Kernel Reading Notes#

A place to record, observations and analysis of the kernel code which has not yet made it into code comments.

How to use this page#

- collect notes together by file name, or by functional-area
- create a level-3 wiki header for the file name. ex: `!ker/nano_sched.c`, if your comments are about specific files
- or create a level -3 wiki header for a new functional area, if your comments are not about a sepecific file.
- if this page gets big, split it into a per-file page and a functional-area page. Then split those pages into per file and per-topic pages.

Per-file Comments#

---

nano_sched.c#

... add notes

1.select_thread_default it will find the first highest priority thread in the ready queue and remove it from the ready queue.

Quite simple!!!

2.mark_running_default Basically it only sets the thp to kernel cpu active thread although we are still running in old thread aspace most likely.

And finally when we return from kernel to userland `__ker_exit` will handle whether address space is changed, process is changed so only `actives[KERNCPU]` changed it will help you understand how `__ker_exit` works.

This is the UP version and it will make things easier.

```
static THREAD * rdecl
select_thread_default(THREAD *act, int cpu, int prio) {
   THREAD      *thp;
   // act may be NULL so we get the dpp from actives
   DISPATCH    *dpp = actives[KERNCPU]->dpp;
   int         hipri;

   hipri = DISPATCH_HIGHEST_PRI(dpp);
   if(hipri < prio) return NULL;

   thp = DISPATCH_THP(dpp, hipri);
   if(thp == NULL) {
      // May occur if there is nothing, not even idle
      return NULL;
   }

   LINK3_REM(DISPATCH_LST(dpp, (thp)->priority), (thp), THREAD);

   /* If that list is now empty, clear the readybit.   */
   if (DISPATCH_THP(dpp, (thp)->priority) == NULL) {
```

```
        DISPATCH_CLR(dpp, (thp));
    }
    return thp;
}

static void rdecl
mark_running_default(THREAD *new) {
    THREAD  *old;

    old = actives[KERNCPU];
    actives[KERNCPU] = new;

    new->state = STATE_RUNNING;
    new->runcpu = KERNCPU;

    _TRACE_TH_EMIT_STATE(new, RUNNING);

    VALIDATE_NEEDTORUN(new)

    WAIT_FOR_CLOCK_DONE(old);

    //We didn't account for any time to the previously running
    //thread. Up the running time by a bit so that watchdog programs
    //see that this thread/process is getting a chance to run.
    //We can be smarter in the future about the value that we add.
    //RUSH3: This actually a bug - we need to do this for any scheduling
    //RUSH3: algorithm, not just round robin
    if (kerop_microaccount_hook != NULL) {
        kerop_microaccount_hook(old, new);
    } else {
        if(IS_SCHED_RR(old) && (!old->schedinfo.rr_ticks)) {
            old->running_time += 1;
            old->process->running_time += 1;
        }
    }
}
```

x86/kernel.S#

... add notes

```
__common_ker_entry:
#ifdef VARIANT_instr
    SAVE_PERFREGS 0
#endif

#ifdef  VARIANT_smp

    //NOTE: If the register used to save the kernel call number is
    //changed, there's code in 'force_kernel' that needs to change as
    //well. See the "NOTE:" comment in that routine.
    mov     %eax,%esi   // save kernel call number /* in userspace every kernel call save kernel call number in eax, see
ker_call_table*/
acquire_kernel_attempt:
    sti
    // Wait for need to run to clear if we're not on the right CPU.
1:
    cmpl    $0,need_to_run                  /* need_to_run thread == 0? */
```

```
    jz      3f                      /* need_to_run==0, that means no thread need to run and preempt me, just go ahead, jump to 3 */
    cmpl    %ebp,need_to_run_cpu    /* need_to_run!=0, compare current cpu == need_to_run_cpu? */
    je      3f                      /* need_to_run_cpu is current cpu, jump to 3*/
    pause                           /* pause, it was suggested by Intel for spin-loop wait, in Intel Xeon, P4 and dual cores we have
    jmp     1b

    // See if anybody else is in the kernel
3:
    mov     inkernel,%eax
    test    $INKERNEL_NOW,%eax      /* and immed32, eax, set SF,ZF,PF according to result*/
    jnz     1b                      /* if (eax & INKERNEL_NOW) jmp 1b*/

    cli                             /* disable my cpu irq*/
end_acquire_kernel_attempt:

    mov     %eax,%edi               /* mov inkernel to edi */
    andl    $0x00ffffff,%edi        /* get edi lower 24 bits*/
    mov     %ebp,%ecx
    shl     $24,%ecx                /* get cpunum, ebp lower 8 bits represent cpunum */
    orl     %edi,%ecx       / Set cpunum
    orl     $INKERNEL_NOW,%ecx                              /* cpunum|inkernel|INKERNEL_NOW*/
    lock; cmpxchg   %ecx,inkernel                           /* lock bus and xchg ecx and inkernel*/
    jnz     acquire_kernel_attempt                          /* if ecx is not same with inkernel, again */
    // We are the kernel
    mov     %esi,%eax   // restore kernel call number
#else
    LOCKOP
    orl     $INKERNEL_NOW,inkernel  / In the kernel (restartable)       /* UP case so inkernel|INKERNEL_NOW*/
#endif

    sti                                                     /* enable my cpu irq*/
    cld

    mov     %eax,SYSCALL(%ebx)
    mov     TFLAGS(%ebx),%ecx
    and     $~(_NTO_TF_KERERR_SET+_NTO_TF_BUFF_MSG+_NTO_TF_KERERR_LOCK),%ecx
    mov     %ecx,TFLAGS(%ebx)
    cmp     $__KER_BAD,%eax
    jae     bad_func
    push    %edx
    push    %ebx
#if defined(VARIANT_instr)
    call    *_trace_call_table(,%eax,4)
#else
    call    *ker_call_table(,%eax,4)
#endif
    / assuming that none of the kernel routines modify the 'act' parm
    / on the stack
    pop     %ebx

    test    %eax,%eax
    jge     set_err


aspace_switch:
/*actives->aspace_prp is in eax*/
/*if it is 0 then jump back*/
    test    %eax,%eax
    je      aspaceret
```

```
    push    %ecx
    push    %eax
/*push vmm_aspace(PROCESS *actprp, PROCESS **pactprp)*/
/*ecx is pactprp and eax is actprp*/
/*call memmgr.aspace vmm_aspace*/
    call    *MEMMGR_ASPACE+memmgr   / Switch address spaces (ebx is not modified)
/* restore stack*/
    add     $8,%esp
    jmp     aspaceret

prp_switch:
/*yzhao actives->process is in eax so set actives_prp*/
    mov     %eax,SMPREF(actives_prp,%ebp,4)
/*yzhao move cpupageptr to ecx*/
    mov     SMPREF(cpupageptr,%ebp,4),%ecx
/*yzhao move actives->process->pls to edx*/
    mov     PLS(%eax),%edx
/*yzhao move actives->process->pls to cpupageptr->pls*/
    mov     %edx,CPUPAGE_PLS(%ecx)
/*yzhao     actives->process->debugger to eax*/
    mov     DEBUGGER(%eax),%eax
    test    %eax,%eax
/*yzhao nobody is debugging actives then jump back to prpret*/
    je      prpret
    andl    $~SYSENTER_EFLAGS_BIT,REG_OFF+REG_EFL(%ebx)     // Single stepping through the SYSEXIT sequence is n
    push    %ebx
/*yzhao call cpu_debug_attach_brkpts*/
    call    *debug_attach_brkpts    / Possibly yes so call to remove soft breakpoints (prp in EAX)
    pop     %ebx
    jmp     prpret
```

...add notes

ppc/kernel.s#

...add notes

Functional Area Comments#

Specret #

...add notes

interrupt handing#

...add notes

SMP mutual exclusion #

...add notes

How Instrumental kernel part works#

How Kernel part works#

Basically it looks like it will replace kernel entry with its own code then it will call original code like a hacker:) and normally if users don't want log any trace events procnto-instr shouldn't have too much performance dropping, then how it is implemented? 1. If we were using procnto-instr then in kernel.S:`__ker_entry` will be "call *_trace_call_table(,%eax,4) and default _trace_call_table is an array contained same contents with normal ker_call_table so procnto-instr will not have any performance effect when we didn't set up to log any trace events.(even you log your user events in kernel). You should know in ker_call_table the functions are all kernel call handlers like thread_destroy and its corresponding kernel handling function is ker_thread_destroy. Code in ker_call_table.c shows that default _trace_call_table has same contents with ker_call_table. In kernel entry

```
__ker_entry:
...
#if defined(VARIANT_instr)
   call    *_trace_call_table(,%eax,4)
#else
   call    *ker_call_table(,%eax,4)
#endif
```

In ker_call_table.c

```
int kdecl (* ker_call_table[])() ={
   MK_KERTABLE(ker)
};

int kdecl (* _trace_ker_call_table[])() ={
   MK_KERTABLE(_trace_ker)
};


int kdecl (* _trace_call_table[])() ={
   MK_KERTABLE(ker)
};

const int ker_call_entry_num = NUM_ELTS(_trace_call_table);
```

Then how various trace events are logged? For example: ker_thread_destroy? Obviously there is another array we do see: _trace_ker_call_table, take a look what is that? In this array all functions start with _trace_ker then ker_thread_destroy will be _trace_ker_thread_destroy so _trace_ker_thread_destroy->_trace_emit_in_w

```
static int _trace_emit_in_w(THREAD *act, void* kap, uint32_t num, uint32_t* arg_arr, uint32_t len)
{
   int h_r_v=1;

   _TRACE_ENTERCALL(act->syscall, num);
   if(_TRACE_GETSTATE(act->syscall)==2&&_TRACE_CHK_ENTRY(act, _TRACE_ENTER_CALL, num)) {
      uint32_t header=_TRACE_MAKE_CODE(
                              RUNCPU,
                              NULL,
                              _TRACE_KER_CALL_C,
                              num
                             );

      if(trace_masks.class_ker_call_enter_ehd_p) {
         h_r_v = exe_pt_event_h_buff(
```

```
                                trace_masks.class_ker_call_enter_ehd_p,
                                header,
                                _TRACE_DER_PTR(act->process,pid),
                                act->tid+1,
                                (void*) arg_arr,
                                sizeof(uint32_t)*len
                               );
        }
        if(trace_masks.ker_call_enter_ehd_p[num]&&h_r_v) {
           h_r_v = h_r_v && exe_pt_event_h_buff(
                                    trace_masks.ker_call_enter_ehd_p[num],
                                    header,
                                    _TRACE_DER_PTR(act->process,pid),
                                    act->tid+1,
                                    (void*) arg_arr,
                                    sizeof(uint32_t)*len
                                   );
        }
        if(h_r_v) {
           add_trace_buffer(header, arg_arr, len);
        }
     }
     if(h_r_v==2) _TRACE_SETEXIT(act->syscall);
     _TRACE_OUTSTATE(act->syscall);
     _TRACE_SETRETSTAT(h_r_v, act);

     return (h_r_v);
}

#define _TRACE_IN_W_0PTR(n,R,S) \
     return (_trace_emit_in_w(act, kap, n, R, S))

int kdecl _trace_ker_thread_destroy(THREAD *act, struct kerargs_thread_destroy *kap)
{
     if(_TRACE_CALL_ARG_WIDE&trace_masks.ker_call_masks[__KER_THREAD_DESTROY]) {
        uint32_t arg_arr[3];

        arg_arr[0] = (uint32_t) kap->tid;
        arg_arr[1] = (uint32_t) kap->priority;
        arg_arr[2] = (uint32_t) kap->status;

        _TRACE_IN_W_0PTR(__KER_THREAD_DESTROY, arg_arr, 3);
     } else {
        _TRACE_IN_F_0PTR(__KER_THREAD_DESTROY, kap->tid, (uint32_t) kap->status);
     }
}
```

2. trace_event kernel call Basically this function call is used to register what kinds of events you are interested in and do some setting because it is the only kernel call userspace talks with kernel.

If (you use procnto) then it will return

else in ker_trace_event will handle all and most important it will hack all of kernel calls, allocating internal buffers...