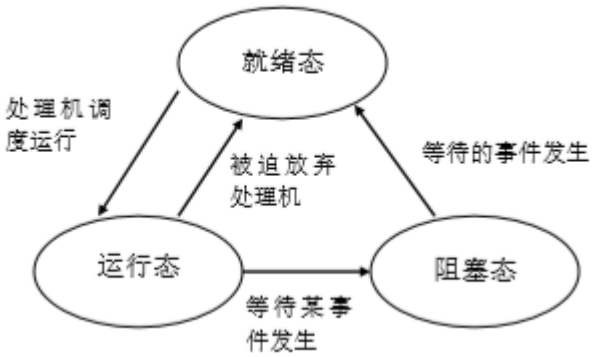
Nachos线程管理

一、进程与线程

1. 进程

系统运行过程中，通常有多个进程同时存在，它们各自执行的指令序列，对系统资源和服务的需求以及状态的变化往往互不相同、千变万化而难以预测。同时还可能接收到需要立即处理的中断信号。而中断信号发生的时间以及频繁程度与系统中许多经常变换着的不确定因素有关。所以每个进程在一种不可预测的次序中交替前进。操作系统内部动作的不可预测、不可重复就是操作系统的不确定性



进程调度算法：略。

2. 线程

略

3. 进程与线程的关系

略

二、Nachos的线程管理

Nachos 是多线程操作系统。线程是 Nachos 处理机调度的单位，在 Nachos 中线程分成两类，一类是系统线程。所谓系统线程是只运行核心代码的线程，它运行在核心态下，并且占用宿主机的资源，系统线程共享 Nachos 操作系统本身的正文段和数据段；一个系统线程完成一件独立的任务，比如在 Nachos 网络部分，有一个独立的线程一直监测有无发给自己的数据报。

Nachos 的另一类线程同 Nachos 中的用户进程有关。Nachos 中用户进程由两部分组成，核心代码部分和用户程序部分。用户进程的进程控制块是线程控制块基础上的扩充。每当系统接 收到生成用户进程的请求时，首先生成一个系统线程，**进程控制块中有保存线程运行现场的空间**，保证线程切换时现场不会丢失。该线程的作用是给用户程序分配虚拟机内存空间，并 把用户程序的代码段和数据段装入用户地址空间，然后调用解释器解释执行用户程序；由于 Nachos 模拟的是一个**单机环境**，多个用户进程会竞争使用 Nachos 唯一的处理机资源，所以在 Nachos 用户进程的进程控制块中增加有虚拟机运行现场空间以及进程的地址空间指针等内容，保证用户进程在虚拟机上的正常运行。

Nachos 除了在线程管理上作了一系列的简化外，和实际的进程管理还有以下的不同：

- 不存在系统中所有线程的列表：在一般的操作系统中，进程的数目总是有限的，但是 Nachos 中的线程数目可以是无限的

- 线程的调度比较简单：在启动了时钟中断的情况下，当时钟中断到来时，如果就绪线程队列中有就绪线程，就必须进行线程切换；当没有启动时钟中断的情况下，Nachos 使用非抢占式调度。
- 没有实现父子线程的关系

三、Nachos线程管理系统的初步实现

3.1 工具模块分析（文件list.cc list.h utility.cc utility.h）

工具模块定义了一些在 Nachos 设计中有关的工具函数，和整个系统的设计没有直接的联系。List 类，在 Nachos 中广泛使用，它定义了一个链表结构

```
class ListElement {                          //   定义了 List 中的元素类型
    public:
      ListElement(void *itemPtr, int sortKey);   //   初始化方法
      ListElement *next;                         //   指向下一个元素的指针
      int key;                                   //   对应于优先队列的键值
      void *item;                                //   实际有效的元素指针
};
其中，实际有效元素指针是(void *)类型的，说明元素可以是任何类型。

class List {
  public:
    List();                                  //   初始化方法
    ~List();                                 //   析构方法
    void Prepend(void *item);                //   将新元素增加在链首
    void Append(void *item);                 //   将新元素增加在链尾
    void *Remove();                          //   删除链首元素并返回该元素
    void Mapcar(VoidFunctionPtr func);       //   将函数 func 作用在链中每个元素上
    bool IsEmpty();                          //   判断链表是否为空
    void SortedInsert(void *item, int sortKey);  //   将元素根据 key 值优先权插入到链中
    void *SortedRemove(int *keyPtr);         //   将 key 值最小的元素从链中删除，
                                             //   并返回该元素
  private:
    ListElement *first;                      //   链表中的第一个元素
    ListElement *last;                       //   链表中的最后一个元素
};
```

3.2 线程启动和调度模块分析（文件switch.s switch.h）

线程启动和线程调度是线程管理的重点。在 Nachos 中，**线程是最小的调度单位**。Nachos 的线程切换借助于宿主机的正文切换，由于这部分内容与机器密切相关，而且直接同宿主机的寄存器进行交道，所以是用汇编来实现的。由于 Nachos 可以运行在多种机器上，不同机器的寄存器数目和作用不一定相同， 所以在 switch.s 中针对不同的机器进行了不同的处理。读者如果需要将 Nachos 移植到其它 机器上，就需要修改这部分的内容。

3.2.1 ThreadRoot函数

Nachos 中，除了 main 线程外，所有其它线程都是从 ThreadRoot 入口运行的。它的语法是： ThreadRoot (int InitialPC, int InitialArg, int WhenDonePC, int StartupPC)

其中，InitialPC 指明新生成线程的入口函数地址，InitialArg 是该入口函数的参数；StartupPC 是在运行该线程是需要作的一些初始化工作，比如开中断；而 WhenDonePC 是当该线程运 行结束时需要作的一些后续工作。。在 Nachos 的源代码中，没有任何一个函数和方法显式地 调用 ThreadRoot 函数，**ThreadRoot 函数只有在线程切换时才被调用到。**一个线程在其初始化的最后准备工作中**调用 StackAllocate 方法**，该方法设置了几个寄存器的值（**InterruptEnable 函数指针，ThreadFinish 函数指针以及该线程需要运行函数的函数指针和 运行函数的参数**），该线程**第一次被切换上处理机运行时调用的就是 ThreadRoot 函数。**其工作过程是：

1. 调用 StartupPC 函数；
2. 调用 InitialPC 函数；
3. 调用 WhenDonePC 函数；

这里我们可以看到，由 ThreadRoot 入口可以转而运行线程所需要运行的函数，从而达到生成线程的目的。

3.2.2 SWITCH函数

Nachos 中系统线程的切换是借助宿主机的正文切换。SWITCH 函数就是完成线程切换的功能。SWITCH 的语法是这样的： void SWITCH (Thread *t1, Thread *t2);

其中 t1 是原运行线程指针，t2 是需要切换到的线程指针。线程切换的三步曲是：

1. 保存原运行线程的状态
2. 恢复新运行线程的状态
3. 在新运行线程的栈空间上运行新线程

3.3 线程模块分析（文件thread.cc thread.h）

Thread 类实现了操作系统的线程控制块，同操作系统课程中进程程管理中的 PCB (Process Control Block) 有相似之处。

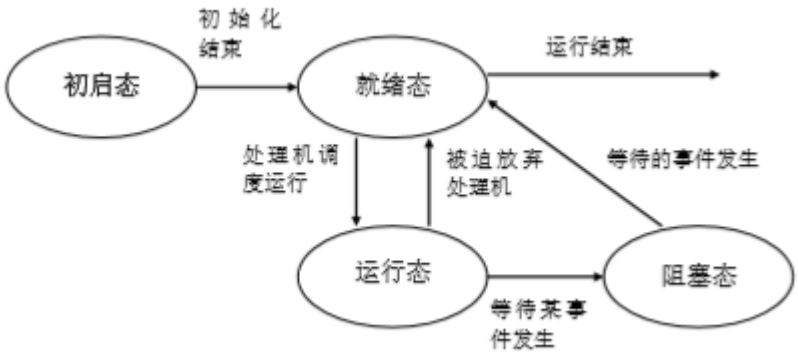
Thread 线程控制类较 PCB 为简单的多，它没有线程标识 (pid)、实际用户标识 (uid)等和线 程操作不是非常有联系的部分，也没有将 PCB 分成 proc 结构和 user 结构。Nachos 对线程的另一个简化是每个线程栈段的大小是固定的，为 4096-5 个字 (word)，而且 是不能动态扩展的。**如果系统线程需要使用的栈空间大于规定栈空间的大小，可以修改 StackSize 宏定义。**

```
class Thread {
  private:
    int* stackTop;                          //   当前堆栈指针
    int machineState[MachineStateSize];     //   宿主机的运行寄存器
  public:
    Thread(char* debugName);                //   初始化线程
```



```
    ~Thread();                              //   析构方法
    void Fork(VoidFunctionPtr func, int arg);   //   生成一个新线程，执行 func(arg)
    void Yield();                           //   切换到其它线程运行
    void Sleep();                           //   线程进入睡眠状态
    void Finish();                          //   线程结束时调用
    void CheckOverflow();                   //   测试线程栈段是否溢出
    void setStatus(ThreadStatus st);        //   设置线程状态
    char* getName() { return (name); }      //   取得线程名（调试用）
    void Print() { printf("%s, ", name); }  //   打印当前线程名（调试用）
  private:
    int* stack;                             //   线程的栈底指针
    ThreadStatus status;                    //   当前线程状态
    char* name;                             //   线程名（调试用）
    void StackAllocate(VoidFunctionPtr func, int arg);
                                            //   申请线程的栈空间
#ifdef USER_PROGRAM
    int userRegisters[NumTotalRegs];            //   虚拟机的寄存器组
  public:
    void SaveUserState();                   //   线程切换时保存虚拟机寄存器组
    void RestoreUserState();                //   线程切换时恢复虚拟机寄存器组
    AddrSpace *space;                       //   线程运行的用户程序
#endif
}
```

线程控制类中的 stackTop 栈指针变量，machineState 机器寄存器数组变量的位置必须是固定的，因为这两个变量和线程的切换有密切的关系，在 SWITCH 函数中： void SWITCH (Thread *t1, Thread *t2); (int *)(t1)实际上指向原有线程的栈空间，而 (int)(*(t1+1))开始则同宿主机的寄存器组一一对应。



- JUST_CREATED 线程初始时的状态。此时线程控制块中没有任何内容
- RUNNING 线程正在处理机上运行
- READY 线程处理就绪态
- BLOCKED 线程处于阻塞态

用户进程在线程切换的时候，除保存宿主机的状态外，必须还要保存虚拟机的寄存器状态。UserRegisters[]数组变量和 SaveUserState(), RestoreUserState()方法就是为了用户进程的切换设计的。

在 UNIX 操作系统中，进程终止时释放大部分空间，有一部分工作留给父进程处理。在 Nachos 中，当一个线程运行结束时，同样需要将线程所占用的空间释放。但是 Nachos 线程不能释放自己的空间，因为此时它还运行在自己的栈段上。所以当线程结束时调用 Finish 方法，Finish 方法的作用是设置全局变量 threadToBeDestroyed，说明该线程已经运行结束，需释放栈空间。Finish 紧接切换到其它线程，该运行线程释放 threadToBeDestoryed 线程栈空间。Scheduler 类中的 Run 方法才有机会删除 threadToBeDestroyed 线程栈空间。当系统中没有就绪线程和中断等待处理时， 系统会退出而不会切换到其它线程 ， 只有借助于系统释放空间的机制来释放 threadToBeDestroyed线程的空间。

3.3.1 Fork方法

语法： void Fork (VoidFunctionPtr func, int arg)

参数： func: 新线程运行的函数 arg: func 函数的参数

功能： 线程初始化之后将线程设置成可运行的。

实现： 1. 申请线程栈空间 2. 初始化该栈空间，使其满足 SWITCH 函数进行线程切换的条件 3. 将该线程放到就绪队列中。

3.3.2 StackAllocate 方法

语法： void StackAllocate (VoidFunctionPtr func, int arg)

参数： func: 新线程运行的函数 arg: func 函数的参数

功能： 为一个新线程申请栈空间，并设置好准备运行线程的条件。

实现：

```
void Thread::StackAllocate (VoidFunctionPtr func, int arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    //    申请线程的栈空间
    stackTop = stack + StackSize - 4;          //    设置栈首指针
    *(--stackTop) = (int)ThreadRoot;
    //    线程准备好运行后进行线程切换，会切换到 ThreadRoot 函数。ThreadRoot 函数
    //    将会开中断，并调用 func(arg)成为一个独立的调度单位。
    *stack = STACK_FENCEPOST;                  //    设置栈溢出标志
    machineState[PCState] = (int) ThreadRoot; //    设置 PC 指针，从 ThreadRoot 开始运行
    machineState[StartupPCState] = (int) InterruptEnable;
    machineState[InitialPCState] = (int) func;
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = (int) ThreadFinish;
    //    以上是为 ThreadRoot 作好准备，ThreadRoot 将分别调用 InterruptEnable，
    //    func(arg)和 ThreadFinish。
}
```

3.3.3 Yield 方法

语法： void Yield ()

功能： 当前运行强制切换到另一个就绪线程运行

实现：

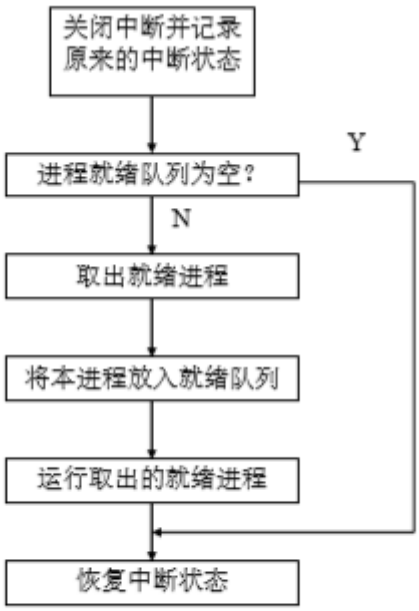


图 3.7　Yield 函数

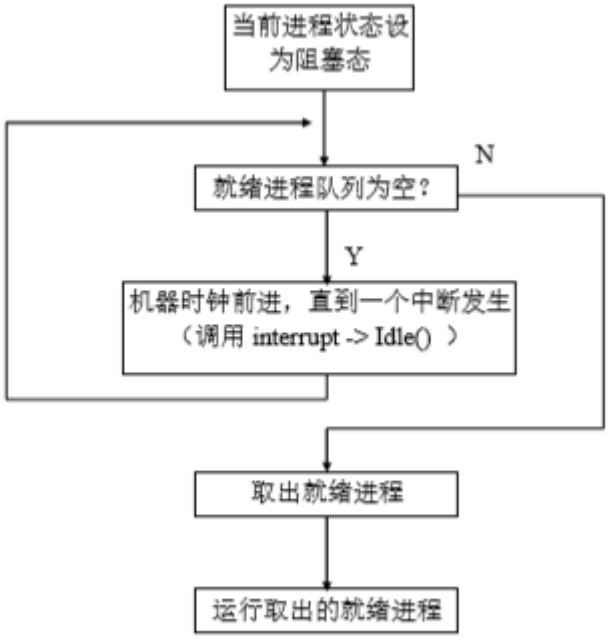
注意：这些操作是原子操作，不允许中断。

3.3.4 Sleep 方法

语法： void Sleep ()

功能： 线程由于某种原因进入阻塞状态等待一个事件的发生（信号量的 V 操作、开锁或者条 件变量的设置）。当这些条件得到满足，该线程又可以恢复就绪状态。

实现：



3.4 线程调度算法模块分析（文件scheduler.cc scheduler.h）

该模块的作用是进行线程的调度。在 Nachos 系统中，有一个线程就绪队列，其中是所有就绪线程。调度算法非常简单，就是取出第一个放在处理机运行即可。由于 Nachos 中线程没有优先级，所以线程就绪队列是没有优先级的。

```
class Scheduler {
  public:
    Scheduler();                        //  初始化方法
    ~Scheduler();                       //  析构方法
    void ReadyToRun(Thread* thread);    //  设置一个线程为就绪态
    Thread* FindNextToRun();            //  找出下一个处于就绪态的线程
    void Run(Thread* nextThread);       //  切换到 nextThread 执行
```



```
    void Print();                       //  打印出处于就绪态的所有线程
  private:
    List *readyList;                    //  线程就绪队列
};
```

3.4.1 Run方法

语法： void Run (Thread *nextThread)

参数： nextThread: 需要切换运行的线程

功能： 当前运行强制切换到 nextThread 就绪线程运行

实现： 1. 如果是用户线程，保存当前虚拟机的状态 2. 检查当前运行线程栈段是否溢出。（由于不是每时每刻都检查栈段是否溢出，所以这时候线程的运行可能已经出错） 3. 将 nextThread 的状态设置成运行态，并作为 currentThread 现运行线程（在调用 Run 方法之前，当前运行线程已经放入就绪队列中，变成就绪态） （以上是运行在现有的线程栈空间上，以下是运行在 nextThread 的栈空间上） 4. 切换到 nextThread 线程运行 5. 释放 threadToBeDestroyed 线程需要栈空间（如果有的话） 6. 如果是用户线程，恢复当前虚拟机的状态

3.5 Nachos主控模块分析（文件main.cc system.cc system.h）

该模块是整个 Nachos 系统的入口，它分析了 Nachos 的命令行参数，根据不同的选项进行不 同功能的初始化设置。选项的设置如下所示：

- -d: 显示特定的调试信息
- -rs: 使得线程可以随机切换
- -z: 打印版权信息和用户进程有关的选项：
- -s: 使用户进程进入单步调试模式
- -x: 执行一个用户程序
- -c: 测试终端输入输出和文件系统有关的选项：
- -f: 格式化模拟磁盘
- -cp: 将一个文件从宿主机拷贝到 Nachos 模拟磁盘上
- -p: 将 Nachos 磁盘上的文件显示出来
- -r: 将一个文件从 Nachos 模拟磁盘上删除
- -l: 列出 Nachos 模拟磁盘上的文件
- -D: 打印出 Nachos 文件系统的内容
- -t: 测试 Nachos 文件系统的效率 和网络有关的选项：
- -n: 设置网络的可靠度（在 0-1 之间的一个小数）
- -m: 设置自己的 HostID
- -o: 执行网络测试程序

main 函数的处理逻辑：

```
int main (int argc, char **argv)
{
    对命令行参数进行处理，并且初始化相应的功能
    currentThread -> Finish ();
    return (0);                    //    此行执行不到。
}
```

在 main函数的最后，是currentThread->Finish()语句。为什么不直接退出呢？这是因为Nachos 是在宿主机上运行的一个普通的进程，当 main 函数退出时，整个占用的空间要释放，进程 也相应的结束。但是实际上在 Nachos 中，main 函数的结束并不能代表系统的结束，因为可 能还有其它的就绪线程。所以在这里我们只是将 main 函数作为 Nachos 中一个特殊线程进行 处理，该线程结束只是作为一个线程的结束，系统并不会退出。

3.6 同步机制模块分析（文件synch.cc synch.h）

线程的同步和互斥是多个线程协同工作的基础。Nachos 提供了三种同步和互斥的手段：信号量、锁机制以及条件变量机制，提供三种同步互斥机制是为了用户使用方便

Nachos 是运行在单一处理器上的操作系统，在单一处理器上，实现原子操作只要在操作之前关中断即可，操作结束后恢复原来中断状态。

3.6.1 信号量（Semaphore）

信号量的私有属性有信号量的值，它是一个阀门。线程等待队列中存放所有等待该信号量的 线程。信号量有两个操作：P 操作和 V 操作，这两个操作都是原子操作。

```
class Semaphore {
  public:
    void P();                    //  信号量的 P 操作
    void V();                    //  信号量的 V 操作
  private:
    int value;                   //  信号量值（>=0)
    List *queue;                 //  线程等待队列
};
```

3.6.1.1 P 操作

1. 当 value 等于 0 时，
   1.1. 将当前运行线程放入线程等待队列。
   1.2. 当前运行线程进入睡眠状态，并切换到其它线程运行。
2. 当 value 大于 0 时，value--。

3.6.1.2 V 操作

1. 如果线程等待队列中有等待该信号量的线程，取出其中一个将其设置成就绪态，准备运行。
2. value++;

3.6.2 锁机制

锁机制是线程进入临界区的工具。一个锁有两种状态，BUSY 和 FREE。当锁处于 FREE 态时，线程可以取得该锁后进入临界区，执行完临界区操作之后，释放锁；当锁处于 BUSY 态时，需要申请该锁的线程进入睡眠状态，等到锁为 FREE 态时，再取得该锁。

```
class Lock {
  public:
    Lock(char* debugName);              //  初始化方法
    ~Lock();                            //  析构方法
    char* getName() { return name; }    //  取出锁名（调试用）
    void Acquire();                     //  获得锁方法
    void Release();                     //  释放锁方法
    bool isHeldByCurrentThread();       //  判断锁是否为现运行线程拥有
  private:
    char* name;                         //  锁名（调试用）
};
```

在现有的 Nachos 中，没有给出锁机制的实现，锁的基本结构也只给出了部分内容，其它内 容可以视实现决定。总体来说，锁有两个操作 Acquire 和 Release，它们都是原子操作。

Acquire: 申请锁：

- 当锁处于 BUSY 态，进入睡眠状态。
- 当锁处于 FREE 态，当前线程获得该锁，继续运行

Release: 释放锁（注意：只有拥有锁的线程才能释放锁）

- 将锁的状态设置成 FREE 态，如果有其它线程等待该锁，将其中的一个唤醒，进入就绪态。

3.6.3 条件变量

条件变量和信号量与锁机制不一样，它是没有值的。（实际上，锁机制是一个二值信号量， 可以通过信号量来实现）当一个线程需要的某种条件没有得到满足时，可以将自己作为一个 等待条件变量的线程插入所有等待该条件变量的队列；只要条件一旦满足，该线程就会被唤 醒继续运行。条件变量总是和锁机制一同使用，它的基本结构如下：

```
class Condition {
  public:
    Condition(char* debugName);                 //  初始化方法
    ~Condition();                               //  析构方法
    char* getName() { return (name); }          //  取出条件变量名（调试用）
    void Wait(Lock *conditionLock);             //  线程进入等待
    void Signal(Lock *conditionLock);           //  唤醒一个等待该条件变量的线程
    void Broadcast(Lock *conditionLock);        //  唤醒所有等待该条件变量的线程
  private:
    char* name;                                 //  条件变量名（调试用）
};
```

在现有的 Nachos 中，没有给出条件变量的实现，条件变量的基本结构也只给出了部分内容， 其它内容可以视实现决定。总体来说，条件变量有三个操作 Wait、Signal 以及 BroadCast， 所有的这些操作必须在当前线程获得一个锁的前提下，而且所有对一个条件变量进行的操作 必须建立在同一个锁的前提下。

void Wait (Lock *conditionLock)	线程等待条件变量 1. 释放该锁 2. 进入睡眠状态 3. 重新申请该锁
void Signal (Lock *conditionLock)	唤醒一个等待该条件变量的线程（如果存在的话）
void BroadCast (Lock *conditionLock)	唤醒所有等待该条件变量的线程（如果存在的话）