Homework 1

Operating System Principles and Implementation

PB22051080 王珏

2024 年 3 月 31 日

目录

1 操作系统的功能和提供的服务

1.1 操作系统的功能

操作系统的功能是管理计算机硬件的程序，为应用程序提供基础，并充当计算机用户和计算机硬件的中介。

计算机系统可分为四个部件：硬件、操作系统、应用程序和用户。硬件为系统提供基本的计算资源，而应用程序利用这些资源解决计算问题。操作系统控制硬件，并协调各个用户应用程序的硬件使用。

计算机系统分为硬件、软件和数据。操作系统提供正确手段以便使用这些资源，为其他程序执行有用工作提供一个方便的环境。

从用户视角来看，单个用户单独使用资源，其目的是优化用户进行工作。**对于这种情况**，操作系统设计的主要目的是用户使用方便，次要是性能，不在乎的是资源利用。在其他情况下，多个用户通过终端访问统一计算机，他们之间可以共享资源并交换信息。这种操作系统需要优化资源利用率，确保所有的 CPU 时间、内存和输入输出设备都能够得到有效利用，并确保没有用户使用超出限额以外的资源。**另一种情况**是用户坐在工作站前，该工作站与其他工作站和服务器相连，这类用户不仅可以使用专用资源，而且可以使用网络和服务器的共享资源。因此，这类操作系统需要具备坚固的使用方便性和资源利用率。

从系统视角来看，操作系统是与硬件紧密相连的程序，因此可将操作系统看作资源分配器。操作系统需要管理这些资源，面对许多指向冲突的资源请求，操作系统需要考虑如何为各个程序和用户分配资源，以便计算机系统能有效且公平地运行。此外，操作系统还强调控制各种输入输出设备和用户程序的需求。**它是一个控制程序**，管理用户程序的执行，以防止计算机资源的错误或不当使用。

1.2 操作系统提供的服务

- 用户功能：

  - 用户界面：
    * 命令行界面
    * 批处理界面
    * 图形用户界面
  - 程序执行：加载程序到内存并运行，同时处理程序的正常或异常结束。
  - 输入输出操作
  - 文件系统操作：读写文件和目录，创建、删除和搜索文件，列出文件信息，以及权限管理。
  - 通信
  - 错误检测和处理

- 系统运行的服务：
  - 资源分配：管理多用户或多作业同时运行时的资源分配。
  - 记账：记录用户使用资源的类型和数量，用于统计使用量和重新配置系统，以提高计算服务。
  - 保护安全：保护多用户或联网的计算机系统的信息安全，确保可以控制系统资源的所有访问。

2 存储层次结构

存储层次结构指的是根据速度、价格、大小、易失性的差异，将计算机系统中的不同类型存储设备组织起来的方式。它通常由多个级别组成，如寄存器、缓存、主存（RAM）、磁盘存储和三级存储。

2.1 缓存的概念和工作原理

缓存是位于 CPU 和主存之间的小型、高速存储组件。它的目的是存储频繁访问的数据和指令，从而减少访问内存的平均时间。

缓存的工作原理涉及利用局部性原理，即程序倾向于频繁访问内存的一小部分，并且附近的内存位置倾向于一起访问。缓存通过存储最近访问的内存内容来工作。当 CPU 请求数据或指令时，缓存控制器首先检查缓存中是否有信息（命中缓存）。如果是，则从缓存中检索数据，这比访问主存要快得多。如果信息不在缓存中（缓存未命中），则从主存中获取，并且也将其存储在缓存中以备将来使用。

3 系统调用

系统调用是操作系统提供的接口，允许用户级进程向内核请求服务。这些服务包括 I/O 操作、进程控制、文件管理、内存管理和通信等。

3.1 系统调用中的参数传递

系统调用通常涉及从用户空间到内核空间传递参数。

向操作系统传递参数有三种方法：通过寄存器、内存块或表、以及堆栈。

参数被整理并复制到内核可访问的指定内存区域，然后调用系统调用指令将控制传递给内核。

3.2 系统调用与库函数的区别

系统调用直接调用内核执行特权操作，如硬件访问或进程管理。库函数则是由库提供的在用户空间执行的例程。库函数可能在内部使用系统调用来执行某些任务，但它们无法直接访问系统资源。

3.3 系统调用与 API 的关系

API（应用程序接口）为方便应用程序规定了一组函数，包括每个函数的输入参数和返回值。API 隐藏了操作系统接口的细节，由运行时库管理。

API（应用程序编程接口）是由库或框架提供的一组函数和协议，用于与软件组件交互。系统调用是操作系统提供的更低级别的接口，而 API 是由库或框架提供的更高级别的接口。API 可能在内部使用系统调用来实现其功能。

4 双模式机制

双模式机制是指现代操作系统中用户模式和内核模式之间的硬件强制分离。

4.1 工作原理

在双模式机制中，CPU 在用户模式和内核模式之间运行。用户模式用于执行用户级进程，在这种模式下，访问某些特权指令和硬件资源是受限的。内核模式，也称为监督或特权模式，允许完全访问系统资源和特权指令。用户模式和内核模式之间的转换由硬件控制，通过特殊的 CPU 指令或标志实现。

4.2 采用双模式的原因

双模式机制通过防止用户级进程直接访问关键系统资源或执行特权指令来提供一定程度的保护和安全性。它有助于执行安全策略，并防止用户应用程序干扰系统稳定性。

5 内核结构

不同的内核结构在性能、灵活性和复杂性方面提供不同的折衷方案。

戳戳这里看详细内容！

5.1 宏内核

在宏内核中，所有操作系统服务都在单个地址空间中运行，并共享同一内存空间。这种架构提供了高性能，但可能更为复杂和不模块化。

内核层中的功能模块都是链接在一起的，并没有一定的层次关系。它们之间可直接通过方法调用进行交互。

假设现在有一个应用程序需要使用到内存分配的功能，那么首先该应用程序会调用到系统提供的内存分配的接口（系统调用），此时 CPU 就会切换到内核态，并执行内存分配相关的代码。内核里的内存管理代码按照特定的算法，分配一块内存。并将分配的内存块的首地址，返回给内存分配的接口函数。当内存分配的接口函数返回时，此时 CPU 又会切换回用户态，应用程序会得到返回的内存块首地址，并开始使用该内存。

我们可以注意到，宏内核的耦合度高，一旦其中一个模块出现问题，其他所有的模块都可能会受到影响。

5.2 分层内核

分层内核将操作系统分成不同的层，每一层提供特定的服务，并且仅与相邻的层进行通信。这种架构提供了模块化和灵活性，但由于层间通信的开销可能会受到影响。

5.3 微内核

微内核设计将内核保持最小化，仅包含必要的功能，如进程调度、内存管理和进程间通信。其他服务，如文件系统和设备驱动程序，则作为用户级服务器运行。

微内核中定义一种进程间通信的机制——消息。当应用程序请求相关服务时，会向微内核发送一条与此服务对应的消息，微内核再把这条消息发送给相关的服务进程（特殊的用户进程），接着服务进程会完成相关的服务。

对比宏内核中，微内核结构主要是多了接收和发送消息的这一过程，实际上也是系统调用，只是并不是直接调用内存管理的接口函数，因为微内核中内存管理功能模块已经不属于系统调用了。所以对比起宏内核，微内核结构的性能会差不少。

但微内核降低了耦合度，模块移除内核后后使得即使某一个模块出现问题，只要重启这个模块的进程即可，不会影响到其他模块，更加的稳定。并且微内核有相当好的伸缩性、扩展性，因为模块功能只是一个进程，可以随时增加或减少系统功能。

5.4 混合内核

混合内核结合了宏内核和微内核的特点，允许一些内核服务在内核空间运行，而其他服务在用户空间运行。这种架构旨在通过提供一些内核和用户级服务的组合来平衡性能和模块化。

6 机制与策略分离

6.1 设计原则解释

举个例子，定时器是一种保护 CPU 的机制，操作系统应该维持控制 CPU，防止用户程序陷入死循环或不调用系统服务，并且不将控制返还给操作系统。定时器可以设置在一定时间后中断计算机，从而确保可以产生中断。策略决定了定时器允许程序执行的时间，在面对不同的用户任务时，定时器可以灵活地修改设置的时间限制。然而，为某个用户应将定时器设置成多长时间是一个策略问题。

根据机制与策略分离的设计原则，系统的机制，如算法和数据结构，被设计成独立于特定策略或规则的。这种分离使得策略可以轻松地更改或替换，而不需要修改系统的机制。换句话说，机制决定了如何实现某个功能，而策略决定了什么功能被实现。这种分离提高了系统的灵活性和可维护性，并且使得系统更容易

适应不同的需求和环境变化。

7 创建子进程

一共创建了 $5+4+3+2+1=15$ 个子进程。

8 PID 问题

Listing 1: 关于 PID 的问题

```
int main(){
        pid_t pid,pid1

        /*fork a child process*/
        pid = fork();

        if (pid < 0){/*error occurred */
                fprint(stderr," Fork failed");
                return 1;
        }
        else if(pid == 0){/*child process*/
                 pid1 = getpid();
                 printf("child:pid = %d",pid);/*A*/
                 printf("child:pid1 = %d",pid1);/*B*/
        }
        else {/*parent process*/
                 pid1 = getpid();
                 printf("parent: pid = %d",pid);/*C*/
```

```
            printf( "" parent:pid1 = %d" ,pid1 );/*D*/
            wait(NULL);
        }
        return 0;
    }
```

A 位置是子进程的 `fork()` 返回值，即为 0；

B 位置的 PID 是子进程的 PID（603）；

而 C 和 D 位置的 PID 分别是子进程的 PID（603）和父进程的 PID（600）。

9 父进程和子进程对于同一块数据的修改的原理

分析以下程序，阐述 `LINE X` 和 `LINE Y` 的输出

父进程和子进程各自拥有独立的内存空间。当子进程修改了数组的值时，它只是修改了自己进程内存中的数组副本，而不会影响父进程中的数组。

Listing 2: 父进程和子进程一起修改同一块数据

```
int nums[SIZE] = {0,1,2,3,4};

int main() {
    int i;
    pid_t pid;

    pid = fork();

    if(pid == 0){
        for(i = 0;i < SIZE;i++){
            nums[i] *= -i;
            printf("chlid␣:%d",nums[i]);/*Line X*/
```

```
        }
    }
    else if(pid > 0){
        wait(NULL);
        for(i = 0;i < SIZE;i++){
            printf("parent : %d",nums[i]);/*Line Y*/
        }
    }
    return 0;
}
```

所以打印结果为:

Listing 3: 父进程和子进程一起修改同一块数据

CHILD : 0 CHILD : −1 CHILD : −4 CHILD : −9 CHILD : −16
PARENT : 0 PARENT : 1 PARENT : 2 PARENT : 3 PARENT : 4 PARENT : 5

10 execl() 的执行

Listing 4: execl() 的执行

```
int main(void) {
    printf("before execl ...\n");
    execl("/bin/ls", "/bin/ls", NULL);
    printf("after execl ...\n"); /*LINE: Y*/
    return 0;
}
```

在这段代码中，`execl()` 函数被调用来执行 `/bin/ls` 命令，这会将当前进程替换为 `ls` 进程。因此，如果 `execl()` 函数成功执行，那么之后的代码就不会被

执行，包括 `printf("after execl ...
n");`

原因是，`execl()` 函数会在执行成功后替换当前进程的内存映像，包括代码、数据和堆栈等。因此，替换后的进程将从 `/bin/ls` 的入口点开始执行，而不会返回到原始的 `main()` 函数中。也就是说，它会遗忘掉该系统调用之后所有执行代码而去执行替换后的代码。