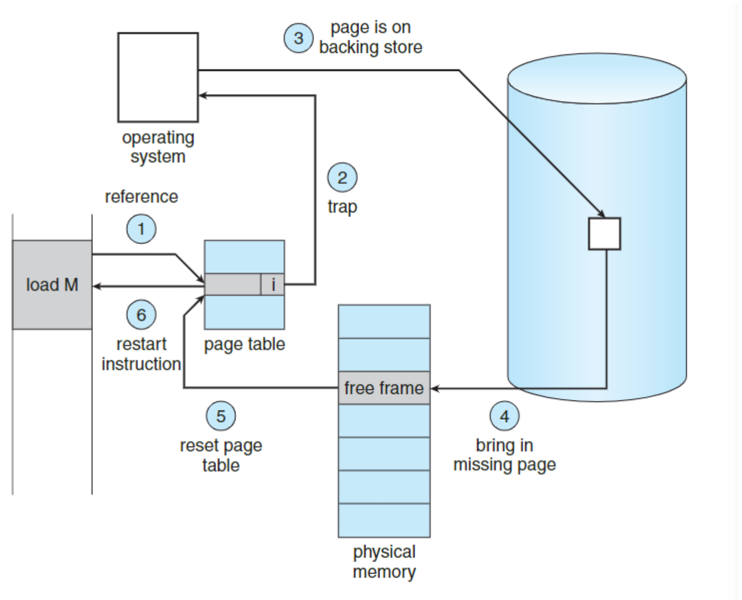


Figure 9.6 Steps in handling a page fault.

- pure demand paging: never bring a page into memory until it is required(区别就是最开始的时候啥也没有，请求调页开始时可能有页面在内存中)
- locality of reference局部引用：让请求调页有一个良好的性能
- 核心要求：restart any instruction after a page fault
- one instruction may modify several different locations
    - microcode computes and attempts to access both ends of both blocks.If a page fault is going to occur, it will happen at this step, before anything is modified
    - uses temporary registers to hold the values of overwritten locations

9.2.2 Performance of Demand Paging

`effective access time = (1 − p) × ma + p × page fault time.`

ma = memory-access time

- 问题：当没有缺页时，不应该是两次内存访问，一次查页表一次引用吗？
    - 这里的ma就是分页机制下读取内存的时间，这里**概念混乱**，注意辨别
    - 另一种解释：这里没有涉及分页，不用考虑两次（TLB hit）或者一次（TLB miss）内存访问

three major components of the page-fault service time:

- Service the page-fault interrupt 处理缺页中断
- Read in the page 将页读入内存
- Restart the process 重启进程

the handling and overall use of swap space

交换空间比文件系统快，故：

- 在进程启动时将整个文件映像复制到交换空间，然后从交换空间执行请求分页。

- 从文件系统请求页面，但在替换页面时将其写入交换空间

9.3 Copy-on-Write

- rapid process creation and minimizes the number of new pages that must be allocated to the newly created process
- lowing the parent and child processes initially to share the
  same pages. These shared pages are marked as copy-on-write pages, meaning
  that if either process writes to a shared page, a copy of the shared page is
  created

9.4 Page Replacement

over-allocate memory - higher CPU utilization and throughput

page replacement:内存过度分配后，某个程序有新的页面请求，但内存已满，需要页面置换

free a frame：

- writing its contents to swap space
- changing the page table (and all other tables) to indicate that the page is no longer in memory

victim frame

- modify bit or dirty bit - indicating that the page has been modified
- set = write the page to the disk.
- not set = need not write the memory page to the disk: -- read-only pages or no modifying

实现请求调页的两个主要问题：
frame-allocation algorithm and **page-replacement** algorithm -->lowest page-fault rate

reference string 引用序列?

- 页号的序列
- 连续重复的页号和为一个

page-replacement algorithm

错误的替换选择会增加页面错误率并减慢进程执行速度。但是，它不会导致错误执行。

- Belady’s anomaly异常：the page-fault rate may increase as the number of allocated frames increases
- FIFO： 先进先出，选最早进入内存的进行换出

- optimal：最佳替换算法，选最长时间不再使用的（在未来最晚引用的那个）
- LRU (least recently used):最久最近（上次）使用算法，选择一个最长时间未使用的 | **good**
    - implementations 1：Counters：更新该页被使用的时间
    - Stack: 引用页后将该页放置在栈顶，最近最久使用的页永远在栈底
- LFU(least frequently used): 最少使用算法，选择最少使用的那个页面
- MFU(most frequently used):最多使用算法，选择最多使用的那个页面

Page-Buffering Algorithms

- free-frame pool
    - 记录需要写回磁盘的帧/页，当系统空闲时，可以选择某个页面写回并设置该页面为不需要写回
    - 记录不需要写回磁盘的帧，组成空闲帧池
- allows the process to restart as soon as possible, without waiting for the victim page to be written out.
- When the victim is later written out, its frame is added to the free-frame pool
- 它可以是对任何页面替换算法的有用增强，以减少选择错误的受害者页面时产生的损失。

9.5 Allocation of Frames

how many frames does each process get? the user process is allocated any free frame.

Minimum Number of Frames

- performance - the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
- 必须有足够的帧来容纳任何单个指令可以引用的所有不同页面
- Whereas the minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory

Allocation Algorithms

- equal allocation 平均分配
- proportional allocation 按比例分配

classify page-replacement algorithms into two broad categories

- global replacement 从全部的帧中选择一个 **常用**
- local replacement   选择的时候只在分配给进程的集合里选

9.6 Thrashing 系统抖动

What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?

- 抖动是由进程所需的最小页面数分配不足引起的，这会导致进程出现连续的缺页错误。
- 系统可以通过评估CPU利用率水平与多道程序程度的比较来检测抖动。
- 它可以通过降低多道程序的程度来消除。

chapter 10 Mass -Storage Structure

10.2 Disk Structure

one-dimensional arrays of logical blocks = the smallest **unit of transfer**

- 扇区 0 是最外柱面第一磁道的第一个扇区。映射按顺序通过该磁道，然后通过该柱面中的其余磁道，然后从最外层到最内层通过其余柱面。
- 将逻辑块号转换为旧式磁盘地址很困难：
  - defective sectors
  - the number of sectors per track is not a constant on some drives.

10.4 Disk Scheduling

acess time:

- seek time： disk arm to move the heads to the cylinder containing the desired sector
- rotational latency : rotate the desired sector to the disk head

bandwidth 带宽：传输的**总字节**数除以第一次服务请求和最后一次传输完成之间的**总时间**

chooses which pending request to service next:

?examples in textbook?

- FCFS Scheduling(first-come, first-served):谁先来就选谁
- SSTF Scheduling(shortest-seek-time-first)：谁的寻道时间最短（最靠近磁头位置）就选谁
  - 可能会造成饥饿(starvation)
- SCAN Scheduling(elevator algorithm电梯调度算法)：The head continuously scans back and forth across the disk 一段到另一端的折返
- C-SCAN Scheduling(Circular SCAN)：C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.When the head reaches the other end, however, it immediately returns to the beginning of the disk **without servicing any requests on the return trip**（与普通的扫描调度的不同点）
  - 提供更同一的等待时间。scan调度在磁头反转时处理的请求是在翻转前刚刚处理过的请求，应该优先处理那些远离这一段的请求，因为他们等待的时间更久
- LOOK/C-LOOK scheduling :it reverses direction immediately, without going all the way to the end of the disk

- performance depends heavily on the **number** and **types** of requests.
- The scheduling algorithms described here consider **only the seek distances**

chapter 11 File -System Interface

the file system consists of two distinct parts: a **collection of files**, each storing related data, and a **directory structure**, which organizes and provides information about all the files in the system

11.1 File Concept

file

- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit操作系统从其存储设备的物理属性中抽象出来定义一个逻辑存储单元
- A file is a named collection of related information that is recorded on secondary storage.
- the smallest allotment of logical secondary storage - 数据无法写入辅助存储，除非它们位于文件内

File Attributes:

- Name
- Identifier : 不是给人看的
- Type

- Location
- Size
- Protection：访问控制
- Time, date, and user identification

The information about all files is kept in the directory structure, which also resides on secondary storage.：

- a directory entry consists of the file’s name and its unique identifier
- locates the other file attribute

minimal set of File Operations：

- create： allocate **space** for the file / an **entry** for the new file must be made in the directory.
- write : write pointer
- read : read pointer ---> current- file-position pointer
- reposition : =seek , need not involve any actual I/O
- delete : release all file **space** / erase the directory **entry**
- truncate files : only file **space** released 只删除文件的内容，但是不改变文件的属性

open-file table：

- avoid constantly searching the directory for the entry associated with the named file
- containing information about all open files
- via index ,no searching
- open() : takes a file name and searches the directory, copying the directory entry into the open-file table
- two levels of internal tables:
    - a per-process table : information regarding the process’s use of the file
    - a system-wide table : process-independent information : location/**open count**

informations associated with an open file

- File pointer
- File-open count : when the open count reaches zero, the file is **no longer in use**, and the file’s entry is removed from the open-file table
- Disk location of the file : Most file operations require the system to modify data within the file.The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.

- Access rights

file locks

- shared lock : 共享锁，读，可以同步读
- exclusive lock： 独占锁， 写，只有一个进程可以写
- mandatory:强制的，操作系统保证锁的获取
- advisory : 建议的，由程序员决定是否上锁

file types:

- A common technique for implementing file types is to include the type as part of the file name.The name is split into two parts—a name and an extension

file structures

文件类型 可用来指示 文件内部的结构

支持多文件结构的缺点

- 操作系统会变得**复杂**，需要包含支持各种文件结构的代码
- 通用性差，每个文件都要定义为操作系用支持的文件类型之一。若用户定义一个操作系统不支持的文件，可能会出现问题

11.2 Access Methods

如何将存储在磁盘上的文件读取到内存?

- Sequential Access：
    - Information in the file is processed **in order**, one record after the other.
    - 编译器，编辑器
- Direct Access （relative access）：
    - a file is made up of fixed-length **logical records**，read and write records rapidly in no particular order
    - 相对块号：防止访问不属于其文件的文件系统部分
- index access:
    - 先搜索索引， 再根据指针进行访问文件

11.3 Directory and Disk Structure

如何存储文件?

- 分区：每个分区都有单独的文件系统-> 卷（volume）
- 设备目录（目录）：记录卷上所有文件的信息：大小位置名称等

计算机有多个存储设备，可以将一个设备分成多个分区，也可以将多个设备合成一个分区，一个分区包含目录和文件

目录：

- 一个符号表，一个特殊的文件，包括一组**文件或子目录**
- 每个目录条目代表一个文件
- 单级目录：所有文件在一个目录下
- 两级目录：用户文件目录（UFD）/主文件目录（MFD）
- 树形目录：
  - 每个进程都有一个**当前目录**，open函数搜索的范围
  - 绝对路径名：从根目录开始的路径名（包括根目录）
  - 相对路径名：从当前路径开始的路径名（路径名里面不包括当前目录）
- 无环目录图：共享的子目录或者文件

11.4 File-System Mounting

a file system must be mounted before it can be available to processes on the system：目录结构构建在卷上，卷必须先安装才能使其可用于文件系统的名称空间

安装：

- 设备名称（文件系统）
- 安装点，通常是空目录

chapter 12 File -System Implementation

12.4 Allocation Methods

- 连续分配：

12.5 Free-Space Management

chapter 13 I/O Systems

13.1 Overview

13.2 I/O Hardware