

简单SIGNAL机制与抢占式用户级多线程

注: 本文使用的图片均来自参考资料[1]。见文章末尾。

SIGNAL Overview

信号机制是一种用于进程间通信的轻量级机制，允许操作系统和进程之间传递异步事件通知。信号可以用于报告各种事件，例如进程的终止、用户请求或硬件异常。

POSIX规定的信号标准接口有

```
void (*signal(int sig, void (*handler)(int)))(int);
int sigaction(int sig, const struct sigaction *act, struct sigaction
*oldact);
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
...
```

YuanShen OS并没有实现POSIX的标准接口。最开始我们只是实现了一个简单的系统调用sigalarm，它来源于MIT6.081的课程实验任务，能够设置一个无参信号处理程序，而一个virtual timer会不断的为进程发送信号从而驱动这个handler。

1. 信号栈帧和用户级多线程实现

然而后来我们注意到这个系统调用通过简单的拓展就可以支持非合作式的，由虚拟定时器驱动的用户级多线程。而一切的关键在于借鉴POSIX标准接口里的struct sigaction, 为信号处理程序增添一个神秘的参数ucontext:

```
struct sigaction {
    void      (*sa_handler)(int);
    void      (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t  sa_mask;
    int       sa_flags;
    void      (*sa_restorer)(void);
};
```

Advanced Signal Handling Support

- UNIX also provides more advanced signal handling:
`int sigaction(int sig, const struct sigaction *act, struct sigaction *oact)`
- `sigaction` struct specifies various details, including the kind of handler function:
 - Either the simple `void handler(int sig)` as before...
 - Or, a more advanced handler function:
`void sigact(int sig, siginfo *info, void *ctxt)`
- The `siginfo` struct includes many details about signals
 - e.g. sending process ID, memory address that caused fault, etc.
- `ctxt` points to a `ucontext_t` structure
 - A platform/architecture-dependent machine context, containing the CPU state of the user process, when it was interrupted by signal
 - Facilitates e.g. user-space threading libraries

当`sa_flags`被设置为`SA_SIGINFO`，那么信号处理程序就会是`sa_sigaction`。

```
void
handler(int sig, siginfo_t *info, void *ucontext)
{
    ...
}
```

我们注意到它的第三个参数是个`void*`，实际上它是由`ucontext_t*`转型过来的，这个东西是实现用户级多线程最为关键的部分。

根据xv6的设计，用户在被时钟中断的时候会把上下文存储在`trapframe`，它只能在内核态被访问。

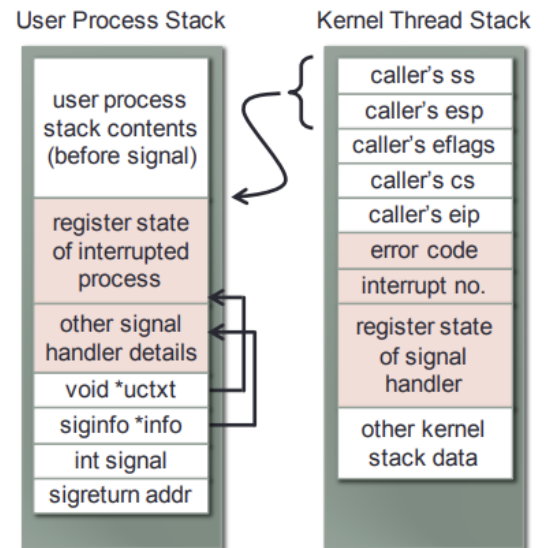
实现抢占式调度的用户多线程的关键就是把这个上下文想办法保存到用户进程的地址空间，这样用户级线程的上下文可以由用户进程的调度器全程管理，但是这该如何实现？

1.1 ucontext_t

根据caltech的课程CS124的课件，我们找到了完美的方法。我们只需要在内核态的时候下调用户的栈指针，然后把`ucontext`这个保存着用户态进程上下文的结构体从内核栈压到用户栈上，正确创立信号栈帧，在函数的第一个入参处存入指向这个`ucontext`的指针，并重设进程返回地址为`p->alarmhandler`。

Delivering Signals (10)

- One last component needed for the stack frame:
 - The return address for when the signal handler returns
- Need the signal handler to return to the kernel:
 - Allow kernel to complete final signal-handling tasks, and restore the interrupted process' original context
- The kernel inserts address of code to invoke the **sigreturn** syscall
 - i.e. a wrapper to code that executes "mov NR_sysreturn, %eax; int \$0x80"
- **sigreturn** has a single purpose:
 - Perform the final task of restoring the interrupted process' CPU context from its location on the stack



那么当信号处理程序运行的时候，它就有了指向ucontext的首地址的指针，如图所示。

```
//in usertrap(void), trap.c
uint64 *user_sp = (uint64*) p->trapframe->sp;
    struct u_context_t *user_context_ptr = (struct u_context_t *)
(user_sp - sizeof(struct u_context_t) / sizeof(user_sp));
    saveContext(p, user_context_ptr);
    p->signal_ucontext_ptr = (uint64) user_context_ptr;
    p->trapframe->sp = (uint64) user_context_ptr;
    p->trapframe->a0 = (uint64) user_context_ptr;
    p->trapframe->epc = (uint64)myproc()->alarmhandler;
```

那么我们再来看看YuanShen OS设置的ucontext到底长什么样

```
struct u_context_t{
uint64 ra;
uint64 sp;
uint64 s0;
uint64 s1;
uint64 s2;
uint64 s3;
uint64 s4;
uint64 s5;
uint64 s6;
```

```
uint64 s7;
uint64 s8;
uint64 s9;
uint64 s10;
uint64 s11;
uint64 t0;
uint64 t1;
uint64 t2;
uint64 t3;
uint64 t4;
uint64 t5;
uint64 t6;
uint64 a0;
uint64 a1;
uint64 a2;
uint64 a3;
uint64 a4;
uint64 a5;
uint64 a6;
uint64 a7;
uint64 epc;
};
```

如我们所料，它就是进程保存在trapframe的上下文的副本。

1.2 信号相关的数据结构

为了支持简易的信号sigalarm，我们在进程控制块内加入了以下字段。inAlarm是pending位，表示信号尚未处理完。限于时间约束，YuanShen OS没有实现信号屏蔽机制，也没有办法处理多种信号。

```
//proc.h
int alarm_int;           // alarm interval
void (*alarmhandler)();  // alarm handler
int ticksSinceLastAlarm; // the number of ticks that passed from last
alarm

uint64 signal_ucontext_ptr;
// struct trapframe *alarmframe; //if the process is alarmed, save its
// trapframe here since the old context will
// be overwritten by the alarm handler.
// we will restore the trapframe when
// handler returns.
int inAlarm;              //if the process is executing the handler,
// then the counting should be stoped temporarily.
```

1.3 抢占式调度用户级多线程的数据结构

YuanShen OS支持的信号机制的设计是专门为实现用户级多线程而服务的，我们来简单看看如何搭配先前设立的sigalarm建立用户级线程池，核心的数据结构如下文所示：

```

struct thread{
    int id;
    struct threadContext context;           /*THREAD CONTEXT SAVED HERE*/
    int state;                             /*FREE, RUNNING OR RUNNABLE*/
};
// We allocate thread struct lazily.
struct threadpool{
    uint32 thread_count;                   /*HOLDS THE NUMBER OF
ACTIVE THREADS*/
    struct thread *queue[MAXTHREADNUM];    /*HOLDS POINTERS TO ALL
THREAD STRUCT*/
    int currentThreadId;
};

```

其中threadContext和ucontext如出一辙。信号处理程序为void handler(void* u_context_ptr),与我们先前陈述一致。它的唯一的作用就是把ucontext保存进线程池,并把ucontext换成另一条线程的上下文。下文展示了用户级多线程实现所需的信号处理程序的代码:

```

void handler(void* u_context_ptr){
    struct threadContext *tc = (struct threadContext*)(u_context_ptr);
    pool.queue[pool.currentThreadId]->context = *tc;
    uthread_yield();
    *tc = pool.queue[pool.currentThreadId]->context;
    sigreturn();
}

```

在上文这段代码中,系统调用sigreturn被调用的时候,这个刚刚被替换的ucontext里的上下文就会写回进程的trapframe,如此一来进程再回到用户态的时候使用的就是另一个线程的上下文。sigreturn的代码如下文所示:

```

uint64 sys_sigreturn(void){
    struct proc *p;
    p = myproc();
    if(p->inAlarm == 1){
        p->inAlarm = 0; //clearing the blocking bit
        p->ticksSinceLastAlarm = 0; // clearing the pending bit
        //since we saved the context of uthread in the user stack, we need to
        load it back to the trapframe!
        restoreContext(p, (struct u_context_t *)p->signal_ucontext_ptr);
        return 0;
    }else {
        panic("sys_sigreturn panic!\n");
    }
    return -1;
}

```

如上文所示,它的职能就是为信号处理程序进行扫尾,清除掉信号的pending位,并且恢复进程在被信号处理程序打断前的上下文。

2. 测试

2.1 testalarm测试

在testalarm这个用户程序中，我们会为进程分别设置三个信号处理程序。这三个信号处理程序会不断地打印alarm!并且将一个全局变量的值自增，不过三个信号处理程序的速度不同。测试的内容就是比较全局变量的值是否符合理论结果。测试结果如下图所示：

```
xv6 kernel is booting

hart 1 starting
init: starting sh
$ testalarm
test0 start
.....alarm!
test0 passed
test1 start
.alarm!
alarm!
alarm!
.alarm!
alarm!
alarm!
.alarm!
alarm!
alarm!
.alarm!
test1 passed
test2 start
.....alarm!
test2 passed
test3 start
test3 passed
```

我们可以看到testalarm如我们的预期打印出了正确的结果，全局变量的值符合理论计算的结果。

2.2 testThread测试

在testThread.c这个用户程序，我们初始化了十条用户级线程，他们每隔一段时间就会打印自己的线程号。在每个线程return前，我们要求它们打印整个线程池。

这是运行的结果，可以发现线程切换正确的进行了，同时可以观察到线程有序且正确的退出了线程池, 运行结果如下图所示：

```
Thread[7] READY
Thread[8] READY
Thread[9] READY
Thread 2 started.
tid:2 print
tid:2 print
tid:2 print
tid:2 print
thread[2] context saved!
thread[3] context restored!
tid:3 print
Thread[0] READY
Thread[10] READY
Thread[9] READY
Thread[3] FINISHED
Thread[4] READY
Thread[5] READY
Thread[6] READY
Thread[7] READY
Thread[8] READY
Thread 3 started.
tid:3 print
tid:3 print
tid:3 print
tid:3 print
thread[3] context saved!
thread[4] context restored!
tid:4 print
Thread[0] READY
Thread[10] READY
Thread[9] READY
Thread[8] READY
Thread[4] FINISHED
Thread[5] READY
Thread[6] READY
Thread[7] READY
```

我们可以看到各用户级线程间发生了有序的上下文切换，结束执行的线程被标记为FINISHED，并从线程池中被清除。

参考资料

[1] California Institute of Technology. (2016). CS124 Lecture 15: Signal Handling.
<http://courses.cms.caltech.edu/cs124/lectures-wi2016/CS124Lec15.pdf>