面向实时MiniGUI体系结构之一体系结构概览 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/252/2021_2022__E9_9D_A2_E5_90_91_E5_AE_9E_E6_c103_252907.htm 1 引言 到目前为止，MiniGUI 的最新发布版本是 0.9.96。我们将 0.9.xx 系列版本定位为 MiniGUI 1.0 版本的预览版。在 0.9.xx 版本足够稳定时，我们将发布 MiniGUI 1.0 版本，同时，目前的代码不会再进行重大调整。在 MiniGUI 1.0 版本发布之后，我们将立即着手开发 MiniGUI 2.0 版本。该版本预期将在体系结构上进行重大调整。为了吸引更多的自由软件程序员加入 MiniGUI 2.0 的开发，也为了更好地帮助 MiniGUI 程序员进行程序开发，我们将撰写一系列的文章介绍 MiniGUI 1.0 版本的体系结构，重点分析其中的一些缺点以及需要在 2.0 版本当中进行优化和改造的地方。介绍体系结构的文章计划如下： 体系结构概览（本文）。将在整体上对 MiniGUI 1.0 的体系结构进行介绍。重点包括：线程的基本概念；多线程的微客户/服务器体系、多线程通讯的关键数据结构??消息队列；面向对象技术在 MiniGUI 中的应用等等。 MiniGUI 的多窗口管理。将介绍 MiniGUI 的多窗口机制以及相关的窗口类技术。其中涉及到窗口剪切处理和 Z 序，消息传递，控件类设计和输入法模块设计等等。 MiniGUI 的图形设备管理。重点介绍 MiniGUI 是如何处理窗口绘制的。其中主要包括图形上下文的概念，坐标映射，图形上下文的局部、全局和有效剪切域的概念等等。 图形抽象层和输入抽象层。图形抽象层（GAL）和输入抽象层（IAL）大大提高了 MiniGUI 的可移植性，并将底层图形设备和上层接口分离开来。这里将重点介绍 MiniGUI 的 GAL 和 IAL 接口

，并以 EP7211 等嵌入式系统为例，说明如何将 MiniGUI 移植到新的嵌入式平台上。 多字体和多字符集支持。MiniGUI 采用逻辑字体实现多字体和多字符集处理。这一技术成功应用了面向对象技术，通过单一的逻辑接口，可以实现对各种字符集以及各种字体的支持。 2 POSIX 线程 MiniGUI 是一个基于线程的窗口系统。为了理解 MiniGUI 的体系结构，我们有必要首先对线程作一番了解。 2.1 什么是线程 线程通常被定义为一个进程中代码的不同执行路线。也就是说，一个进程中，可以有多个不同的代码路线在同时执行。例如，常见的字处理程序中，主线程处理用户输入，而其他并行运行的线程在必要时可在后台保存用户的文档。我们也可以说线程是“ 轻量级进程 ”。在 Linux 中，每个进程由五个基本的部分组成：代码、数据、栈、文件I/O 和信号表。因此，系统对进程的处理要花费更多的开支，尤其在进行进程调度和任务切换时。从这个意义上，我们可以将一般的进程理解为重量级进程。在重量级进程之间，如果需要共享信息，一般只能采用管道或者共享内存的方式实现。如果重量级进程通过 fork() 派生了子进程，则父子进程之间只有代码是共享的。 而我们这里提到的线程，则通过共享一些基本部分而减轻了部分系统开支。通过共享这些基本组成部分，可以大大提高任务切换效率，同时数据的共享也不再困难??因为几乎所有的东西都可以共享。 从实现方式上划分，线程有两种类型：“ 用户级线程 ”和“ 内核级线程 ”。 用户线程指不需要内核支持而在用户程序中实现的线程，这种线程甚至在象 DOS 这样的操作系统中也可实现，但线程的调度需要用户程序完成，这有些类似 Windows 3.x 的协作式多任务。另外一种则需要内核的

参与，由内核完成线程的调度。这两种模型各有其好处和缺点。用户线程不需要额外的内核开支，但是当一个线程因 I/O 而处于等待状态时，整个进程就会被调度程序切换为等待状态，其他线程得不到运行的机会；而内核线程则没有各个限制，但却占用了更多的系统开支。 Linux 支持内核级的多线程，同时，也可以从 Internet 上下载一些 Linux 上的用户级的线程库。Linux 的内核线程和其他操作系统的内核实现不同，前者更好一些。大多数操作系统单独定义线程，从而增加了内核和调度程序的复杂性；而 Linux 则将线程定义为“ 执行上下文 ”，它实际只是进程的另外一个执行上下文而已。这样，Linux 内核只需区分进程，只需要一个进程/线程数组，而调度程序仍然是进程的调度程序。Linux 的 clone 系统调用可用来建立新的线程。 2.2 POSIX 线程 POSIX 标准定义了线程操作的 C 语言接口。我们可以将 POSIX 线程的接口划分如下： 线程的建立和销毁。用来创建线程，取消线程，制造线程取消点等等。 互斥量操作接口。提供基本的共享对象互斥访问机制。 信号量操作接口。提供基本的基于信号量的同步机制。不能与 System V IPC 机制的信号量相混淆。 条件量操作接口。提供基本的基于条件量的同步机制。尽管信号量和条件量均可以划分为同步机制，但条件量比信号量更为灵活一些，比如可以进行广播，设置等待超时等等。但条件量的操作比较复杂。 信号操作接口。处理线程间的信号发送和线程信号掩码。 其他。包括线程局部存储、一次性函数等等。目前，Linux 上兼容 POSIX 的线程库称为 LinuxThreads，它已经作为 glibc 的一部分而发布。这些函数的名称均以 pthread_ 开头（信号量操作函数以 sem_ 开头）。 为了对线程有一些

感性认识，我们在这里举两个例子。 第一个例子在进入 main () 函数之后，调用 pthread_create 函数建立了另一个线程。pthread_create 的参数主要有两个，一个是新线程的入口函数（thread_entry），另一个是传递给入口函数的参数（data），而新线程的标识符通过引用参数返回（new_thread）。见清单 1。 清单 1 新线程的创建 void* thread_entry (void* data) { ... // do something. return NULL. } int main (void) { pthread_t new_thread. int data = 2. pthread_create (amp.data). pthread_join (new_thread, NULL). } main () 函数在建立了新线程之后，调用 pthread_join 函数等待新线程执行结束。pthread_join 类似进程级的 wait 系统调用。当所等待的线程执行结束之后，该函数返回。利用 pthread_join 可用来实现一些简单的线程同步。注意在上面的例子中，我们忽略了函数调用返回值的错误检查。 第二个例子是利用信号量进行同步的两个线程。这里所使用的例子利用信号量解决了经典的“ 生产者/消费者 ”问题（清单 2）。我们首先解释信号量的基本概念。 信号量的概念由 E. W. Dijkstra 于 1965 年首次提出。信号量实际是一个整数，进程（也可以是线程）在信号量上的操作分两种，一种称为 DOWN，而另外一种称为 UP。DOWN 操作的结果是让信号量的值减 1，UP 操作的结果是让信号量的值加 1。在进行实际的操作之前，进程首先检查信号量的当前值，如果当前值大于 0，则可以执行 DOWN 操作，否则进程休眠，等待其他进程在该信号量上的 UP 操作，因为其他进程的 UP 操作将让信号量的值增加，从而它的 DOWN 操作可以成功完成。某信号量在经过某个进程的成功操作之后，其他休眠在该信号量上的进程就有可能成功完成自己的操作，这时，系统负

责检查休眠进程是否可以完成自己的操作。 为了理解信号量，我们想象某机票定购系统。最初旅客在定票时，一般有足够的票数可以满足定票量。当剩余的机票数为 1，而某个旅客现在需要定两张票时，就无法满足该顾客的需求，这时售票小姐让这个旅客留下他的电话号码，如果其他人退票，就可以优先让这个旅客定票。如果最终有人退票，则售票小姐打电话通知上述要定两张票的旅客，这时，该旅客就能够定到自己的票。 我们可以将旅客看成是进程，而定票可看成是信号量上的 DOWN 操作，退票可看成是信号量上的 UP 操作，而信号量的初始值为机票总数，售票小姐则相当于操作系统的信号量管理器，由她（操作系统）决定旅客（进程）能不能完成操作，并且在新的条件成熟时，负责通知（唤醒）登记的（休眠的）旅客（进程）。 在操作系统中，信号量的最简单形式是一个整数，多个进程可检查并设置信号量的值。这种检查并设置操作是不可被中断的，也称为“ 原子 ”操作。检查并设置操作的结果是信号量的当前值和设置值相加的结果，该设置值可以是正值，也可以是负值。根据检查和设置操作的结果，进行操作的进程可能会进入休眠状态，而当其他进程完成自己的检查并设置操作后，由系统检查前一个休眠进程是否可以在新信号量值的条件下完成相应的检查和设置操作。这样，通过信号量，就可以协调多个进程的操作。 信号量可用来实现所谓的“ 关键段 ”。关键段指同一时刻只能有一个进程执行其中代码的代码段。也可用信号量解决经典的“ 生产者/消费者 ”问题，“ 生产者/消费者 ”问题和上述的定票问题类似。这一问题可以描述如下： 两个进程共享一个公共的、固定大小的缓冲区。其中的一个进程，即

生产者，向缓冲区放入信息，另外一个进程，即消费者，从缓冲区中取走信息（该问题也可以一般化为 m 个生产者和 n 个消费者）。当生产者向缓冲区放入信息时，如果缓冲区是满的，则生产者进入休眠，而当消费者从缓冲区中拿走信息后，可唤醒生产者；当消费者从缓冲区中取信息时，如果缓冲区为空，则消费者进入休眠，而当生产者向缓冲区写入信息后，可唤醒消费者。 清单 2 中的例子实际是“ 生产者/消费者 ”问题的线程版本。 清单 2 利用信号量解决“ 生产者/消费者 ”问题 /* The classic producer-consumer example, implemented with semaphores. All integers between 0 and 9999 should be printed exactly twi 