**Pthread\_condition\_t condition**

**Example:**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#define BUFFER\_SIZE 10

int buffer = 0; // 当前缓冲区中的商品数量

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

pthread\_cond\_t not\_full = PTHREAD\_COND\_INITIALIZER;

pthread\_cond\_t not\_empty = PTHREAD\_COND\_INITIALIZER;

// 生产者线程函数

void\* producer(void\* arg) {

while (1) {

pthread\_mutex\_lock(&mutex); // 加锁

// 如果缓冲区满，等待消费者消费

while (buffer == BUFFER\_SIZE) {

printf("Buffer is full. Producer is waiting...\n");

pthread\_cond\_wait(&not\_full, &mutex);

}

// 批量生产，将尽可能多的商品放入缓冲区

int produce\_amount = (BUFFER\_SIZE - buffer) / 2; // 每次最多放一半

if (produce\_amount == 0) produce\_amount = 1; // 保证至少生产1个

buffer += produce\_amount;

printf("Producer produced %d items. Buffer = %d\n", produce\_amount, buffer);

// 唤醒等待的消费者线程

pthread\_cond\_signal(&not\_empty);

pthread\_mutex\_unlock(&mutex); // 解锁

sleep(1); // 模拟生产时间

}

return NULL;

}

// 消费者线程函数

void\* consumer(void\* arg) {

while (1) {

pthread\_mutex\_lock(&mutex); // 加锁

// 如果缓冲区空，等待生产者生产

while (buffer == 0) {

printf("Buffer is empty. Consumer is waiting...\n");

pthread\_cond\_wait(&not\_empty, &mutex);

}

// 批量消费，将尽可能多的商品取出

int consume\_amount = buffer / 2; // 每次最多消费一半

if (consume\_amount == 0) consume\_amount = 1; // 保证至少消费1个

buffer -= consume\_amount;

printf("Consumer consumed %d items. Buffer = %d\n", consume\_amount, buffer);

// 唤醒等待的生产者线程

pthread\_cond\_signal(&not\_full);

pthread\_mutex\_unlock(&mutex); // 解锁

sleep(1); // 模拟消费时间

}

return NULL;

}

int main() {

pthread\_t prod, cons;

// 创建生产者和消费者线程

pthread\_create(&prod, NULL, producer, NULL);

pthread\_create(&cons, NULL, consumer, NULL);

// 等待线程完成（实际中不会结束，因为这是一个无限循环）

pthread\_join(prod, NULL);

pthread\_join(cons, NULL);

// 销毁互斥锁和条件变量

pthread\_mutex\_destroy(&mutex);

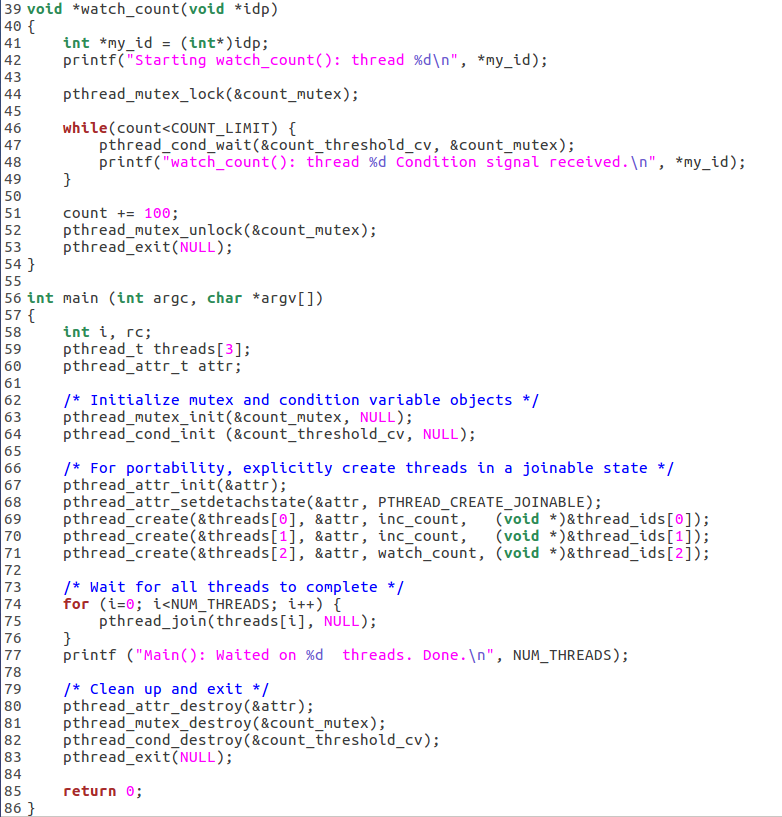
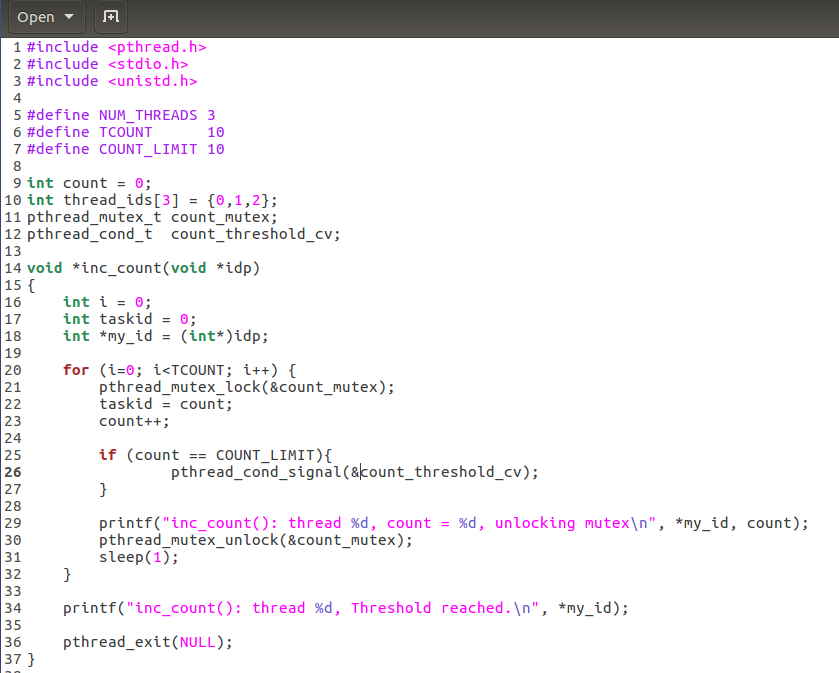
pthread\_cond\_destroy(&not\_full);

pthread\_cond\_destroy(&not\_empty);

return 0;

}

Another example:

****

**文件描述符**（File Descriptor）是一个**整数**，用于**表示**进程正在使用的一个文件或I/O资源（如套接字、终端等）。在 Unix 和类 Unix 系统中，每个打开的文件或设备都被分配了一个文件描述符。文件描述符是内核用来追踪文件访问的标识符。

* **标准输入（STDIN\_FILENO = 0）**、**标准输出（STDOUT\_FILENO = 1）**、和 **标准错误（STDERR\_FILENO = 2）** 是默认的文件描述符。标准输入通常是键盘，这意味着键盘的输入会被映射为文件描述符 0。

**I/O 操作**（Input/Output Operation）是指对文件或设备的读写操作。例如，通过文件描述符，你可以使用函数来对相应文件进行读取（如 read()）、写入（如 write()）等操作。

通过文件描述符读取相应文件的例子：  
#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

int main() {

char buffer[100];

int bytesRead;

// 将标准输入设置为非阻塞模式

int flags = fcntl(STDIN\_FILENO, F\_GETFL, 0);

fcntl(STDIN\_FILENO, F\_SETFL, flags | O\_NONBLOCK);

printf("请输入内容（非阻塞模式）: ");

// 尝试读取输入

bytesRead = read(STDIN\_FILENO, buffer, sizeof(buffer) - 1);

if (bytesRead > 0) {

buffer[bytesRead] = '\0'; // 确保字符串以 '\0' 结尾

printf("\n读取到的数据: %s\n", buffer);

} else {

printf("\n没有输入数据或者输入被阻塞\n");

}

return 0;

}

作业中，fcntl() 被用来修改文件描述符的状态，以便对键盘输入实现非阻塞读取。这样，程序就可以继续执行，而不必在读取输入时一直等待：  
oldf = fcntl(STDIN\_FILENO, F\_GETFL, 0);

fcntl(STDIN\_FILENO, F\_SETFL, oldf | O\_NONBLOCK);

**终端**（Terminal）是指用户与系统交互的输入输出设备，可以是物理上的终端（如键盘和显示器）或者虚拟终端（如你在操作系统上打开的终端窗口）。它提供了一个用户与系统之间直接交互的界面，接受用户的输入并显示输出。

具体到“终端输入属性”这个概念，它指的是控制终端设备输入行为的一些配置选项，例如：

* **是否需要按回车键才能提交输入**（规范模式 ICANON）。
* **输入字符是否会回显到屏幕上**（ECHO）。

通过设置这些属性，程序可以控制用户在终端上输入时的行为方式，从而实现不同的输入处理方式。

**设置终端输入属性** (tcsetattr()): 这是针对终端设备本身的设置，影响用户如何输入，例如：

* + 是否需要按回车才能提交输入（规范模式 vs. 非规范模式）。
  + 输入字符是否会显示在屏幕上（回显模式）。
  + 换句话说，这是控制用户在终端上输入时的交互方式以及输入数据的表现形式。

**设置文件描述符的标志属性** (fcntl()): 这是针对操作系统如何处理输入数据的行为。例如：

* + 设置是否以**非阻塞模式**读取输入，如果没有数据可以读取，程序是否应该等待（阻塞）或者立即返回。
  + 这种设置控制了操作系统和文件描述符（在这里是标准输入）之间的交互方式。
  + 主要是控制程序在与文件或设备交互时，如何处理数据的读取与写入。

所以，前者控制了**你（用户）如何输入、输入的数据会如何被处理和呈现**；后者则是**控制程序和操作系统如何处理这些输入数据**。

在文件描述符的标志属性中，除了 O\_NONBLOCK，还有许多其他的标志，以下是一些常见的文件描述符标志属性：

O\_RDONLY（只读模式）：

表示文件以只读模式打开。

适用于文件打开函数（如 open()），可以确保文件只能读取，不能写入。

O\_WRONLY（只写模式）：

表示文件以只写模式打开。

适用于打开文件以进行数据写入而不需要读取的场景。

O\_RDWR（读写模式）：

表示文件以读写模式打开。

文件既可以读取，也可以写入。

O\_APPEND（追加模式）：

每次写入时数据都会被追加到文件末尾。

适用于日志文件等需要在文件末尾持续写入内容的场景。

O\_CREAT（创建文件）：

如果文件不存在，则创建一个新文件。

通常与 open() 一起使用，用于确保文件存在。

O\_TRUNC（截断文件）：

打开文件时会将其长度截断为 0（删除文件中的所有内容）。

适用于重新创建或清空文件的情况。

O\_EXCL（排他性创建）：

通常与 O\_CREAT 结合使用。如果文件已存在，调用会失败。

适用于避免覆盖已有文件的场景。

O\_SYNC（同步写入）：

表示每次写入操作都会等到物理 I/O 操作完成后再返回，确保写入操作是同步的。

适用于需要保证数据可靠写入磁盘的应用程序。

O\_NOCTTY（不分配控制终端）：

打开文件时，如果是一个终端设备，不要将该设备分配为该进程的控制终端。

适用于后台服务程序（守护进程）打开终端设备时避免其作为控制终端。

这些标志都可以通过按位或操作组合使用，以便在打开或修改文件描述符属性时实现特定的行为。例如，当使用 fcntl() 设置标志时，可以组合 O\_NONBLOCK 和 O\_APPEND，使得读取操作是非阻塞的，同时写入时会追加到文件末尾。标志的组合和使用，给了开发者更多控制文件描述符的方式。

### 终端控制序列说明

**清屏 (**\033[2J**)**

* 1. 代码：\033[2J
  2. 功能：清除终端屏幕上的所有内容，使屏幕变成空白。
  3. 说明：\033 表示转义字符（ESC），[2J 是清屏命令。
  4. 使用：例如在 printf 中，printf("\033[2J"); 会清除终端上的所有内容。

**将光标移到屏幕左上角 (**\033[H**)**

* 1. 代码：\033[H
  2. 功能：将光标移动到终端的左上角（位置 1,1）。
  3. 说明：光标会回到初始位置，但不会清除屏幕内容。
  4. 使用：例如 printf("\033[H"); 会将光标移到左上角。

**将光标移动到指定位置 (**\033[r;cH**)**

* 1. 代码：\033[r;cH
  2. 功能：将光标移动到指定的行 r 和列 c。
  3. 说明：行和列的索引从 1 开始。比如 \033[5;10H 表示将光标移到第 5 行、第 10 列。
  4. 使用：printf("\033[5;10H"); 将光标移到第 5 行、第 10 列。

**隐藏光标 (**\033[?25l**)**

* 1. 代码：\033[?25l
  2. 功能：隐藏终端中的光标。
  3. 说明：在需要更新屏幕的应用中，隐藏光标可以防止用户看到光标在屏幕上跳动。
  4. 使用：例如 printf("\033[?25l"); 将隐藏光标。

**清除从光标到行尾的内容 (**\033[K**)**

* 1. 代码：\033[K
  2. 功能：清除从当前光标位置到该行末尾的所有内容。
  3. 说明：适用于更新行内容或清除当前行的剩余部分。
  4. 使用：例如 printf("\033[K"); 会清除光标到行尾的内容。

### 应用场景

这些控制序列广泛应用于**命令行界面（CLI）应用**和**终端动画**，用于优化用户界面、更新显示内容、调整光标位置等。例如：

* **清屏**：在每次更新数据之前清空屏幕，保持输出整洁。
* **移动光标**：将光标移到特定位置进行数据更新，适合实时数据显示（如进度条、状态更新等）。
* **隐藏光标**：在更新屏幕内容时隐藏光标，避免视觉干扰。
* **清除行尾**：在一行内容更新时清除行尾的旧内容，保持整洁。

### 示例代码

假设你希望在终端中显示一个计数器，同时动态清屏和更新光标位置，可以使用这些控制代码：

c

复制代码

#include <stdio.h>#include <unistd.h>

int main() {

for (int i = 0; i <= 10; i++) {

printf("\033[2J"); // 清屏

printf("\033[H"); // 移动光标到左上角

printf("Counter: %d\n", i); // 显示计数器

fflush(stdout); // 刷新输出缓冲区

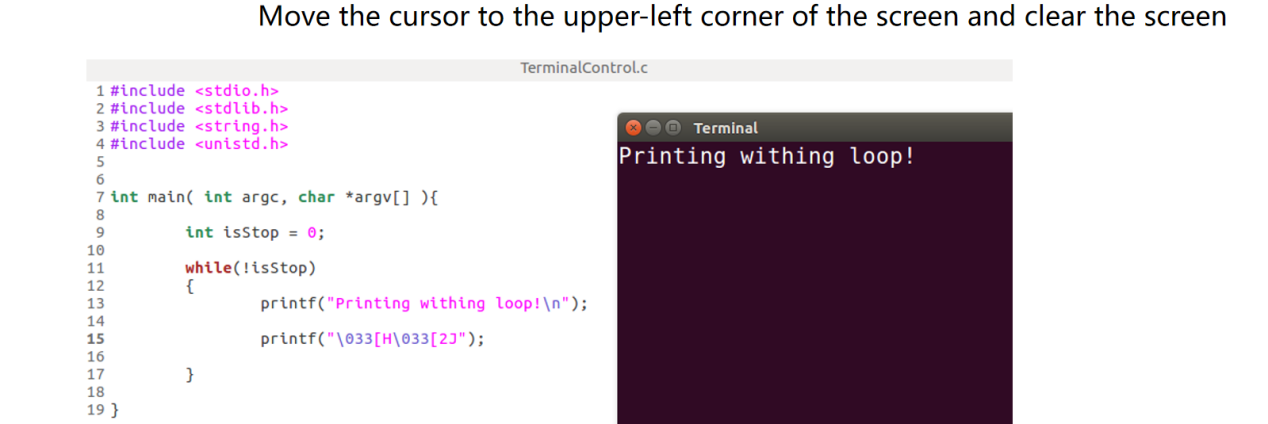
sleep(1); // 等待 1 秒

}

return 0;

}

Another example：  

空指针与解引用：

