

# CPT104

内容完全来自于LearningMall的PPT。仅供学习和交流使用。任何机构或个人不得将其用于商业用途或未经授权的传播。如需引用或分享，请注明出处并保留此声明。

 GITHUB

REPOSITORY

 VISIT NOTES AUTHOR'S PAGE

CLICK ME

CPT104 Operating Systems Concepts 操作系统概念

## ■ [Week 1 Lecture 0]

进程 Process。

### 课程信息

学分：5 credits

项目	占比	其他信息
Final Exam	80%	2 小时。
CW1	20%	Week 7
CW2	20%	Week 12

#### 模块负责人及联系方式

- 姓名: Gabriela Mogos
- 电子邮件: [Gabriela.Mogos@xjtlu.edu.cn](mailto:Gabriela.Mogos@xjtlu.edu.cn)
- 办公室电话: 88161515
- 办公室地点及办公时间: SD547; 周一 13:00-14:00, 周二 15:00-16:00, 或预约

[Module Handbook] 学生须达到不低于80%的出勤率。未能遵守此要求可能导致考试失败或被禁止参加补考。

Failure to observe this requirement may lead to failure or exclusion from resit examinations or retake examinations in the following year.

### 内容

1. 操作系统（Operating Systems concept）的概念
2. 进程（Process Concept）概念
3. 进程调度（Process Scheduling）
4. 进程操作（Operations on Processes）
5. 进程间通信（Inter-process Communication, IPC）

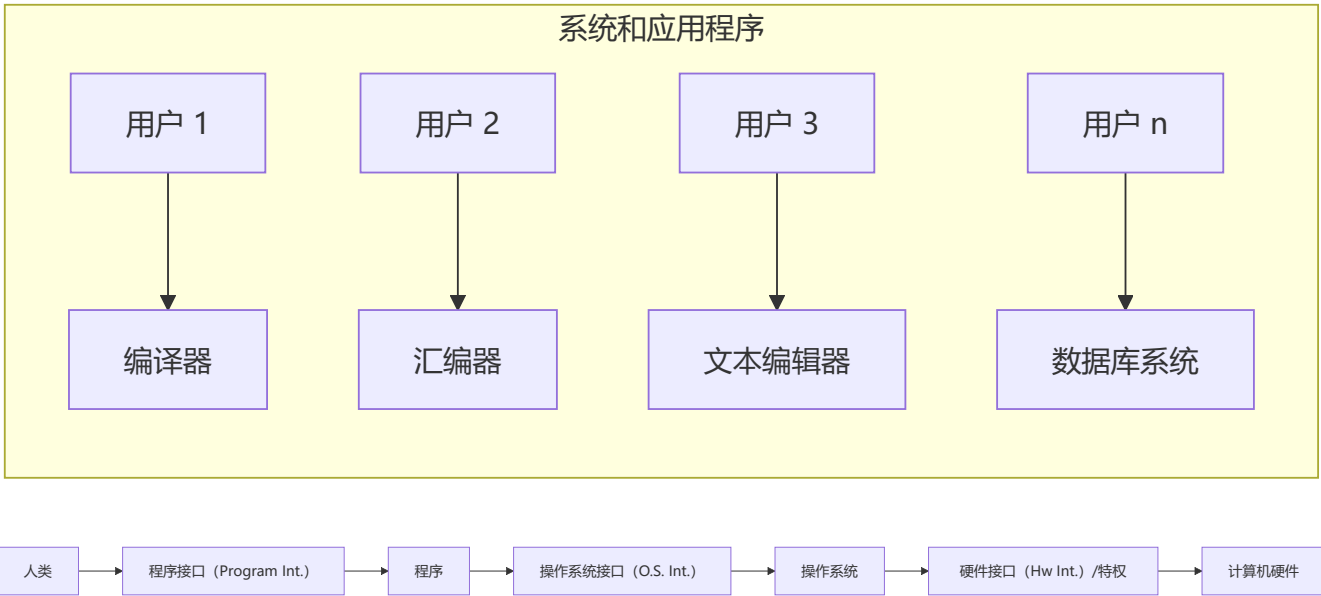
# [1] 操作系统的概念

操作系统可以理解为：**用户和硬件之间的接口**。或者说，一种“**架构 (architecture)**”环境。

- 允许方便的使用；隐藏繁琐的部分
- 允许高效的使用；**并行活动 (parallel activity)**，避免**浪费的周期**
- 提供信息保护
- 为每个用户提供一个**资源片段 (a slice of the resources)**
- 作为控制程序发挥作用

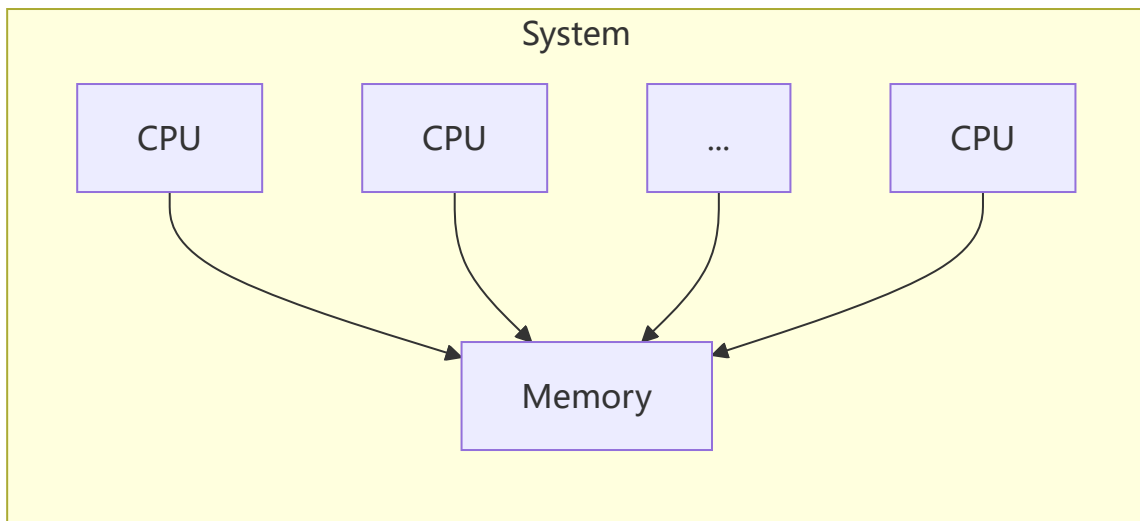
**并行活动**：指的是操作系统能够同时执行多个任务或进程，让多个程序或用户共享系统资源，从而提高系统的效率和响应速度。

**避免浪费的周期**：指的是操作系统通过合理分配和利用计算资源，避免资源闲置或浪费，确保每个资源（如 CPU、内存等）都被充分利用。

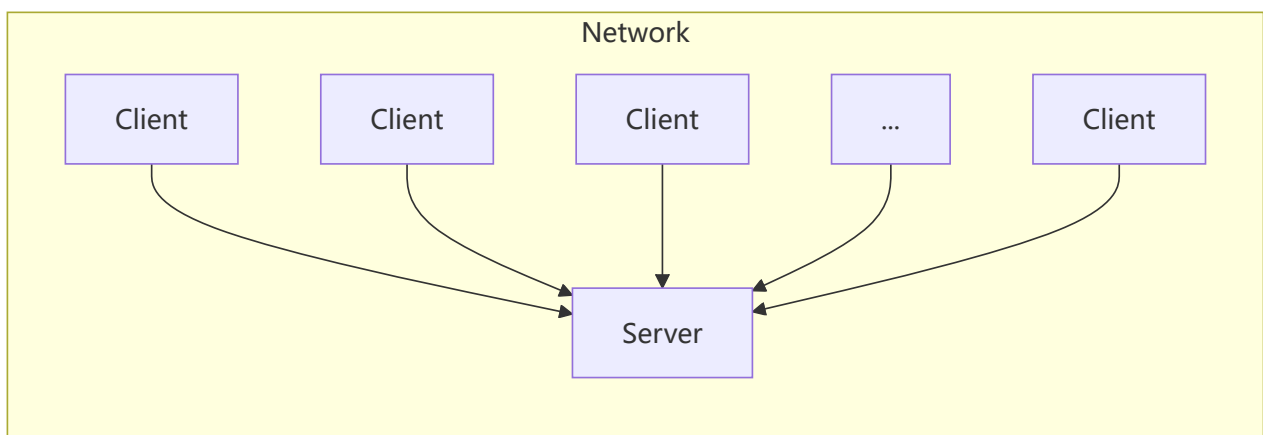


## 操作系统的特征：

1. **时间共享 (Time Sharing)** - 允许多个用户**同时**使用一台计算机系统。
2. **多处理 (Multiprocessing)** - 通过**共享内存**实现紧密联系的系统。通过将多个现成的处理器组合在一起，**提升计算速度**。



3. **分布式系统 (Distributed Systems)** - 通过**消息传递**实现松散联系的系统。其优点包括**资源共享**、**加快处理速度**、**提高可靠性**以及**增强通信能力**。



操作系统中最常见的操作：

1. 进程管理 (Process Management)
2. 内存管理 (Memory management)
3. 文件系统管理 (File System Management)
4. 输入/输出系统管理 (I/O System Management)
5. 保护与安全 (Protection and Security)

## [2] 进程概念

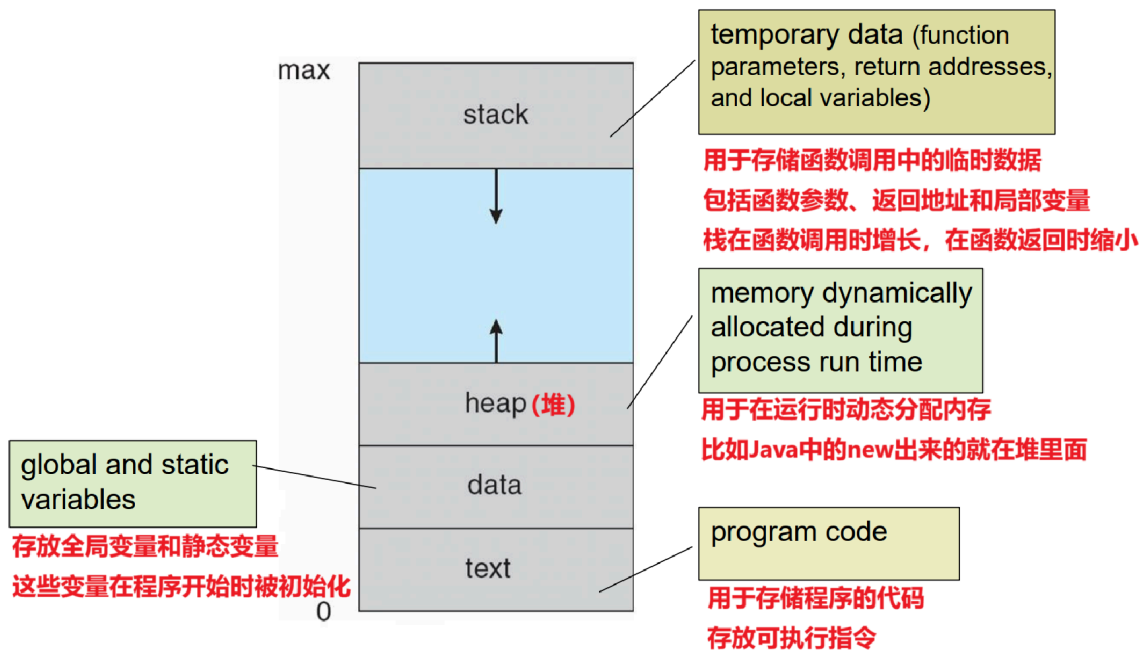
**进程 = 正在执行的程序** (Process = a program in execution)

进程的执行必须**按顺序进行**(In sequential fashion)。

进程被视为“**活动 (active)**”实体。

程序被视为“**被动 (Passive)**”实体 (存储在磁盘上的可执行文件 (executable file))

程序在**可执行文件加载到内存时**成为进程。 (Program becomes process when **executable file loaded into memory**)

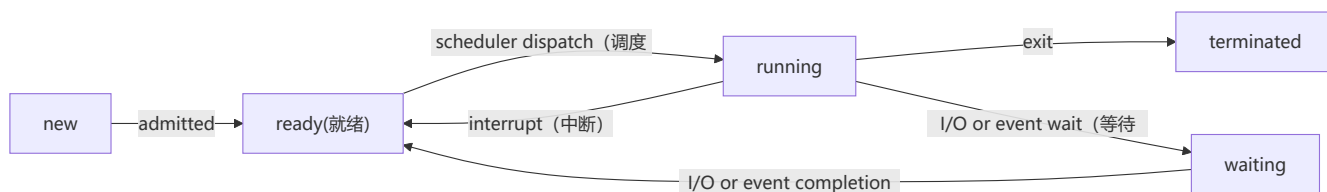


## 进程状态

### Process Status

一个进程执行时改变**状态(status)**，进程的状态在某种程度上由该进程**当前的活动**来定义。

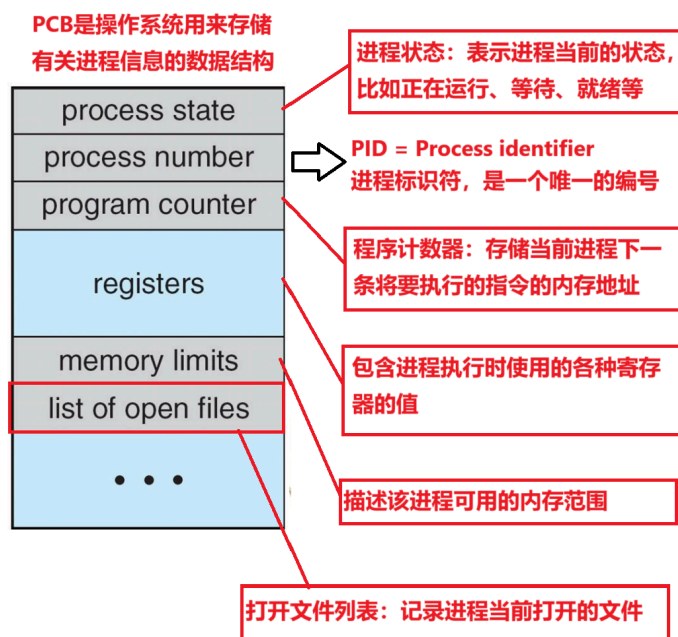
- **new (新建)**：进程正在被创建
- **running (运行)**：指令正在被执行
- **waiting (等待)**：进程正在等待某个事件发生
- **ready (就绪)**：进程正在等待被分配给处理器
- **terminated (终止)**：进程已完成执行



**进程控制块 (Process Control Block, PCB)** 是一种用于存储进程相关信息的数据结构。

- 每个进程都有其自身的PCB。
- PCB也被称为进程的**上下文 (context)**。
- 每个进程的PCB都存储在主内存中。
- 所有进程的PCB都存在于一个链表中。
- 在**多道程序(Multiprogramming)**设计环境中，PCB至关重要，因为它包含与同时运行的多个进程相关的信息。

**多道程序**：一种计算机操作系统技术，允许多个程序（程序1、程序2、程序3等）同时存在于内存中，并**轮流使用CPU**。虽然CPU同一时间只能执行一个程序，但操作系统通过快速切换，让用户感觉多个程序在同时运行。



## [3] 进程调度

进程的执行由**CPU执行**和**I/O等待**的**交替序列**组成。

**CPU执行**：进程在运行时，CPU会执行程序中的指令

**I/O等待**：当进程需要进行 **输入/输出** 操作（如从硬盘读取文件、向显示器输出数据）时，CPU会切换到等待状态，因为这些操作需要外部设备的响应。

**交替序列**：进程的运行不是一直占用CPU，而是时而工作（CPU执行），时而等待（I/O等待）

## 三种调度器和三种队列

调度器：Process Scheduler

队列：queue

**进程调度器(Process scheduler)** 从内存中选择已准备好执行的进程，并将CPU分配给其中一个进程。

进程的调度队列：

- **作业队列 (Job queue)**：系统中所有进程的集合。
- **就绪队列 (Ready queue)**：所有已准备好并且等待执行的进程的集合。
- **设备队列 (Device queues)**：等待某个I/O设备（如硬盘、打印机等）的进程的集合。

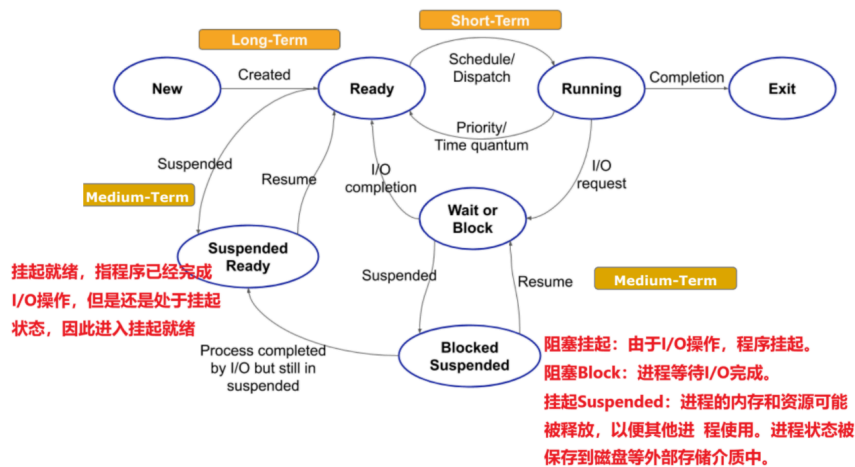
调度器种类

**长期调度器(Long-Term Scheduler)**：也被称为**作业调度器(Job Scheduler)**。控制系统的多道程序设计程度，管理处于**就绪 (Ready)** 状态的进程数量。

**短程调度器 (Short-Term Scheduler)**：也被称为**CPU调度器(CPU scheduler)**：负责从就绪队列中**选择**一个进程，并将其调度到**运行 (Running)** 状态 (CPU执行)。

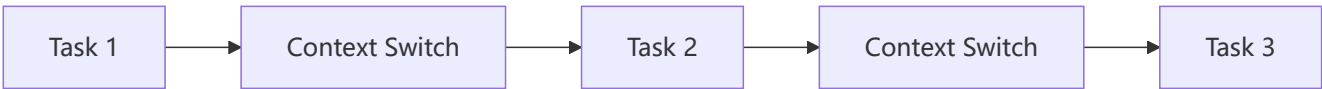
**中期调度器 (Medium-term scheduler)**：负责进程的内存与外存交换 (如将进程从主内存换出到磁盘，或从磁盘换入主内存)，对系统性能产生中期影响

上学期期末考有一题就是页面切换 (内存到硬盘之前切换)，算时间的，就是中期调度器做的事情。



## 上下文切换

当CPU切换到另一个进程时，系统必须保存旧进程的状态并通过**上下文切换 (CONTEXT SWITCH)** 加载新进程的保存状态。进程的上下文信息在**进程控制块 (PCB)** 中表示。



## [4] 进程操作

操作系统必须提供必要的进程操作：**进程创建 (Process creation)**、进程终止 (Process termination)。

### 进程创建

父进程创建子进程，子进程又创建其他进程，形成一棵**进程树**。

**资源共享**有三种选项

1. 父进程和子进程共享所有资源 (类似 `public`)
2. 子进程共享父进程资源的一个子集 (类似 `protected`)
3. 父进程和子进程不共享资源 (类似 `private`)

**执行选项**

1. 父进程和子进程**并发执行 (execute concurrently)**

## 2. 父进程等待直到子进程终止

# 进程终止

★ 进程终结的流程如下

### 1. 进程从所有队列中移除：

进程会从就绪队列、阻塞队列和其他相关队列中移除，表示它不再参与调度。

### 2. 释放进程占用的资源：

包括释放内存、关闭打开的文件、释放其他系统资源等。

### 3. 将终止状态返回给父进程(如果有)：

如果存在父进程，子进程会通过 `exit()` 系统调用将终止状态 (exit status) 传递给父进程。父进程可以通过 `wait()` 或 `waitpid()` 系统调用来获取这些信息。

### 4. 销毁或回收进程控制块 (PCB)：

当父进程获取了终止状态后，操作系统会销毁或回收子进程的 PCB。

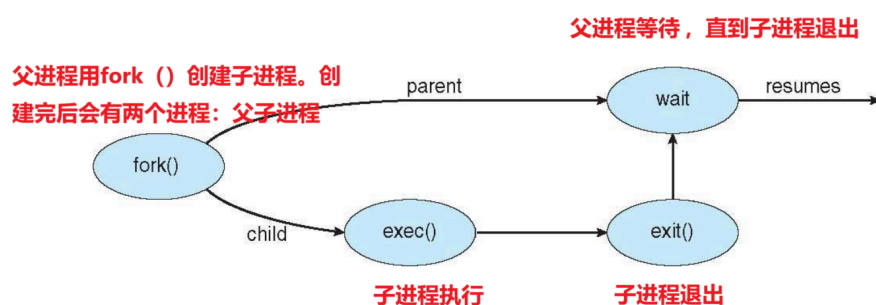
进程执行完毕最后一条语句，然后使用 `exit()` 系统调用请求操作系统删除它

子进程主动终结

父进程可以等待子进程终止执行

这是指父进程的行为。父进程可以使用诸如 `wait()` 或 `waitpid()` 的系统调用来等待其子进程的终止。在这个过程中，父进程可能会暂停执行，直到子进程完成并退出。这样做的好处是，父进程可以在子进程结束后获取子进程的退出状态信息，进行资源清理或执行其他逻辑。

- 子进程超出分配的资源



## [5] 进程间通信 IPC

进程间通信 Inter-Process Communication (IPC)

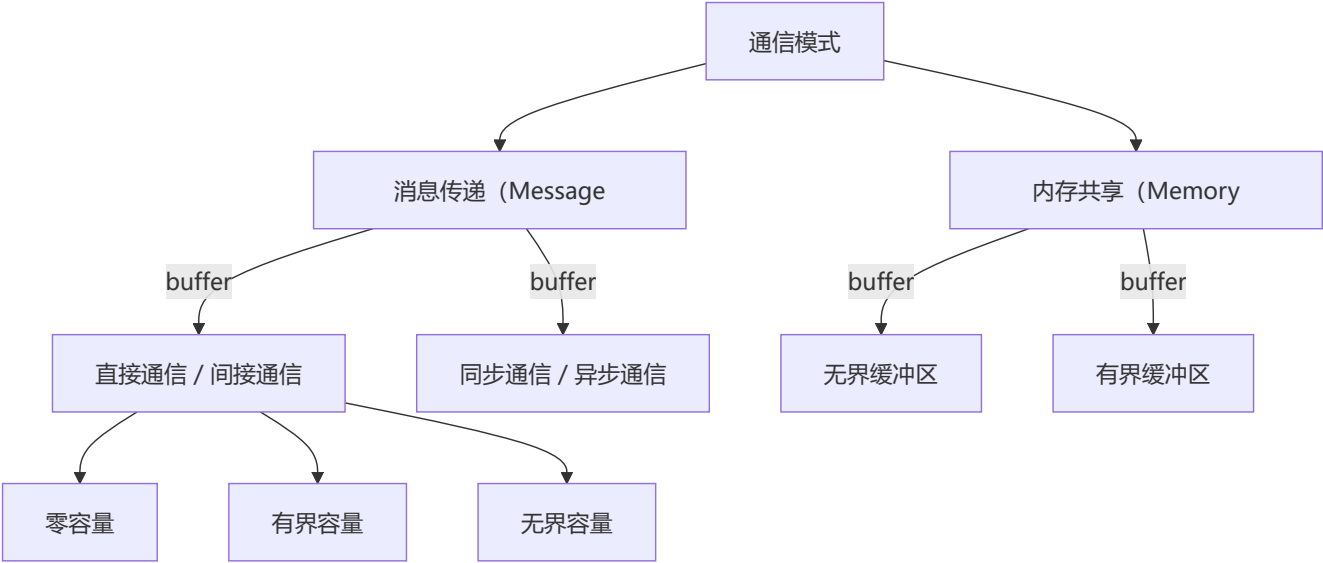
**独立进程 (Independent Process)** - 既不影响其他进程，也不受其他进程影响。

**协作进程 (Cooperating Process)** - 可以影响或被其他进程影响。

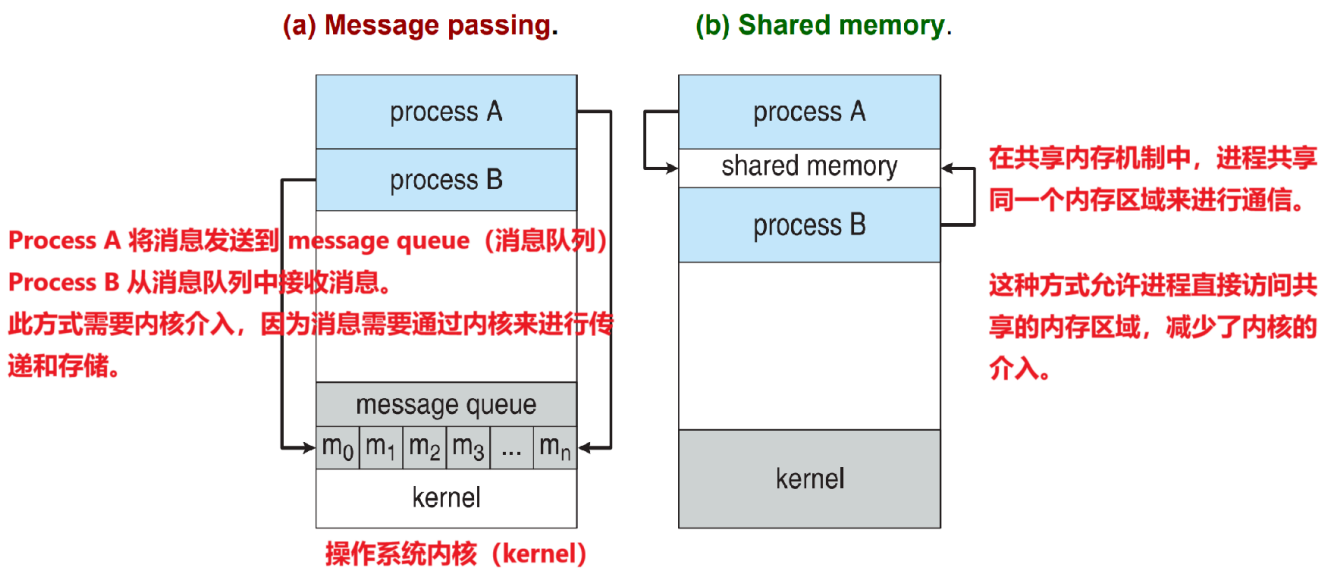
为什么需要协作进程？

- **信息共享 (Information Sharing)** - 例如，需要访问同一个文件的进程。
- **计算加速 (Computation speedup)** - 如果一个问题可以分解成同时解决的子任务，那么问题可以更快地解决。（比如java中的 `new Thread()`）
- **模块化 (Modularity)** - 将系统分解为协作的模块（例如，具有客户端-服务器架构的数据库）。
- **方便性 (Convenience)** - 即使是单个用户也可能在多任务处理，例如在不同的窗口中编辑、编译、打印和运行相同的代码。

## [6] IPC 的2种通信模式



有 **Message Passing (消息传递)** 和 **Shared Memory (共享内存)** 两种。





	优点	缺点
Message Passing (消息传递)	1. 提供了严格的进程间隔离， <b>安全性较高</b> 。2. 适用于分布式系统中的进程间通信。	1. 内核介入增加了通信开销，性能可能较低。2. 消息队列可能成为瓶颈，影响系统效率。
Shared Memory (共享内存)	1. 由于直接访问共享内存，通信速度快，效率高。2. 适用于需要频繁、大量数据交换的场景。	1. 共享内存需要进程间的同步机制，以防止数据竞争和一致性问题。2. 如果没有适当的访问控制和 <b>同步机制</b> ，安全性较低。（例如并发操作）

综合比较

- **消息传递**适合需要强隔离和简单通信的场景，尽管其性能可能稍差。
  - **共享内存**适合需要频繁、高速数据交换的场景，但需要有额外的同步和安全考虑。
- 

消息传递

[6] 通信模式中的第一种。Message-Passing

★ 消息传递是IPC的一种通信模式。其中消息 (Message) 可以是以下两种模式

1. **固定大小的消息(Fixed-size message)**: 消息的长度是预先定义的。
2. **可变大小的消息(Variable-size messages)**: 消息的长度可以根据实际需要动态调整。

消息的传递至少提供如下两种操作：

1. 发送
2. 接受

如果进程 P 和 Q 需要通信，必须在他们之间建立一个**通信链路**(communication link)。

逻辑上实现链路和 `send()` / `receive()` 操作的方法有几种：

- **直接通信 / 间接通信 [Direct / Indirect communication]**
  - **同步通信 / 异步通信 [Synchronous / Asynchronous communication]**
- 

直接通信 / 间接通信

[Direct / Indirect communication]

在消息传递系统中，通信进程交换的消息存储在一个**临时队列 (temporary queue)** 中。

三种缓冲机制 Buffering

1. **零容量 (Zero capacity)**：队列的最大长度为零，因此通信链路中不能有任何等待的消息。

发送方和接收方必须同时准备好，发送方发送消息后，必须等待接收方接收到消息后才能继续操作。

如果没有接收方准备好，发送方会一直阻塞，直到接收方接收消息。因此也叫做**没有缓冲空间**。
2. **有界容量 (Bounded capacity)**：队列的长度为有限值  $n$ ，因此最多可以容纳  $n$  条消息。
3. **无界容量 (Unbounded capacity)**：队列的长度没有限制，理论上可以存储无限多的消息。

## 直接通信

- 进程必须显式地命名彼此：
  - `send(P, message)` —— 向进程 `P` 发送消息。
  - `receive(Q, message)` —— 从进程 `Q` 接收消息。
- 直接通信的实现方式是进程使用特定的**进程标识符 (specific process identifier)** 进行通信，但在某些场景下，预先确定发送方是困难的。

## 间接通信

- 创建一个新的**邮箱(mailbox)** (端口 `port`) 。
- 通过**邮箱**发送和接收消息：
  - `send(A, message)` —— 向邮箱 `A` 发送消息。
  - `receive(A, message)` —— 从邮箱 `A` 接收消息。
- 销毁邮箱。

## 同步通信 / 异步通信

[Synchronous / Asynchronous communication]\*\*

消息传递可以是**阻塞 (blocking)** 或**非阻塞 (non-blocking)** 的。

### 阻塞 (BLOCKING) 被认为是同步的

- 阻塞发送 (Blocking send) : 发送方会被阻塞，直到消息被接收为止。
- 阻塞接收 (Blocking receive) : 接收方会被阻塞，直到有消息可用为止。

### 非阻塞 (NON-BLOCKING) 被认为是异步的。

- 非阻塞发送 (Non-blocking send) : 发送方发送消息后可以继续执行。
- 非阻塞接收 (Non-blocking receive) : 接收方会收到以下两种情况之一：
  - 一个有效的消息
  - 空消息 (Null message)

## 内存共享

[6] 通信模式中的第二种。Shared Memory

共享内存区域：由协作进程共享的一块内存区域。

信息交换：进程通过读取和写入共享区域中的所有数据来交换信息。

### 两种缓冲区类型：

- **无界缓冲区 (Unbounded-buffer)** : 对缓冲区的大小没有实际限制。
- **有界缓冲区 (Bounded-buffer)** : 假设缓冲区的大小是固定的。

# ▲ [Week 1 Tutorial]

## Q&A

来自LMO的quiz (不计分, Optional)

1. A Process Control Block(PCB) does not contain which of the following?

- A. Code B. Stack **C. Bootstrap program** D. Data

进程控制块 (PCB) 不包含以下哪一项?

答案: C; 进程控制块 (PCB) 是操作系统用于**管理进程的数据结构**, 通常包含以下信息:

进程状态 (Process State)、程序计数器 (Program Counter)、寄存器值 (Register Values)、栈 (Stack)、数据 (Data)、代码 (Code)、资源信息 (Resource Information)、调度信息 (Scheduling Information) 。

2. A process can be terminated due to \_\_\_\_\_

- A. normal exit** B. fatal error C. killed by another process D. all of the mentioned

一个进程可能由于以下哪种原因被终止?

答案: A;

3. What is a short-term scheduler?

- A. It selects which process has to be executed next and allocates CPU**

B. It selects which process has to be brought into the ready queue

C. None of the mentioned

D. It selects which process to remove from memory by swapping

什么是短期调度?

答案: A; 参考 [3] **进程调度 -> 三种调度器和三种队列**

4. **Messages** sent by a process \_\_\_\_\_

A. have to be a variable size

- B. can be fixed or variable sized**

C. have to be of a fixed size

D. None of the mentioned

答案: B; 首先这里的Message是消息的意思, 我们知道进程间通信模式IPC: **消息传递+内存共享**。因为这题提到的是**Message**, 和另一种**内存共享**无关。所以这题应该在**消息传递**中找。在 [6] IPC 的2种通信模式 -> **消息传递** 中写了, 消息既可以是固定长度, 也可以是可变长度。

5. Operating system's fundamental responsibility is to control the

A. Termination of Processes (终结进程)

- B. Execution of Processes(执行进程)**

C. Control of Processes(控制进程)

D. Access of Processes (访问进程)

操作系统的基本职责是控制\_\_\_\_\_

答案：C；操作系统的基本职责集中在 **程序（进程）执行** 上

6. What is a Process Control Block (PCB) ?

A. A secondary storage section

B. Process type variable

**C. Data Structure**

D. A Block in memory

答案：C；PCB是操作系统中用于描述和管理进程的数据结构；

7. Essential element for a process is

A. Time

B. Code

C. Program

**D. Process ID (PID)**

进程的基本要素是什么？

答案：C；

进程是操作系统中的一个执行实体，其基本要素包括：

- **进程 ID (Process ID, PID)**：唯一标识一个进程，是操作系统管理和调度进程的关键。
- **代码段 (Code)**：程序的执行指令。
- **数据段 (Data)**：程序运行时的变量和数据结构。
- **程序计数器 (Program Counter, PC)**：指向下一条要执行的指令。
- **资源**：分配的内存、文件句柄等。

8. If process is running currently executing, it is in running

A. Mode (模式)

B. Process (进程)

C. State (状态)

D. Program (程序)

如果进程当前正在执行，它处于运行\_\_。

答案：C；进程Process 在执行时被描述为处于 **运行状态 (Running State)**。这是进程生命周期中的一个状态，表示进程正在 CPU 上执行其指令。参考 [2] **进程概念 -> 进程状态**。

9. What will happen when a process terminates?

A. Its process control block is de-allocated (它的进程控制块被释放)

B. Its process control block is never de-allocated (它的进程控制块永远不会被释放)

C. It is removed from all, but the job queue (它从所有队列中移除，除了作业队列)

**D. It is removed from all queues** (它从所有队列中移除)

当一个进程终结时，会发生什么？

答案：D；简而言之，一个进程结束会发生：队列中移除进程，释放资源，`exit()` 返回结束状态给父进程，系统销毁该进程的PCB。

A项知识其中一个。BC都不对。答案是D。

具体参考 [4] **进程操作** -> 进程终止

10. The state of a process is defined by \_\_\_\_\_

- A. the activity just executed by the process (进程刚刚执行的活动)
- B. the activity to next be executed by the process (程接下来要执行的活动)
- C. the current activity of the process (进程当前的活动)
- D. the final activity of the process (进程的最终活动)

进程的状态是\_\_\_\_\_定义的。

答案：C；进程的状态 (Process State) 是由其 **当前的活动** 决定的。操作系统会根据进程当前在执行什么操作来定义其状态。参考 [2] **进程概念** -> **进程状态**

11. In indirect communication between processes P and Q \_\_\_\_\_

- A. none of the mentioned
- B. there is another machine between the two processes to help communication
- C. there is another process R to handle and pass on the messages between P and Q
- D. there is a mailbox to help communication between P and Q

在进程 P 和 Q 之间的间接通信中\_\_\_\_\_。

答案：D；参考 [6] **同步通信 / 异步通信** -> **异步通信**。

12. Information in the proceeding list is stored in a data structure called \_\_\_\_\_

- A. Process Access Block
- B. Data Blocks
- C. Process Control Block
- D. Process Flow Block

答案：C；进程控制块PCB。

13. Message passing system allows processes to \_\_\_\_\_

- A. communicate with one another by resorting to shared data
- B. name the recipient or sender of the message
- C. communicate with one another without resorting to shared data.
- D. share data

答案：C；消息传递（系统）允许进程：**在不依赖共享数据的情况下相互通信**。消息传递（系统）不依赖数据共享。

14. What is the ready state of a process?

- A. when process is scheduled to run after some execution (当进程被调度为在某些执行后运行)
- B. when process is using the CPU (当进程正在使用 CPU)

- C. when process is unable to run until some task has been completed (当进程无法运行直到某些任务完成)
- D. none of the mentioned (以上都不是)

进程的就绪状态是什么？

答案：A；B是**运行状态 (Running State)**。C是**阻塞状态 (Blocked State)**。

14. When a program is loaded into memory it is called as

- A. Procedure (过程)
- B. Register
- C. Table
- D. Process (进程)

当一个程序被加载到内存中时，它被称为

答案：A。当一个程序从存储设备（如硬盘）加载到内存中时，操作系统会为其创建一个 **进程 (Process)**。进程是程序的一个实例。

## Binary Number

The number 394 is said to be written in **base 10(以10为基数)**.

$$394 = 3 \times 100 + 9 \times 10 + 4 \times 1 = 3 \times 10^2 + 9 \times 10^1 + 4 \times 10^0,$$

在数字 394 中，数字 3 的**权重(weight)**是 100，数字 9 的权重是 10，数字 4 的权重是 1。

**注意：**数字的权重从**右到左**递增。

二进制数以 2 为基数，仅需要数字 0 和 1。

为了区分，下文用下标代表几进制。例如：10<sub>10</sub> 代表十进制的10，用10<sub>2</sub>代表二进制的2。

## 任意进制之间转换

如果想让 8 进制257<sub>8</sub>转二进制，那么可以：

- 1. 257<sub>8</sub> 转 10 进制
- 2. 10 进制再转 2 进制

### 任意进制转十进制

例如 8 进制257<sub>8</sub>转10进制

$$ans = 2 * 8^2 + 5 * 8^1 + 7 * 8^0 = 128 + 40 + 7 = 175$$

### 十进制转任意进制

例如 175<sub>10</sub> 转 2 进制

步骤	被除数 (十进制)	商	余数
1	175 ÷ 2	87	1

步骤	被除数（十进制）	商	余数
2	$87 \div 2$	43	1
3	$43 \div 2$	21	1
4	$21 \div 2$	10	1
5	$10 \div 2$	5	0
6	$5 \div 2$	2	1
7	$2 \div 2$	1	0
8	$1 \div 2$	0	1

不断进行触发求余后，把余数**倒过来写**。得到 $10101111_2$

小数形式二进制

将二进制数  $0.1101_2$  转换为 10 进制形式。

对于这种类型的二进制数，小数点后的第一位权重为  $2^{-1}$ ，第二位为  $2^{-2}$ ，依此类推。

二进制权重	$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$
权重值	0.5	0.25	0.125	0.0625
二进制数字	1	1	0	1

$$ans = 1 * 2^{-1} + 1 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} = 0.5 + 0.25 + 0.0625 = 0.8125_{10}$$

加减法

加法：

求  $101_2 + 110_2 + 1011_2$

1

0

1

1

1

0

+

1

0

1

1

1

2

2

2

加完之后进行进位，满2进1

1

1

1

1

0

减法：

求  $1101_2 - 111_2$

i

i

0

1

—

1

1

1

0

1

1

0

## Q&A

1. 使用  $n$  位**非小数(non-fractional)**的二进制数，能够表示的最大（十进制）数是多少？

答案：  $2^n - 1$ 。

例如，不考虑小数。使用2位二进制数，能表示 00, 01, 10, 11 四个数。其中最大的是  $11_2 = 3_{10}$ ，因此是  $2^2 - 1 = 3_{10}$ 。

2. 求  $11.1101_2$  的十进制

答案： 3.8125。

$$ans = (1 * 2^1 + 1 * 2^0) + (1 * 2^{-1} + 1 * 2^{-2} + 2^{-4}) = 3 + 0.5 + 0.25 + 0.0625 = 3.8125$$

## ■ [Week 2 Lecture 1]

Thread 线程

## 内容

### 线程 Thread

- 概述 / 什么是线程？
- 多核编程 (Multicore Programming)
- 多线程模型 (Multithreading Models)
- 线程库 (Thread Libraries)
- 隐式线程 (Implicit threading)
- 线程问题 / 设计多线程程序 (Threading issues / Designing multithreaded programs)

## [1] 什么是线程

### 基本概念

回顾：

1. **CPU（中央处理器）** 是计算机中的一块硬件，用于执行二进制代码。
2. **操作系统（OS）** 是一种软件，负责**调度(schedule)**程序的 CPU 使用权。
3. **进程** 是一个正在执行的程序。

**Process** is a program that is being executed.

4. **线程** 是进程的一部分。

Thread is part of a process.

具体而言：



1. **执行线程(Thread of execution) 是最小的可编程指令序列**，它能够被调度器独立管理并由 CPU 独立于父进程执行。

最小的可编程指令序列: smallest sequence of programmed instructions.

[补充]：

线程是由一系列指令组成的最小执行单元。线程可以直接被OS调度给CPU执行而不依赖于进程。

2. **线程** 是一种轻量级进程，可以被调度器独立管理。

[补充]：

线程比进程更加轻量。创建和管理线程的**开销**更小。

线程与进程的主要区别在于：线程共享进程的资源（如内存空间），而进程拥有独立的资源。

3. 线程 按顺序执行一系列指令（同一时间内只能执行一件事）。

[补充]：

线程是一个**单控制流**，它按指令顺序执行代码。**在单个线程中**，一次只能执行一个操作，**不会出现并发执行的现象**。

4. **线程** 是**进程内**的控制流。

A thread is a flow of control within a process.

[补充]：

一个进程可以包含多个线程。这些**线程共享进程的资源**（如内存、文件等），但各自拥有独立的执行路径。

5. 当一个进程中有多个线程运行时，进程会提供**共享内存 (the process provides common memory)**。

6. 应用程序的多个任务可以通过独立线程实现。

Multiple tasks with the application can be implemented by separate threads.

[补充]：

通过多线程，程序可以同时处理多个任务。每个线程可以独立执行不同的操作，从而提高程序的并发性和效率。例如，在一个程序中，一个线程可以处理用户输入，另一个线程可以执行计算任务。

操作系统的视角 (O.S. view)：线程是一个独立的指令流，可以由操作系统调度运行。

软件开发者的视角 (Software developer view)：线程可以看作是一个“过程”，它**独立**于主程序运行。




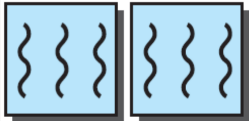
## 顺序程序和多线程程序

1. **顺序程序 (Sequential program)：**

程序中只有一个指令流。

2. **多线程程序 (Multi-threaded program)：**

程序中包含多个指令流（需要使用多个核心/CPU）。

	单程序	多程序
单线程		
多线程		

线代表：Instruction Trade

例如，Microsoft Word 编辑器是一个程序（存在磁盘中的静态代码）。当用户双击 Word，OS 就会创建 Word 进程。随后，

用户在 Word 编辑器中输入文本 -> 相关线程的变化：

- 打开 Word 编辑器中的文件（一个线程），
- 用户输入文本（一个线程），
- 文本自动格式化（另一个线程），
- 文本自动检查拼写错误（另一个线程），
- 文件自动保存到磁盘（另一个线程）。

## 线程的优点

### 1.响应性 (Responsiveness)

- 允许程序在部分被阻塞的时候，或者执行耗时操作的时候，继续运行。

### 2.资源共享与经济性 (Resource Sharing and Economy)

- 线程共享地址空间、文件、代码和数据。
- 避免资源消耗。
- 执行更快的上下文切换。

还记得上下文切换吗？意思是切换执行对象。

### 3.可扩展性 (Scalability)

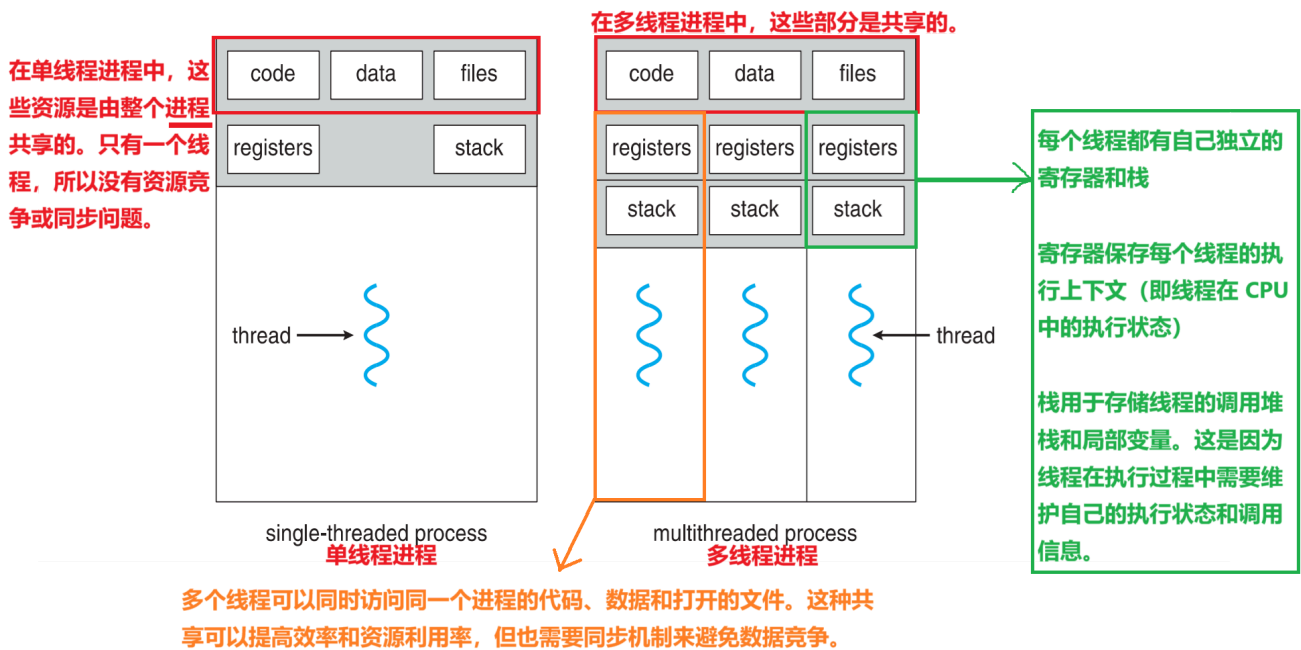
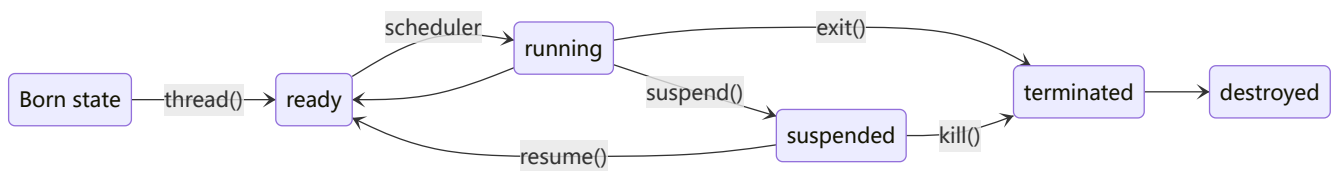
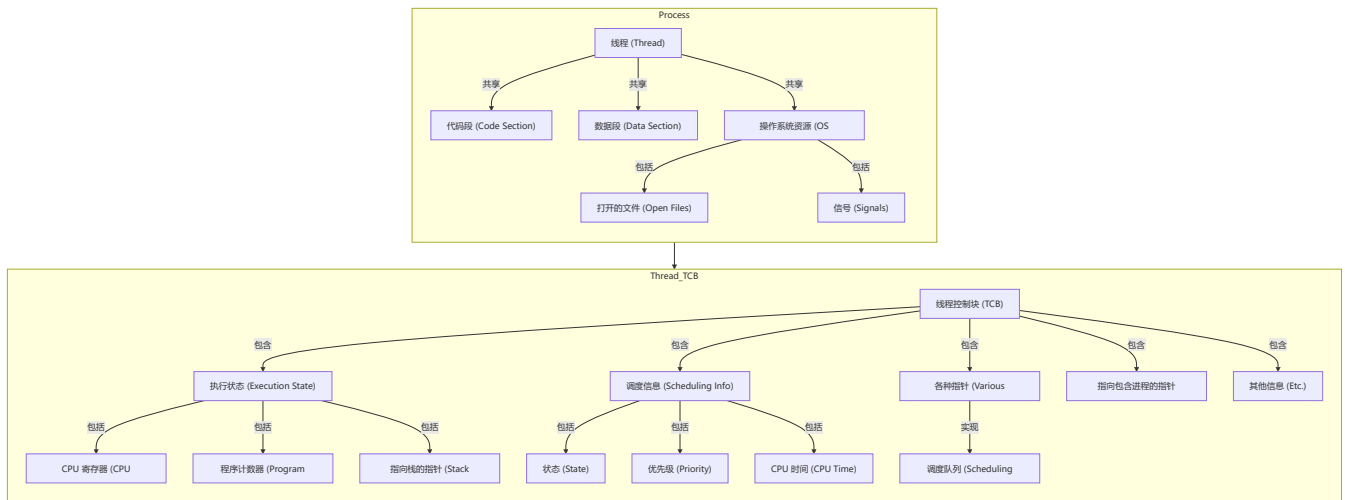
- 将代码、文件和数据放置在主内存中。
- 将线程分配到每个 CPU 上。
- 线程可以在不同的处理核心上并行运行 Running in parallel。

### 4.死锁避免 (Deadlock Avoidance)

## 线程控制块 TCB

Thread Control Block, TCB

线程控制块 (Thread Control Block, TCB) 存储了线程的相关信息。



## [2] 多核编程

### Multicore Programming

#### 单核系统上的 并发执行(Concurrent execution)

并发意味着多个任务能够启动、运行，并且操作系统在它们之间快速切换 ➡ 任务是**交错执行(interleaved)**的。并发创造了**并行执行**的假象。

- 在单核系统中，同一时间只能运行一个任务，但操作系统通过快速的上下文切换（context switching）让多个任务交替执行，从而实现 **任务的交错（interleaved）**。
- 例如，假设有三个任务A、B、C，操作系统可能会先运行A一小段时间，然后切换到B，再切换到C，再回到A，如此反复。
- **并发创造了一种并行执行的错觉。**
- 虽然单核系统同一时间只能执行一个任务，但对于用户来说，多个任务似乎在同时进行（例如，你在听音乐的同时写文档）。这种错觉是通过快速的上下文切换实现的。

### 多核系统上的并行执行(Parallelism)

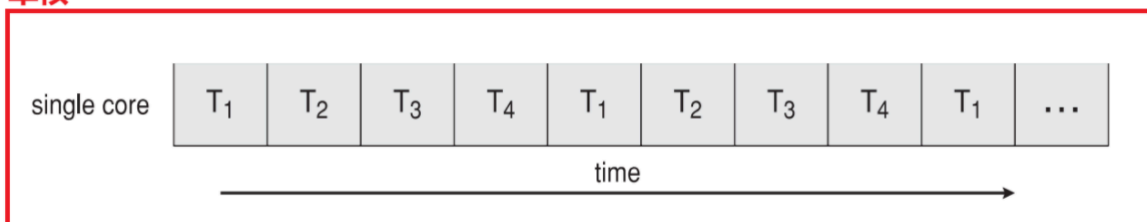
如果一个系统能够同时执行多个任务，则该系统是并行的。

- 如果一个系统能够同时执行多个任务，它就是并行的。
- 在多核系统中，每个核心可以独立执行任务，因此多个任务可以真正同时运行，而不是通过切换来实现。这就是 **真正的并行（parallelism）**。
- 例如，在一个四核系统中，四个任务可以分别在不同的核心上同时运行。

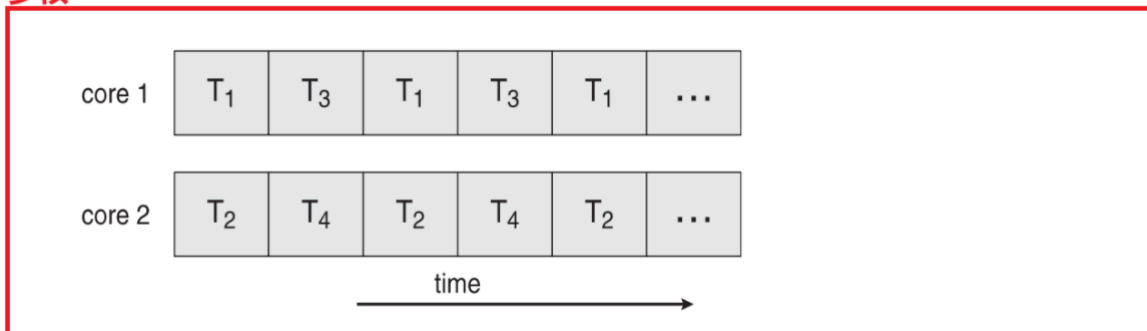
### 核心区别：

- **并发（Concurrency）**：任务**交替执行**，通过快速切换实现多个任务的“同时运行”。它更关注任务的调度和组织，适用于单核或多核系统。
- **并行（Parallelism）**：任务**真正同时执行**，通常需要多核或多处理器系统的硬件支持。

#### 单核



#### 多核



## [3] 多线程模型

# 用户模式和内核模式

## 用户模式 (User Mode) 和内核模式 (Kernel Mode)

处理器会根据当前运行的代码类型在两种模式之间切换：

- 应用程序在 **用户模式** 下运行。

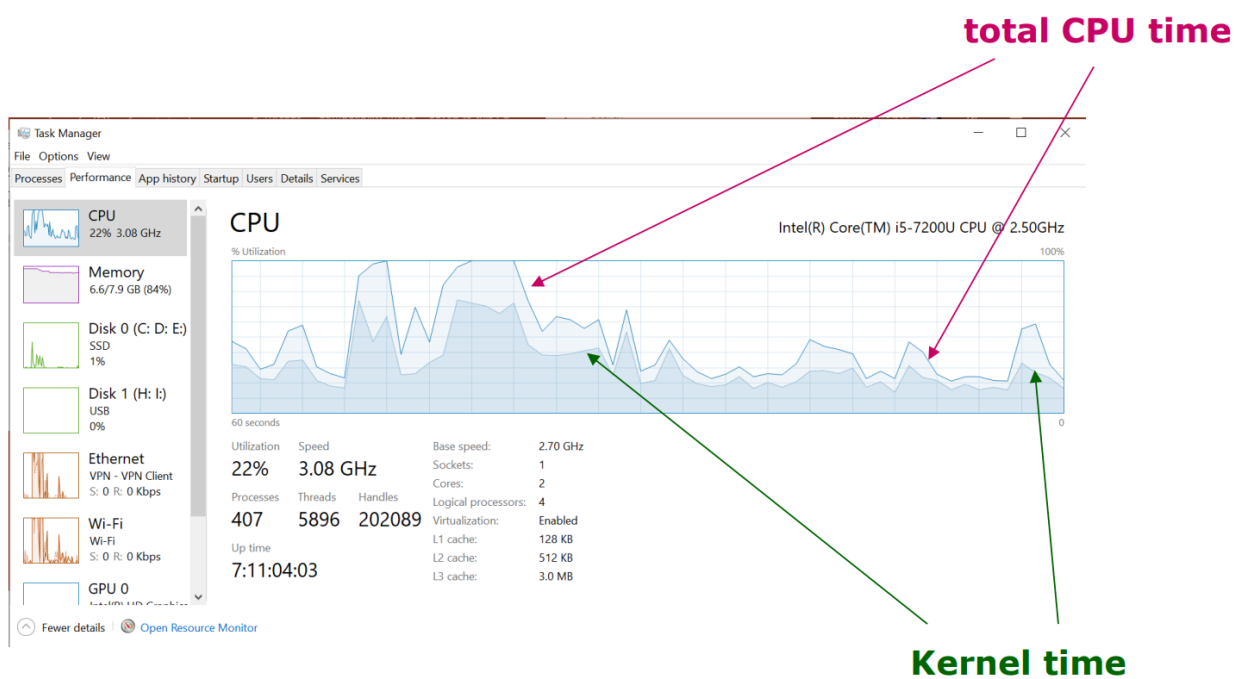
**核心操作系统组件在内核模式下运行。**

- 虽然许多驱动程序在内核模式下运行，但也有一些驱动程序可能在用户模式下运行。

使用任务管理器 **Ctrl** + **Alt** + **.** 查看 CPU 时间

其中最上面的一条线是总CPU时间，而下方深色部分是内核时间。那么两者之差（浅色部分）就是用户时间。

PPT 上说这是CPU时间，但实际上是利用率。



## 用户级线程和内核级线程

线程的类型有两种实现类别：

- 用户级线程 **User-Level Threads, ULT**
- 内核级线程 **Kernel-Level Threads, KLT**

### 用户级线程 ULT

**用户级线程**是由用户实现的执行单元，内核并不感知这些线程的存在。用户级线程的管理由**用户级线程库(ULT Library)**完成。

三个主要的线程库：

- POSIX Pthreads

- Windows 线程
- Java 线程
- 编程语言中的线程实现（如C#、Python中的线程）

内核级线程 KLT

内核级线程直接由操作系统处理，线程管理由内核完成。

几乎所有通用操作系统都采用内核级线程，包括：

- Windows
- Linux
- Mac OS X
- iOS
- Android

总结

- **用户级线程**：由用户空间实现，内核不可见，由 **用户级线程库** 管理。
- **内核级线程**：由操作系统内核直接管理，效率和调度更优。

用户级线程（ULT）与内核级线程（KLT）的区别

特性	用户级线程（ULT）	内核级线程（KLT）
创建和管理的速度	快	慢
实现方式	通过 <b>用户级的线程库</b> 实现。	由操作系统支持内核线程的创建。
通用性 Generality	用户级线程是通用的，可以运行在任何操作系统上。	内核级线程是 <b>特定于</b> 操作系统的。
多处理器利用	多线程应用程序无法利用多处理器（多核）的优势。	内核例程本身可以是多线程的，可以利用多处理器。
内核感知	内核并不感知用户线程的存在。	内核直接管理和调度内核线程。

[补充]

为什么 多线程应用程序无法利用多处理器（多核）的优势？

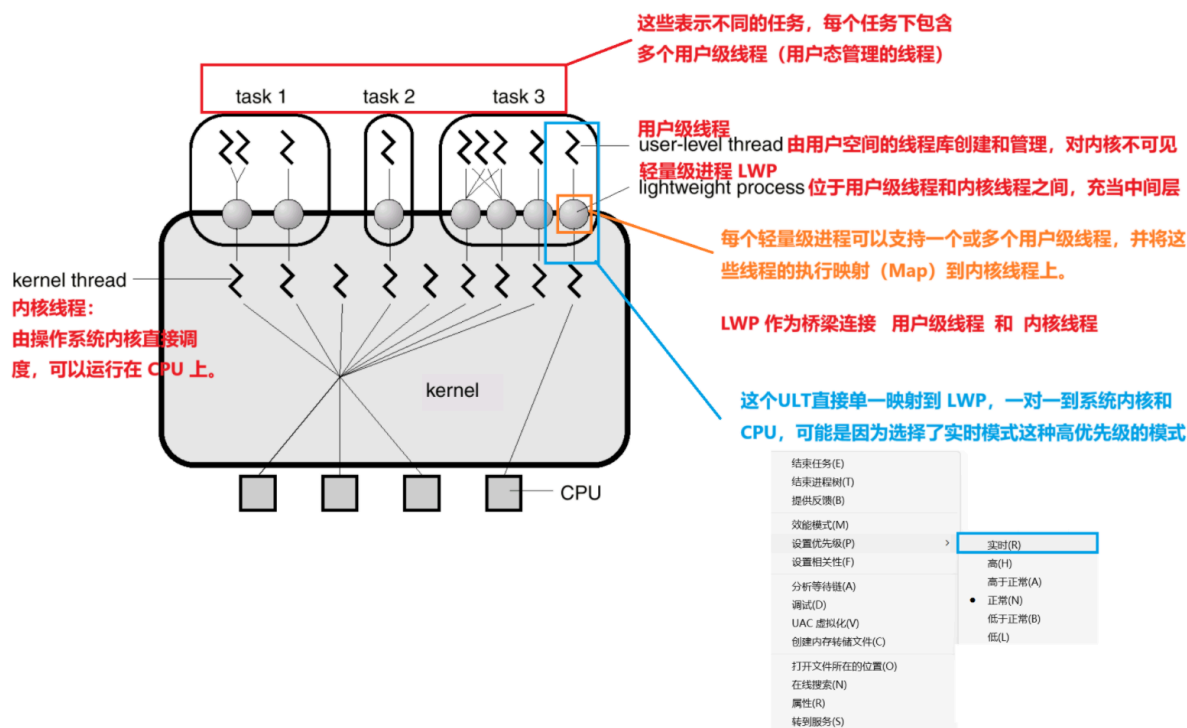
1. 用户级线程对内核不可见

2. 缺乏内核的调度支持

## ULT 和 KLT 的关系

ULT: User-Level Thread 用户级线程

KLT: Kernel-Level Thread 内核级线程



上图展示了如何将**用户级线程 ULT** 映射为**内核级线程 KLT**

一个程序（Task）中的多个线程，可以在多个CPU处理器上并行运行。

多线程模型分为三种类型：

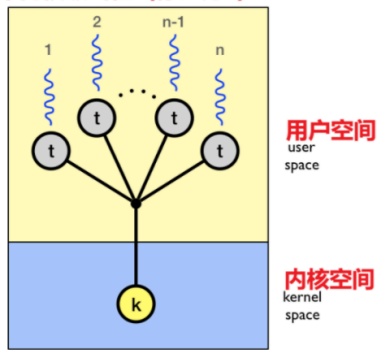
- 多对一（Many-to-One）多个 **ULT** 映射到单个 **KLT**
- 一对一（One-to-One）比如图上蓝色标注的 **ULT** 和 **KLT** 一对一
- 多对多（Many-to-Many）多个 **ULT** 映射到多个 **KLT**

### 多对一

多个 **ULT** 映射到单个 **KLT**

这种模型适用于：**不需要多处理器支持的简单应用**，但在 多核CPU的现代应用中可能存在局限。

多个用户级线程（标记为t）被映射到  
单个内核级线程（标记为k）

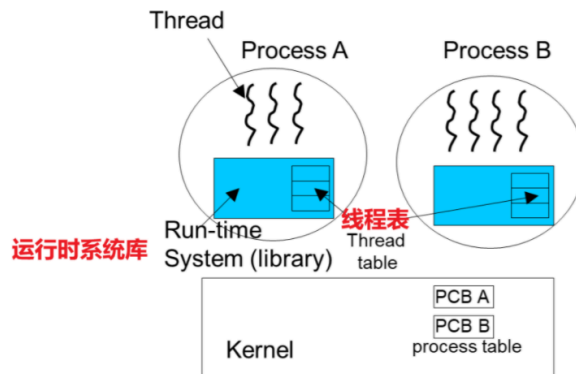


用户级线程运行在用户空间，由于是多对一模型，它们共享一个内核线程，因此在任意时刻只能有一个线程实际占用CPU。

内核空间中只有一个内核线程负责执行这些用户线程。

这种多对一模型的线程管理完全由用户级库负责

例如Solaris的Green Threads和GNU Portable Threads。



每个进程中都有一个运行时库，管理线程和线程表。该表维护所有用户线程的信息。

ULT 在进程内部由运行时系统进行切换，并不需要内核的直接参与。

内核只了解每个进程的进程控制块（PCB），而不了解进程内部的线程。

关于上下文切换：

上下文切换（Context Switch）、并发执行，通常发生在内核中。

单核系统中的上下文切换：

- **线程上下文切换**：在一个单核系统中，CPU同一时刻只能执行一个线程。当操作系统决定暂停当前线程并开始执行另一个线程时，就会进行线程上下文切换。
- **进程上下文切换**：类似地，操作系统也可以在进程间进行切换。在这种情况下，需要保存和恢复更多的状态信息（例如，内存映射、文件描述符等），因为不同进程有各自独立的资源。

多核系统中的上下文切换：

- 在多核系统中，多个CPU核心可以同时独立运行多个进程或线程。
- **进程/线程上下文切换**：即使在多核系统中，一个核心上的进程或线程也可能需要被另一个进程或线程替换，这个过程也是上下文切换。
- 为什么还会发生上下文切换？尽管有多个核心，任务可能有更高的优先级需要被立即执行，或者当前任务需要等待I/O操作完成等。

图中右边的这种模型使线程管理变得灵活，但不支持在多处理器系统中的并行执行，因为内核只看到**单一的执行上下文**。（看完上下文切换的解释，应该就能理解这句话了）

## 一对一

一对一模型

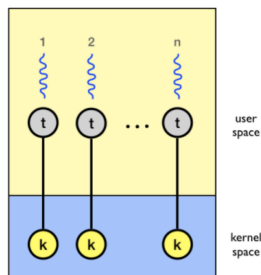
每个用户线程映射到一个内核线程

- 内核可以实现线程并管理线程、调度线程。
- 内核能够感知线程的存在。
- 提供了更多的并发性（Provides more concurrency）；当一个线程阻塞时，另一个线程可以继续运行。



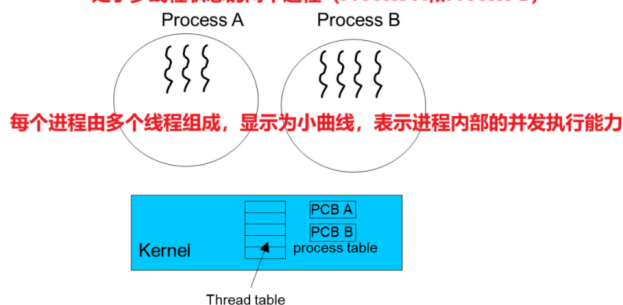
### Examples

- Windows NT/XP/2000
- Linux
- Solaris 9 and later **处于多线程状态的两个进程 (Process A和Process B)**



每个ULT通过一条连接线映射到一个KLT

在这种模型中，每个ULT都有一个对应的KLT来为其提供 底层支持和调度



每个进程由多个线程组成，显示为小曲线，表示进程内部的并发执行能力

内核包含一个线程表和一个进程表。

线程表：管理所有线程的信息，包括与进程相关的信息。

进程表：包含进程控制块（PCB），用于管理进程的状态和资源。

在这种模型中，内核能够感知每个线程，因为每个用户线程都与一个内核线程直接映射。这意味着内核可以更有效地进行线程调度和资源管理。

该模型的优点包括可以更好地支持并发，因为当一个线程阻塞时，另一个线程仍然可以运行。然而，代价是每个线程占用内核资源，可能导致较高的资源消耗。

## 多对多

允许将多个用户级线程映射到多个内核级线程

- 允许操作系统创建足够数量的 KLT
- KLT 的数量可以特定于某个特定应用程序或特定机器
- 用户可以创建任意数量的线程也就是ULT，并且相应的内核级线程可以在多处理器上**并行**运行

## 总结

并发和并行：

- 并发 (Concurrency)**：指的是系统能够在同一时间段内处理多个任务。并发意味着多个任务能够进展，但不一定同时执行。并发通常用于描述单核系统中不同任务通过时间切片机制交替执行的能力。
- 并行 (Parallelism)**：指的是在同一时刻真正同时执行多个任务。并行通常用于描述多核系统中，可以通过同时在多个核心上执行任务来提高计算效率。

## 单核系统中的并发

在单核系统中，程序通常通过**上下文切换**来实现并发。在这种环境下，多任务之间虽然不能并行执行，但可以通过快速切换任务来营造出同时进行的感覺。

## 多核系统中的并行

多核系统的设计使其可以真正同时执行多个任务

三种模型对比：

特性	一对一 (One-to-One)	多对一 (Many-to-One)	多对多 (Many-to-Many)
管理方式	内核管理	用户级线程库管理	用户级线程库和内核共同管理
线程映射机制	每个用户级线程对应一个内核线程	多个用户级线程映射到一个内核线程	多个用户级线程映射到多个内核线程
优点	1. 优化 <b>并行性</b>	1. 线程创建和切换速度快	1. 灵活性高，适应性强
	2. 线程阻塞不影响其他线程	2. 资源消耗小	2. 可以利用多核处理器
			3. 阻塞一个线程不影响其他线程
缺点	1. 创建和管理成本高	1. <b>不能利用多核处理器</b>	1. 实现复杂度高
	2. 上下文切换开销大（因为一个 ULT 对应一个 KLT，切换的时候麻烦。）	2. 阻塞一个线程会阻塞所有线程	
适用场景	高并发和 <b>并行</b> 处理应用	轻量级线程且不需要并行处理的系统	复杂应用和异构工作负载

## [4] 线程库

### Thread Libraries

线程可以通过线程API（线程库的函数）来 创建、使用和终止。

### API： Application Programming Interface 应用程序编程接口

**线程库**为程序员提供了一种**API**，用于创建和管理线程。

- 这个线程可以在用户空间或内核空间中实现。
- **用户级库 (User-level library)** （无需内核支持）。
- 库的所有代码和数据结构都**存在于用户空间中**。
  - 调用库中的函数会导致**用户空间中的本地函数调用**，而不是系统调用。
- **内核级库 (Kernel-level library)** （直接由操作系统支持）。
- 库的代码和数据结构存在于**内核空间中**。
  - 调用库API中的函数通常会导致对**内核的系统调用**。

三种主要的线程库：POSIX线程、Java线程、Win32线程。

### [5] 隐式线程

### Implicit threading

线程技术可以分为两类：**显式线程**和**隐式线程**。

- **显式(Explicit)线程**：程序员显式地创建和管理线程。
- **隐式(Implicit )线程**：**编译器**或**运行时库**负责创建和管理线程。

设计多线程程序的三种替代方案：

1. **线程池 Thread pool** - 在进程启动时创建一定数量的线程并将其放入池中，这些线程在池中等待任务。
2. **OpenMP** - 是一组适用于C、C++和Fortran程序的编译器指令，用于告诉编译器：在适当的地方自动生成**并行代码**。

一个简单的例子是，对于 `for` 循环，通过 OpenMP 优化，让不同的核心计算不同的区间段，加快 `for` 循环

```
1 // 使用 OpenMP 实现并行化
2 #pragma omp parallel for
3 for (i = 0; i < N; i++) {
4     b[i] = a[i] * a[i];
5 }
```

3. **Grand Central Dispatch (GCD)** - 是Apple的MacOS X和iOS操作系统上为C和C++提供的扩展，用于支持并行编程。

## [5] 线程问题 / 设计多线程程序

Threading issues / Designing multithreaded programs

1. `fork()` 和 `exec()` 系统调用
2. 信号处理
3. 线程取消

### `fork()` 和 `exec()` 系统调用

`fork()` 是 仅复制调用线程 还是所有线程？

`fork()` 会创建一个新进程，但在多线程程序中，它只会复制调用 `fork()` 的那个线程，而不会复制父进程中的所有线程。

因此，新创建的子进程将是单线程的。

**`exec()` 系统调用**

`exec()` 会用参数中指定的新程序替换当前进程的代码和数据，同时会终止当前进程的所有线程，并从新程序的入口点开始执行一个单线程的进程。

总结：

1. `fork()` 仅复制调用线程，子进程是单线程的。
2. `exec()` 替换当前程序及其所有线程，新程序以单线程运行。

# 信号处理

信号是一种软件中断，它发送给程序以指示某个重要事件已经发生。

信号在 **UNIX** 系统中被广泛使用。  
**Windows** 系统没有显式支持信号。

所有信号的处理过程遵循以下模式：

1. 信号的**生成**：特定事件的发生会生成一个信号。
2. 信号的**传递**：该信号被传递给某个进程。
3. 信号的**处理**：信号一旦被传递，就必须被处理。

例子 1：

非法内存访问和除以零操作。

如果正在运行的程序执行了上述操作中的任何一个，就会生成一个信号。  
信号会被传递给**引发该操作的同一进程**。

例子 2：

**UNIX/Linux 系统**

## Ctrl-C

按下此键会导致系统向正在运行的进程发送一个 **INT** 信号（**SIGINT**）。默认情况下，此信号会立即终止（中断）该进程。

## Ctrl-Z

按下此键会导致系统向正在运行的进程发送一个 **TSTP** 信号（**SIGTSTP**）。默认情况下，此信号会挂起该进程的执行。

## Ctrl-\

按下此键会导致系统向正在运行的进程发送一个 **QUIT** 信号（**SIGQUIT**）。默认情况下，此信号会立即终止（杀死）该进程。

进程通过运行特殊的**信号处理程序**来捕获并处理信号。信号处理程序执行完毕后，OS 会从进程原先被中断的地方**重新启动该进程**。

信号可以由以下两种可能的处理程序来处理：

### 1. 默认信号处理程序

默认操作是指：应用程序没有捕获并处理信号时会发生的行为。

对于许多信号类型，默认操作是**立即终止进程**。在某些情况下，默认操作是**简单地忽略信号**。

### 2. 用户自定义信号处理程序

操作是用户定义的，可以根据需要自定义信号的处理方式。

# 线程取消

线程取消通常有两种主要方式：

- 1. **异步取消 (Asynchronous Cancellation)**
  - **立即终止目标线程。**
  - 如果要取消的线程正在更新共享数据，可能会导致问题。
- 2. **延迟取消 (Deferred Cancellation)**
  - 允许目标线程定期**检查是否应该被取消。**

与目标线程相关的关键问题包括：

- 如果已经为被取消的目标线程分配了资源，这些资源会如何处理？

**[补充]**

线程被取消时，资源可能未被释放，导致泄漏。

**解决：**使用清理函数（如 `pthread_cleanup_push`）或在线程取消点手动释放资源，确保资源被正确回收。

- 如果目标线程在更新与其他线程共享的数据时被终止，会发生什么情况？

**[补充]**

共享数据可能处于不一致状态，导致程序错误或死锁。

**解决：**使用锁机制保护共享数据，确保线程在终止前完成更新并释放锁，或使用原子操作保证数据一致性。

## [Lab]

基于 Lab。部分非PPT会标注，作为补充。

在**学期的第一部分**，将快速介绍 C 语言的基础知识。在**学期的第二部分**，我们将学习 Linux 操作系统的基础知识以及 C 语言的开发。

规范：

```
1  #include <stdio.h>                // 总是使用这个库！
2  int main() {
3      printf("Hello world!");
4      return 0;                    // 总是 return 0;为结尾
5  }
```

## 转义字符

尝试输出

```
1 | Hello world!
2 | I'm learning "C in Linux" coding
```

```
1 | #include <stdio.h>
2 | int main(){
3 |     printf("Hello world!\n");
4 |     printf("I'm learning \"C in Linux coding\"\n");
5 |     return 0;
6 | }
```

C++中一些需要转义的字符

转义字符	含义
\n	换行符 (Newline)
\t	水平制表符 (Tab)
\b	退格符 (Backspace)
\r	回车符 (Carriage Return)
\f	换页符 (Form Feed)
\a	响铃符 (Alert/Bell)
\\	反斜杠 (Backslash)
\'	单引号 (Single Quote)
\"	双引号 (Double Quote)

## 循环

```
1 | Test case 1 :
2 |
3 | Input:
4 | 5
5 |
6 | Output:
7 | 0x5 = 0
8 | 1x5 = 5
9 | 2x5 = 10
10 | 3x5 = 15
11 | 4x5 = 20
12 | 5x5 = 25
13 | 6x5 = 30
14 | 7x5 = 35
15 | 8x5 = 40
16 | 9x5 = 45
17 | 10x5 = 50
```

```
1 #include <stdio.h>
2 int main(){
3     int number;
4     scanf("%d",&number);
5     for(int iterate = 0; iterate <=10 ;iterate++){
6         printf("%dx%d = %d\n",iterate,number,number*iterate);
7     }
8     return 0;
9 }
```

# 基本数据类型和输入输出

非PPT内从

## 1. 整型 (Integer Types)

数据类型	描述	大小 (通常)	范围 (有符号)	范围 (无符号)	输入格式	输出格式
short	短整型	2 字节	-32,768 到 32,767	0 到 65,535	scanf("%hd", &a);	printf("%hd", a);
int	整型	4 字节	-2,147,483,648 到 2,147,483,647	0 到 4,294,967,295	scanf("%d", &a);	printf("%d", a);
long	长整型	4 或 8 字节	取决于平台	取决于平台	scanf("%ld", &a);	printf("%ld", a);
long long	超长整型	8 字节	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	0 到 18,446,744,073,709,551,615	scanf("%lld", &a);	printf("%lld", a);
unsigned short	无符号短整型	2 字节	0 到 65,535		scanf("%hu", &a);	printf("%hu", a);
unsigned int	无符号整型	4 字节	0 到 4,294,967,295		scanf("%u", &a);	printf("%u", a);
unsigned long	无符号长整型	4 或 8 字节	0 到取决于平台		scanf("%lu", &a);	printf("%lu", a);
unsigned long long	无符号超长整型	8 字节	0 到 18,446,744,073,709,551,615		scanf("%llu", &a);	printf("%llu", a);

## 2. 浮点型 (Floating-Point Types)

数据类型	描述	大小 (通常)	精度 (十进制有效位数)	范围 (大致)	输入格式	输出格式
float	单精度浮点数	4 字节	6-7 位	±3.4e-38 到 ±3.4e38	scanf("%f", &a);	printf("%f", a);

数据类型	描述	大小 (通常)	精度 (十进制有效位数)	范围 (大致)	输入格式	输出格式
<code>double</code>	双精度浮点数	8 字节	15-16 位	$\pm 1.7\text{e-}308$ 到 $\pm 1.7\text{e}308$	<code>scanf("%lf", &amp;a);</code>	<code>printf("%lf", a);</code>
<code>long double</code>	扩展精度浮点数	10 或 16 字节	19-20 位	取决于平台	<code>scanf("%Lf", &amp;a);</code>	<code>printf("%Lf", a);</code>

### 3. 字符型 (Character Types)

数据类型	描述	大小 (通常)	范围 (有符号)	范围 (无符号)	输入格式	输出格式
<code>char</code>	字符型	1 字节	-128 到 127	0 到 255	<code>scanf("%c", &amp;a);</code>	<code>printf("%c", a);</code>
<code>unsigned char</code>	无符号字符型	1 字节	0 到 255		<code>scanf("%c", &amp;a);</code>	<code>printf("%c", a);</code>
<code>wchar_t</code>	宽字符型 (用于 Unicode 字符)	2 或 4 字节	取决于平台		使用 <code>wscanf</code> 和 <code>wprintf</code> 特殊处理	
<code>char16_t</code>	16 位 Unicode 字符型 (C++11 引入)	2 字节	0 到 65,535		使用 <code>wscanf</code> 和 <code>wprintf</code> 特殊处理	
<code>char32_t</code>	32 位 Unicode 字符型 (C++11 引入)	4 字节	0 到 4,294,967,295		使用 <code>wscanf</code> 和 <code>wprintf</code> 特殊处理	

### 4. 布尔型 (Boolean Type)

数据类型	描述	大小 (通常)	范围	输入格式	输出格式
<code>bool</code>	布尔型 (真或假)	1 字节	<code>true</code> 或 <code>false</code>	使用 <code>int</code> 类型间接输入	使用 <code>int</code> 类型间接输出



## 5. Void 类型

数据类型	描述	输入格式	输出格式
<code>void</code>	表示“无类型”，通常用于函数返回值或指针。	无输入	无输出