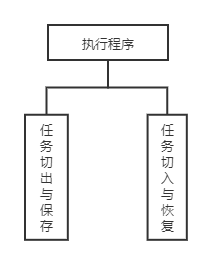
**项目设计**

本设计文档分成两个部分，主要介绍了申威众核处理器从核上执行程序的设计、协程之间的通道通信设计。

1执行程序设计

1.1执行程序概要设计

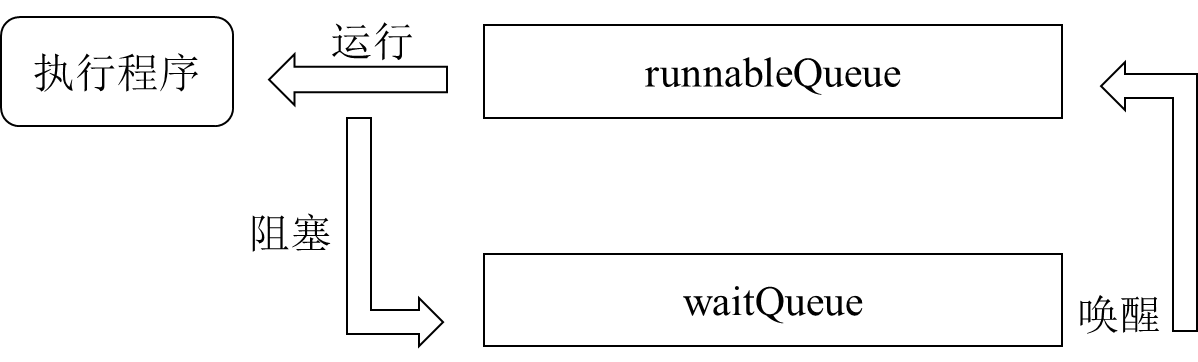
传统的操作系统上的调度算法涉及到用户态与内核态的切换，且比较复杂，进程或者线程的任务状态与阻塞条件较多，不适合直接应用在申威众核处理器中。申威众核处理器的从核是整个申威众核处理器的计算核心单元，从核仅能运行在用户模式下，并且不支持中断，单个从核上仅能运行单个线程。本研究设计的执行程序只需要运行任务队列，切换协程任务。每个协程任务都是一个计算任务，大量的计算任务需要从核的算力，所以将执行程序设计运行在从核上。

执行器在初始化并加载到从核上之后，就会一直等待执行自己所管理的协程任务队列中的协程任务。每当执行一个协程任务时，执行器就会根据协程任务中函数地址去执行它。协程的阻塞与切换、上下文的保存与恢复、协程资源的回收均由执行器完成。当每个从核上的执行器的可运行任务队列和等待任务队列中均无可执行或者挂起的协程任务，并且调度器也无任何新创建协程任务时，执行器会收到来自调度器的停止信号，停止自身运行，并回收相关资源。

**图1.1 执行程序所含模块**

每个从核上都运行着一个执行程序，每个执行器都管理着自己的两个协程任务队列：可运行任务队列（runnableQueue）与等待任务队列（waitQueue），新创建的任务和被唤醒的任务都会加入到可运行任务队列中等待被执行。阻塞等待任务队列中存放的是阻塞状态的协程任务。执行程序包含有协程任务切出与协程任务切入两个模块，如图1.1、图1.2所示。

任务切出接口是在任务函数内被调用的，当协程任务未获取到需要的数据，则主动调用执行程序的任务切出接口。调用接口之后自身的执行状态被中断，上下文状态和参数等信息被保存在开辟的共享栈中，传递到主存中存储。寄存器状态通过数组保存。之后执行程序将被阻塞的协程任务从可运行任务队列中取出，放入到等待任务队列的队尾中。然后继续执行可运行任务队列中下一个协程任务。



**图1.2 协程任务切出与切入过程**

任务切入接口是执行程序调用的，执行程序在每一次执行完可运行任务队列中的协程任务时，都会询问一下数据是否到来。到数据到来时，执行程序将之前被阻塞的协程任务唤醒，唤醒的过程是将其从等待任务队列中取出，放入到可运行任务队列。当执行到该协程任务时，直接调用任务切入接口，从主存读取上下文信息，从开辟的数组中读取寄存器状态，恢复协程任务的上下文，让协程从被阻塞的地方继续运行。

1.2　执行程序详细设计

### 1.2.1 协程任务相关接口函数

执行程序数据结构在调度系统开始运行时就调用sc\_pr\_init函数初始化，并通过申威提供的athread\_spawn接口加载执行程序（swgo\_pr\_start）到从核上执行。

执行程序运行流程图如下图所示。执行程序主要功能就是管理两个任务队列上的协程，与协程运行相关的函数有三个，第一个是协程执行函数co\_resume，第二个是协程任务切出函数co\_swao\_out，第三个协程任务切入函数co\_swap\_in。

co\_resume声明为

**void** co\_resume(coctx\_t\* ctx1,coctx\_t\* ctx2,coctx\_t\* ctx3,**char**\* sp,coctx\_param\_t\* params);

其中ctx1与ctx3是与协程上下文和执行程序上下文相关的结构体，ctx2为中间暂存变量，sp为中间缓冲栈，params为参数列表。在系统中调用情况如下。

1.co\_resume(swco->coctx,&bctx\_base,&bctx\_temp,sp,para);

执行程序通过co\_resume函数执行协程任务。第一个参数swco->coctx为当前协程的上下文结构体，第二个参数bctx\_base为当前执行程序自身的上下文，第三个参数bctx\_temp作为中间缓冲的上下文，sp为协程栈空间，para为参数列表。由于从核上只能运行单线程，同一时间只能执行一个代码文件，所以在进入到协程任务执行前，需要将执行程序自身的上下文和寄存器内容存储起来，再去执行协程任务。当协程执行完毕或者由于阻塞切出时，再跳转到执行程序。

co\_swap\_out声明为

**void** co\_swap\_out(coctx\_t\* ctx1,coctx\_t\* ctx2,coctx\_t\* ctx3,**char**\* sp);

co\_swap\_in声明为

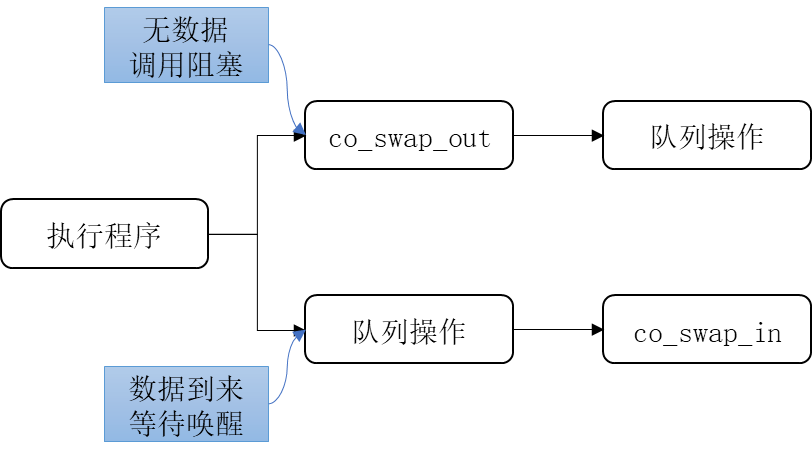
**void** co\_swap\_in(coctx\_t\* ctx1,coctx\_t\* ctx2,coctx\_t\* ctx3,**char**\* sp);

其中co\_swap\_out中ctx2与ctx3是与切入切出有关的协程上下文结构体，ctx1作为中间缓冲的上下文结构体，sp作为中间缓冲栈。co\_swap\_in中ctx1与ctx2是与上下文切入有关的上下文结构体，ctx3为中间缓冲的上下文结构体，sp也是作为中间缓冲栈。在系统中调用情况如下。

1.co\_swap\_out(&bctx\_temp,swco->coctx,&bctx\_base,sp);

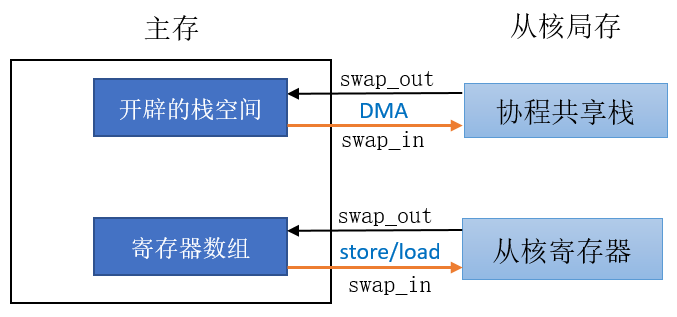
2.co\_swap\_in(&bctx\_base,swco->coctx,&bctx\_temp,sp);

本研究中在协程结构体内嵌套了上下文结构体，用以保存协程执行过程中的数据，并在申威众核处理器下使用其指令调用其寄存器用于保存参数和恢复（运行栈，返回地址，寄存器状态）。如图1.3所示。



**图1.3 两个协程任务模块作用**

协程间的依赖关系需要用户自己明确，当协程任务获取所需数据无法得到时，将自身状态变为wait（阻塞），之后主动调用co\_swap\_out接口，我们为了保证协程任务被唤醒时能继续执行，需要正确保存并恢复其运行时的上下文。协程上下文的保存与恢复主要是运行栈与寄存器状态的保存与恢复。由于从核的私有存储（LDM）的大小极小，只有64kb，无法支撑大量协程的协程栈的保存。所以，我们采用了共享栈的思想，在每个从核上预先开辟一段空间作为协程的共享栈空间。执行程序自身的上下文全部存放在从核局存中，这是与协程上下文保存与恢复所不同的地方。如图1.4所示。



**图1.4 共享栈设计思路**

在协程阻塞时，将共享栈空间的内容用DMA传输的方式传输到处理器的主内存进行存储，对于寄存器状态的保存，我们根据处理器寄存器的数目开辟相应的数组进行寄存器状态的保存。然后执行程序才进行相应的队列操作。接口函数内的处理流程如下所示。

1. co\_swap\_out.c //切出函数
2. asm **volatile**    //C语言中内嵌汇编
3. //使用申威平台的store指令(stl)将当前程序（任务函数）寄存器的值
4. //存入到相关地址（主存）的寄存器数组中
5. stl 寄存器,寄存器数组
6. //使用申威平台的athread调用，通过DMA方式
7. //将当前程序（任务函数协程）栈信息存入到主存中保存
8. athread\_put  运行栈,地址空间
9. //将执行程序运行栈从从核局存中读取回来（直接读取）
10. asm **volatile**
11. //使用申威平台的load指令(ldl)
12. //将之前保存的执行程序寄存器数组的值读取回来
13. //返回到原来执行程序上下文继续执行

在协程切入时，与协程阻塞相反，是由执行程序先进行队列操作，然后再调用co\_swap\_in接口，将切换进来的协程的栈空间用DMA传输方式从主核内存拷贝到共享栈空间，完成协程栈的切换。对于寄存器状态，我们传入当前协程与切入协程的寄存器状态数组首地址，使用申威众核处理器平台相应的汇编指令在汇编层面完成寄存器状态的保存与切换。接口函数内的处理流程如下所示。

1. co\_swap\_in.c //切入函数
2. asm **volatile**  //C语言中内嵌汇编
3. //使用申威平台的store指令(stl)将当前程序（执行程序）寄存器的值
4. //存入到相关地址空间（从核局存）中
5. stl 寄存器,寄存器数组
6. //使用申威平台的athread调用，通过DMA方式
7. //将当前程序（执行程序）栈信息存入到从核局存中
8. //将主存中保存的协程栈信息读入到从核共享栈
9. athread\_get  地址空间,运行栈
10. asm **volatile**
11. //使用申威平台的load指令(ldl)
12. //将之前保存的协程寄存器数组的值读取回来
13. //跳转(call)到切入的协程任务函数

### 1.2.2 执行程序处理任务流程

**图1.5 执行程序运行状态**

图1.5为执行程序运作状态图，由于每个从核上运行的执行程序代码相同，所以执行程序设计时需要考虑协程正常运行状态和阻塞状态两种情况。执行程序总体上是不断while循环执行的，当收到来自调度程序发送来的停止信号时（stop\_signal）才会退出循环。在循环内，当可运行任务队列不为空时，执行程序取队头协程，首先根据协程状态是运行（run）还是等待（wait），确定该协程是第一次执行还是之前阻塞过的协程，是否需要切入上下文继续执行。执行完之后，若协程状态为完成（done），说明正常执行完成，则回收资源。若协程状态为wait，则说明这个协程自身调用co\_swap\_out被阻塞了，自身的状态被保存起来，然后跳出计算任务，回到执行程序，执行程序将这个被阻塞的协程放入到等待任务队列中。

唤醒相关协程执行，首先需要将阻塞协程从等待任务队列中取出，插入到可运行任务队列第二个位置。然后等待执行程序执行到它，执行程序判断一下协程状态，主动调用co\_swap\_in接口，将所有的寄存器状态和参数恢复，从协程切出的地方继续执行。

2 　通道通信接口设计

2.1 通道通信接口概要设计

本项目使用生产者-消费者模式进行协程间的消息传递，基本的使用方式就需要包含数据的发送以及接收两个接口。在参考了Go语言中的channel实现以及C++协程库libgo中的通信实现后，我们决定以通道（channel）作为中间体的方式进行生产者-消费者模式的消息发送与接收。为了能够在主核与从核上运行的程序之间进行数据的通信，需要主核与从核都能够访问通道中的数据，于是我们将通道的数据保存在主存的主从核共享内存区域中。由于主核与从核上的程序需要分别进行编写与编译，我们将在主核与从核环境下进行不同的实现，但保持向上层应用提供的通道的接口一致。我们首先对通道的基础功能——从消息生产者发送特定的值到消息的消费者来开始，进行申威众核平台协程间通信的接口设计。最终设计了具体的接口如下所示。

1. channel\* channel\_init\_p(uint32\_t capacity,uint32\_t elem\_size, **enum** PATTERN pattern);
2. uint32\_t push(channel \* chan,uint8\_t \*buf);
3. uint32\_t pop(channel \*chan,uint8\_t \*buf);
4. **bool** \_close(channel \*chan);
5. **bool** empty(channel \*chan);
6. **bool** full(channel \*chan);

其中channel\_init\_p()函数用于创建并初始化接口，传入的参数为通道容量capacity，消息体大小elem\_size，通道运行模式pattern。函数返回一个通道结构体指针。程序在调用消息发送和接收接口时，会根据不同的通道模式改变消息发送与读取的方式，通道的模式PATTERN包含MANY\_TO\_MANY，ONE\_TO\_ONE，ONE\_TO\_MANY，MANY\_TO\_ONE四种。模式决定了发送与接收接口的具体行为，将在3.3节通道的同步机制中详细介绍。

push()函数是生产者向通道发送数据所使用的接口，将数据复制到通道中；参数为通道指针chan，要发送的消息指针buf，发送成功时返回0；

pop()函数是消费者从通道中读取消息的接口，从通道中取得数据；参数为通道地址chan，接收数据的变量的指针buf，接收成功时返回0；

close()函数将通道关闭，关闭后的通道不能继续进行消息的发送于接收，参数为通道地址chan；

empty()函数用于判断通道是否为空，为空时返回true，不为空时返回false，参数为通道地址chan；

full()函数用于判断通道是否为满，已满时返回true，未满时返回false，参数为通道地址chan。

2.2 通道通信详细设计

2.2.1 通道通信接口详细设计

我们选择的通道通信的主要数据结构为循环缓冲区（Ring buffer），循环缓冲区是一种能较为方便地实现生产者-消费者机制的数据结构，在没有生产者之间或消费者之间互相竞争的条件下可以无锁化的进行数据的传递，循环缓冲区有多种实现方式，并在数据通信中得到广泛应用。本研究使用基于循环缓冲区的数据读写实现在主核与从核上通用的通道消息的存取，包含了循环缓冲区结构的通道的结构体定义如下所示。

1. **struct** channel{
2. unsigned **char** \*buffer;
3. unsigned **int** elem\_size;
4. unsigned **int** capacity;
5. unsigned **int** read;
6. unsigned **int** to\_read;
7. unsigned **int** write;
8. unsigned **int** to\_write;
9. **enum** PATTERN pattern;
10. list read\_wait;
11. list write\_wait;
12. **bool** closed;
13. };

通道的数据存取如图2.1所示。



**图2.1　通道的存取**

2.2.2　同步机制的实现

我们首先考虑主核上同步机制的实现。在x86架构的系统中可以用CAS（compare-and-swap）原子操作来保证数据的一致性。CAS操作常用于无锁化数据结构的实现。在主核上支持CAS操作，而在从核上并不支持。我们可以直接使用CAS方式进行协程间并行时数据一致性的保证，申威26010处理器的C语言编译器sw5cc支持类似于GCC的编译环境，其中的CAS接口如下：

bool \_\_sync\_bool\_compare\_and\_swap (type \*ptr, type oldval, type newval, ...)

使用CAS操作实现对通道进行消息发送的同步代码如下所示。

1. **do** {
2. **if** (full(chan))
3. **continue**;
4. temp = chan->write;
5. **if**(temp != chan->to\_write)
6. **continue**;
7. ok = CAS(&chan->write, temp, temp+1);
8. } **while** (!ok);
9. //此处复制数据到缓冲区
10. CAS(&chan->to\_write, temp, temp+1);

我们通过使用CAS原子操作对主核上对通道的发送与接收操作实现了同步。但是在从核上的指令支持和编译环境与主核不同，不能支持CAS操作，因此我们不能像在主核上那样的方式直接利用CAS操作来完成协程之间的数据同步。从核也不支持线程的锁，信号量以及其他的线程同步机制。只有一种基本的原子操作可以利用。从核上支持原子化的数据加减操作。对数据进行原子操作的接口如下：

updt\_addw(\_n\_, \_addr\_)

参数\_n\_是需要对变量增加的数，\_addr\_是变量的地址。原子操作是不可分割的，在执行完毕之前不会被任何其它任务或事件中断。

使用原子操作实现从核上多生产者消息发送的代码如下所示。

1. **while**(**true**){
2. **if**(full(chan))
3. co\_swap\_out();
4. temp = chan->write;
5. **if** (temp == chan->to\_write)
6. updt\_addw(1,&(chan->write));
7. **else**
8. **continue**;
9. **if**(chan->write == temp+1){
10. //此行复制数据到缓冲区
11. updt\_addw(1,&chan->to\_write);
12. **return** 0;
13. }**else**{
14. updt\_addw(-1,&chan->write);
15. **continue**;
16. }
17. }

2.2.3　通道通信的四种模式

由于在实际应用的开发中，使用通道的生产者与消费者并不一定都需要竞争，因此，为了最大化向通道发送消息与从通道读取消息的执行效率，本研究设计了通道的不同运行模式，在通道的创建参数中，pattern是一个决定通道运行时，消息发送与接收的具体执行方式的一个参数，其类型PATTERN的定义如下。

1. **enum** PATTERN{
2. MANY\_TO\_MANY ,
3. ONE\_TO\_ONE ,
4. ONE\_TO\_MANY,
5. MANY\_TO\_ONE
6. };

PATTERN共有四种模式：MANY\_TO\_MANY，ONE\_TO\_ONE，ONE\_TO\_MANY

，MANY\_TO\_ONE代表着不同的生产者-消费者情况，为了获得通道通信时的最高效率，程序开发时可以根据需要自行选择所需要的模式。