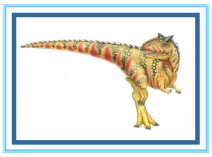
Chapter 6: Synchronization Tools

同步工具



Chapter 6: Synchronization Tools

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Mutex Locks
- Semaphores

Objectives

- Describe the **critical-section problem** and illustrate the **race condition**
- Describe hardware solutions to the critical-section problem using **compare-and-swap operations**, and **atomic variables**
- Demonstrate how **mutex locks, semaphores**, and **condition variables** can be used to solve the critical section problem

Background

- Processes execute concurrently
  - Processes may be interrupted at any time, partially completing execution, due to a variety of reasons.
- 并发访问 Concurrent access to any shared data may result in data inconsistency 数据不一致
- Maintaining data consistency requires OS mechanisms to ensure the orderly execution of cooperating processes

Illustration of the Problem

- Think about the Producer-Consumer problem
- An integer **`counter`** is used to keep track of the number of buffers occupied.
  - Initially, **`counter`** is set to 0
  - It is incremented each time by the producer after it produces an item and places in the buffer
  - It is decremented each time by the consumer after it consumes an item in the buffer.

Producer-Consumer Problem

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

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

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

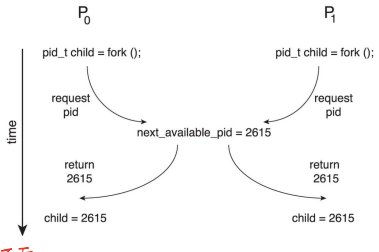
```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer execute register1 = counter          {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = counter          {register2 = 5}
S3: consumer execute register2 = register2 - 1    {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4}
```


Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (`pid`)



- Unless there **is mutual exclusion**, the same pid could be assigned to two different processes!

Critical Section Problem

- A **Race Condition** is an undesirable situation where several processes access or/and manipulate a shared data concurrently and the outcome of the executions depends on the particular order in which the accesses or executions take place - The results depend on the timing execution of programs. With some bad luck (i.e., context switches that occur at untimely points during execution), the result become **non-deterministic**
- Consider a system with *n* processes {P_0, P_1, … P_{n-1}}
- A process has a Critical Section segment of code (can be short or long), during which
  - A process or thread may be changing shared variables, updating a table, writing a file, etc.
  - We need to ensure when one process is in Critical Section, no other can be in its critical section
  - In a way, **mutual exclusion** and **critical section** imply the same thing
- Critical section problem is to design a protocol to solve this
  - Specifically, each process must ask permissions before entering a critical section in entry section, may follow critical section with exit section, then remainder section

Critical Section

- The general structure of process p_i is

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```


Solution to Critical-Section Problem

1. **Mutual exclusion** - If process P_i is executing in its critical section, no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, the selection of a process that will enter the critical section next *cannot* be postponed *indefinitely* – selection of one process entering
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted – any waiting process

- Assume that each process executes at a nonzero speed, and there is no assumption concerning relative speed of each individual process

Critical-Section Problem in Kernel

- Kernel code - the code the operating system is running, is subject to several possible race conditions
  - A kernel data structure that maintains a list of all open files can be updated by multiple kernel processes, i.e., two processes were to open files simultaneously
  - Other kernel data structures such as the one maintaining memory allocation, process lists, interrupt handling etc.
- Two general approaches are used to handle critical sections in operating system, depending on whether the kernel is preemptive or non-preemptive
  - Preemptive – allows preemption of process when running in the kernel mode, not free from the *race condition*, and increasingly more difficult in SMP architectures.
  - Non-preemptive – runs until exiting the kernel mode, blocks, or voluntarily yields CPU. This is essentially free of race conditions in the kernel mode, possibly used in single-processor system

Synchronization Tools 同步工具

- Many systems provide hardware support for implementing the critical section code. On uniprocessor systems – it could simply disable interrupts, currently running code would execute without being preempted or interrupted. But this is generally inefficient on multiprocessor systems
- Operating systems provide hardware and high level API support for critical section code

Programs	Share Programs
Hardware	Load/Store, Disable Interrupts, Test&Set, Compare&Swap
High level APIs	Locks, Semaphores

禁用中断
锁 信号量

Synchronization Hardware 同步硬件

- Modern OS provides special atomic 原子 hardware instructions
  - **Atomic** = non-interruptible (不可中断)
  - This ensures the execution of atomic instruction can not be interrupted, thus, no race condition can occur
  - The building block for more sophisticated synchronization mechanisms
- There are two commonly used atomic hardware instructions, which can be used to construct more sophisticated synchronization tools
  - Test a memory word and set a value – `Test_and_Set()`
  - Swap contents of two memory words – `Compare_and_Swap()`

test_and_set Instruction

通常用于实现锁机制

- Definition:

```
boolean test_and_set (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
 }
```


Solution using test_and_set()

利用它实现锁机制的方法:

- Shared Boolean variable **lock**, initialized to **FALSE**
- Solution:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
   /* critical section */
   lock = false;
   /* remainder section */
} while (true);
```

① 如果 lock 为 false (未被占用), 则 test_and_set() 将其设置为 true 并返回 false, 表示成功获取锁

② 如果 lock 为 true (被占用), 则 test_and_set() 将其设置为 true, 表示获取锁失败, 进程将继续在这个 while 循环中等待

4. **临界区**: `/* critical section */`
在成功获取锁后，进程进入临界区。在这个区域内，进程可以安全地访问共享资源，因为其他进程无法同时进入。

5. **释放锁**: `lock = false;`
一旦进程完成了对临界区的操作，它将 `lock` 设置回 `FALSE`，表示锁已被释放，其他进程可以尝试获取锁。

compare_and_swap Instruction

- Definition:

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


Solution using compare_and_swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

```
do {
   while (compare and swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   /* critical section */
   lock = 0;
   /* remainder section */
} while (true);
```

3. **尝试获取锁**: `while (compare_and_swap(&lock, 0, 1) != 0)`
这行代码调用 `compare_and_swap()` 函数来尝试获取锁。`compare_and_swap(&lock, 0, 1)` 的作用是:
   - 检查 `lock` 的当前值是否为 `0` (表示未被占用)。
   - 如果是，则将 `lock` 设置为 `1` (表示已被占用)，并返回 `0`。
   - 如果不是，则返回当前的 `lock` 值 (即 `1`)，表示获取锁失败，进程将继续在这个 `while` 循环中等待。

4. **临界区**: `/* critical section */`
在成功获取锁后，进程进入临界区。在这个区域内，进程可以安全地访问共享资源，因为其他进程无法同时进入。

5. **释放锁**: `lock = 0;`
一旦进程完成了对临界区的操作，它将 `lock` 设置回 `0`，表示锁已被释放，其他进程可以尝试获取锁。

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false; /* no one is waiting, so release the lock */
   else
      waiting[j] = false; /* Unblock process j */
   /* remainder section */
} while (true);
```



```
do {
    waiting[i] = true; // 标记进程 i 正在等待
    key = true;        // 初始化 key 为 true
    while (waiting[i] && key) // 当进程 i 等待且 key 为 true 时循环
        key = test_and_set(&lock); // 尝试获取锁，更新 key 的值
    waiting[i] = false; // 退出等待状态，表示进程 i 已经获得锁
    /* critical section */ // 进入临界区，安全访问共享资源
    j = (i + 1) % n; // 设置 j 为下一个进程的索引
    while ((j != i) && !waiting[j]) // 检查是否有其他进程在等待
        j = (j + 1) % n; // 如果没有，继续检查下一个进程
    if (j == i) // 如果没有其他进程在等待
        lock = false; // 释放锁
    else
        waiting[j] = false; // 解除进程 j 的阻塞状态
    /* remainder section */ // 退出临界区，执行剩余部分
} while (true); // 无限循环
```

1. **等待标志**：`waiting[i] = true;`
   这行代码将当前进程 `i` 的等待状态设置为 `true`，表示它正在尝试获取锁。
2. **初始化 key**：`key = true;`
   初始化一个布尔变量 `key` 为 `true`，用于控制进程是否可以继续尝试获取锁。
3. **获取锁**：`while (waiting[i] && key)`
   这个循环会持续执行，直到进程 `i` 不再等待或 `key` 被设置为 `false`。在循环内部，调用 `test_and_set(&lock)` 尝试获取锁：
   - 如果成功获取锁，`key` 将被更新为 `false`，进程 `i` 将继续执行。
   - 如果锁已被其他进程占用，`key` 将保持为 `true`，进程 `i` 将继续等待。
4. **退出等待**：`waiting[i] = false;`
   一旦进程 `i` 成功获取锁，它将其等待状态设置为 `false`，表示不再等待。
5. **临界区**：`/* critical section */`
   在这一部分，进程 `i` 可以安全地访问共享资源，因为它已经获得了锁。
6. **检查其他进程**：`j = (i + 1) % n;`
   设置 `j` 为下一个进程的索引，以检查是否有其他进程在等待锁。
7. **寻找等待的进程**：`while ((j != i) && !waiting[j])`
   这个循环检查是否有其他进程在等待锁。如果没有，`j` 将继续指向下一个进程。
8. **释放锁**：`if (j == i)`
   如果 `j` 回到了 `i`，这意味着没有其他进程在等待，进程 `i` 可以释放锁。
9. **解除阻塞**：`else waiting[j] = false;`
   如果有其他进程在等待，解除进程 `j` 的阻塞状态，允许它继续执行。
10. **剩余部分**：`/* remainder section */`
    这部分代码表示进程在完成临界区操作后可以执行的其他任务。

Sketch Proof

- **Mutual-exclusion**: P_i enters its critical section only if either `waiting[i]==false` or `key==false`. The value of `key` can become `false` only if `test_and_set()` is executed. Only the first process to execute `test_and_set()` will find `key==false`; all others must wait. The variable `waiting[i]` can become `false` only if another process leaves its critical section; only one `waiting[i]` is set to `false`, thus maintaining the mutual-exclusion requirement.
- **Progress**: since a process existing its critical section either sets `lock` to `false` or sets `waiting[j]` to `false`. Both allow a process that is waiting to enter its critical section to proceed.
- **Bounded-waiting**: when a process leaves its critical section, it scans the array `waiting` in cyclic order {`i+1, i+2, …,n-1,0,1,…i-1`}. It designates the first process in this ordering that is in the entry section (`waiting[j]==true`) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within `n-1` turns.

Atomic Variables 原子变量

- Typically, instructions such as compare-and-swap are used as building blocks for other (more sophisticated) synchronization tools.
- One tool is an **atomic variable** that provides atomic (uninterruptible) updates on basic data types such as integers and Booleans.
- For example, the **`increment()`** operation on the atomic variable **`sequence`** ensures **`sequence`** is incremented without interruption - **`increment(&sequence)`**;
  - The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
   int temp;

   do {
         temp = *v;
   }
   while (temp !=
(compare_and_swap(v,temp,temp+1));
}
```


Mutex Locks 互斥锁

- OS builds a number of software tools to solve the Critical Section problem
- The simplest tool that most OSes use is mutex lock
- To access the critical regions with it by first **`acquire()`** a lock then **`release()`** it afterwards
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic** (non-interruptible)
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**. This lock therefore called a **spinlock** 自旋锁
  - 缺点
  - Spinlock wastes CPU cycles due to busy waiting, but it has one distinct advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlock is useful
  - 优点
  - Spinlocks are often used in multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor

acquire() and release()

出现不可用时，忙等待

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}

do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

Solutions based on the idea of **lock** to protect critical section

- Operations are **atomic** (non-interruptible) – at most one thread acquires a lock at a time
- Lock before entering critical section for accessing share data
- Unlock upon departure from critical section after accessing shared data
- Wait if locked - all synchronization involves busy waiting, should "sleep" or "block" if waiting for a long time

Semaphore

- **Semaphore *S*** – non-negative integer variable, can be considered as a generalized lock
  - First defined by Dijkstra in late 1960s. It can behave similarly as mutex lock, but it has more sophisticated usage - the main synchronization primitive used in original UNIX
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called `P()` and `V()`, where `P()` stands for "`proberen`" (to test) and `V()` stands for "`verhogen`" (to increment) in Dutch
- It is essential that semaphore operations are executed atomically which guarantees that no more than one process can execute **`wait()`** and **`signal()`** operations on the same semaphore at the same time – serialization
- The semaphore can only be accessed via these two atomic operations except initialization

```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
signal (S) {
    S++;
}
```


Semaphore Usage

- **Counting semaphore** – An integer value can range over an unrestricted domain
  - Counting semaphore can be used to control access to a given set of resources consisting of a finite number of instances; semaphore value is initialized to the number of resources available
- **Binary semaphore** – integer value can range only between `0` and `1`
  - This can be behave like mutex locks, can also be used in different ways
- This can also be used to solve various synchronization problems
- Consider P_1 and P_2 that shares a common semaphore **`synch`**, initialized to `0`; it ensures that P_1 process executes S_1 before P_2 process executes S_2

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```


Semaphore Implementation with no Busy waiting

- Each semaphore is associated with a waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record on the queue
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it on the ready queue
- Semaphore values may become negative, whereas this value can never be negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes currently waiting on the semaphore.

Semaphore Implementation with no Busy waiting (Cont.)

```
typedef struct{
   int value;
   struct process *list;
} semaphore;
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Noticing that

- Increment and decrement are done before checking the semaphore value, unlike the busy waiting implementation
- The **`block()`** operation suspends the process that invokes it.
- The **`wakeup(P)`** operation resumes the execution of a suspended process P.


```c
typedef struct {
    int value; // 信号量的值，表示可用资源的数量
    struct process *list; // 等待队列，存储等待该信号量的进程
} semaphore;

wait(semaphore *S) {
    S->value--; // 将信号量的值减 1
    if (S->value < 0) { // 如果信号量的值小于 0
        add this process to S->list; // 将当前进程添加到等待队列
        block(); // 阻塞当前进程，等待信号量可用
    }
}

signal(semaphore *S) {
    S->value++; // 将信号量的值加 1
    if (S->value <= 0) { // 如果信号量的值小于等于 0
        remove a process P from S->list; // 从等待队列中移除一个进程 P
        wakeup(P); // 唤醒进程 P，使其可以继续执行
    }
}
```

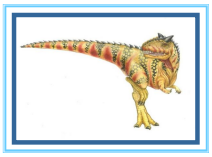
Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes (to be examined in Chapter 8)
- Let S and Q be two semaphores initialized to 1

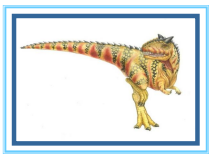
P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
...	...
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- Consider if P_0 executes **`wait(S)`** and P_1 executes **`wait(Q)`**. When P_0 executes **`wait(Q)`**, it must wait until P_1 executes **`signal(Q)`**, However, P_1 is waiting until P_0 executes **`signal(S)`**. Since these **`signal()`** operations will never be executed, P_0 and P_1 are **deadlocked**. This is extremely difficult to debug
- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue, in which it is suspended. For instance, if we remove processes from the queue associated with a semaphore using LIFO (last-in, first-out) order or based on certain priorities.

End of Chapter 6



Chapter 7: Synchronization Example



Synchronization Examples

- Classic Problems of Synchronization
  - Bounded-Buffer Problem 有界缓冲区问题
  - Readers and Writers Problem
- Window Synchronization
- POSIX Synchronization

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1 信号量
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```


这段代码是一个典型的生产者-消费者模型中的生产者部分，使用信号量来管理对共享缓冲区的访问。以下是对代码的逐行解释：

1. **do { ... } while (true);**: 这是一个无限循环，表示生产者将持续不断地生产项目。
2. **/*produce an item in next_produced */**: 这一行是注释，表示生产者在这里生成一个新的项目，并将其存储在变量 `next_produced` 中。
3. **wait(empty);**: 这是一个信号量操作，表示生产者在尝试添加新项目之前，首先检查缓冲区是否有空位。`empty` 信号量表示缓冲区中空位的数量。如果 `empty` 的值为0，生产者将被阻塞，直到有空位可用。
4. **wait(mutex);**: 另一个信号量操作，表示生产者请求对缓冲区的互斥访问。`mutex` 信号量确保在同一时刻只有一个进程可以访问缓冲区，以避免数据竞争。
5. **/*add next produced to the buffer */**: 这一行是注释，表示生产者将 `next_produced` 中的项目添加到缓冲区中。
6. **signal(mutex);**: 释放互斥锁，允许其他进程（如消费者）访问缓冲区。
7. **signal(full);**: 这是一个信号量操作，表示生产者在成功添加新项目后，通知消费者缓冲区中有新项目可用。`full` 信号量表示缓冲区中已填充项目的数量。

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
        ...
        /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
        /* consume the item in next consumed */
        ...
    } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data; they **do not** perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read the data set at the same time, but at most only one single writer can access shared data at a time
- Several variations of how readers and writers are treated – involve different priorities.
- The simplest solution, referred to as the first readers-writers problem, requires that no reader be kept waiting unless a writer has already gained access to the shared data
  - Shared data update (by writers) can be delayed
  - This gives readers priority in accessing shared data
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
   wait(rw_mutex);
      ...
      /* writing is performed */
      ...
   signal(rw_mutex);
} while (true);
```

1. **do { ... } while (true);**: 这是一个无限循环，表示写者将持续不断地进行写操作。
2. **wait(rw_mutex);**: 这是一个信号量操作，表示写者请求对共享资源的互斥访问。`rw_mutex` 是一个互斥锁，确保在同一时刻只有一个写者可以访问共享资源，以避免数据竞争。
3. ***/ writing is performed */**: 这一行是注释，表示在这里进行实际的写操作。写者将对共享资源进行修改。
4. **signal(rw_mutex);**: 释放互斥锁，允许其他进程（如其他写者或读者）访问共享资源。

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex)
   ...
   /* reading is performed */
   ...
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Note:

- `rw_mutex` controls the access to shared data (critical section) for writers, and the first reader. The last reader leaving the critical section also has to release this lock
- `mutex` controls the access of readers to the shared variable `count`
- Writers wait on `rw_mutex`, first reader yet gain access to the critical section also waits on `rw_mutex`. All subsequent readers yet gain access wait on `mutex`

1. **do { ... } while (true);**
   这是一个无限循环，表示读者将不断尝试读取共享资源。
2. **wait(mutex);**
   通过调用 `wait(mutex)`，读者请求对 `mutex` 的访问，以确保对 `read_count` 的操作是互斥的。
3. **read_count++;**
   读者数量增加，表示有一个新的读者开始读取。
4. **if (read_count == 1) wait(rw_mutex);**
   如果这是第一个读者（即 `read_count` 变为 1），则请求对 `rw_mutex` 的访问。这是为了确保在有读者时，写者不能访问共享资源。
5. **signal(mutex);**
   释放 `mutex`，允许其他读者或写者访问。
6. */ reading is performed /*
   在这里，读者执行实际的读取操作。
7. **wait(mutex);**
   读者完成读取后，再次请求对 `mutex` 的访问，以更新 `read_count`。
8. **read_count--;**
   读者数量减少，表示有一个读者结束了读取。
9. **if (read_count == 0) signal(rw_mutex);**
   如果这是最后一个读者（即 `read_count` 变为 0），则释放 `rw_mutex`，允许写者访问共享资源。
10. **signal(mutex);**
    释放 `mutex`，允许其他读者或写者继续访问。

Readers-Writers Problem Variations

- **First variation** – no reader kept waiting unless a writer has gained access to use shared object. This is simple, but can result in starvation for writers, thus can potentially significantly delay the update of the object.
- **Second variation** – once a writer is ready, it needs to perform update asap. In another word, if a writer waits to access the object (this implies that there could be either readers or a writer inside), no new readers may start reading, i.e., they must wait after the writer updates the object
- A solution to either problem may result in starvation.
- The problem can be solved or at least partially by the kernel providing **reader-writer locks**, in which multiple processes are permitted to concurrently acquire a reader-writer lock in read mode, but only one process can acquire the reader-writer lock for writing (exclusive access). Acquiring a reader–writer lock thus requires specifying the mode of the lock: either read or write access

Synchronization Examples

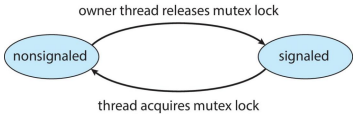
- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutex for efficiency when protecting data from *short code segments*, usually less than a few hundred (machine-level) instructions
  - Starts as a standard semaphore implemented as a spinlock in a multiprocessor system
  - If lock held, and by a thread running on another CPU, spins to wait for the lock to become available
  - If lock held by a non-run-state thread, block and sleep waiting for signal of lock being released
- Uses condition variables
- Uses readers-writers locks when longer sections of code need access to data. These are used to protect data that are frequently accessed, but usually in a read-only manner. The readers-writer locks are relatively expensive to implement.

Windows Synchronization

- The kernel uses interrupt masks to protect access to global resources in uniprocessor systems
- The kernel uses spinlocks in multiprocessor systems (to protect short code segments)
  - For efficiency, the kernel ensures that a thread will never be preempted while holding a spinlock
- For thread synchronization outside the kernel (user mode), Windows provides dispatcher objects, threads synchronize according to several different mechanisms, including mutex locks, semaphores, events, and timers
  - Events are similar to condition variables; they may notify a waiting thread when a desired condition occurs
  - Timers are used to notify one or more thread that a specified amount of time has expired
  - Dispatcher objects either signaled-state (object available) or non-signaled state (this means that another thread is holding the object, therefore the thread will block)



Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive kernel
- Linux provides:
  - semaphores
  - Spinlocks – for multiprocessor systems
  - atomic integer, and all math operations using atomic integers performed without interruption
  - reader-writer locks
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption

Atomic Variables

- Atomic variables - `atomic_t` is the type for atomic integer
- Consider the variables

```
atomic_t counter;
int value;
```

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variables
- Widely used on UNIX, Linux, and MacOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Condition Variables

- POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```


POSIX Condition Variables

- Thread waiting for the condition `a == b` to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- `pthread_cond_wait()` `&mutex` as the second parameter - in addition to putting the calling thread to sleep, releases the lock when putting said caller to sleep. If not, no other thread can acquire the lock and signal it to wake up

POSIX Condition Variables

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

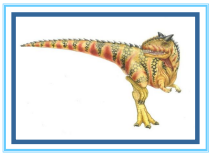
- When signaling (as well as when modifying the condition variable), make sure to have the lock held. This ensures that no race condition is accidentally introduced

- Before returning after being waked up, the `pthread_cond_wait()` re-acquires the lock, thus ensuring that any time the waiting thread is running between the lock acquire at the beginning of the wait sequence, and the lock release at the end, it holds the lock.

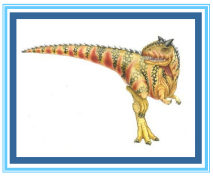
这段话的意思是，在使用 `pthread_cond_wait()` 函数时，当一个线程被唤醒并准备继续执行之前，它会重新获取锁。这确保了在整个等待过程中，从线程在等待开始时获取锁到等待结束时释放锁的这段时间内，线程始终持有该锁。

简单来说，这个机制保证了线程在等待条件变量时，能够安全地访问共享资源，避免了数据竞争和不一致的问题。

End of Chapter 7



Chapter 8: Deadlocks



Chapter 8: Deadlocks

- Deadlock Examples
- Deadlock Characterization 特征
- Resource Allocation Graph 资源分配图
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection and Recovery from Deadlock

Chapter Objectives

- Illustrate how deadlock can occur when mutex locks are used.
- Define the *four necessary conditions* that characterize deadlock.
- Identify a deadlock situation in a resource allocation graph.
- Evaluate the four different approaches for preventing deadlocks.
- Apply banker's algorithm for deadlock avoidance.
- Apply the deadlock detection algorithm.
- Evaluate approaches for recovering from deadlock.

Deadlock in Multithreaded Application

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

- The order in which the threads run depends on how they are scheduled by the CPU scheduler
- This example illustrates the fact that it is difficult to identify and test for deadlocks that may occur only under certain scheduling circumstances.

Deadlock Example with Lock Ordering

具有锁顺序的死锁示例

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

3. 获取锁：
   `lock1 = get_lock(from);`
   `lock2 = get_lock(to);`
   通过 `get_lock` 函数获取与转出账户和转入账户相关的锁。
4. 获取互斥锁：
   `acquire(lock1);`
   `acquire(lock2);`
   获取 `lock1` 和 `lock2`，确保在执行转账操作时，其他线程无法同时访问这两个账户。
5. 执行转账操作：
   `withdraw(from, amount);`
   `deposit(to, amount);`
   从转出账户中提取指定金额，并将该金额存入转入账户。
6. 释放锁：
   `release(lock2);`
   `release(lock1);`
   在完成转账后，释放 `lock2` 和 `lock1`，允许其他线程访问这两个账户。

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$
  CPU cycles, memory space, files, I/O devices, semaphores
- Each resource type R_i has W_i instances. 实例
- Each process P_i utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization

Deadlock involving multiple processes can arise if the following **four** conditions hold **simultaneously** – they are **necessary** but **not sufficient** conditions

同时发生 必要但不是充分条件

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resource(s) held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task

不抢占 自愿

- **Circular wait**: 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, …, 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.

Resource-Allocation Graph

资源分配

A set of vertices V and a set of edges E.

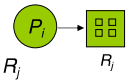
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$ 请求边 表示进程对资源的请求
- **assignment edge** – directed edge $R_j \rightarrow P_i$ 分配边 表示资源已分配给进程

Resource-Allocation Graph (Cont.)

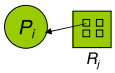
- Process

- Resource Type with 4 instances

- P_i requests instance of R_j

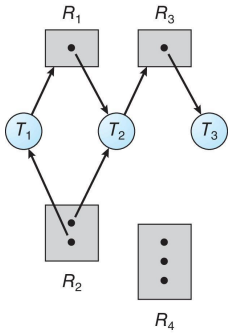


- P_i is holding an instance of R_j

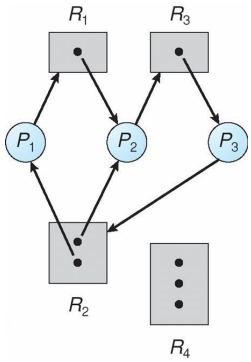


Resource Allocation Graph Example

- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



Resource Allocation Graph With A Deadlock



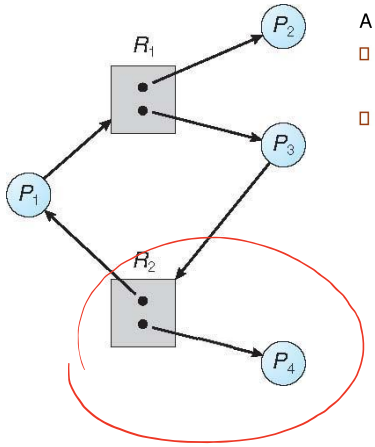
Cycles exist

- P1 → R1 → P2 → R3 → P3 → R2 → P1
- P2 → R3 → P3 → R2 → P2
- Processes P1, P2, and P3 are deadlocked

资源分配图法.
如果图中存在循环、则说明存在死锁

特例

Graph With A Cycle But No Deadlock



A cycle exists

- P1 → R1 → P3 → R2 → P1
- However, there is no deadlock. Observe that thread P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.

因为没有满足 占有且等待.
P4 占有一个资源、但是它并不需要等待获取其它资源.

Basic Facts

- If a graph contains no cycles ⇒ no deadlock
- If a graph contains a cycle ⇒ the system may or may not be in a deadlocked state
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state:
  - Deadlock prevention: it provides a set of methods to ensure at least one of the necessary conditions cannot hold
  - Deadlock avoidance: this requires **additional information** given in advance concerning which resources a process will request and use during its lifetime. Within such knowledge, the OS can decide for each resource request whether a process should wait or not
- Deadlock detection - allow the system to enter a deadlock state, periodically detect if there is a deadlock and then recover from it
- Many commercial operating systems, esp., for desktops, laptops, and smart phones ignore the deadlock problem because of the overhead and pretend that deadlocks never occur in the system
  - It will cause the system's performance to deteriorate, because resources are being held by processes that cannot run and because more and more processes, as they make requests for resources, will enter a deadlocked state – restart the system manually

Deadlock Prevention

Restrain the ways request can be made

- Mutual Exclusion – not required for sharable resources (e.g., read-only files); but it must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require each process to request and be allocated all its resources before it begins execution, or request resources only when the process has none
  - The disadvantages - low resource utilization, and possible starvation
- No Preemption –
  - If a process that is holding some resources requests another that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources added to list of resources for which the process is waiting
  - Process will be restarted only when it can regain all of its old resources, as well as the new ones that it is currently requesting
  - This can only be applied to resources whose state can be easily saved and restored such as registers, memory space and database transactions. It cannot generally be applied to resources such as locks and semaphores

Deadlock Prevention (Cont.)

- Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration – R = {<R_1, R_2, …, R_m>}
  - This requires that a process cannot request a resource R_j before requesting a resource R_i if j > i
- This can be proved by contradiction
  - Let the set of processes involved in a circular wait be P = {<P_0, P_1, …, P_n>}, where P_i is waiting for a resource R_i, which is held by process P_{i+1}, so that P_n is waiting for a resource R_n held by P_0.
  - Since process P_{i+1} is holding resource R_i while requesting resource R_{i+1}, we must have $R_i < R_{i+1}$ for all *i*.
  - This implies $R_0 < R_1 < R_2 \ldots < R_n < R_0$
  - $R_0 < R_0$, this is impossible, therefore there can be no circular wait

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e. mutex locks) a unique number.
- Resources must be acquired **in order.**
- If:

```
first_mutex = 1
second_mutex = 5
```

code for `thread_two` could not be written as follows:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```


Deadlock Avoidance

Requires that the system has some additional **a priori** information available

For instance, with the knowledge of complete sequence of request and release for each process, system can decide for each request whether the process should wait to avoid a possible future deadlock.

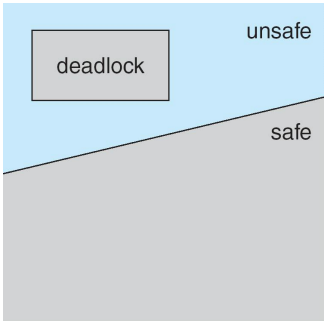
- The simplest and most useful model requires that each process declares the *maximum number* of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist
- Resource-allocation state is defined by the number of (1) available and (2) allocated resources, and (3) the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide whether such an allocation will leave the system in a **safe state**
- System is in safe state if there exists a safe sequence $<P_1, P_2, \ldots, P_n>$ consisting of **all processes** in the systems such that for each P_i, the resources that P_i can still request (based on prior declaration) can be satisfied by currently available resources plus resources held by **all** P_j, with $j < i$. That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- If no such sequence exists, then the system state is said to be **unsafe**.

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- **Avoidance** ⇒ ensure that a system will never enter an unsafe state
  - In this scheme, if a process requests a resource that is currently available, it may still have to wait (if the allocation leads to unsafe state).
  - The resource utilization may be lower than it would be otherwise



Avoidance algorithms

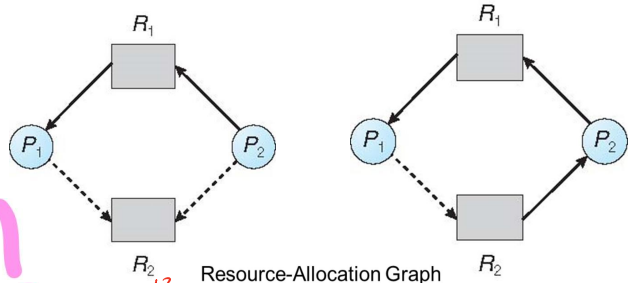
- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's algorithm

Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converts to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to claim edge
- Resources must be claimed a priori in the system

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge **does not result** in the formation of a cycle in the resource allocation graph



Resource-Allocation Graph

Banker's Algorithm

- Multiple instances
- Each process must declare a priori maximum usage
- When a process requests a resource, it may have to wait – check to see if this allocation results in a safe state or not
- When a process gets all its resources it must return them in a finite amount of time after use
- This is analogous to banking loan system, which has a maximum amount, total, that can be loaned at one time to a set of businesses each with a credit line.

Data Structures for the Banker's Algorithm

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

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

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***
   (b) $Need_i \le Work$
   If no such ***i*** exists, go to step 4
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2
4. If ***Finish*** [***i***] == ***true*** for all ***i***, then the system is in a safe state, otherwise unsafe

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants ***k*** instances of resource type R_j

1. If $Request_i \le Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \le Available$, go to step 3. Otherwise, P_i must wait, since resources are not available
3. Pretend to have allocated requested resources to P_i by modifying the state as follows:
   $Available = Available - Request_i$;
   $Allocation_i = Allocation_i + Request_i$;
   $Need_i = Need_i - Request_i$;
   - Run safety algorithm: If safe ⇒ the resources can be allocated to P_i
   - If unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
  3 resource types:
  A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

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

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

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

Example (Cont.)

- 5 processes P_0 through P_4; 3 resource types:
  A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T_0:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

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

Example: P_1 Request (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
- Can request for (3,3,0) by P_4 be granted? – resource not available
- Can request for (0,2,0) by P_0 be granted? – state is not safe

Deadlock Detection

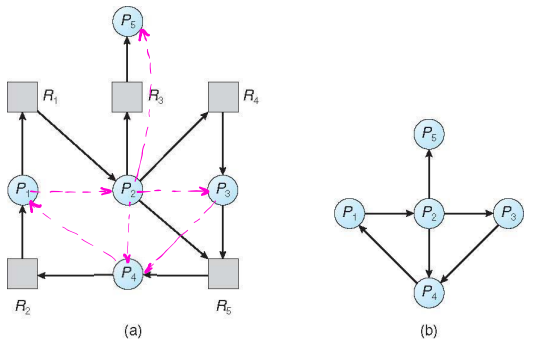
If a system does not use either a deadlock-prevention, or deadlock-avoidance algorithm, then a deadlock situation may occur. In this environment, the system may provide

- An algorithm that examines the state of the system to determine whether a deadlock can occur
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- Maintain wait-for graph 等待导待图
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a *cycle* in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph
- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances for each resource type

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding Wait-for Graph

Several Instances for a Resource Type

- **Available**: A vector of length m indicates the number of available resources of each type
- **Allocation**: An n x m matrix defines the number of resources of each type currently allocated to each process
- **Request**: An n x m matrix indicates the current request of each process. If $Request[i][j] = k$, then process P_i is requesting k instances of resource type R_j.

Detection Algorithm

1. Let $Work$ and $Finish$ be vectors of length m and n, respectively Initialize:
   (a) $Work = Available$
   (b) For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then $Finish[i] = false$; otherwise, $Finish[i] = true$
2. Find an index i such that both:
   (a) $Finish[i] == false$
   (b) $Request_i \leq Work$
   If no such i exists, go to step 4
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   go to step 2
4. If $Finish[i] == false$, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if $Finish[i] == false$, then P_i is deadlocked

This algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

这个序列是指定的

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- P_2 requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will be affected by a deadlock when it occurs
    - one for each disjoint cycle 不相交回路

任意调用

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph; we would not be able to tell which of the many deadlocked processes "caused" the deadlock.
- Invoking the deadlock detection algorithm for every resource request will incur considerable overhead in computation.
  - A less expensive alternative is to invoke the algorithm at defined intervals – for example, once per hour, or whenever CPU utilization drops below 40%

缺点

时隔

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes: This clearly breaks the deadlock cycle, but at great expense

中止

- Abort one process at a time until the deadlock cycle is eliminated: This also incurs considerable overhead, since after each process is aborted, the deadlock-detection algorithm needs to run
- In which order should we choose to abort? – many factors:
  1. Priority of the process
  2. How long process has computed, and how much longer to complete?
  3. Resources the process has used
  4. Resources the process needs to complete
  5. How many processes will need to be terminated?
  6. Is process interactive or batch?

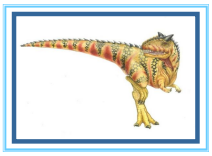
Recovery from Deadlock: Resource Preemption

To successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken

- Selecting a victim – minimize cost (which resources and which processes are to be preempted)
- Rollback – return to some safe state, restart process from that state
- Starvation – the same process may always be picked as victim, including the number of rollback in cost factor might help to reduce the starvation

End of Chapter 8

Chapter 9: Main Memory



Chapter 9: Memory Management Strategies

- Background
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table
- Examples: the Intel 32 and 64-bit Architectures and ARM Architecture

Objectives

- Explain the difference between a logical and a physical address and the role of the memory management unit (MMU) in address translation.
- Dynamic storage-allocation problem - apply **first-**, **best-**, and **worst-fit** strategies for allocating memory contiguously.
- Explain the distinction between **internal** and **external fragmentation.**
- Translate logical to physical addresses in a paging system that includes a translation look-aside buffer (TLB).
- Describe hierarchical paging, and hashed paging
- Describe address translation for IA-32, x86-64, and ARMv8 architectures

Background

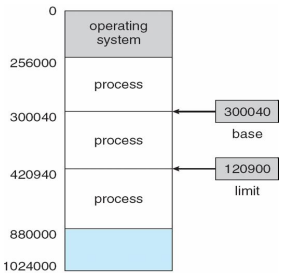
- Main memory is central to the operation of computer systems
- Memory consists of a large array of **bytes**, each with its own address – byte-addressable
- Program must be brought (from disk or other secondary storage) into memory and placed within a process for it to run, i.e., PCB points to the address space of the process
- A typical instruction-execution cycle include
  - CPU first fetches an instruction from memory (or cache)
  - The instruction is then decoded and may cause operands to be fetched from memory.
  - After the instruction executed on the operands, results may be stored back in memory.
- The memory management unit or MMU only sees a stream of addresses
  - It does not know how they are generated, could be by the instruction counter, indexing, indirection, literal addresses, and so on or what they are for (instructions or data).
  - Accordingly, MMU is interested only in the sequence of memory addresses generated by a running program.

Background (Cont.)

- Main memory (including cache) and registers are the only storage CPU can access directly; in another word, CPU cannot access secondary storage such as disk directly, but through I/O controllers
- Memory management unit or MMU only sees a stream of addresses + read requests, or address + data and write requests
  - Registers can be accessed in one CPU clock (cycle)
  - Accessing main memory (through memory bus) may take many CPU clocks, causing a **memory stall**, when it does not yet have the data required to complete the instruction it is executing
  - **Cache** sits between main memory and CPU registers, residing on CPU chips for fast access
- Memory protection is required to ensure correct operation
  - We must protect the operating system memory space from being accessed by user processes, as well as protect user processes from one another
  - This protection must be provided by the hardware for performance – performance or speed

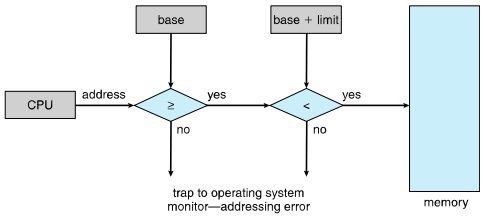
Per-process memory space separated from each other

- Separate per-process memory space protecting the processes from each other is fundamental to have multiple processes loaded in memory for concurrent execution – multiprogramming systems
- A pair of base and limit registers define the legal range of a process address space
  - For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive)



Hardware Address Protection

- Protection of memory space is accomplished by having CPU compare every address generated in user mode with these two registers
- The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction (executed only in the kernel mode)
- The operating system, executing in kernel mode, however, is given unrestricted access to both operating-system memory and users' memory



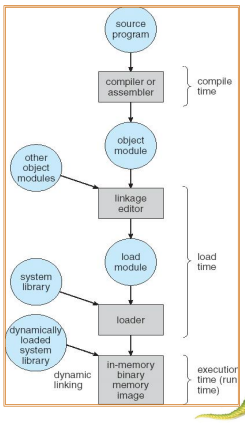
Address Binding

- Usually, a program resides on a disk as a binary executable file, which must be brought into memory and placed within a process (part of its address space) for it to run
- Most systems allow a user process to reside in any part of the physical memory. Although the computer may start at 00000, the first address of a user process need not be 00000
- A user program goes through several steps - some of which may be optional before being executed. Addresses may be represented in different ways during these steps:
  - Source code addresses usually symbolic – the variable count
  - A compiler typically binds these symbolic addresses to relocatable addresses, such as "14 bytes from beginning of this module"
  - Linker or loader in turn binds the relocatable addresses to absolute addresses, i.e., 74014
  - Each binding is essentially a mapping from one address space to another

Multistep Processing of a User Program

Address binding of instructions and data to memory addresses can happen in three different stages

- **Compile time**: If memory location is known a priori, absolute code can be generated; must recompile code if starting location changes (e.g., "gcc"). MS-DOS uses this
- **Link or Load time**: Compiler must generate relocatable code if memory location is not known at compile time (e.g, Unix "ld" does link). The binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. This needs hardware and operating system support for address maps (e.g., base and limit registers), e.g., dynamic libs. **Most general-purpose operating systems use this method.**



Address Translation and Protection

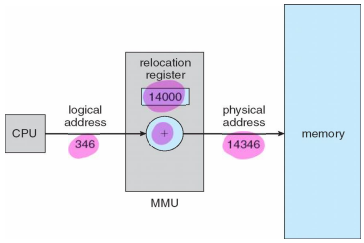
- In the old days of uni-programming (e.g., MS-DOS), only one program can run at the time, thus it occupies "nearly" the entire physical memory
  - No address translation is needed, nor is there any protection
- In the earlier stage of multi-programming such as Windows 3.1 (1990-1992)
  - No address translation, binding occurs at link or loader time – address adjusted when programs are loaded into memory. Bugs in programs can crash the system
  - Protection was added later using base and limit registers, preventing illegal access by another user program

Logical vs. Physical Address Space

- Recall: Address Space
  - All the addresses space a process can "touch"
  - Each process has its own unique address space, different from kernel address space
- Consequently, there are two views of memory
  - Logical address – generated by the CPU; also referred to as virtual address
  - Physical address – address seen by the memory
  - The translation is done by memory management unit or MMU – hardware device
- Translation makes it much easier to implement protection
  - To ensure a process can not access another process's address space.
- Logical and physical addresses are the same if compile-time and load-time address-binding schemes are used (in old days); logical (virtual) and physical addresses differ only in execution-time address-binding scheme
  - Logical address space is the set of all logical addresses generated by a program
  - Physical address space is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- MMU is a hardware mechanism that at runtime maps virtual address to physical address
- There are many different mapping methods, covered in the rest of this chapter
- To start, consider a simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - User programs never accesses the real physical addresses
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to the actual location in memory

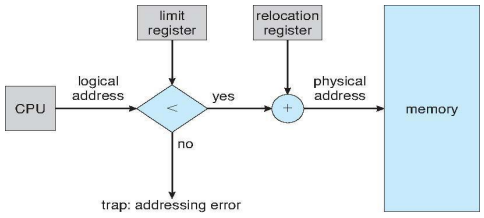


Contiguous Allocation

- Contiguous allocation is one of the early memory allocation methods
- Main memory is usually divided into two partitions – one for operating system and one for user processes
- The operating system can be placed in either low memory addresses or high memory addresses, depending on many factors, such as the location of the interrupt vector
- Many operating systems (including Linux and Windows) place the operating system in high memory
- In a multi-programmed operating system, several user processes reside in memory at the same time. In contiguous memory allocation, each process contained in a single section of memory that is contiguous to the section containing the next process.

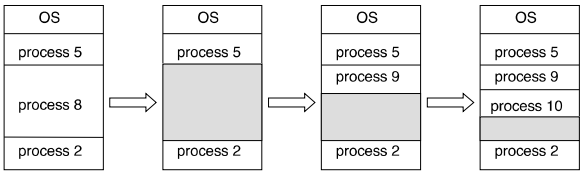
Hardware Support for Relocation and Limit Registers

- Relocation register and limit register are used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of the 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



Contiguous Allocation (Cont.)

- Multiple-partition allocations
  - Degree of multiprogramming is bounded by number of partitions
  - Variable-partition sizes (sized to a given process' needs)
  - **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
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about a) allocated partitions; b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of (variable) size n from a list of free holes?

- First-fit: Allocate the *first* hole that is big enough
- Best-fit: Allocate the *smallest* hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole – intended not use it
- Worst-fit: Allocate the *largest* hole
  - Must also search entire list
  - Produces the largest leftover hole – intended to reuse the remaining hole

Experiments have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Fragmentation

- External Fragmentation – total memory space available to satisfy a request, but it is not contiguous – scattered holes
  - The storage is fragmented into a large number of small holes.
- Internal Fragmentation – memory allocated to a process may be larger than requested memory; this size difference is memory internal to a partition, but not being used

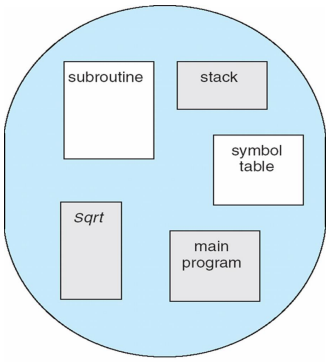
Fragmentation (Cont.)

- Reduce external fragmentation by a technique called **compaction**
  - 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. In another word, if relocation is static and is done at assembly or load time, compaction cannot be done
  - The compaction is expensive (time-consuming)
- The backing store (the secondary storage) has similar fragmentation problems, which will be discussed later
- Another more feasible solution is to permit logical address space of the processes to be **non-contiguous**, by dividing into multiple pieces (variable-sized segments or fixed-size pages), thus allowing a process to be allocated physical memory wherever such a piece of memory is available. These techniques include **segmentation** and **paging**

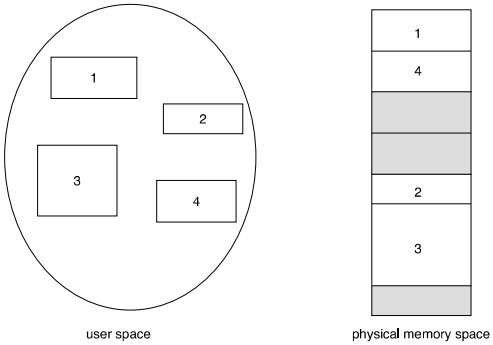
Segmentation- User View

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

main program
procedure
function
method
object
local variables, global variables
common block
stack
symbol table
arrays



Segmentation: Logical vs. Physical



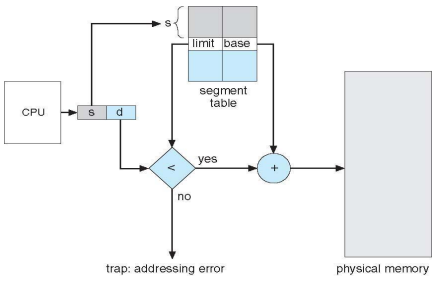
Segmentation Architecture

- The Logical address now consists of a **two-tuple**:
  <segment-number, offset>,
- **Segment table** – maps two-dimensional programmer-defined addresses (virtual address) into one-dimensional physical addresses; each entry in the segmentation table has:
  - base – contains the starting physical address where the segments reside in memory
  - limit – specifies the length of the segment
- Both STBR and STLR stored in the PCB of a process – Recall a process PCB contains all the information about the process
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates the number of segments used by a program;

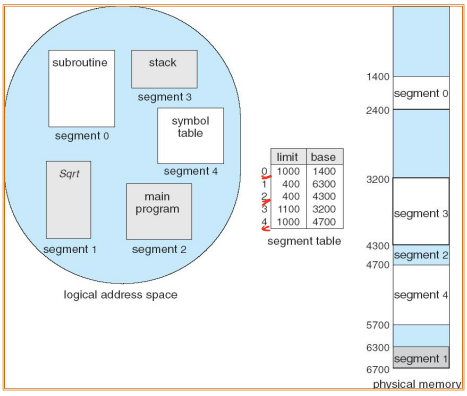
  segment number s is legal if s < STLR

Segmentation Architecture (Cont.)

- **Protection**. each entry in a segment table associates with:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- A protection bit is associated with each segment; code sharing occurs at segment level
- Since segments vary in length, memory allocation for a segment is also a dynamic storage-allocation problem
- External fragmentation still exists, but is much less severe than that in a contiguous allocation – since each segment is considerably smaller than the total memory space of a process



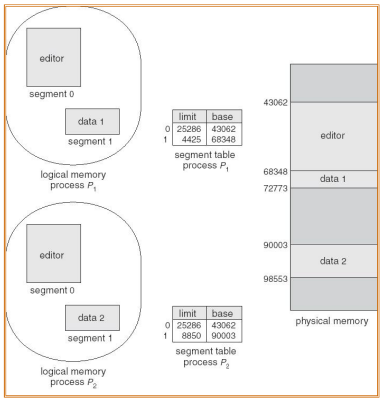
Example of Segmentation



- <0, 800> → 2200
- <2, 420> illegal address
- <4, 930> → 5630

Sharing of Segments

多个进程可以访问同一段的代码或数据，通常只读主存只读段中



Summary - Segmentation

- Protection is easy in a segmentation scheme – in segment table
  - Code segment would be read-only, data and stack would be read-write (stores allowed), shared segment could be read-only or read-write
- Can address be outside the valid range?
  - Yes, this is how stack and heap are allowed to grow. For instance, stack takes fault, system automatically increases the size of stack
- The main problem with segmentation
  - It must fit variable-sized chunks into physical memory – classical Dynamic Storage Allocation Problem
  - It may move processes multiple times to fit everything
  - External fragmentation

Paging 分页

- Physical address space of a process can be non-contiguous; process is allocated physical memory whenever the physical memory space is available
  - Avoids external fragmentation
  - Avoids the problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames
  - Usually, the size is power of 2, between 4KB (12 bit offset in physical memory address) and 1 GB (30 bit offset) – defined by the hardware
- Divide logical memory into blocks of same size called pages - the same size of a frame
- The OS needs to keep track of all free (physical) frames available in main memory
- To run a program of size N pages, need to find N free frames (non-contiguous) in memory and load the program into that
- A **page table** is used to translate logical to physical addresses – keep track of allocated frames
- This, however, suffers from internal fragmentation in the last page/frame 内部碎片

Address Translation Scheme 地址转换方案

- 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

page number	page offset
p	d
m - n	n

  - Total m bits logical address, the logical address space 2^m bytes, and the page size or frame size is 2^n, the number of pages this logical address contains is 2^{m-n}

Example of Paging Scheme

- Suppose logical address of a system is

1 byte = 8 bits
1 KB = 1024 bytes = 2^{10} bytes.

表示字节地址的. 12位作为页内偏移量时
意味着可以表示 2^{12} 个不同的地址.
每个地址对应一个字节(byte)

page number	page offset
p	d
20-bit	12-bit

2^{12} = 4 KB

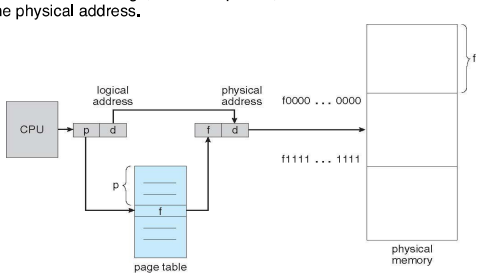
  - The page size - 12 bits are used to offset into a page, the page size is 2^{12} = 4 KB
  - 32 bits virtual address indicates virtual address space is 2^{32} = 4 GB
  - 20 bits page number, the number of page is 2^{20}

Paging Hardware

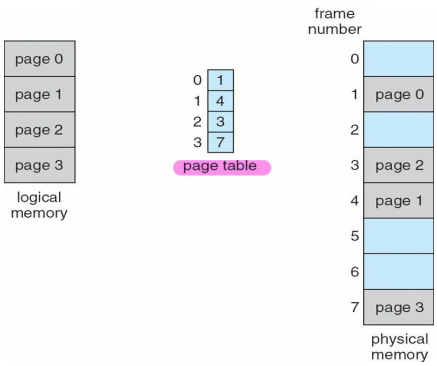
- The following outlines the steps taken by the MMU to translate a logical address generated by the CPU to a physical address:
  1. Extract the page number p and use it as an index into the page table.
  2. Extract the corresponding frame number f from the page table.
  3. Replace the page number p in the logical address with the frame number f

步骤

- As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

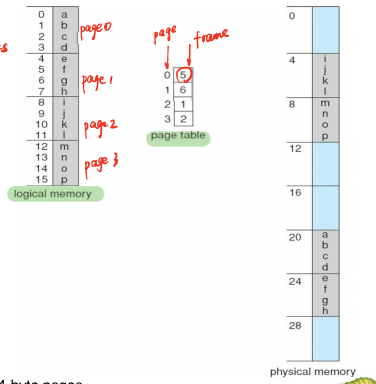


Paging Model of Logical and Physical Memory



Paging Example

- Frame number can automatically derive the starting address of each frame
- m=4 represents logical address of 16 bytes
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 × 4) +0]
- Logical address 3 (page 0, offset 3) maps to physical address 23 [= (5 × 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 [= (6 × 4) + 0].
- Logical address 13 maps to physical address 9.

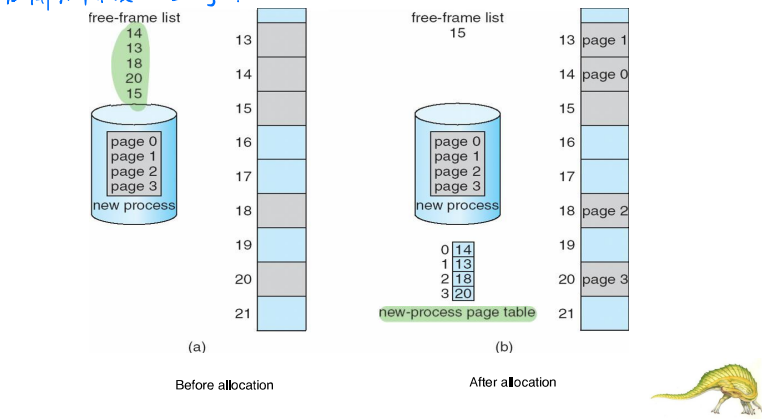


n=2 and *m*=4 32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
  - Page size = 2,048 bytes (2KB)
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable? - not really
  - The overhead involved in each page table entry reduces when page size increases.
  - Each page table entry takes memory to track, smaller page size results increased number of page table entries (one per page), thus it leads to larger page table
- Page size has grown over the time
  - SunMicro Solaris OS supports two-page sizes – 8 KB and 4 MB
- Process view and physical memory now very different
  - The programmer views memory as one single space, containing only this one program. But in fact, user program is scattered throughout physical memory, which also holds other programs.

Free Frames



Implementation of Page Table

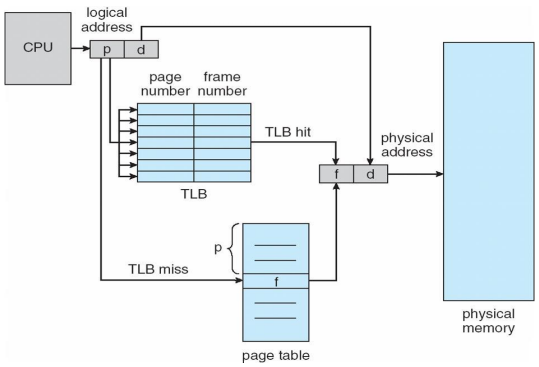
- Page table is kept in main memory – PCB has to keep track of memory allocation for the process
- Page-table base register (PTBR) points to the page table (starting address)
- Page-table length register (PTLR) indicates size of the page table
- In this scheme every data/instruction access requires **two memory accesses**
  - One for the page table (translation) and one for fetching the data / instruction – same with segmentation scheme
- 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 (TLBs)
- TLBs typically small (64 to 1,024 entries). Some CPUs implement separate instruction and data address TLBs. This is shared and used by all processes (kernel and user processes) in a system
- On a TLB miss (if the page number is not in the TLB), value is loaded into the TLB for faster access next time. This can be done by hardware (MMU) or software (running kernel codes)
  - Replacement policies must be considered – LRU, round-robin or even random are possible
  - Some entries can be wired down for permanent fast access, for example TLB entries for key kernel code for fast access

Associative Memory

- Associative memory (TLB) – parallel search

- Address translation (p, d)
  - **Check all entries in parallel** (hardware) – TLB shared and used by all processes (process ID and page number)
  - If p is in associative register (TLB), get frame # out – TLB hit
  - Otherwise get frame # from page table in memory, and also bring this entry to the TLB
- Locality on TLB
  - Instruction usually stays on the same page (sequential access nature)
  - Stack exhibits locality (pop in and out)
  - Data less locality, still quite a bit

Paging Hardware With TLB



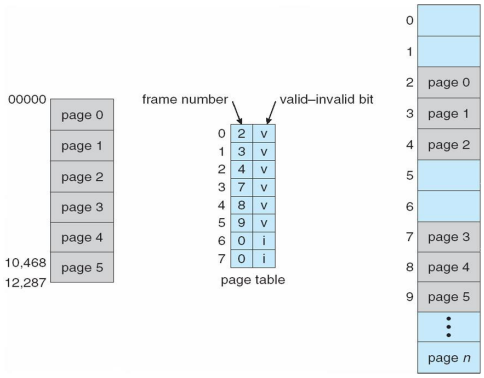
Effective Access Time

- Associative Lookup = ε time unit
  - Usually < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- **Effective Access Time** (EAT) – memory access time normalized to 1
  $EAT = (1 + \varepsilon)\alpha + (2 + \varepsilon)(1 - \alpha)$
  $= 2 + \varepsilon - \alpha$
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121ns

Memory Protection

- Memory protection in paging is accomplished by protection bits associated with each frame. Normally, these bits are kept in a page table.
  - One bit can define a page to be read–write or read-only, or more bits to indicate execute-only
- Valid-invalid bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space
  - "invalid" indicates that the page is not in the process' logical address space
  - The operating system sets this bit for each page to allow or disallow access to the page
- Any violations result in a trap to the kernel
  - Example: a 14-bit address space (0 to 16383). If a program only uses address 0 to 10468, with a page size 2KB, pages 0-5 are valid, pages 6-7 are invalid.
- Rarely does a process use all it address range, usually only a small fraction of the address space available. It would be wasteful to create a page table with entries for every page in the address range, and most of the table entries are unused but would take up memory space
  - use page-table length register (PTLR) to indicate the size of the page

Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

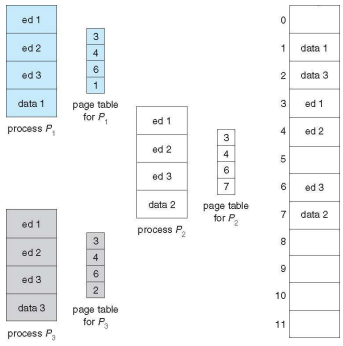
允许不同的进程访问相同的数据

Shared code

- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space
- Also useful for IPC if sharing of read-write pages is allowed

Private code and data

- Each process keeps a separate copy of the code and data
- The pages for the private code and data can appear anywhere in the logical address space



An Example: Intelx86 Page Table Entry

PTE

- What is in a Page Table Entry or PTE?
  - For page translation, each page table consists of a number of PTEs
  - Permission bits: valid, read-only, read-write, write-only
- Example: Intel x86 architecture PTE:
  - Address same format previous slide (10, 10, 12-bit offset)
  - Intermediate page tables called "Directories"

Page Frame Number (Physical Page Number)	Free (OS)	0	L	D	A	PCD	PWT	U	W	P
31-12	11-9	8	7	6	5	4	3	2	1	0

P: Present (same as "valid" bit in other architectures)
W: Writeable
U: User accessible
PWT: Page write transparent: external cache write-through
PCD: Page cache disabled (page cannot be cached)
A: Accessed: page has been accessed recently
D: Dirty (PTE only): page has been modified recently
L: L=1⇒4MB page (directory only).
Bottom 22 bits of virtual address serve as offset within a page

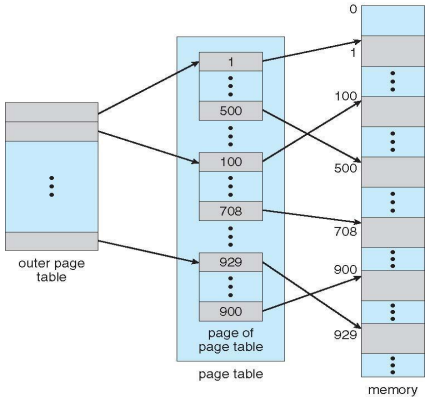
Structure of the Page Table

- Memory structures for paging can grow enormously using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Page table cannot be allocated contiguously in main memory either, which will be allocated into multiple pages (frames) – paging the page table, meaning another page table is needed to track the page table inside memory
  - Page size 4 MB (2^{22}) results in a page table with 1,000 entries, less of a problem
  - What about 64-bit logical address?
- Hierarchical Paging
- Hashed Page Tables

Hierarchical Page Tables 层次页表

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table or hierarchical page tables
- To page the page table

Two-Level Page-Table Scheme



Two-Level Paging Example

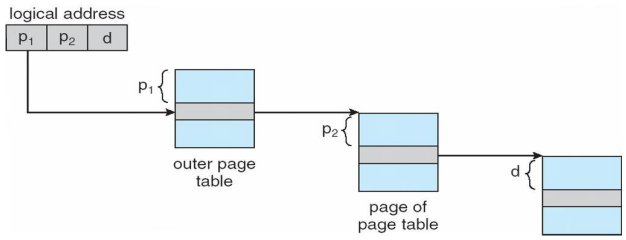
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number P_2, this must be 10 bits if each PTE occupies 4 bytes, **ensuring each inner page table can be stored within one frame (page)**,
  - a 10-bit page offset P_1. If PTE also occupies 4 bytes, the number of bits in P_1 can not be more than 10, or it needs to be further divided – more than two levels
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

- where p_1 is an index into the outer page table (in Intel architecture, this is called "directories", and p_2 is the displacement within the page of the inner page table
- because the address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table

正向映射页表.

Address-Translation Scheme



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like

outer page	inner page	page offset
p_1	p_2	d
42	10	12

  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes (16 GB) in size
    - And possibly 4 memory access to get to one physical memory location
    - The 64-bit UltraSPARC would require seven levels of paging – a prohibitive number of memory accesses – to translate each logical address

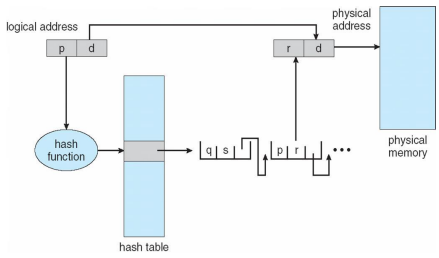
Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

Hashed Page Tables 哈希页表

- Common in address space > 32 bits
- The page number is hashed into a page table
  - This page table contains a chain of elements hashing into the same location
- Each element contains (1) the page number (2) the value of the mapped page frame (3) a pointer to the next element
- Page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted 提取

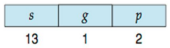


Example: The Intel 32 and 64-bit Architectures

- Dominant industry chips 优势行业芯片
  - 16-bit Intel 8086 (late 1970s) and 8088 was used in original IBM PC
- Pentium CPUs are 32-bit and called IA-32 architecture
  - It supports both segmentation 分段 and paging 分页 - The CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory.
- Current Intel CPUs are 64-bit and called IA-64 architecture
  - Currently most popular PC operating systems run on Intel chips, including Windows, MacOS, and Linux (Linux runs on several other architectures as well)
  - Intel's dominance has not spread to mobile systems, where they mainly use ARM architecture
- Many variations in the chips, only the main ideas are covered here

Example: The Intel IA-32 Architecture

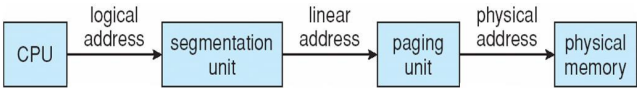
- Supports both segmentation and segmentation with paging
  - Each segment can be as large as 4GB (32 bits) further divided into pages
  - Each process can have up to 16K segments per process (14 bits), divided into two partitions
    - First partition of up to 8 K segments private to process (kept in local descriptor table (LDT))
    - Second partition of up to 8K segments shared among all processes (kept in global descriptor table (GDT))
- CPU generates logical address
  - Selector given to segmentation unit - which produces linear addresses - s for segment number, g indicates whether the segment is in LDT or GDT, p deals with protection



  - Linear address is then given to the paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Intel IA-32 Segmentation

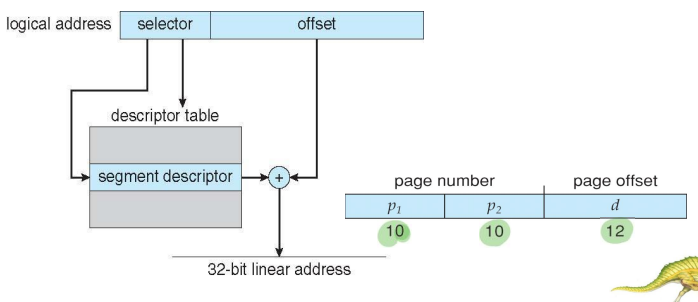
- Each entry in the LDT and GDT consists of an 8-byte segment descriptor with detailed information about a particular segment, including the **base location** and **limit**
- The segment register points to the appropriate entry in the LDT or GDT. The base and limit information about the segment is used to generate a linear address.



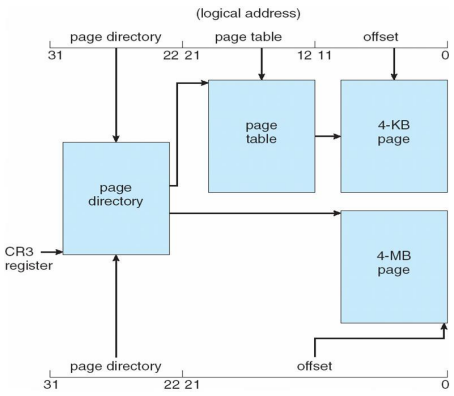
Intel IA-32 Segmentation

The linear address on the IA-32 is 32 bits long and is formed as follows

- The **limit** is used to check for address validity. If the address is not valid, a memory fault is generated, trap to the operating system. If it is valid, the value of the offset is added to the value of the **base**, resulting in a 32-bit linear address (i.e., each segment can be 4GB)

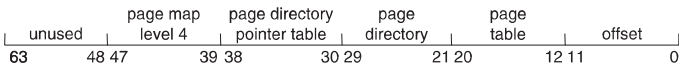


Intel IA-32 Paging Architecture



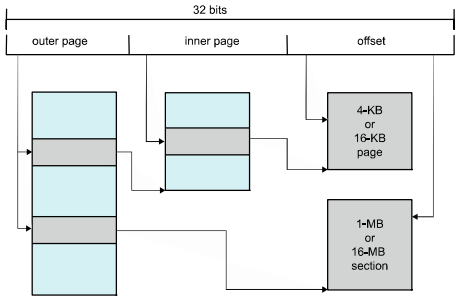
Intel x86-64

- Current generation Intel x86-64 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE (page address extension) so virtual addresses are 48 bits and physical addresses are 52 bits (4096 terabytes)

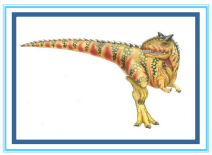


Example: ARM Architecture

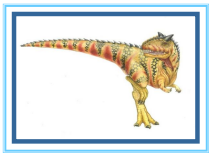
- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



End of Chapter 9



Chapter 10: Virtual Memory



Chapter 10: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Frame Allocation
- Thrashing
- Other Considerations

Objectives

- Describe **virtual memory** and its benefits.
- Illustrate how pages are loaded into memory using **demand paging**.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the **working set** of a process and explain how it is related to program locality.

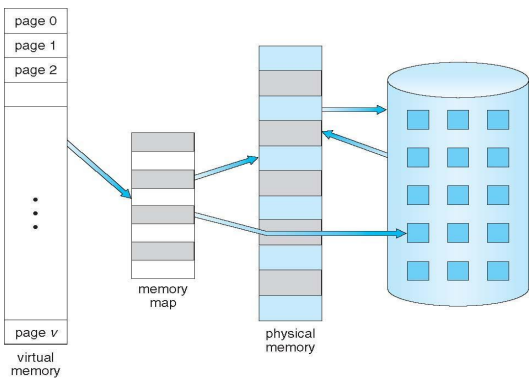
Background

- **Codes need to be in memory to execute, but not necessarily the entire program**
  - Error code, unusual routines. Some errors seldom, if ever, occur in practice, this code is almost never executed
  - Large data structures such as arrays, lists and tables are often allocated more memory than they need. For example, an array may be declared 100x100 elements, even though it is seldom larger than 10x10
  - Certain options and features of a program may be used rarely
- **Even if the entire program is needed, it may not all be needed at the same time**
- Consider ability to execute partially-loaded programs
  - Programs no longer constrained by limits of physical memory. Programs can be written with an extremely large virtual memory address, simplifying the programming task
  - Each user program could take less physical memory, more programs could be run at the same time, which increases CPU utilization (**degree of multiprogramming**) and throughput
  - Less I/O would be needed to load or swap user programs into physical memory, so each user program would run faster.

Background

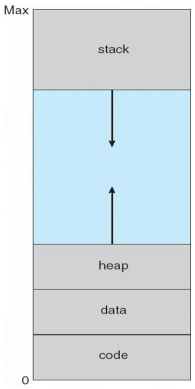
- **Virtual memory** – separation of user logical memory (referred as address space earlier) perceived by programmers from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than the actual physical address space
  - Allows address spaces to be shared by several processes. For instance, system libraries can be shared by several processes
  - Allows more efficient process creation, as pages can be shared during process creation, thus speeding up the process creation
  - More programs running concurrently – increase the degree of multiprogramming
  - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
  - Demand paging or demand segmentation – in principle, they are similar, details are different with respect to fix-sized frame/page and variable-sized segment

Virtual Memory That is Larger Than Physical Memory

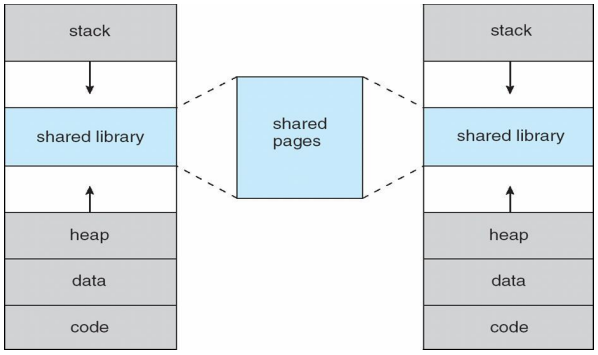


Virtual-address Space

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until the end of address space
  - Meanwhile, physical memory organized in frames
  - MMU must map logical to physical address
- Heap can grow upward in memory, used in dynamic memory allocation. Stack can grow downward in memory through successive function calls
- The large blank space (or hole) between the heap and stack is part of the virtual address space but will require actual physical pages (space) only if the heap or stack grows.
- Enables sparse address spaces with holes left for growth, dynamically linked libraries, and etc
- System libraries can be shared via mapped into virtual address space
- Pages can be shared during process creation with the fork() system call, speeding up process creation

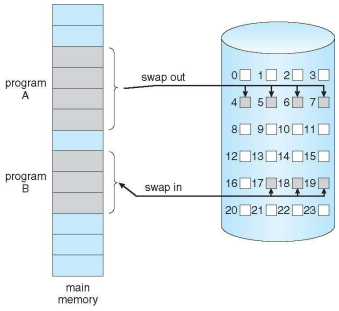


Shared Library Using Virtual Memory



Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when the page is needed or referenced
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users to be running
- Similar to paging system with swapping (diagram on right)
- Page is needed => reference to it
  - invalid reference (illegal memory address) => abort
  - not-in-memory => bring to memory
- Lazy swapper – never swaps a page into memory unless page is needed
  - Swapper that deals with pages is a pager

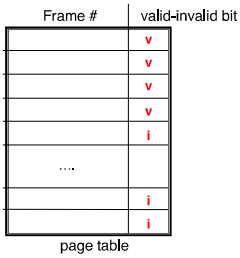


Basic Concepts

- **Pager** brings in only those "needed" pages into memory
- How to determine the set of pages brought inside memory?
  - Need new MMU functionality to implement demand paging
- If pages needed are already memory-resident
  - No difference from non demand-paging MMU
- If pages needed are not memory resident
  - Need to detect and load the page into memory from a secondary storage device
    - Without changing program behavior
    - Without programmer needing to change code – application not aware of this

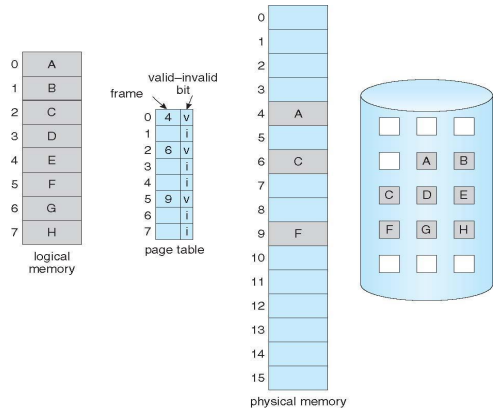
Valid-Invalid Bit

- With each page table entry, a valid–invalid bit is associated (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



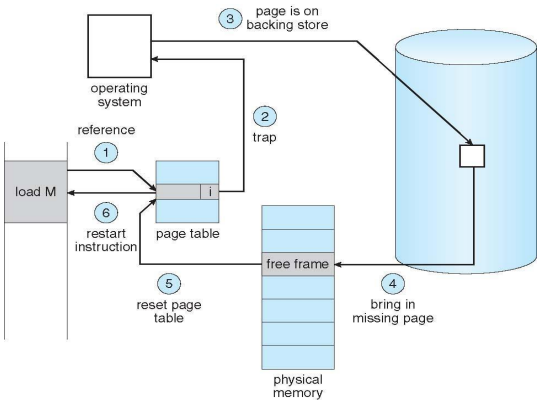
Page Fault

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

步骤:

1. Operating system looks at the corresponding page table entry to decide:
   - Invalid reference (illegal address) ⇒ abort 中止
   - Just not in memory
2. Get an empty frame if any - OS maintains free-frame list
3. Swap the page (on the secondary storage) into **the frame** via scheduled disk operation
4. Update the corresponding entry in page table
5. Reset page table to indicate this page is now in memory,
   Set validation bit = **v**
6. Restart the instruction (depending on CPU scheduling) that caused the page fault

Steps in Handling a Page Fault



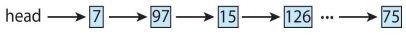
Aspects of Demand Paging

- The **extreme case** – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on the first access
  - This is referred as **Pure demand paging** 纯需求分页.
- A given instruction could access multiple pages -> result in multiple page faults
  - Consider fetch and decode of instruction which adds two numbers from memory and stores result back to memory
  - Pain caused by page faults decreases after process starts running for some time because of locality of memory reference 内存引用的局限性
- Usually, in order to minimize the initial and potentially high page fault, the OS pre-page some pages into the memory before the start of a process execution.
- Hardware support is needed for demand paging
  - Page table with valid / invalid bit as indication
  - Secondary memory (swap device with swap space) for page in and page out
  - Instruction restart

辅助

Free-Frame List 自由帧列表.

- When a page fault occurs, the operating system must bring the desired page from the secondary storage into main memory.
- Most operating systems maintain a free-frame list -- a pool of free frames for satisfying such requests (a kernel data structure that only OS can access)



按需零填充

- Operating system typically allocate free frames using a technique known as zero-fill-on-demand -- the content of the frames zeroed-out before being (re)-allocated.
  - The technique of the writing of zeros into a page before it is made available to a process - thus erasing their previous contents or to keep any old data from being available to the process
  - Consider the potential security implications of not clearing out the contents of a frame before reassigning it.
- When a system starts up, all available memory is placed on the free-frame list.

Performance of Demand Paging

- **Stages in Demand Paging – handle page faults** 步骤

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame (if available) in physical memory:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU core to some other processes
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other process (depending on the CPU scheduling)
9. Determine that the interrupt was from the disk
10. Update the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

- There are three major tasks in page-fault service time:
  1. Service the interrupt – careful coding might result in several hundred instructions
  2. Read the page – lots of time (accessing the secondary storage, typically a hard disk)
  3. Restart the process – a small amount of time
- The page switch time will probably be close to 8 milliseconds (for a typical hard disk)

- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\begin{aligned} EAT = (1 - p) \times &\text{ memory access} \\ &+ p \text{ (page fault overhead} \\ &+ \text{swap page out} \\ &+ \text{swap page in} \\ &+ \text{restart overhead)} \end{aligned}$$

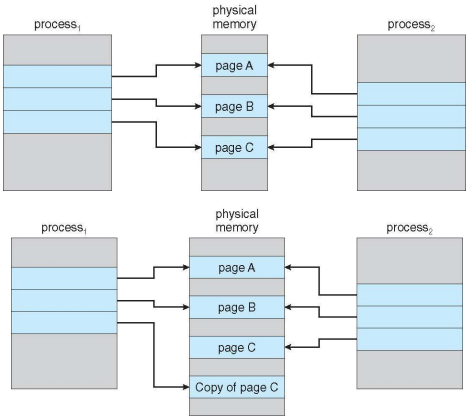
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds. This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses
- Fortunately, the memory locality usually satisfies this, as each memory miss brings an entire page

Copy-on-Write

- Traditionally, `fork()` works by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
  - 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 must be created for that process. In that case, the non-modified page is left with the other process with no sharing
- COW allows more efficient process creation as only modified pages are copied or duplicated
- This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process
  - Considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary anyway

Before and After Process 1 Modifies Page C



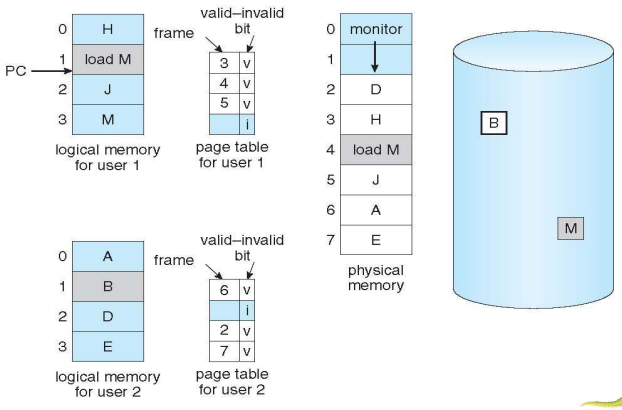
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc.
- How much memory to allocate to each process? - frame-allocation algorithm
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithms – terminate the process? swap out the entire process image? replace the page?
  - Performance – want an algorithm which will result in the minimum number of page faults
  - This needs to be transparent to a process or program execution
- Noticing that it may be inevitable that same page may be brought into memory several times

Page Replacement

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

Need For Page Replacement



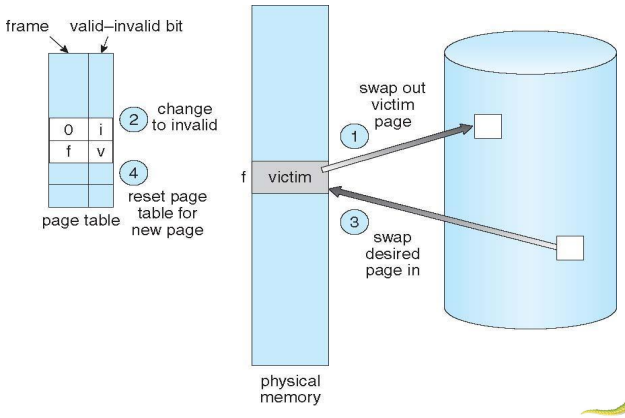
Basic Page Replacement

步骤

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame
   - Write victim frame to disk if "dirty" (modified since last time it was brought into the memory)
3. Bring the desired page into the (newly) free frame; update the page table
4. Continue the process by restarting the instruction that caused the trap

Note now potentially two page transfers for a page fault – which can significantly increase EAT

Page Replacement



Page and Frame Replacement Algorithms

- Page-replacement algorithm 页面替换算法
  - Decide which frames to replace – the objective is to minimize the page-fault rate
- Frame-allocation algorithm 帧分配算法 determines - how many frames allocated to each process – which is decided by how a process accesses the memory - locality

Page and Frame Replacement Algorithms

- Evaluate algorithms 评估 by running on a particular string of memory references - reference string and computing the number of page faults on that string
  - First, for a given page size, we need to consider only the page number, rather than the complete physical memory address.
  - If we have a reference to a page **p**, then any references to page **p** that immediately follow will never cause a page fault. Page **p** will be in memory after the first reference, so the immediately following references will not have page fault.
- For example, for a particular process, we might record the following address sequence:

  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,

  0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
- At 100 bytes per page, this sequence is reduced to the following reference string:

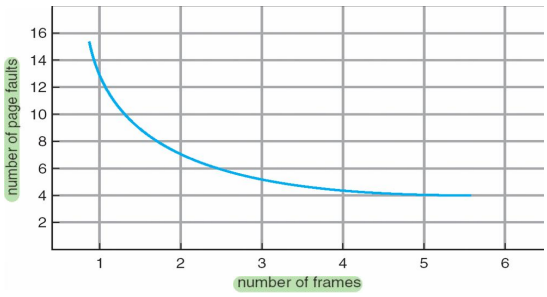
  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
- In all our examples, the reference string is

  7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

只关注每个页号的首次出现. 所以有些连续重复的页号会被省略

0100 ÷ 100 → 1
0432 ÷ 100 → 4
0101 ÷ 100 → 1 ⇒ 1416161641
0612 ÷ 100 → 6
0102 ÷ 100 → 1
0103 ÷ 100 → 1 (省略)
⋮

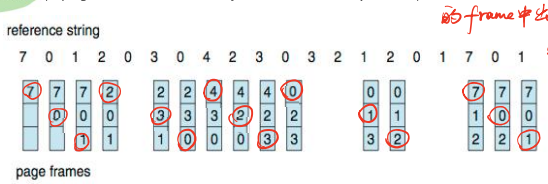
Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time for the process)

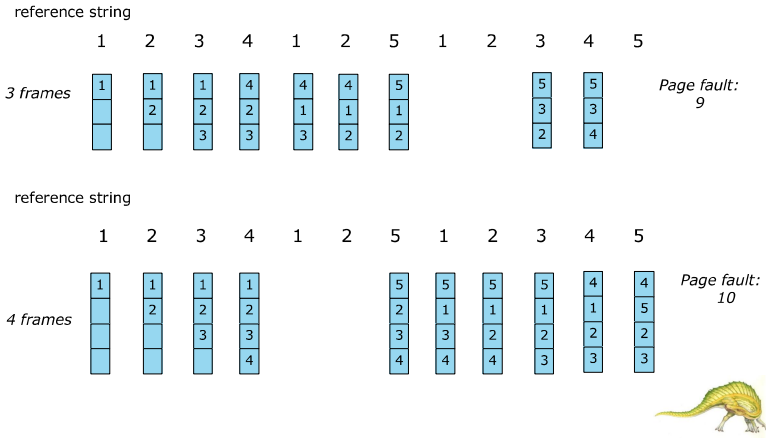
只要当前 string 在已有的 frame 中出现过，则不会进行任何替换



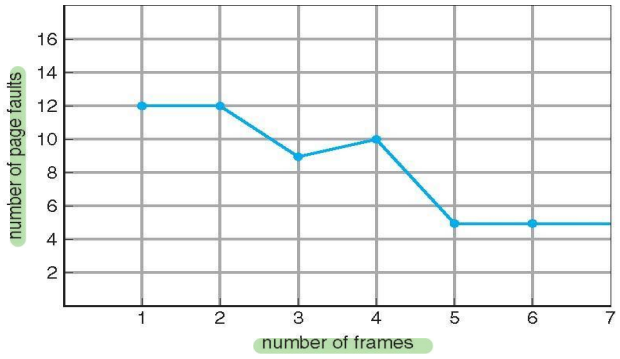
15 page faults

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly** 贝拉迪异常
- How to track ages of pages? – implementation easy
  - Just use a FIFO queue (the time this page brought into the memory)

FIFO Page Replacement

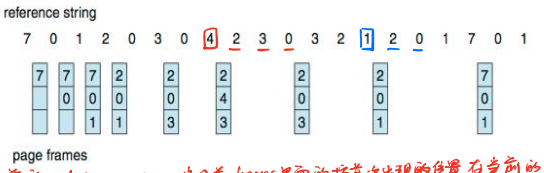


FIFO Illustrating Belady's Anomaly 说明



Optimal Algorithm

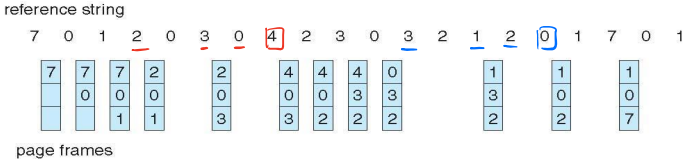
- Replace page that will not be used for longest period of time in the future
  - If possible, ideally select a page that will not be used at all in the future. In practice, this may not be feasible, so a page can be brought into the memory multiple times
  - 9 is the optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs as a comparison



规律：对于当前的 reference string，将目前 frames 里面的按首次出现的位置在当前的 reference string 后面找出，然后替换位于最右端的那个

Least Recently Used (LRU) Algorithm

- Use the past knowledge rather than future as an approximation for OPT
- Replace page that has not been used for the most amount of time
- Associate the time of last use with each page – complex, as this need to be updated with each memory reference

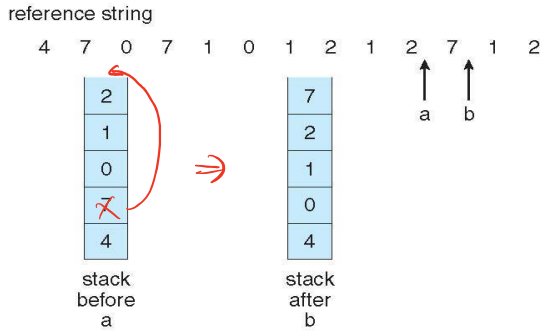


- 12 faults – better than FIFO but worse than OPT
- Generally a good algorithm and frequently used
- But how to implement? - An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use.

LRU Algorithm Implementation 实现

- **Counters** implementation
  - Every page entry adds a *time-of-use field* recording a logical clock or counter. The clock is incremented for every memory reference.
  - Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
  - When a page needs to be replaced, look for the counters to find the smallest value
    - Search through table needed to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access
- **Stack** implementation
  - Keep a stack of page numbers
  - Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom
  - This requires multiple pointers to be changed upon each reference
  - Each update more expensive, but no need to search for replacement

Use of A Stack to Record the Most Recent Page References



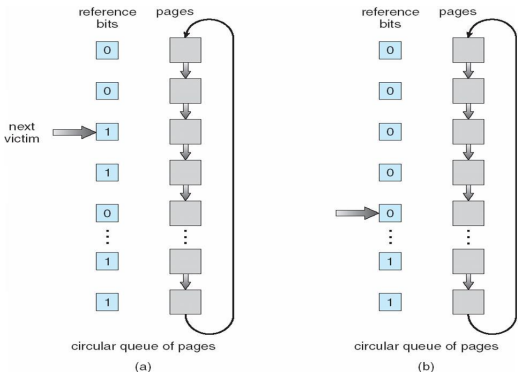
LRU Algorithm Discussions

- LRU and OPT are cases of **stack algorithms** that don't suffer from Belady's Anomaly
  - A stack algorithm can be shown that the set of pages in memory for **n** frames is always a subset of the set of pages that would be in memory with **n+1** frames allocated
  - For LRU replacement, the set of pages in memory would be the **n** most recently referenced pages**.** If the number of frames allocated is increased to (**n+1**), these same **n** pages will still be part of the **n+1** most recently referenced, so will still be in memory
- Both the implementation of LRU A(counter and stack) requires extra hardware assistance
- The updating of the clock fields or stack must be done for **every memory reference**
  - If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten!

LRU Approximation Algorithms

- **Reference bit**
  - Each page associate a bit, initially = 0, associated with each entry in the page table
  - When page is referenced (read or write), the reference bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - Corse-grained approximation, but do not know the order of use
- **Second-chance algorithm**
  - FIFO order, plus hardware-provided reference bit - clock replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it (select it as the victim)
    - reference bit = 1 then:
      - set reference bit 0, leave page in memory (second chance)
      - replace next page, subject to same rules (FIFO and clock)

Second-Chance (Clock) Page-Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- The least frequently used (**LFU**) Algorithm: replaces the page with the smallest count
- The most frequently used (**MFU**) Algorithm: replace the page with the largest count based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Neither LFU nor MFU replacement is commonly used**.** The implementation of such algorithms is expensive, and they do not approximate OPT replacement well

Allocation of Frames

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory, or different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs certain **minimum** number of frames in order to execute its program
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- The maximum, of course, is the total frames required for a process
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation 固定分配

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
- Proportional allocation – Allocate according to the size of process
  - Dynamic as the degree of multiprogramming and process sizes change over the time

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

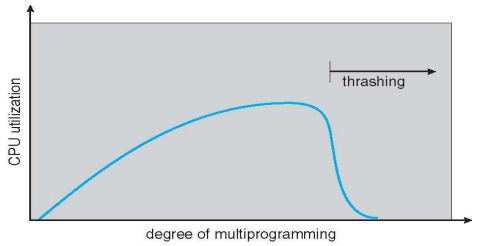
$a_2 = \frac{127}{137} \times 64 \approx 59$

Global vs. Local Allocation

- Global replacement – process selects a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; thus, one process can take a frame from another process
  - For instance, this can be based on priority – priority allocation in which a process can preempt memory from a lower priority process
  - This can result in better system throughput
  - But process execution time can vary greatly, as a process cannot control its own page-fault rate.
- Local replacement – each process selects from only its own set of allocated frames
  - What we used in the page replacement algorithms earlier
  - The set of page for a process is only affected by the paging behaviour of that process
  - More consistent per-process performance
  - But possibly underutilized memory, since pages allocated to a process can not be utilized by another process, even if this page is not currently used by the process holding it

Thrashing 抖动

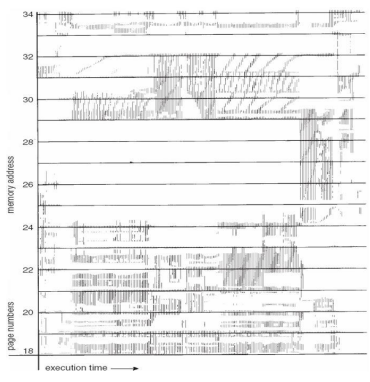
- If a process does not have "enough" pages, the page-fault rate would be very high
  - Page fault to get page, and replace an existing frame, but quickly need replaced frame back
  - This leads to low CPU utilization – OS might think "by mistakes" that it needs to increase the degree of multiprogramming in order improve the CPU utilization – aggravate the problem
- Thrashing ≡ a process or a set of processes is busy swapping pages in and out
- This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing, which results in serious performance problems



Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - A locality is a set of pages that are actively used together. A running program is generally composed of several different localities over the time, which may overlap
  - Memory access or subsequent memory access tends to stay in the same set of pages
  - Process migrates from one locality to another, e.g., operating on a different set of data or call a function (different code segment)
  - Localities may overlap, instructions or part of the data being manipulated
- Why does thrashing occur?
  Σ size of locality (of all processes) > total memory size
  - If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.
  - We can limit the effects by using local replacement, this way thrashing in one process can not steal frames from another process and cause the latter to thrash as well

Locality In A Memory-Reference Pattern 局限性

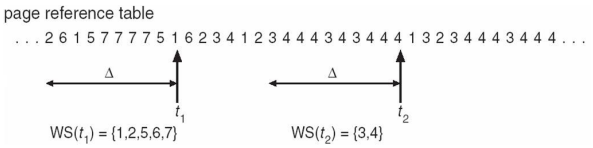


- Recall program memory access patterns exhibit temporal and spatial locality
- The left Figure illustrates the concept of locality and how a process's locality changes over time. At time (a), the locality is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}. At time (b), the locality changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.

Working-Set Model 工作集模型

- Δ ≡ working-set window ≡ a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working set of Process P_i) =
  total number of distinctive pages referenced in the most recent Δ - this varies in time
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of the current locality in the system (of all processes)
- if $D > m \Rightarrow$ Thrashing – at least one process is short of memory
- Policy if $D >$ m, then suspend or swap out one of the processes
- The working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible, thus optimizes CPU utilization
- The difficulty with a working-set model is how to keep track of the working set.

Working-Set Model (Cont.)

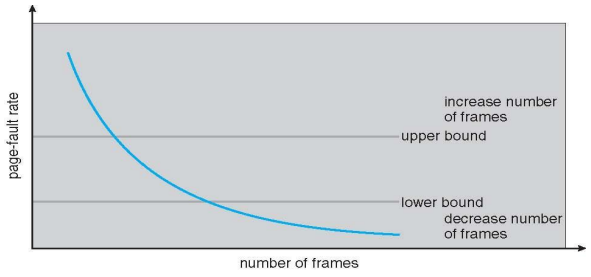


Keeping Track of the Working Set

- It is difficult to keep track of the working set, as working-set window is a moving window which needs to be updated **for each memory reference**
- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page, (0,0), (0,1),(1,0),(1,1)
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 ⇒ page in working set
- This is of course not accurate, as we cannot tell where, within an interval of 5,000, a reference occurred
- Improvement = 10 bits and interrupt every 1000 time units, more accurate but cost is higher
- Accuracy versus complexity

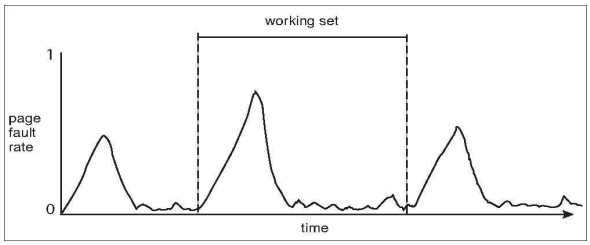
Page-Fault Frequency

- More direct approach than WSS
- Establish "acceptable" page-fault frequency (PFF) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Working Sets and Page Fault Rates

- Relationship between working set of a process and its page-fault rate
- Working set changes over time
- The page-fault rate of a process transitions between peaks and valleys over time.



Other Considerations

- Prepaging
- Page size
- TLB reach
- Program structure

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume *s* pages are prepaged and *α* of the pages is used
  - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
  - α near zero ⇒ prepaging loses

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration a conflicting set of criteria:
  - Fragmentation – calls for smaller page size
  - Page table size – calls for larger page size
  - Resolution – isolate the memory actually be used
  - I/O overhead – larger page size requires longer I/O time
  - Number of page faults – smaller page size can increase the number of page faults
  - Locality – ideally each page should match the current locality
  - TLB size and effectiveness – larger page size improves the TLB reach
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise, there might be a high degree of page faults, or the access time slows down
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
  - `Int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
     for (i = 0; i < 128; i++)
          data[i,j] = 0;
```

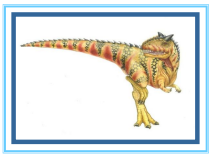
128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
          data[i,j] = 0;
```

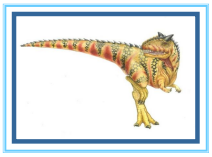
128 page faults

End of Chapter 10



Chapter 11: Mass-Storage Systems

大容量存储系统



Chapter 11: Mass-Storage Systems

- Overview of Mass Storage Structure
- Disk Structure
- Disk Scheduling
- RAID Structure

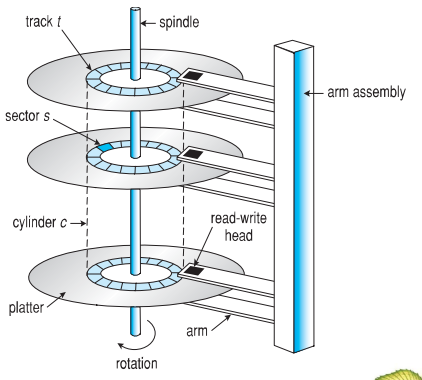
Objectives

- Describe the physical structure of secondary storage devices and the effect of a device's structure on its uses
- Explain the performance characteristics of mass-storage devices
- Evaluate I/O scheduling algorithms
- Discuss operating-system services provided for mass storage, including RAID

Moving-head Disk Mechanism

移动磁头磁盘机制

- Each disk platter has a flat circular shape with diameters of 1.8, 2.5 to 3.5 inches
- Two surfaces of a platter covered with a magnetic materials for storing information
- A read-write head "flies" just above each surface of every platter
- The heads are attached to a disk arm that move all heads as a unit
- The surface of a platter is logically divided into circular **tracks**, which are subdivided into hundreds of **sectors** (per track)
  轨道 扇区
- The set of tracks that are at one arm position makes up a **cylinder** - thousands of concentric cylinders in a disk drive
- There could be thousands of concentric cylinders in a disk drive



Overview of Mass Storage Structure

- Magnetic disks provide bulk of secondary storage of modern computers
  辅助
  - Drives rotate at 60 to 250 times per second
  - Transfer rate is rate at which data flow between drive and computer
  - Positioning time (random-access time) is time to (1) move disk arm to desired cylinder (seek time) and (2) time for desired sector to rotate under the disk head (rotational latency)
    定位时间
  - Head crash results from disk head making contact with the disk surface – that's bad!
    磁头崩溃
- Disks can be removable
- Drive attached to computer via I/O bus
  - Busses vary, including EIDE, ATA, SATA, USB, Fibre Channel, SCSI, SAS, Firewire
  - Host controller in computer uses bus to talk to disk controller built into drive or storage array

Hard Disk Drive

硬盘驱动器

- Platters range from .85" to 14" (historically)
  - Commonly 3.5", 2.5", and 1.8"
- Range from 30GB to 3TB per drive
- Performance
  - **Transfer Rate** – theoretical – 6 Gb/sec
  - **Effective Transfer Rate** – real – 1Gb/sec
  - **Seek time** from 3ms to 12ms – 9ms common for desktop drives
  - Average seek time measured or calculated based on 1/3 of tracks
  - RPM typically, 5,400, 7,200, 10,000 and 15,000 rotations per minute
  - Latency based on spindle speed
    延迟
    - 1/(RPM/60) = 60/RPM
  - **Average latency** = ½ latency
  - For example, with 7200 rpm, that is 120 rps, the average latency is 1/240 = 4.17 mini-seconds

Spindle [rpm]	Average latency [ms]
4200	7.14
5400	5.56
7200	4.17
10000	3
15000	2

Hard Disk Performance 性能

- Access latency or average access time = average seek time + average latency
  - For fast disk 3ms + 2ms = 5ms
  - For slow disk 9ms + 5.56ms = 14.56ms
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead
- For example, to transfer a 4KB block on a 7200 RPM disk with a 5ms average seek time, 1Gb/s transfer rate with a 0.1ms controller overhead =
  - 5ms + 4.17ms + 4KB / 1Gb/sec + 0.1ms =
    - 4KB = 4 * 2^10 * 8 = 32 * 2^10 bits
    - 1Gbps = 1 * 2^30 bits per seconds
  - 9.27ms + 32 / 2^20 s =
  - 9.27ms + 0.0305 ms ≈ 9.3ms
  - Thus, it takes an average 9.3ms to transfer 4KB, thus effective bandwidth is 4KB/9.3ms ≈ 3.5 Mb/sec only (with a transfer rate at 1 Gb/sec given the overhead).

average latency

$\frac{7200}{60} = 120$ RPS (每秒转)

即：每圈需：$\frac{1}{120} = 8.33$ ms.

平均寻道时间 = $8.33/2 \approx 4.17$ ms

Hard Disk Performance (Cont.)

- There is a huge gap 差距 in hard drive performance between random and sequential 顺序 workloads
- Consider a disk with 300GB capacity, average seek time is 4 milliseconds (4 ms), RPM is 15,000 RPM or 250 RPS (the average rotation time is 2 ms), transfer rate is 125MB/s, and a 4KB read occurs at a random location, recall

  average access time = average seek time + average latency
- Average I/O time = average access time + (amount to transfer / transfer rate) + controller overhead (ignored) = 4 ms + 2 ms + 30 microseconds ≈ 6ms
- The effective bandwidth or transfer rate is 4KB/6ms = 0.66MB/s
- With sequential access of a 100 MB file, suppose there is only one seek and rotation, this would yield an effective bandwidth or transfer rate close to 125MB/s

Solid-State Disk (SSD) 固态硬盘

- An SSD is nonvolatile memory (NVM) used like a hard drive
  - Many technology variations, e.g., from DRAM with a battery to maintain its state in a power failure, through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips
  - SSDs can be more reliable than HDDs because they have no moving (mechanical) parts
  - They are much faster because they have no seek time or rotation latency.
  - They consumes less power – power efficiency
  - But they are more expensive per MB, have less capacity, and may have shorter life span
- Because they are much faster than magnetic disk drives, standard bus interface can be too slow, causing a major limit on throughput
  - Some connect directly to system bus (e.g., PCI)
  - Some use them as a new cache tier, moving data between magnetic disk, SSDs, and memory to optimize performance

Magnetic Tape 磁带

- Magnetic tape was an early secondary-storage medium
  - It is relatively permanent and can hold large quantities of data
  - Access time is slow, as moving to the correct spot on a tape can take minutes
  - Random access ~1000 times slower than magnetic disk, so they are not very useful for secondary storage in modern computer systems
- Mainly used for backup, storage of infrequently-used data, or as a medium of transferring information from one system to another
- Tape capacities vary greatly, depending on the particular kind of tape drive, with current capacities exceeding several terabytes, and typically between 200GB and 1.5TB

Disk Structure 磁盘结构

- Disk drives are addressed as large one-dimensional arrays of **logical blocks**, where a logical block is the smallest unit of transfer. In another word, disk is represented by a number of disk blocks, each block has a unique block number – disk address 逻辑块.
  - The size of a logical block is usually 512 bytes
  - Low-level formatting creates logical blocks on physical media
- The one-dimensional array of logical blocks is mapped into sectors 扇区 of the disk, sequentially:
  - Sector 0 is the first sector of the first track on the outermost cylinder
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost
  - Logical to physical address (consist of a ***cylinder number***, a ***track number*** within the cylinder, and a ***sector number*** with the track) should be easy, except
    - Defective sectors, mapping hides this by substituting spare sectors from elsewhere on disk 损坏的
    - The number of sectors per track may not be a constant on some devices. Non-constant # of sectors per track via constant angular velocity

特殊情况

Disk Scheduling 磁盘调度

- The operating system is responsible for using hardware efficiently — for disk drives, this means having fast access time and large disk bandwidth
- The seek time is the time for the disk head arm to move the heads to the corresponding cylinder containing the desired sector, which can be measured by the seek distance in term of number cylinders/tracks.
- The rotational latency is the additional time for the disk to rotate the desired sector to the disk head.
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.
- We can improve both access time and the bandwidth by managing the order in which disk I/O requests are serviced.

Disk Scheduling (Cont.)

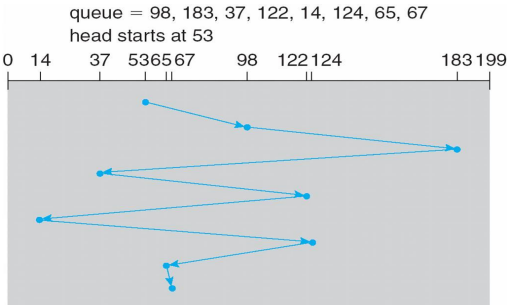
- There are many sources of disk I/O requests, from OS, system processes, and user processes
  - I/O request includes input/output modes, disk address, memory address, number of sectors to transfer
- OS maintains a queue of requests per disk or device
  - In a multiprogramming system with many processes, the disk queue often has several pending requests.
- Idle disk can immediately work on I/O request, busy disk means that requests must be queued.
  - Optimization only make sense when a queue of I/O request exists.
- Disk drive controllers have small buffers and manage a queue of I/O requests (of varying "depth").
  - When one request is completed, which pending request to select to service next?
  - **Disk scheduling**
- We next illustrate scheduling algorithms with a request queue (0-199), 0-199 are cylinder numbers.

  98, 183, 37, 122, 14, 124, 65, 67

  The current Head position is **53**

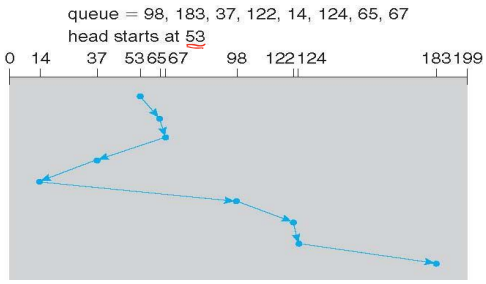
FCFS First-Come-First-Serve

- **FCFS** is intrinsically fair, but it generally does not provide the fastest service
- Illustration shows total head movement of **640** cylinders



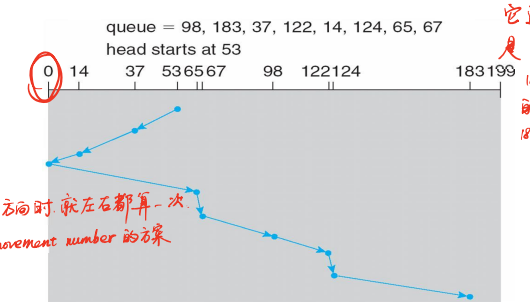
SSTF Shortest-Seek-Time-First

- The **Shortest Seek Time First** (**SSTF**) selects the request with the least seek time from the current head position, i.e., choose the pending request closest to the current head position (either direction)
- SSTF scheduling is a form of SJF scheduling (greedy algorithm); may cause starvation of some requests, as requests may arrive at any time dynamically
- Illustration shows total head movement of **236** cylinders



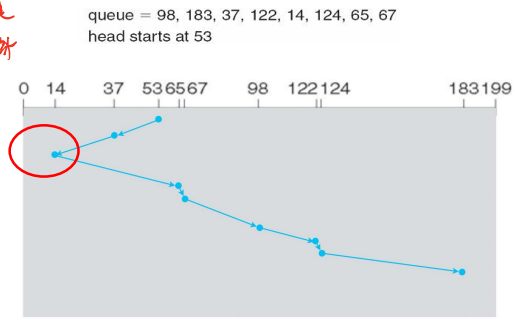
SCAN Scheduling

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. This sometimes called the elevator algorithm.
- Note if requests are uniformly distributed across cylinders, the heaviest density of requests are at other end of disk and those wait the longest. Also, we need to know direction of head movement.
- Illustration shows total head movement of **236** cylinders



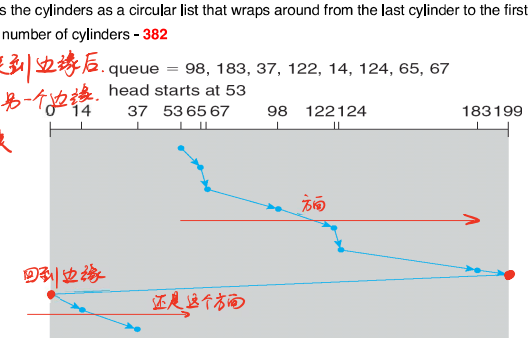
LOOK Scheduling

- LOOK Scheduling: Similar to SCAN scheduling, but the disk arm only goes as far as the final request in each direction (instead of the end of the disk)
- In the following example, the total head movement is **208** cylinders



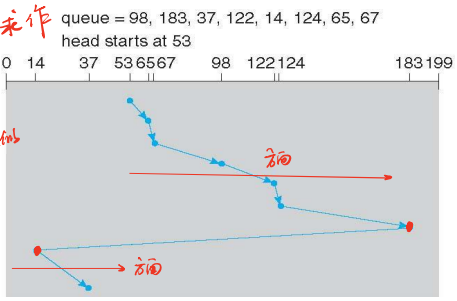
C-SCAN

- **C-SCAN**, **Circular-SCAN**, a variant of SCAN, provides a more uniform waiting time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one
- Total number of cylinders - **382**



C-LOOK

- LOOK a version of SCAN, C-LOOK a version of C-SCAN
- Disk arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk
- Total number of cylinders? – 322 (for C-LOOK) and 308 for LOOK



Selecting a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal as it increases performance over FCFS. SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause starvation problems.
- The scheduling performance also depends on the number and type of requests. If only one request, all scheduling algorithms should behave the same (like FCFS scheduling)
- Requests for disk service are greatly influenced by the file-allocation method (to be discussed)
  - Contiguously allocated file will generate several requests close together on the disk, resulting in limited head movement, while a linked or indexed file may include blocks widely scattered on the disk, resulting in greater head movement.
- The location of directories and index blocks are also important, which are accessed frequently.
  - Directory entry and file data on different cylinders cause excessive head movement
  - Caching directory and index block in memory help
- The disk-scheduling algorithm should be written as *a separate module* of the operating system, allowing it to be replaced with a different algorithm if necessary. Either SSTF or LOOK is a reasonable choice as the default algorithm.

RAID - Improving Reliability via Redundancy

- RAID – Redundant Arrays of Independent Disks
  - In the past, RAID composed of small, cheap disks were viewed as a cost-effective alternative to large, expensive disks (once called redundant arrays of inexpensive disks)
  - Now RAIDs are used for higher reliability via redundancy and higher data-transfer rate (access in parallel)
- Increases mean time to failure
  - The chance that a disk out of N disks fails is much higher than the chance that a specific single disk fails. Suppose that the mean time to failure of a single disk is 100,000 hours, the mean time to failure of some disk in an array of 100 disks will be 100,000/100 = 1,000 hours, or 41.66 days!
- The data loss rate is unacceptable if we store only one copy of the data
  - The solution is to introduce redundancy; the simplest (but most expensive) approach is to duplicate every disk, called mirroring. Every write is carried out on two physical disks. Data will be lost only if the second disk fails before the first failed disk is replaced.
- The mean time to repair is the time it takes (on average) to replace a failed disk and to restore data on it – exposure time when another failure could cause data loss
  - Suppose the mean time to failure of a single disk is 100,000 hours and the mean time to repair is 10 hours. The mean time to data loss is 100, 000^2 / (2 * 10) = 500 * 10^6 hours, or 57,000 years!

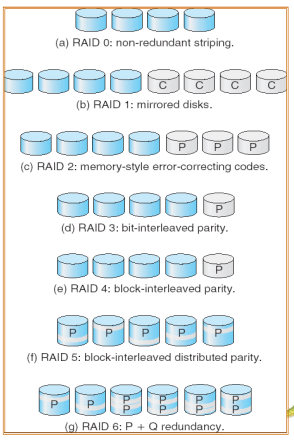
$$\text{mean time to data loss} = \frac{(\text{mean time to failure})^2}{2 \times (\text{mean time to repair})}$$

RAID – Improving Performance via Parallelism

- Parallelism in a disk system, via data striping, has two main goals:
  - Increase the throughout of multiple small access by load balancing
  - Reduce the response time of large access
- Bit-level striping
  - For example, if we have an array of 8 disks, we can write bit *i* of each byte to disk *i*. The array of 8 disks can be treated as a single disk with sectors that are 8 times the normal sector size. The access rate can be improved by 8 times!
  - Bit-level striping can be generalized to include a number of disks that either is a multiple of 8 or divides 8. For example, with an array of 4 disks, bit *i* and *4+i* of each byte cane be stored in disk *i*
- Block-level striping
  - Blocks of a file are striped across multiple disks
  - With *n* disks, block *i* of a file goes to disk (*i* mod *n*)+*i*

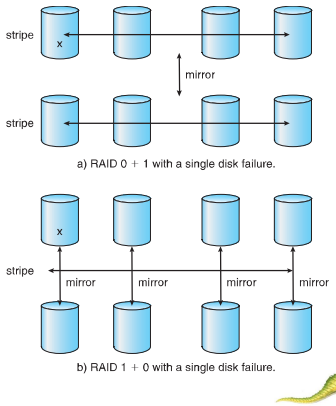
RAID Structure

- Mirroring provides high reliability, but expensive
- Striping provides high data-transfer rates, but does not provide reliability
- Many schemes provide redundancy at lower cost by using striping combined with "parity" bits, generally, classified into 6 RAID levels
- In the RAID levels, 4 disks' worth of data are stored. P indicates error-correcting bits, C indicates a second copy of the data
  - RAID 0 refers to disk array with striping at the level of blocks with non redundancy
  - Mirroring or shadowing (RAID 1) keeps duplicate of each disk
  - Striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance and high reliability
  - Block interleaved parity (RAID 4, 5, 6) uses much less redundancy



RAID (0 + 1) and (1 + 0)

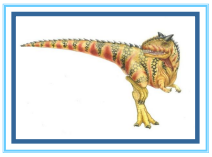
- Both striped mirrors (RAID 1+0) or mirrored stripes (RAID 0+1) provides high performance (RAID 0) and high reliability (RAID 1)
- (RAID 0+1) a set of drives are striped, and then the stripe is mirrored to another, equivalent stripe
- (RAID 1+0) drives are mirrored in pairs and then the resulting mirrored pairs are striped



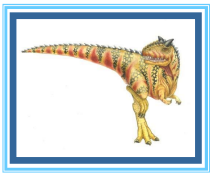
Other Features

- Regardless of what RAID implemented, other useful features can be added at each level
- Snapshot is a view of file system before the last update took place (for recovery)
- Replication is automatic duplication of writes between separate sites for redundancy and disaster recovery. This can be synchronous (each block must be written locally and remotely before the write is considered complete) or asynchronous (writes are grouped together and written periodically)
- A hot spare disk is not used for data but is configured to be used as a replacement in case of disk failure
  - For instance, a hot spare can be used to rebuild a mirrored pair should one of the disks in the pair fails. In this way, RAID level can be reestablished automatically, without waiting for the failed disk to be replaced/repaired.

End of Chapter 11



Chapter 13: File-System Interface



Chapter 13: File System Interface

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection

Objectives

- To explain the functions of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including *access methods*, *file sharing*, and *directory structures*
- To explore file-system protection

File Concept

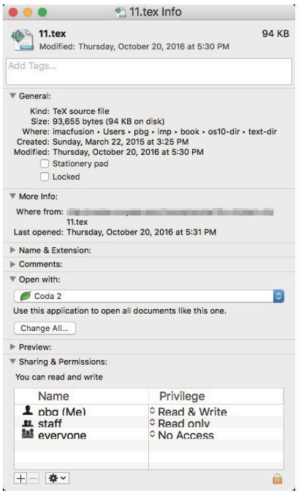
- Contiguous logical address space
- Types:
  - Data
    - numeric 数字
    - character 字符
    - binary
  - Program
- Contents defined by the file's creator
  - Many types, consider text file, source file, executable file

文本文件. 源文件. 可执行文件.

File Attributes 文件属性.

- Name – information kept in human-readable form
- Identifier – unique tag (number) identifies files within a file system 标识符
- Type – needed by systems that support different types
- Location – pointer to file location on device
- Size – current file size
- Protection – controls who can do reading, writing, executing, etc.
- Time, date, and user identification 标识 – data for protection, security, and usage monitoring
- Information about files are kept in a **directory structure**, maintained on the disk - part of which currently in use can be cached in main memory for fast access
- Many variations, including extended file attributes such as file checksum

File info Window on Mac OS X



File Operations 文件操作

- File is an ADT or abstract data type 抽象数据类型
- Create – create a file
- Write – at write pointer location
- Read – at read pointer location
- Reposition within file - seek
- Delete
- Truncate 截断.
- Open(F_i) – search the directory structure on disk for entry F_i, and move the content of entry to memory, preparing file for subsequent access
- Close (F_i) – move the content of entry F_i in memory to directory 目录 structure on disk

- Such operations involve the changes of various OS kernel data structures

Open Files

- Several data structures are needed to manage open files:
  - Open-file tables: tracks open files, system-wide open-file table, and per-process open-file table
  - File pointer: pointer to last read/write location, per process that has the file open
  - File-open count: counting the number of processes that the file has been opened – to allow removal of data from the open-file table when the last processes closes it (when file-open count is zero)
  - Disk location of a file: cache of data access information
  - Access rights: per-process access mode information

File Types – Name, Extension

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Access Methods 访问方法

- Sequential Access – simplest access method 顺序访问

```
read next
write next
reset
no read after last write
    (rewrite)
```

- Direct Access – file is fixed length logical records

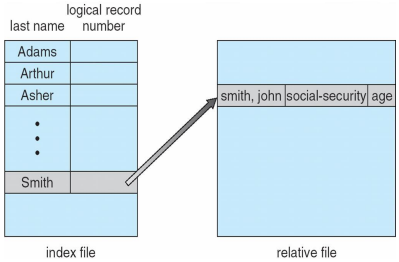
```
read n
write n
position to n
       read next
       write next
rewrite n
```

n = relative block number

- Relative block numbers allow OS to decide where file should be placed
  - See disk block allocation problem in Chapter 14

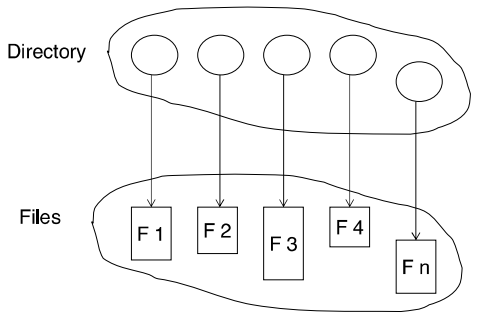
Other Access Methods

- Other file access methods can be built on top of direct-access method
- Generally, involve creation of an index for a file 文件索引.
  - Keep index in memory for fast location of the data to be operated on
  - If too large, index (in memory) of the index (on disk)
- IBM indexed sequential-access method (ISAM) is an example
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS
- VMS operating system provides index and relative files as another example



Directory Structure 目录结构

- A collection of nodes containing information about all files

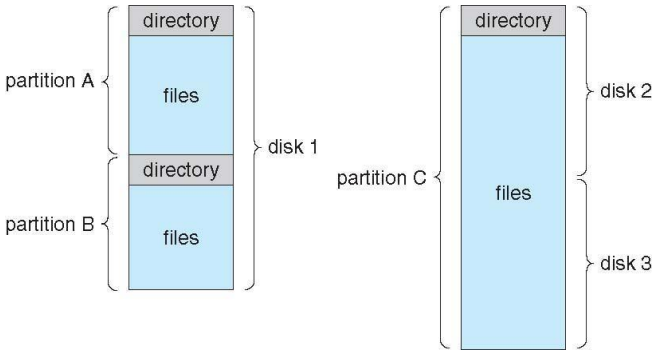


Both the directory structure and files reside on disk 驻留

Disk Structure 磁盘结构

- Disk can be subdivided into partitions
- Disks or partitions can be RAID protected against failure
- Disk or partition can be used raw – without a file system, or formatted with a file system
- Partitions are also known as minidisks, slices 切片
- An entity on a disk containing a file system known as a **volume** 卷
- Each volume containing a file system also keeps track of the file system info in device directory or volume table of contents
- Other than general-purpose file systems, there are many special-purpose file systems, frequently within the same operating system or computing systems

A Typical File-system Organization



Operations Performed on Directory

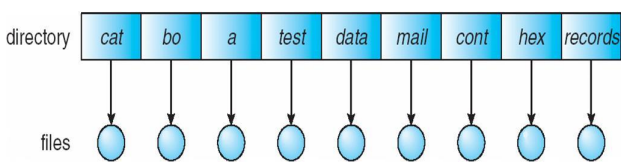
- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system 遍历文件系统

Organize the Directory (Logically) to Obtain 组织目录以获取

- **Efficiency** – locating a file quickly
- **Naming** – convenient to users
  - Two users can have same name for different files
  - The same file can have several different names
- **Grouping** – logical grouping of files by properties 属性, (e.g., all Java programs, all games, my comp3511, …)

Single-Level Directory 单级目录

- A single directory for all users

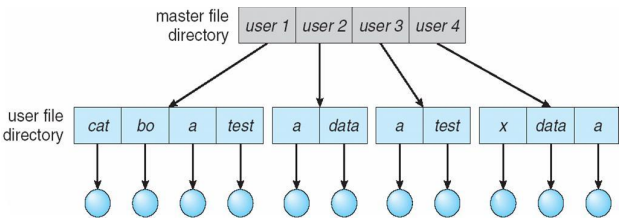


Naming problem

Grouping problem

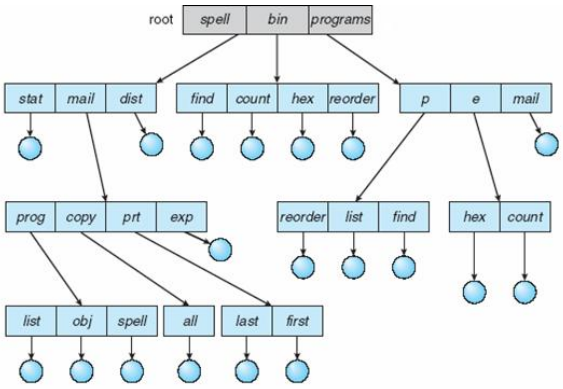
Two-Level Directory

- Separate directory for each user



- Path name – need a pathname to identify a file/dir, e.g., /user1/cat
- Can have the same file name under different users (paths)
- More efficient searching than single-level directory
- No grouping capability

Tree-Structured Directories 树状结构目录



Tree-Structured Directories (Cont.)

- Efficient searching
- Grouping Capability
- Current directory (working directory)
  - `cd /spell/mail/prog`
  - `type list`

Tree-Structured Directories (Cont)

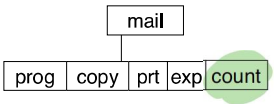
- Absolute or relative path name
- Creating a new file is done in the current directory
- Delete a file in the current directory

  `rm <file-name>`
- Creating a new subdirectory is done in current directory

  `mkdir <dir-name>`

  Example: if in current directory `/mail`

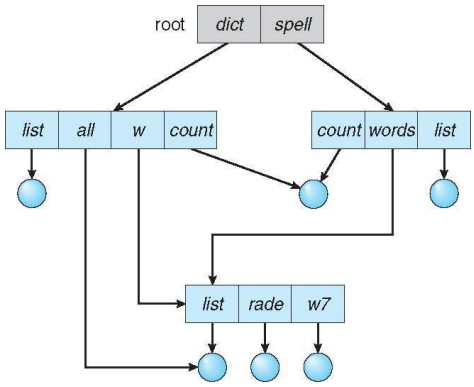
  `mkdir count`



Deleting "mail" ⇒ deleting the entire subtree rooted by "mail"

Acyclic-Graph Directories 无环图目录

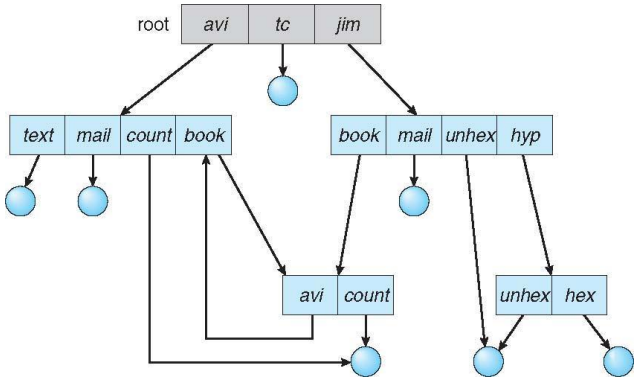
- Have shared subdirectories and files – more flexible and complex



Acyclic-Graph Directories (Cont.)

- New directory entry type 新目录条目类型
  - Link – another name (pointer) to an existing file
  - Resolve the link – follow pointer to locate the file 解析
- Two different (path) names (aliasing) 别名
  - Ensure not traversing shared structures more than once
- Deletion might lead to that dangling pointers that point to empty files or even wrong files 悬空指针
- There is also difficulty ensuring there is no cycles in a graph – complexity associated with it

General Graph Directory 通用图目录

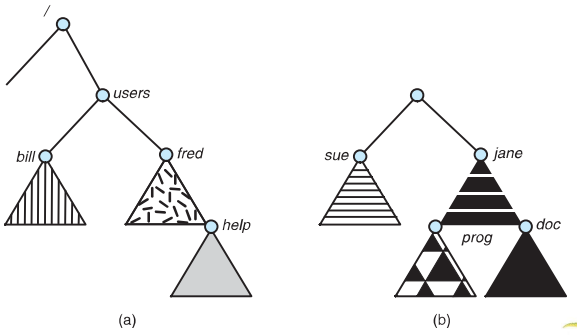


General Graph Directory (Cont.)

- How do we guarantee no cycles?
  - Allow only links to file not subdirectories – sometime not convenient
  - Every time a new link is added use a cycle detection algorithm to determine whether there is a cycle or not – time consuming

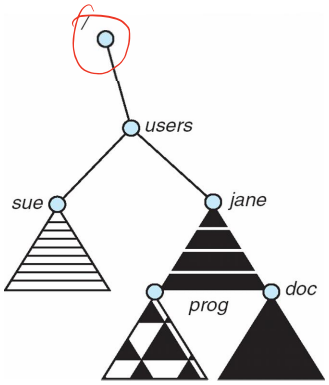
File System Mounting

- A file system must be mounted before it can be accessed – just like a file must be opened before it is used
- A unmounted file system (i.e., Fig. (b)), to be mounted at a mount point



Mount Point

- Volume is mounted at /users



File Sharing

- Sharing of files in multi-user systems is desirable
- Sharing may be done through a protection scheme
- In distributed systems, files may be shared across a network
  - Network File System (NFS) is a common distributed file-sharing method
- With a multi-user system
  - User IDs identify users, allowing permissions and protections to be per-user
  - Group IDs allow users to be in groups, permitting group access rights
  - Owner of a file / directory
  - Group of a file / directory

Protection

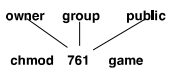
- File owner/creator of the file should be able to control:
  - what can be done
  - by whom
- Types of access
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users on Unix / Linux

a) owner access	7	⇒	RWX 1 1 1
b) group access	6	⇒	RWX 1 1 0
c) public access	1	⇒	RWX 0 0 1

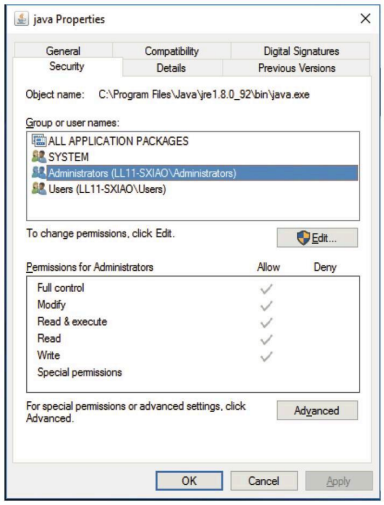
- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file or subdirectory, define an appropriate access.



Attach a group to a file

```
chgrp    G    game
```

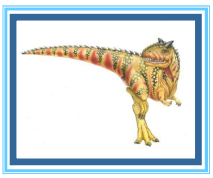
Windows 10 Access-Control List Management



A Sample UNIX Directory Listing

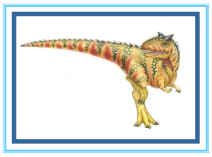
-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

End of Chapter 13



文件系统实现

Chapter 14: File-System Implementation



Chapter 14: Implementing File Systems

- File-System Structure
- File-System Implementation
- Directory Implementation
- Allocation Methods
- Free-Space Management

Objectives

- To describe the details of implementing file systems and directory structures
- To discuss block allocation and free-block algorithms and trade-offs

文件的分配方法有三种主要类型：连续分配、链接分配和索引分配。连续分配提供最佳性能，但可能导致外部碎片；链接分配避免了外部碎片，但不支持随机访问；索引分配通过索引块集中管理指针，支持随机访问，但可能浪费空间。

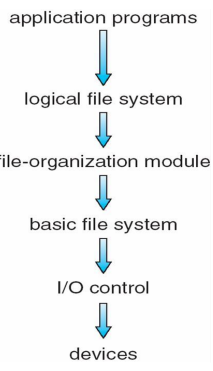
File-System Structure

- Disks provide most of the secondary storage on which file systems are maintained.
- Two characteristics of disks make them convenient for this usage:
  1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back onto the same place on the disk
  2. A disk can access directly any block of information it contains. Thus it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for disk to rotate – discussed in Chapter 11
- To improve I/O efficiency, I/O transfers between memory and disk are performed in units of **blocks**. Each block contains one or more sectors. A sector size varies from 32 bytes to 4,096 bytes (4KB), usually 512 bytes (0.5 KB)

File-System Structure (Cont.)

- File structure
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage – hard drive or disks
  - It provides efficient and convenient access to disk by allowing data to be stored, located, and retrieved easily
  - It provides user interface: file and file attributes, operations on files, directory for organizing files
  - It provides data structures and algorithms for mapping logical file system onto physical secondary storage devices
- File systems are organized into different layers

Layered File System



File System Layers

- **I/O control** and **device drivers** manage I/O devices at the I/O control layer
  - It consists of device drivers and interrupt handlers to transfer information between memory and disks
  - Given commands like "read drive1, cylinder 72, track 2, sector 10" (disk physical address), into memory location 1060" outputs low-level hardware specific commands to hardware controller
- **Basic file system** issues generic commands to the appropriate device driver to read and write physical blocks on the disk
  - given commands like "retrieve block 123", translates it to a specific device driver
  - It manages memory buffers and caches that hold various file-system, directory, and data block. (allocation, freeing, replacement)
  - Buffers hold data in transit. A block in memory buffer is allocated before the transfer of a disk block occurs.
  - Caches hold frequently used file-system metadata to improve performance
- **File organization module** knows files, and their logical blocks, as well as physical blocks
  - Translates logical block # (address) to physical block #, pass this to basic file system for transfer
  - Manages free disk space, disk block allocation

File System Layers (Cont.)

- **Logical file system** manages metadata information
  - Metadata includes all of the file-system structures except the actual data, i.e., the contents of files
  - It manages directory structure to provide the information needed by file-organization module.
  - Translates file name into file number or file handle, location by maintaining file control blocks
  - A file control block (FCB) (called inode in Unix file systems) contains all information about a file including ownership, permissions, and location of the file contents (on the disk)
  - It is also responsible for file protection
- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Many file systems are in use today, and most operating systems support more than one file system
  - Each with its own format - CD-ROM is ISO 9660; Unix has UFS (Unix File System) based on FFS; Windows has FAT, FAT32, NTFS (or Window NT File System) as well as floppy, CD, DVD Blu-ray, Linux has more than 40 types of file systems, with extended file system ext2 and ext3; plus distributed file systems, etc.
  - New ones still arriving – ZFS, GoogleFS, Oracle ASM, FUSE

File-System Implementation

Several on-disk and in-memory structures are used to implement a file system.

- On-disk structure, it may contain information about how to boot an operating system stored on the disk, the total number of blocks, number and location of free blocks, directory structure, and individual files
- In-memory information used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount time.

On-Disk File-System Structure

- Boot control block (per volume) contains info needed by system to boot OS from that volume
  - If the disk does not contain an OS, this block can be empty
  - Usually the first block of a volume. In UFS, it is called the boot block. In NTFS, it is the partition boot sector
- Volume control block (per volume) contains volume (or partition) details
  - Total # of blocks, # of free blocks, block size, free block count and pointers, a free FCB count and pointer
  - In UFS, this is called superblock. In NTFS, it is stored in the master file table
- A directory structure (per file system) is used to organize files
  - In UFS, this includes file names and associate inode numbers (FCB in Unix). In NTFS, it is stored in the master file table
- Per-file File Control Block (FCB) contains many details about a file
  - It has a unique identifier number to associate with a directory entry.
  - In UFS, inode number, permissions, size, dates
  - NFTS stores into in master file table using relational DB structure, with a row per file

A Typical File Control Block

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

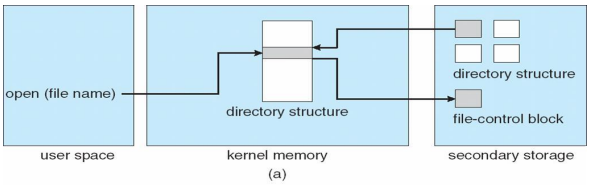
In-Memory File System Structures

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount

- An in-memory mount table contains information about each mounted volume
- An in-memory directory-structure cache holds the directory information of recently accessed directories.
- The system-wide open-file table contains a copy of the FCB of each open file to locate the files, as well as other information
- The per-process open-file table contains a pointer to the appropriate entry in the system-wide open-file tale, as well as other information such as per-process file protection and access rights.
- Buffers hold file-system blocks when they are being read from disk or written to disk

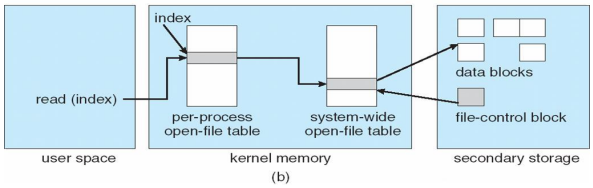
In-Memory File System Structures

- Figure below refers to opening a file
  - The **open() operation** passes a file name to the logical file system.
  - If the file is already in use by another process, only the per-process open-file table entry is created pointing to the corresponding entry of this file in the system-wide open-file table.
  - If not, it searches the directory structure (part of it may be cached in memory). Once the file is found, an entry in system-wide open-file table and in per-process open-file table are created, respectively. The **FCB** is copied into the system-wide open-file table in memory for subsequent access
  - The system-wide open-file table not only stores the FCB but also tracks the number of processes that have open, and thus are using this file – file count
  - The other fields in the per-process open-file table may include a pointer to the current location in the file (for next read() or write() operation) and access mode in which the file is open.



In-Memory File System Structures

- Figure below refers to reading a file
  - The open() call returns a pointer to the appropriate entry in the per-process open-file table. In another word, open() operation creates the corresponding entries in both per-process and system-wide open-file tables, and subsequent operations use this pointer to locate file content without the need to access directory structure – this speeds up the subsequent operations on the file
  - All file operations are then performed via this pointer. UNIX systems refer to it as a file descriptor; Windows refers to it as a file handle.
  - Data from read() eventually copied to specified user process memory address (part of process address space)



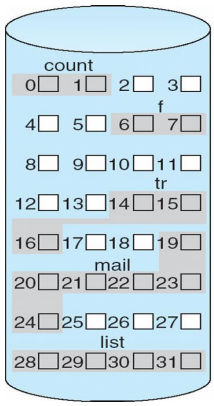
Directory Implementation

- The selection of directory-allocation and directory management algorithms significantly affects the efficiency, performance, and reliability of the file system.
- **Linear list** of file names with pointer to the data blocks
  - Simple to program
  - Time-consuming to execute
  - The major disadvantage of a linear list is that finding a file requires a linear search time.
    - Cache in memory the frequently used directory information
    - Could keep ordered alphabetically via linked list or use B+ tree
- **Hash Table** – linear list with hash data structure
  - Decreases directory search time
  - Collisions – situations where two file names hash to the same location
  - Only good if entries are fixed size, or use chained-overflow method
  - The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

Allocation Methods - Contiguous

- **An allocation method** refers to how disk blocks are allocated for files, such that the disk space is utilized effectively, and files can be accessed quickly.
- There are three major methods of allocating disk space that are widely in use, **contiguous**, **linked** and **indexed**.
- **Contiguous allocation** – each file occupies a set of contiguous blocks of the disk
  - Best performance in most cases – support sequential and direct access easily
  - Simple – only starting location (block #) and length (number of blocks) are required
  - Problems with finding space for a new file, and when file size grows
  - This is also a dynamic storage-allocation problem discussed earlier, which involves how to satisfy a request of size n (variable) from a list of free variable-sized holes – external fragmentation exists
  - Best-fit and first-fit are common strategies and shown to be more efficient than worst-fit.
  - The cost of compaction is particularly high for large disk, which may take hours. Some system require that compaction be done only when off-line, with the file system unmounted
  - This is preferred for files that the file size must be known at the time of file creation – overestimation leads to large amount of internal fragmentation

Contiguous Allocation of Disk Space

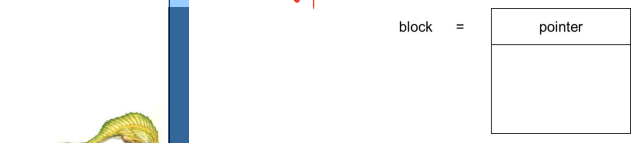


directory

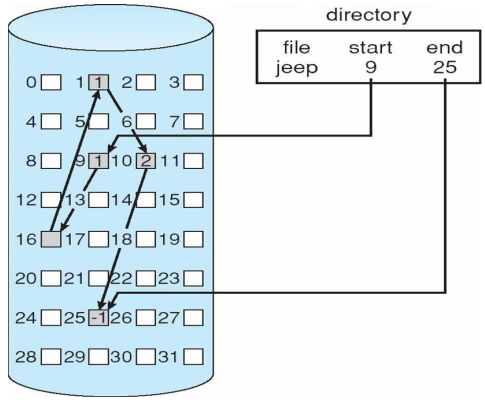
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Allocation Methods - Linked

- **Linked allocation** – each file consists of a linked-list of blocks
  - Each file is a linked list of disk blocks, which may be scattered anywhere on the disk
  - The directory contains a pointer to the first and last blocks of a file
  - File ends at null pointer (the end-of-list pointer value)
  - Each block contains pointer to next block
  - No compaction needed, and no external fragmentation
  - A file can continue to grow as long as free blocks are available
  - It is **inefficient** to support direct access of the file, only good for **sequential access**
  - Extra disk space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78% of the disk space is being used for pointers.
  - Reliability can be a problem; for instance, what happen if a pointer is lost or damaged.



Linked Allocation



Allocation Methods - FAT (文件分配表)

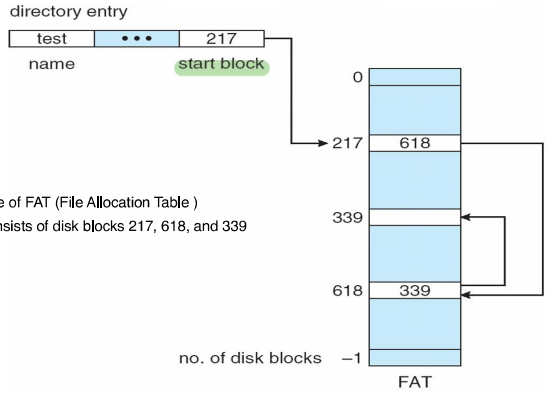
- FAT (File Allocation Table) – an important variation on linked allocation
  - This simple but efficient method of disk space allocation was used by the MS-DOS
  - A section of disk at the beginning of volume is set aside to contain a table called FAT.
  - The table has one entry for each disk block and is indexed by block number
  - The FAT is used in much the same way as a linked list.
  - The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This continues until it reaches the last block, which has a special end-of-file value as the table entry
  - An unused block is indicated by a table entry value 0. Allocating a new block to a file is a simple matter of finding the first 0-value table entry.
  - FAT can be cached. Random (direct) access time is improved, because the disk head can find the location of any block by reading the information in the FAT, instead of moving through blocks stored on the disk in the linked-allocation scheme.

4.2 (3 points) What are the advantges of using FAT over a linked allocation?

Answer:

Linked allocation performs the worst when accessing a block that is stored at the middle or near the end of a file as it needs to access all of the individual blocks of the file in a sequential manner to find the pointer to the target block starting from the first block. This may involve considerable overhead in moving disk arm and waiting for rotation. This is much easier in FAT by chasing the pointers stored within the FAT, with no or significantly less disk arm movement. In addition, most of the FAT can be cached in memory and therefore the pointers can be determined with just memory accesses instead of having to access the disk blocks.

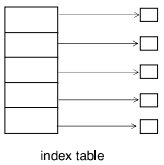
File-Allocation Table



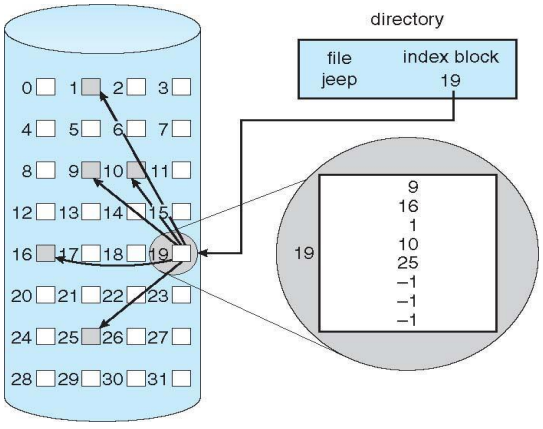
- An example of FAT (File Allocation Table)
- The file consists of disk blocks 217, 618, and 339

Allocation Methods - Indexed

- Indexed allocation – brings all the pointers together into one location, the index block
  - Each file has its own index block, which contains an array of pointers to its data blocks, or disk-block addresses
- Logical view



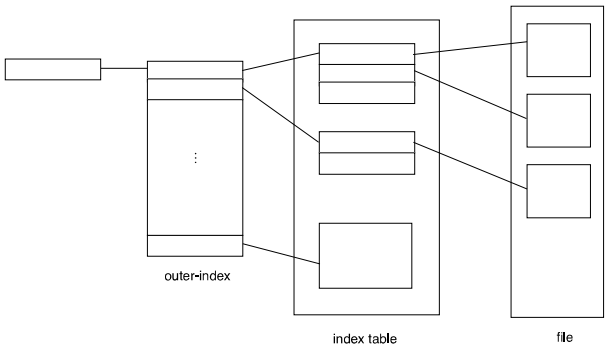
Example of Indexed Allocation



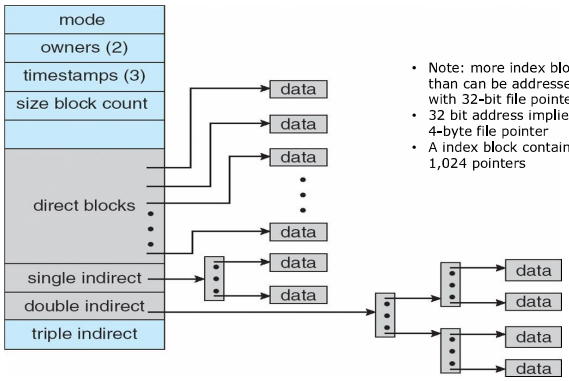
Indexed Allocation (索引分配)

- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space
- Indexed allocation suffers from some of the same performance problem as linked allocation. Specifically, index block(s) can be cached in memory, but data blocks may still be spread all over a volume (disk)
- Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead in the linked allocation
  - Suppose a file only has one or two blocks. Indexed allocation lose an entire index block, while linked allocation lose the space of only one pointer per block
- So far, an index block occupies one disk block. What happen if the file is too large such that one index block is too small to hold enough pointers
  - Linked scheme – to link several together index blocks
  - Multilevel scheme – a first-level index block points to a set of second-level index blocks, which in turn point to the file blocks. This could be continued to a third or fourth level, depending on the desired maximum file size. With a 4,096-byte block, we could store 1,024 four-byte pointers in one index block. Two levels of index allow 1,048,576 data blocks, and a file size up to 4GB.
  - Combined scheme – direct blocks for small files, and indirect blocks (single indirect, double indirect, and triple indirect blocks) for larger files, used in UNIX-based file systems.

Indexed Allocation – Multilevel Scheme



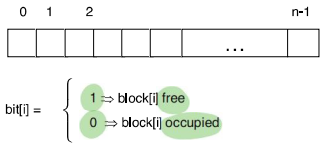
Combined Scheme: UNIX UFS (4K bytes per block, 32-bit addresses)



- Note: more index blocks than can be addressed with 32-bit file pointer
- 32 bit address implies a 4-byte file pointer
- A index block contains 1,024 pointers

Free-Space Management

- File system maintains free-space list to track free disk space
  - (Using term "block" for simplicity)
- **Bit vector** or **bit map** (n blocks)



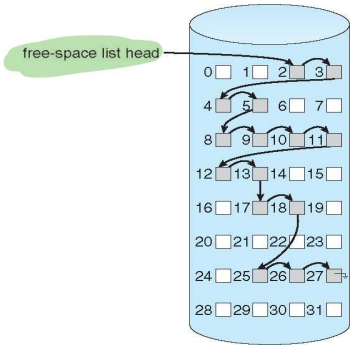
- The main advantage is its simplicity and efficiency in finding the first free blocks or n consecutive free blocks on the disk
- Bit vector is inefficient unless the entire vector can be kept in memory, but requires extra space

block size = 4KB = 2^{12} bytes
disk size = 2^{40} bytes (1 terabyte)
n = $2^{40}/2^{12} = 2^{28}$ bits (or 32MB)

Linked Free Space List on Disk

Linked-List - link together all the free disk blocks

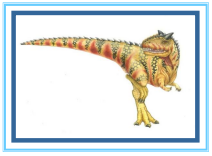
- Keeping a pointer to the first free block, in a special location on the disk, can be cached in memory
- The first block contains a pointer to the next free blocks, and so on
- Easily locate one free block, not easy to obtain contiguous blocks
- It is not efficient to traverse the entire list, since it must read each block, which requires substantial I/O time. Fortunately, this is not a frequent action



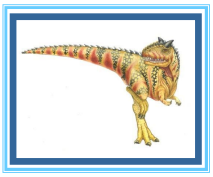
Free-Space Management (Cont.)

- **Grouping**
  - Modify linked-list to store addresses of n free blocks in first free block, The first n-1 of these blocks are free. The last block contains addresses of another n free blocks, and so on.
  - The addresses of a large number of free blocks can be found more quickly than linked-list
- **Counting** - Because several contiguous blocks may be allocated and freed simultaneously, particularly when contiguous-allocation algorithm or extents is used
  - By taking advantage of this, rather than keeping a list of n free disk addresses, we can keep address of first free block and count of following contiguous free blocks
  - Each entry in the free-space list consists of a disk address and a count
  - Although each entry requires more space than would a simple disk address, the overall list is shorter, as long as the count is generally greater than one
  - Note this method of tracking free space is similar to the extent method of allocating blocks.

End of Chapter 14



Chapter 17: Protection



Chapter 17: Protection

- Goals of Protection
- Principles of Protection
- Protection Rings
- Domain of Protection
- Access Matrix
- Implementation of Access Matrix

Objectives

- Discuss the goals and principles of protection in modern computer systems
- Explain how **protection domains** combined with **an access matrix** are used to specify the resources that a process may access
- Examine capability-based protection system

Goals of Protection

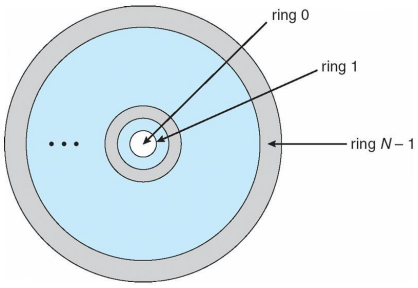
- In a protection model, computer system consists of a collection of **objects**, hardware or software
  - Hardware objects: CPU, memory segments, printers, disks, and tape
  - Software objects: files, programs, and semaphores
- Each object has a unique name and can be accessed through a well-defined set of operations
- **Protection problem** is to ensure that each object is accessed correctly and only by those processes allowed to do so
- Mechanisms are distinct from policies, in which mechanisms determine how something will be done, and policies decide what will be done.
  - The separation is important for **flexibility**, as policies are likely to change from place to place or from time to time.
  - The separation ensures that not every change in policy would require a change in the underlying mechanism.

Principles of Protection

- The guiding principle – **principle of least privilege**
  - Programs, users and systems should be given just enough privileges to perform their tasks - mitigate the attack
  - In file permissions, this principle dictates that a user have read access but not write or execute access to a file. The principle of least privilege would require that the OS provides a mechanism to only allow read access but not write or execute access
- Properly set permissions (i.e., the access rights to an object) can limit damage if entity has a bug or gets abused

Protection Rings

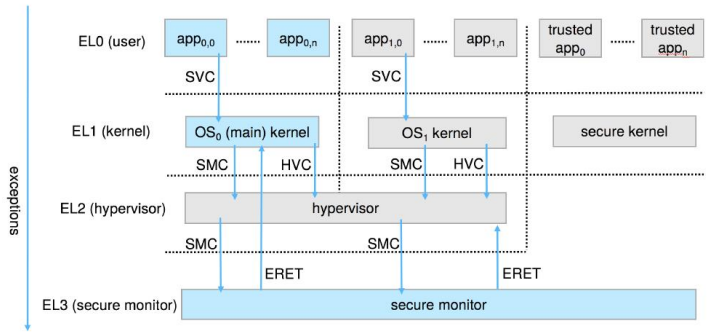
- **User mode** and **kernel mode** – privilege separation
- Hardware support required to support the notion of separate execution
- Let D_i and D_j be any two domain rings
- If $j < i \Rightarrow D_i \subset D_j$
- The innermost ring, ring 0, provides the full set of privileges



Protection Rings (Cont.)

- Components ordered by amount of privilege and protected from each other
  - For example, the kernel is in one ring and user applications in another
  - This privilege separation requires hardware support
  - "Gates" used to transfer between rings, for example the syscall Intel instruction, also traps and interrupts
- **Hypervisors** (Intel) is introduced (another ring) - virtual machine managers, which create and run virtual machines, and have more capabilities than the kernels of the guest operating systems
- ARM processors added **TrustZone** or **TZ** ring to protect crypto functions with access (more privileged than kernel)
  - This most privileged execution environment has exclusive access to hardware-backed cryptographic features, such as the NFC Secure Element and an on-chip cryptographic key, that make handling passwords and sensitive information more secure.

ARM CPU Architecture



Domain of Protection

- Protection rings separate functions into different domains and order them hierarchically
- **Domain** can be considered as a generalization of rings without a hierarchy
- A computer system can be treated as processes and objects
  - **Hardware objects** (such as CPU, memory, disk) and **software objects** (such as files, programs, semaphores)
- Process for example should only have access to objects it currently requires to complete its task – the need-to-know principle (policy)
- Implementation can be via process operating in a protection domain
  - Protection domain specifies the set of resources a process may access
  - Each domain specifies set of objects and types of operations may be invoked on each object

Domain of Protection (Cont.)

- The ability to execute an operation on an object is an **access right**
- A domain is a collection of access rights, each of which is an ordered pair <object-name, rights-set>
  - An example: if domain D has the access right <file F, {read,write}>, then a process executing in domain D can both read and write file F. It cannot, however, perform any other operation on that object.
- Domains may share access rights
- Associations between processes and domains can be static if the set of resources available to the process is fixed throughout the process's lifetime, or can be dynamic
- If the association is dynamic, a mechanism is available to allow domain switching, enabling the process to switch from one domain to another during different stage of execution

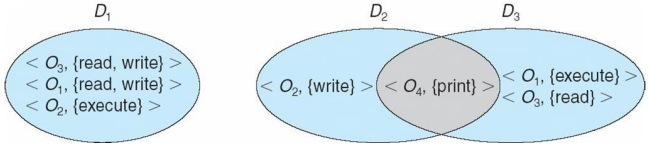
Domain of Protection (Cont.)

Domain can be realized in a variety of ways:

- Each user may be a domain - the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed
- Each process may be a domain - the set of objects that can be accessed depends on the identity of the process. Domain switching occurs when one process sends a message to another process and then waits for a response.
- Each procedure may be a domain - the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made

Domain Structure

- **Access-right** = <object-name, rights-set>
  where rights-set is a subset of all valid operations that can be performed on the object
- **Domain** = set of access-rights
- The access right <O4, {print}> shared by domains D2 and D3, thus, a process executing in either of these two domains can print object O4.



Access Matrix 访问矩阵

- View protection as a matrix (**access matrix**)
- **Rows** represent domains, and **columns** represent objects
- **Access(i,j)** consists of a set of access rights - the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Use of Access Matrix

- The **access matrix** scheme provides the mechanism for specifying a variety of policies - mechanism and policy separation
- The mechanism consists of implementing the access matrix and ensuring that the semantic properties hold.
  - To ensure that a process executing in domain D_i can access only those objects specified in row i.
- The policy decisions specify which rights should be included in the (i,j)th entry, and determine the domain in which each process executes
- If a process in Domain D_i tries to do "op" on object O_j, then "op" must be in the access matrix
- User who creates an object can define access column for that object
  - When a user creates a new object O_j, the column O_j is added to the access matrix with the appropriate initialization entries, as dictated by the creator. The user may decide to enter some rights in some entries in column j and other rights in other entries, as needed.

Use of Access Matrix (Cont.)

This can be expanded to dynamic protection

- Operations to add, delete access rights
- Special access rights:
  - owner of O_i - can add and remove any right in any entry in column
  - copy op from O_i to O_j (denoted by "*") - only within the column (that is, for the object)
  - control – D_i can modify D_j access rights – modify domain objects (a row)
  - transfer – switch from domain D_i to D_j
- Copy and Owner applicable to an object - change the entries in a column
- Control applicable to domain object - change the entries in a row
- New objects and new domains can be created dynamically and included in the access-matrix model
- In a dynamic protection system, we may sometimes need to revoke access rights to objects shared by different users – revocation of access right 撤销

Access Matrix of Figure A with Domains as Objects

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

"switch权限"通常指的是在计算机系统或网络中，用户或进程切换到不同权限级别或角色的能力。这种权限管理在多用户操作系统和网络环境中非常重要。以下是一些关键点：

1. **权限切换**：用户或进程可以在不同的权限级别之间切换，例如从普通用户权限切换到管理员权限。这种切换通常需要特定的认证或授权。

Access Matrix with *Copy* Rights

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Access Matrix With *Owner* Rights

所有者权限

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Modified Access Matrix of Figure B

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Implementation of Access Matrix

- In general, the access matrix is sparse; that is, most of the entries will be empty
- **Option 1 – Global Table**
  - Store ordered triples <domain, object, rights-set> in table
  - A requested operation M on object O_j within domain D_i -> search table for $< D_i, O_j, R_k >$ with $M \in R_k$
  - But the table could be large -> might not fit in main memory, requires additional I/O – virtual memory techniques are often used
  - Difficult to group objects - For example, if everyone can read a particular object, this object must have a separate entry in every domain.

Implementation of Access Matrix (Cont.)

- Each column = **Access-control list** for one object
  Defines who can perform what operation

  Domain 1 = Read, Write
  Domain 2 = Read
  Domain 3 = Read

- Each row = **Capability List** (like a key)
  For each domain, what operations allowed on what objects

  Object F1 – Read

  Object F4 – Read, Write, Execute

  Object F5 – Read, Write, Delete, Copy

Implementation of Access Matrix (Cont.)

- **Option 2 – Access lists for objects**
  - Each column implemented as an access list for one object
  - Resulting per-object list consists of ordered pairs <domain, rights-set> defining all domains with non-empty set of access rights for the object
  - Obviously, the empty entries can be discarded.
  - This can be easily extended to define **default** set of access rights -> If M ∈ default set, also allow access (for all domains)

Implementation of Access Matrix (Cont.)

- **Option 3 – Capability list for domains**
  - Instead of object-based, list is domain-based
  - A capability list for domain is a list of objects together with operations allowed on them
  - An object represented by its name or address, called a **capability**
  - To execute operation M on object O_j, a process requests operation M, specifying the capability (or pointer) for object O_j as a parameter
  - Possession of capability means access is allowed
- Capability list associated with a domain, but never directly accessible by a process executing in that domain
  - Rather, the capability list itself is a protected object, maintained by OS and accessed by users only indirectly
  - This avoids the possibility of capability list modification by users
  - If all capabilities are secure, the object they protect is also secure against unauthorized access

Implementation of Access Matrix (Cont.)

- **Option 4 – Lock-key**
  - Compromise between access lists and capability lists
  - Each object has list of unique bit patterns, called **locks**
  - Each domain as list of unique bit patterns called **keys**
  - Process in a domain can only access object if domain has key that matches one of the locks of the object
  - As with capability lists, the list of keys for a domain must be managed by the operating system on behalf of the domain.
  - Users are not allowed to examine or modify the list of keys (or locks) directly.

user 不能直接检查或修改列表

Comparison of Implementations

Choosing a technique for implementing an access matrix involves various trade-offs.

- Global table is simple, but large, lack of grouping of objects or domains
- Access lists correspond directly to the needs of users
  - An access list on an object is specified when a user creates the object
  - Determining set of access rights for each domain is difficult - every access to the object must be checked, requiring a search of the access list.
- Capability lists useful for localizing information for a given process
  - But revocation capabilities can be inefficient
- Lock-key can be effective and flexible depending on the length of the keys
  - Keys can be passed freely from domain to domain, easy revocation
- Most systems use combination of access lists and capabilities

End of Chapter 17