**北京科技大学实验报告**

学院： 计通学院 专业： 计算机科学与技术 班级： 计172

姓名： 金玉卿 学号： 41724051 实验日期： 2019年 11月30 日

**实验名称：操作系统实验2 线程状态及转换（4分）**

**实验目的：**以一个教学型操作系统EOS为例，熟悉线程状态及其转换，理解线程状态转换与线程调度的关系；能对核心源代码进行分析和修改；训练分析问题、解决问题以及自主学习能力，逐步达到能独立对小型操作系统的功能进行分析、设计和实现。

**实验环境：**EOS操作系统及其实验环境。

**实验内容：**

跟踪EOS线程在各种状态间的转换过程，分析EOS中线程状态及其转换的相关源代码，描述EOS定义的线程状态以及状态转换的实现方法；修改EOS的源代码，为线程增加挂起状态。

**实验步骤：**

**1）EOS线程状态转换过程的跟踪与源代码分析**

（分析EOS中线程状态及其转换的核心源代码，总结EOS定义的线程状态以及状态转换的实现方法，包括数据结构和算法等；简要说明在本部分实验过程中完成的主要工作，包括对EOS线程状态转换过程的跟踪等）

在实验中主要对EOS中的下列线程状态转换过程有一个全面的认识：

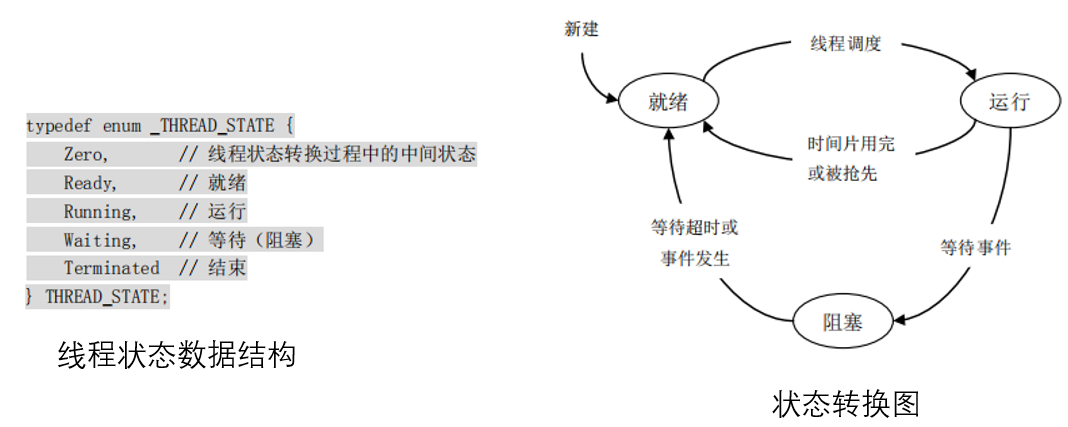
*  线程由阻塞状态进入就绪状态。
*  线程由运行状态进入就绪状态。
*  线程由就绪状态进入运行状态。
*  线程由运行状态进入阻塞状态。

1. **背景知识**

* 就绪状态：当进程已分配到除CPU以外的所有必要的资源，只要获得处理机便可立即执行，这时的进程状态称为就绪状态。
* 运行状态：当进程已获得处理机，其程序正在处理机上执行，此时的进程状态称为运行状态。
* 阻塞状态：正在执行的进程，由于等待某个事件发生而无法执行时，便放弃CPU而处于阻塞状态。引起进程阻塞的时间可有多种，例如，等待I/O完成、申请缓冲区不能满足、等待信号等。

1. **相关实现**

EOS线程的状态由线程控制块TCB中的State域保存，在文件ps/psp.h中定义的线程状态如下：



EOS线程的状态和转换过程如上图，由于是运行在单处理器上的多任务操作系统，所以在任意时刻最多只能有一个处于运行状态的线程占有处理器。

EOS中线程状态的函数定义在ps/sched.c中：

* **PspReadyThread**：将指定线程插入其优先级对应的就绪队列的队尾，并修改其状态码为

Ready。除了当前运行线程因被抢先而进入就绪状态的情况外，其它任何情况下，都是通过调用此函数使线程进入就绪状态的。注意：当前运行线程因被抢先而进入就绪状态时，应将其插入其优先级对应的就绪队列的队首，因为其时间片尚未用完。

* **PspUnreadyThread**：将指定线程从就绪队列中移除，并修改其状态码为 Zero。无论是

要将就绪线程转入运行状态，还是要结束处于就绪状态的线程，都必须先调用这个函数使线程脱离就绪状态。

* **PspWait**：将当前运行线程插入指定等待队列的队尾，并修改状态码为 Waiting，然后执

行线程调度，让出处理器。通过 PspWait 的第二个参数可以指定等待的时限，如果等待超时则会被系统自动唤醒进入就绪状态。当前运行的线程都是调用这个函数而进入阻塞状态的。

* **PspUnwaitThread**：将阻塞线程从其所在的等待队列中移除，并修改其状态码为 Zero。

不管是因事件条件满足或等待超时而使阻塞线程进入就绪状态，还是要结束处于阻塞状态的线程，都必须先调用这个函数使线程脱离阻塞状态。

* **PspWakeThread**：该函数会先调用 PspUnwaitThread 函数使线程脱离阻塞状态，然后再

调用PspReadyThread 函数使线程进入就绪状态，从而唤醒被阻塞的线程。线程被唤醒后将从函数 PspWait 返回并继续运行。PspWakeThread 的第二个参数将作为被唤醒线程从 PspWait 返回时的返回值。

* **PspSelectNextThread**：线程调度程序函数，使被抢先的线程从运行状态进入就绪状态，

并决定哪个就绪线程应该进入运行状态。任何线程进入运行状态都是这个函数执行的结果。

**（3）核心代码及EOS状态转换跟踪**

1. **控制台输入”loop”执行loop线程**

“loop”命令的命令函数在ke/sysproc.c文件的ConsoleCmdLoop函数中使用LoopThreadFunction函数作为线程函数创建一个优先级为8的线程，函数参数中第一个“0”对应于LPSECURITY\_ATTRIBUTES，即对象描述符，这个结构为很多函数创建对象是提供安全性设置；第二个“0”和 “NULL”为CreateThread的配置参数；LoopThreadFunction是创建loop循环线程。

ThreadHandle = (HANDLE)CreateThread(

0, LoopThreadFunction, (PVOID)StdHandle, 0, NULL);

LoopThreadFunction函数主要执行一个死循环，此时查看“进程线程”窗口系统空闲线程处于就绪状态，控制台派遣线程和所有控制台线程处于阻塞状态，而优先级为8的loop线程处于运行状态。

for (i=0;;i++) {

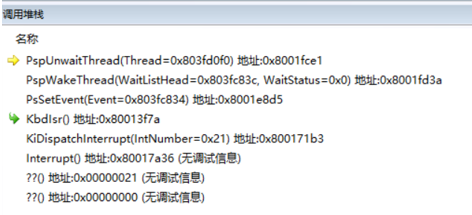
SetConsoleCursorPosition(StdHandle, CursorPosition);

fprintf(StdHandle, "Loop thread ID %d : %u ", ThreadID, i);

}

1. **控制台派遣线程由阻塞状态进入就绪状态**

当在虚拟机窗口中按下一次空格键后，发生键盘中断。调用堆栈情况如图所示。



由键盘中断服务程序KdbIsr函数进入PspUnwaitThread函数，在该键盘中断服务程序的最后会将键盘事件放入缓冲区，然后唤醒控制台派遣线程，由控制台派遣线程将键盘事件派遣到活动的控制台。

IopWriteRingBuffer(Ext->Buffer, &KeyEventRecord, sizeof(KEY\_EVENT\_RECORD));

PsSetEvent(&Ext->BufferEvent);

控制台派遣线程会先调用PspUnwaitThread函数使处于等待状态的线程脱离等待队列转入Zero状态，即退出阻塞状态，后调用PspReadyThread函数进入就绪状态。两个函数的实现代码如下：

* **PspUnwaitThread 函数**

功能描述：使处于等待状态的线程脱离等待队列并转入 Zero 状态。

参数：Thread -- 目标线程对象指针。

VOID

PspUnwaitThread(IN PTHREAD Thread)

{

ASSERT(Waiting == Thread->State);

// 将线程从所在等待队列中移除并修改状态码为Zero。

ListRemoveEntry(&Thread->StateListEntry);

Thread->State = Zero;

// 如果线程注册了等待计时器，则注销等待计时器。

if (STATUS\_TIMEOUT == Thread->WaitStatus) {

KeUnregisterTimer(&Thread->WaitTimer);

}

}

在PspUnwaitThread函数中，参数Thread是目标线程对象指针，首先测试线程是不是在等待队列，如果不是则结束函数的执行，在程序执行时，主要用于调试，使其不容易出现错误或异常。

ASSERT(Waiting == Thread->State);

将线程从所在等待队列中移除并修改状态码为Zero，传递的函数参数&Thread->StateListEntry是线程队列，调用ListRemoveEntry函数来将线程从等待队列中移除。

ListRemoveEntry(&Thread->StateListEntry);

Thread->State = Zero;

* **PspReadyThread 函数**

功能描述：使 Zero 状态或者运行状态的线程转入就绪状态。

参数：Thread -- 线程指针。

VOID

PspReadyThread(PTHREAD Thread)

{

ASSERT(NULL != Thread);

ASSERT(Zero == Thread->State || Running == Thread->State);

// 将线程插入其优先级对应的就绪队列的队尾，并设置就绪位图中对应的位。

// 最后将线程的状态修改为就绪状态。

ListInsertTail(&PspReadyListHeads[Thread->Priority], &Thread->StateListEntry);

BIT\_SET(PspReadyBitmap, Thread->Priority);

Thread->State = Ready;

}

在PspReadyThread函数中，首先测试线程是否处于Zero或Running状态，将线程插入其优先级对应的就绪队列的队尾，并设置就绪位图中对应的位，后修改线程状态为就绪状态。

ListInsertTail(&PspReadyListHeads[Thread->Priority], &Thread->StateListEntry);

BIT\_SET(PspReadyBitmap, Thread->Priority);

Thread->State = Ready;

1. **loop线程由运行状态进入就绪状态**

继续执行，会在PspSelectNextThread函数中的断点中断，查看当前线程为loop线程处于运行状态。函数执行结束后loop线程进入就绪状态。PspSelectNextThread函数实现代码如下：

* **PspSelectNextThread 函数**

功能描述：线程调度函数。当最外层中断服务程序执行完成后，并不是立刻返回被中断运行的线程，而是调用这个函数选择一个合适的线程继续运行（被中断的线程可能继续运行）。

参数：无。

返回值：应执行线程的CPU环境块指针。

PCONTEXT

PspSelectNextThread(VOID)

{

ULONG HighestPriority;

SIZE\_T StackSize;

// 扫描就绪位图，获得当前最高优先级。注意：就绪位图可能为空。

BitScanReverse(&HighestPriority, PspReadyBitmap);

if (NULL != PspCurrentThread && Running == PspCurrentThread->State) {

if (0 != PspReadyBitmap && HighestPriority > PspCurrentThread->Priority) {

// 如果存在比当前运行线程优先级更高的就绪线程，当前线程应被抢先。

// 因为当前线程仍处于运行状态，所以被高优先级线程抢先后应插入其

// 优先级对应的就绪队列的队首。注意，不能调用 PspReadyThread。

ListInsertHead( &PspReadyListHeads[PspCurrentThread->Priority],

&PspCurrentThread->StateListEntry );

BIT\_SET(PspReadyBitmap, PspCurrentThread->Priority);

PspCurrentThread->State = Ready;

} else {

// 当前线程继续运行。

// 注意：中断程序执行时可能换出了当前线程绑定运行的地址空间。

MmSwapProcessAddressSpace(PspCurrentThread->AttachedPas);

return &PspCurrentThread->KernelContext;

}

} else if(0 == PspReadyBitmap) {

// 被中断运行线程处于非运行状态，必须存在一个可运行的就绪线程。

ASSERT(FALSE);

KeBugCheck("No ready thread to run!");

}

if (NULL != PspCurrentThread) {

// 如果当前线程结束了自己则在这里释放线程的内核栈，因为线程在执行时不能

// 释放自己正在占用的栈。

if (Terminated == PspCurrentThread->State) {

StackSize = 0;

MmFreeVirtualMemory( &PspCurrentThread->KernelStack,

&StackSize,

MEM\_RELEASE,

TRUE );

}

// 取消指针 PspCurrentThread 对线程对象的引用。

ObDerefObject(PspCurrentThread);

}

// 选择优先级最高的非空就绪队列的队首线程作为当前运行线程。

PspCurrentThread = CONTAINING\_RECORD(PspReadyListHeads[HighestPriority].Next, THREAD, StateListEntry);

ObRefObject(PspCurrentThread);

PspUnreadyThread(PspCurrentThread);

PspCurrentThread->State = Running;

// 换入线程绑定运行的地址空间。

MmSwapProcessAddressSpace(PspCurrentThread->AttachedPas);

// 返回线程的上下文环境块，恢复线程运行。

return &PspCurrentThread->KernelContext;

}

在PspSelectNextThread函数中，首先扫描就绪位图，获得当前最高优先级

BitScanReverse(&HighestPriority, PspReadyBitmap)

由于控制台派遣线程已经由阻塞状态进入就绪状态，优先级比loop线程高，所以当前线程被抢先。因为当前线程仍处于运行状态，所以被高优先级线程抢先后应插入其优先级对应的就绪队列的队首，所以这里不能用PspReadyThread函数，因为PspReadyThread函数会将线程插入到就绪队列的队尾，和我们想要的效果不一致。在插入后，设置就绪位图中对应的位，最后修改线程状态为Ready。

ListInsertHead( &PspReadyListHeads[PspCurrentThread->Priority],

&PspCurrentThread->StateListEntry );

BIT\_SET(PspReadyBitmap, PspCurrentThread->Priority);

PspCurrentThread->State = Ready;

1. **控制台派遣线程由就绪状态进入运行状态**

继续执行，会在PspUnreadyThread函数中的断点处中断。快速监视可知控制台派遣线程处于就绪状态。当PspUnreadyThread函数执行结束返回PspSelectNextThread函数，将线程状态修改为Running。PspUnreadyThread函数实现代码如下：

* **PspUnreadyThread 函数**

功能描述：取消线程的就绪状态，使线程转入 Zero 状态。

参数：Thread -- 当前处于就绪状态的线程的指针。

VOID

PspUnreadyThread(PTHREAD Thread)

{

ASSERT(NULL != Thread && Ready == Thread->State);

// 将线程从所在的就绪队列中取出，如果线程优先级对应的就绪队列变为空，

// 则清除就绪位图中对应的位。

ListRemoveEntry(&Thread->StateListEntry);

if(ListIsEmpty(&PspReadyListHeads[Thread->Priority])) {

BIT\_CLEAR(PspReadyBitmap, Thread->Priority);

}

Thread->State = Zero;

}

在PspUnreadyThread函数中，首先测试线程状态是否有效，当线程处于就绪状态时，将线程从所在的就绪队列中取出，并修改线程状态为Zero。

ListRemoveEntry(&Thread->StateListEntry);

Thread->State = Zero;

转回PspSelectNextThread函数后，将函数状态修改为Running。

PspUnreadyThread(PspCurrentThread);

PspCurrentThread->State = Running;

1. **控制台派遣线程由运行状态进入阻塞状态**

继续执行，会在PspWait函数中的断点处中断。快速监视可知控制台派遣线程处于运行状态。函数执行结束后，将线程状态改为Waiting。PspWait函数实现代码如下：

* **PspWait 函数**

功能描述：当前线程按照 FCFS 的原则插入指定的等待队列的队尾，线程阻塞等待直到等待

超时或者 PspWakeThread 被调用。

参数：WaitListHead -- 欲加入的等待队列的指针。

Milliseconds -- 有限等待时间(单位 ms)，如果等待时间超出，则被系统自动唤醒并返回 STATUS\_TIMEOUT。如果为 0，则立即返回 STATUS\_TIMEOUT。如果为 INFINIT，则永

久等待直到 PspWakeThread 被调用。

返回值：如果线程等待超时则返回 STATUS\_TIMEOUT，否则返回 PspWakeThread 的第二个

参数 WaitStatus。

STATUS

PspWait(

IN PLIST\_ENTRY WaitListHead,

IN ULONG Milliseconds

)

{

ASSERT(0 == KeGetIntNesting());

ASSERT(Running == PspCurrentThread->State);

ASSERT(0 != PspReadyBitmap);

if(0 == Milliseconds) {

return STATUS\_TIMEOUT;

}

// 将当前线程插入等待队列的队尾并修改线程状态码为Waiting。

ListInsertTail(WaitListHead, &PspCurrentThread->StateListEntry);

PspCurrentThread->State = Waiting;

// 如果不是永久等待，就注册一个用于超时唤醒线程的等待计时器。

if (INFINITE != Milliseconds) {

KeInitializeTimer( &PspCurrentThread->WaitTimer,

Milliseconds,

PspOnWaitTimeout,

(ULONG\_PTR)PspCurrentThread );

KeRegisterTimer(&PspCurrentThread->WaitTimer);

PspCurrentThread->WaitStatus = STATUS\_TIMEOUT;

} else {

PspCurrentThread->WaitStatus = STATUS\_SUCCESS;

}

// 当前线程进入等待状态后需要让出处理器（让权等待），执行线程调度。

PspThreadSchedule();

// zzZ...

// 线程被唤醒继续执行，返回等待结果状态码。

return PspCurrentThread->WaitStatus;

}

在PspWait函数中，首先测试线程状态是否有效，当线程处于运行状态且就绪位图不为空时，将当前线程插入等待队列的队尾并修改线程状态码为Waiting。

ListInsertTail(WaitListHead, &PspCurrentThread->StateListEntry);

PspCurrentThread->State = Waiting;

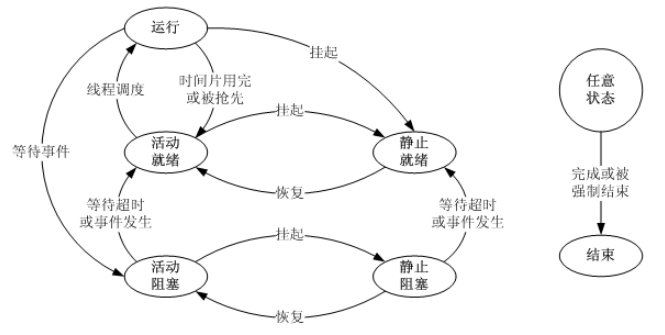
之后触发线程调度功能，中断执行当前已经处于阻塞状态的控制台派遣线程，并将处理器上下文保存到该线程的线程控制块中。

PspThreadSchedule();

**2）为线程增加挂起状态**

（给出实现方法的简要描述、源代码、测试及结果等）

1. **添加挂起状态后线程的状态和转换过程**



1. **简要描述**

分析可知，在命令函数ConsoleCmdResumeThreadd（在 ke/sysproc.c 文件的第 897 行定义）中调用Resume原语，将一个被Suspend原语挂起的线程（处于精致就绪状态）恢复为就绪状态，可以通过以下流程实现resume功能。

a）首先调用 ListRemoveEntry 函数将线程从挂起线程队列中移除。

b）然后调用 PspReadyThread 函数将线程恢复为就绪状态。

c）最后调用 PspThreadSchedule 宏函数执行线程调度，让刚刚恢复的线程有机会执行。

分析：本实验添加了三行代码，先将线程从挂起线程队列移除，将线程恢复为就绪状态，后调用函数使刚刚恢复的线程有机会执行。

1. **源代码**

STATUS

PsResumThread(IN HANDLE hThread)

{

STATUS Status;

BOOL IntState;

PTHREAD Thread;

// 根据线程句柄获得线程对象的指针

Status = ObRefObjectByHandle(hThread, PspThreadType, (PVOID\*)&Thread);

if (EOS\_SUCCESS(Status)) {

IntState = KeEnableInterrupts(FALSE); // 关中断

if (Zero == Thread->State) {

// 在此添加代码将线程恢复为就绪状态

ListRemoveEntry(&Thread->StateListEntry);

PspReadyThread(Thread);

PspThreadSchedule();

//

Status = STATUS\_SUCCESS;

} else {

Status = STATUS\_NOT\_SUPPORTED;

}

KeEnableInterrupts(IntState); // 开中断

ObDerefObject(Thread);

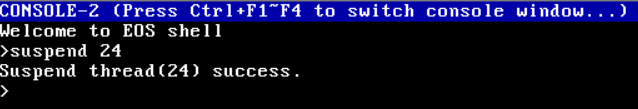
}

return Status;

}

1. **测试及结果**

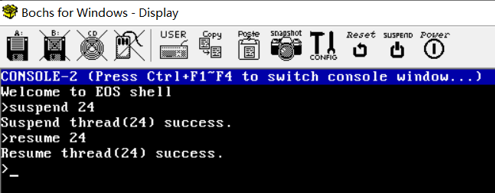
在控制台1输入命令“loop”后回车，loop线程执行计数不停增长。在控制台2输入“suspend 24”使之挂起，控制台显示如下图。



此时控制台1的loop线程的执行计数停止增长。



在控制台2输入命令”Resume 24”后回车，控制台2显示如下图。



切换回控制台1，loop线程的执行计数恢复增长，即Resume原语可以正常工作了。



**结果分析：**

（对本实验所做工作及结果进行分析，包括EOS线程状态及其转换方法的特点、不足及改进意见；结合EOS对线程状态及其转换的相关问题提出自己的思考，分析线程状态转换与线程调度的关系；分析为线程增加挂起状态实现方法的有效性、不足和改进意见，如果同时采用了多种实现方法，则进行对比分析；其他需要说明的问题）

1. EOS线程状态及其转换方法的特点

EOS线程的状态由线程控制块TCB中的State域保存，主要分成三种：就绪、运行、阻塞，由于EOS被设计为运行在单处理器上的多任务操作系统，所以，在任意时刻，最多只能有一个处于运行状态的线程占用处理器，而处于其他状态的线程数量可以为 0 个或多个。同时，在线程转换过程中涉及到的中间状态Zero状态在状态转换图中没有体现，在实际应用时应注意。

线程在不同的状态间转换时，最终都是通过调用ps/sched.c文件中的PspReadyThread、PspUnreadyThread、PspWait、PspUnwaitThread、PspWakeThread、PspSelectNextThread几个函数完成的。

一般情况下从运行到阻塞是线程自己阻塞自己，为了等待IO的发生，而被唤醒则需要外部唤醒，即通过操作系统的线程调度，为线程增加的挂起状态可以成功实现线程的挂起以及重新唤醒。

1. 线程状态转换与线程调度的关系

线程有三种基本状态，并可以在这三种状态之间进行转换。线程的状态由于不同的事件发生而相互转化。线程的调度可以分为可剥夺式和不可剥夺式，并且有线程调度算法，包括先来先服务调度算法，短作业优先调度算法，最高响应比优先调度算法，时间片轮转法，多级反馈队列等。在调度的过程中线程的状态发生改变。

1. 为线程增加挂起状态实现方法分析

在增加挂起状态时，首先调用函数将线程从挂起线程队列中移除，然后调用函数将线程恢复为就绪状态，最后调用函数执行线程调度，让刚刚恢复的线程有机会执行。

为确保完整性，上述所有的转换过程是不应该被打断的，即这些转换过程都是原语操作，而在代码中原语操作的实现是通过开关中断实现的。同时注意到，在EOS实现挂起和恢复部分的代码中，存在一些不足。PsSuspendThread 函数只能挂起处于活动就绪状态的线程，对于挂起处于运行或者阻塞状态的线程的工作还没有完成。

EOS 目前所实现的 suspend 原语在将线程挂起时，挂起线程的状态为 Zero，这违背了 EOS 定义 Zero 状态的初衷。显然，应该在枚举类型 THREAD\_STATE 中定义一个新的项用来表示静止就绪状态，并需要对 PsSuspendThread 函数进行适当修改。此外，处于阻塞状态和运行状态的线程也应该可以被挂起并被恢复，可以尝试继续修改 PsSuspendThread 函数和 PsResumeThread 函数来补充这部分功能。