```c
#include <errno.h>
#include <sys/helix_config.h>
#include <sys/kernel.h>
#include <sys/io.h>
#include <stdio.h>
#include <malloc.h>
#include <daemons.h>
#include <sys/time.h>
#include <signal.h>
#include <pthread.h>
#include <stdlib.h>
#include <trace.h>
#include <mc.h>

/*! \file
 * \brief Implementation of all kernel functions.
 *
 * Note that all function starting with k should only be called from within the kernel
 context: priviledged mode, in a syscall.
 * Do not do a syscall from within a syscall.
 */

#define DBLED_ALIVE     0
#define DBLED_INIT      1
#define DBLED_SCHEDULE  2
#define DBLED_PANIC     3
#define DBLED_NEWPROC   4

#ifdef KERNEL_DEBUG
#define DEBUG_ON(led)   led_on(led)
#define DEBUG_OFF(led)  led_off(led)
#define DEBUG_ALLOFF()  leds_value(0)
#define DEBUG(str,a...) printk("KERNEL %d: " str "\n",GetProcID(),##a)
#define DEBUG0(a...)    do{if(GetProcID()==0)DEBUG(a);}while(0)
#define TDEBUG_SIZE 13
#define TDEBUG(a...)
    \
    do{
        \
        struct timeval now;
            \
        const unsigned int  ts=(1<<TDEBUG_SIZE)*GetProcID(),
            \
                            ts_mask=((1<<TDEBUG_SIZE)*NUM_PROCESSORS-1)&~((1<<TDEBUG_SIZE)-1);
                                \
        while(gettimeofday(&now,NULL)==0&&(now.tv_usec&ts_mask)!=ts);
            \
        DEBUG(a);
            \
    }while(0)

#else
#define DEBUG_ON(led)
#define DEBUG_OFF(led)
#define DEBUG_ALLOFF()
#define DEBUG(str,a...)
#define DEBUG0(a...)
#define TDEBUG(a...)
#endif

#define ASSIGN_CONST_PTR(type,var,val)  *(type**)&(var)=(val)
#define kSetOSState(state)              *(os_state_t*)&os_state=state

#ifdef KERNEL_SIGNALS
#define kIsInSigHandler(pt)             (pt->sig_havetriggered!=0)
#else
#define kIsInSigHandler(pt)             false
```

```c
#endif

#define td_proc (&kernel_context->proc_table[0])


#ifdef __cplusplus
extern "C" {
#endif

extern void kInitSignalContext(proc_context_t* sig_context, proc_context_t* proc_context,char*
triggered,int* havetriggered);
extern void kResetSignalContext(proc_context_t* sig_context);
static void kInitKernel(kernel_boot_param_t* param);
static void kStartTaskDaemon(kernel_boot_param_t* param);
static void kStartScheduling() __attribute__ ((noreturn));

#ifdef SCHEDULING_IN_LOCAL_RAM
extern void* _text_schedule;
extern void* _text_schedule_end;
extern void* _text_schedule_base;
#endif

#ifdef STACK_CHECK
extern void* _stack_end;
#endif

//////////////////////////////////////////////////////
// Types
//////////////////////////////////////////////////////

#define STACK_PROTECTION_VALUE  0xdeadbeef

#ifdef KERNEL_CONTEXT_LOCAL
/*! \brief The one and only kernel context */
kernel_context_t kernel_contexts[1];
#else
/*! \brief All kernel contexts, located in shared memory */
kernel_context_t kernel_contexts[NUM_PROCESSORS] __attribute__ ((section
(".bss.kcontext"),aligned(CACHELINE_ALIGN)));
#endif

/*! \brief The name of the kernel to print during boot */
static const char* kernel_name __attribute__((unused)) = KERNEL_NAME;

/*! \brief Flags to influence scheduling */
typedef struct {
    pthread_sch_t* pthread_sch;
    volatile int just_did_work;
    volatile int have_work;
    int was_idle;
#ifdef SCHEDULE_SLOTS
    volatile int skip_taskdaemon;
    volatile unsigned int remaining_slots;
#endif
} sch_t;

extern sch_t sch;

#define DID_WORK_NO     0
#define DID_WORK_MAYBE  1
#define DID_WORK_YES    2

#define TRACE ((trace_t*)kGetHookTable()[HOOK_TRACE])

//////////////////////////////////////////////////////
// newlib support
//////////////////////////////////////////////////////
```

```c
#include <reent.h>
// there is a typo in the _REENT_INIT_PTR macro:
#define __sf_fake_stdio __sf_fake_stdin

/*! \brief All impure data (as defined by newlib), reserved for kernel */
struct _reent kernel_impure_data __attribute__ ((section (".bss.proc")));
/*! \brief Pointer to global_impure_data */
struct _reent *_CONST __ATTRIBUTE_IMPURE_PTR__ _kernel_impure_ptr = &kernel_impure_data;

//////////////////////////////////////////////////////
// Kernel lifetime
//////////////////////////////////////////////////////

#include <stdio.h>

/*!
 * \brief Entry point of C-code of the kernel
 *
 * After initialization of crt0.S and crtbegin.S, this function is called.
 * It will handle all initialization routines of the kernel and will eventually
 * start the task daemon.
 *
 * The master core will usually be invoked without additional arguments, a
 * slave core will receive a #kernel_boot_param_t* that contains a
 * reference to a mailbox to notify a successful boot.
 *
 * \param arg NULL or #kernel_boot_param_t*
 */
void kBoot(kernel_boot_param_t* arg) __attribute__ ((noreturn));
void kBoot(kernel_boot_param_t* arg){
    kSetOSState(OS_BOOT);

    kernel_boot_param_t param;
    param.notify_booted=NULL;
    param.arg=NULL;
    // copy to local variable; InitKernel will overwite the _fifomem
    if(arg!=NULL)
        param=*((kernel_boot_param_t*)arg);

    DEBUG_ALLOFF();
    InitHardware(param.arg);
    DEBUG_ON(DBLED_ALIVE);

    DEBUG_ON(DBLED_INIT);
    kInitKernel(&param);

    kStartTaskDaemon(&param);
    DEBUG_OFF(DBLED_INIT);

    kStartScheduling();
    // unreachable
}

void Disclaimer(){
    printk("Built for %d processors, running %d processes each.
\n",NUM_PROCESSORS,MAX_PROCESSES);
#ifdef KERNEL_BUILD_FLAGS
    printk("Kernel build flags: " KERNEL_BUILD_FLAGS "\n");
#endif
#if defined(SVNREV) && SVNREV>0
    printk("svn:r" STRINGIFY(SVNREV) " ");
#endif
#ifdef EDKVERSION
    printk("edk:" EDKVERSION " ");
#endif
#ifdef BUILDDATE
```

```
    printk("date:" BUILDDATE " ");
#endif
#ifdef GCCVERSION
    printk("cc:" GCCVERSION " ");
#endif
#ifdef _NEWLIB_VERSION
    printk("newlib:" _NEWLIB_VERSION " ");
#endif
    printk("\nCopyleft Jochem Rutgers, 2012\n\n");
}

#ifdef KERNEL_CHECKSUM
extern unsigned int _ddrbase;
unsigned int _chksum __attribute__((unused,section(".chksum")));

static void kChecksum(){
#ifdef KERNEL_DEBUG
    _puts("...\nChecksum over ");
    _putp(&_ddrbase);
    _puts("-");
    _putp(&_chksum);
    _puts(" is ");
    _putn(_chksum);
    _puts(": ");
#endif

    if(_chksum==0){
#ifdef KERNEL_DEBUG
        _puts("ignored\n");
#endif
        return;
    }

    unsigned int *w=&_ddrbase;
    unsigned int sum=0;
    for(int i=(int)(((unsigned int)&_chksum-(unsigned int)&_ddrbase)/sizeof(unsigned
int))-1;i>=0;i--)
        sum+=w[i];
    if(sum!=_chksum){
        // invalid checksum
        _puts("Invalid checksum ");
        _putn(sum);
        _puts("\n");
        kReset();
    }
#ifdef KERNEL_DEBUG
    else
        _puts("ok\n...");
#endif
}
#endif

#ifdef KERNEL_DEBUG
#define KERNEL_PRINT_PROGRESS       if(GetProcID()==0)outbyte(kernel_name[init_progress++])
#define KERNEL_PRINT_PROGRESS_END
if(GetProcID()==0)do{printk("%s\n",&kernel_name[init_progress]);
if(GetProcID()==0)Disclaimer();}while(0)
#else
#define KERNEL_PRINT_PROGRESS
#define KERNEL_PRINT_PROGRESS_END
#endif

/*!
 * \brief Basic initialization of the kernel
 *
 * It will initialize the heaps, kernel context, and process table.
 *
```

```
 * \param param the same argument as the one main(int,char**) got
 */
static void kInitKernel(kernel_boot_param_t* param){
    kSetOSState(OS_INIT);
#ifdef KERNEL_DEBUG
    int init_progress=0;
#endif
    KERNEL_PRINT_PROGRESS;

#ifdef KERNEL_CHECKSUM
    if(GetProcID()==0)
        kChecksum();
    KERNEL_PRINT_PROGRESS;
#endif

    // Speed up the processor: enable cache
    InitICache();
    KERNEL_PRINT_PROGRESS;

#ifdef KERNEL_DEBUG
    if(sizeof(kernel_context_t)%CACHELINE_ALIGN!=0)
        kPanic("Kernel context size not aligned to cacheline!");
    if(((unsigned int)kernel_contexts)%CACHELINE_ALIGN!=0)
        kPanic("Kernel context not aligned to cacheline!");
#endif
#ifdef HELP_ME_DEBUGGING
    InvalidateDCache();
#else
    InitDCache();
#endif
    KERNEL_PRINT_PROGRESS;

#ifdef SCHEDULING_IN_LOCAL_RAM
    // Copy .text.schedule section from DDR to local ram.
    // This should make context switching faster, because no DDR and caches are used
    size_t _text_schedule_size=(size_t)((unsigned int)&_text_schedule_end-(unsigned
int)&_text_schedule);
    if(GetLocalMemSize()<(unsigned int)&_text_schedule_end)
        kPanic("Local memory not large enough for scheduling code");
    memcpy(&_text_schedule,&_text_schedule_base,_text_schedule_size);
    KERNEL_PRINT_PROGRESS;
#endif

    // Initialize context
#ifdef KERNEL_CONTEXT_LOCAL
    kernel_context=&kernel_contexts[0];
#else
    kernel_context=&kernel_contexts[GetProcID()];
#endif
    kernel_context->hooks=NULL;
    KERNEL_PRINT_PROGRESS;

    // Set PID to 0, so we mimic being the TaskDaemon.
    // Otherwise, we cannot create processes.
    ASSIGN_CONST_PTR(proc_table_t,current_proc,td_proc);
    td_proc->pid=0;

    // newlib support: reentrancy structs
    KERNEL_PRINT_PROGRESS;
    _REENT_INIT_PTR(_kernel_impure_ptr);
    KERNEL_PRINT_PROGRESS;
    _impure_ptr=_kernel_impure_ptr;
    KERNEL_PRINT_PROGRESS;

    // Initialize memory
    HEAP* p=NULL;
    malloc_init(&p,kernel_context->heap,PROC_HEAP);
```

```
    KERNEL_PRINT_PROGRESS;
    if(!p)kPanic("Initialization of processor heap failed.");
    KERNEL_PRINT_PROGRESS;

    p=NULL;
    malloc_init(&p,_fifomem,GetFifoMemSize()-TASK_DAEMON_FIFO_SIZE-RING_CONSISTENCY_SIZE);
    KERNEL_PRINT_PROGRESS;
    if(!p)kPanic("Initialization of FIFO memory heap failed.");
    KERNEL_PRINT_PROGRESS;

    if(!(kernel_context->hooks=(hook_t*)lcalloc(HOOKS,sizeof(hook_t))))
        kPanic("Cannot initialize hooks.");
    KERNEL_PRINT_PROGRESS;

    // Shared heap will not be managed from the kernel, so do the next call from some master
daemon:
    // malloc_init([fake pointer],__sheap,SHARED_HEAP_SIZE);

    KERNEL_PRINT_PROGRESS_END;

    // Basic memory management is up and running, so it is safe to use printf now.
    TDEBUG("Booting kernel on processor %d...",GetProcID());
    DEBUG0("boot param      : %p",param);
    if(param&&param->notify_booted)
        DEBUG0("boot notify     : %p",param->notify_booted);
    DEBUG0("kernel_context  : %p",kernel_context);

    kernel_stack=&kernel_context->stack[KERNEL_STACK_SIZE];
    DEBUG0("kernel_stack    : %p",kernel_stack);
    DEBUG0("reset vector    : 0x%08x 0x%08x",*((unsigned int*)0x0),*((unsigned int*)0x4));
    DEBUG0("exception vector: 0x%08x 0x%08x",*((unsigned int*)0x8),*((unsigned int*)0xC));
    DEBUG0("interrupt vector: 0x%08x 0x%08x",*((unsigned int*)0x10),*((unsigned int*)0x14));
    DEBUG0("hardware vector : 0x%08x 0x%08x",*((unsigned int*)0x20),*((unsigned int*)0x24));

    DEBUG0("caches          : I=0x%08x D=0x%08x",(unsigned int)GetICacheSize(),(unsigned
int)GetDCacheSize());
    DEBUG0("memories        : L=0x%08x F=0x%08x",(unsigned int)GetLocalMemSize(),(unsigned
int)GetFifoMemSize());

#ifdef KERNEL_DEBUG
    if(GetProcID()==0){
        malloc_dump((HEAP*)kernel_context->heap);
        malloc_dump(_fifomem);
    }
#endif

    // Initialize hardware
    kStopTimer();

    // Set current context to an unused context.
    // When the first interrupt arrives to start scheduling, the current context must be
stored somewhere.
    // However, the current context will be thrown away, so dump to an unused proc_context
#if MAX_PROCESSES<2
#error "Not enough processes supported by the kernel, need at least 2."
#endif
    proc_context_t* dummy_context=&kernel_context->proc_table[MAX_PROCESSES-1].context;
    // Make sure that the context is written, otherwise the stack overflow detection might
kick in.
    dummy_context->donttouch=0;
    ASSIGN_CONST_PTR(proc_context_t,current_context,dummy_context);

    // Initialize all process information
    for(pid_t i=0;i<MAX_PROCESSES;i++){
        kernel_context->proc_table[i].state=NOPROC;
        kernel_context->proc_table[i].pid=i;
        kernel_context->proc_table[i].next_entry=&kernel_context->proc_table[i];
```

```
        kernel_context->proc_table[i].prev_entry=&kernel_context->proc_table[i];
    }

#ifdef SCC_REQUIRE_WRITETHROUGH
    if(GetMBFlags()&MB_FLAG_DCACHE_WRITEBACK)
        kPanic("Current software cache coherency without distributed locks only works with
        write-through cache.");
#endif
#ifndef ALL_MEM_CACHED
    if(GetMBFlags()&MB_FLAG_SHARED_HEAP_CACHED)
        kPanic("Hardware caches all memory, but kernel is not compiled for that.");
#endif
#ifdef USE_DISTRIBUTED_LOCK
    if(GetSoCCores()!=NUM_PROCESSORS){
        DEBUG("The kernel is compiled for %d processors, but %d physical cores are
        found.",NUM_PROCESSORS,GetSoCCores());
        kPanic("Distributed lock requires that the number of physical cores corresponds to
        NUM_PROCESSORS.");
    }
#endif

    // Initialize pthread specific stuff
    ASSERT(param->notify_booted!=NULL||GetProcID()==0,"Usually, core 0 is the master core...
    We are now %d.",GetProcID());
    int res;
    // when notify_booted==NULL, this must be the master core
    if(!param->notify_booted){
        if((res=pthread_global_init())!=0){
            DEBUG("pthread global initialization failed: %d",res);
            kPanic("pthread global initialization failed");
        }
    }
    if((res=pthread_local_init())!=0){
        DEBUG("pthread local initialization failed: %d",res);
        kPanic("pthread local initialization failed");
    }else
        sch.pthread_sch=(pthread_sch_t*)kGetHookTable()[HOOK_SCHEDULER];

    kSetOSState(OS_SINGLE);

    // Initialize tracing
    trace_t* t __attribute__((unused))=kTraceInit();

#ifdef KERNEL_DEBUG
    struct timeval tv;
    gettimeofday(&tv,NULL);
    DEBUG0("Kernel is up and running since %u.%03u s after reset.",(unsigned int)tv.tv_sec,
    (unsigned int)(tv.tv_usec/1000));

    if(GetProcID()==0){
        printk("\n");
        DumpConfig();
        printk("\n");
    }
#endif
}

/*!
 * \brief Starts the task daemon
 *
 * This is essentially a wrapper around kCreateProcess() and kStartProcess().
 * It will call InitTaskDaemon(), which should be implemented by the application.
 *
 * \param param as received by main(int,char**)
 */
static void kStartTaskDaemon(kernel_boot_param_t* param){
    DEBUG0("Starting task daemon...");
```

```c
    // copy param (which is a local variable) to the heap;
    // the local variable will be destroyed when the scheduler kicks in
    kernel_boot_param_t* p=(kernel_boot_param_t*)malloc(sizeof(*param));
    if(!p)
        kPanic("Cannot save boot parameters");
    *p=*param;
    pid_t pid=0;
    if(kCreateProcess(pid,&InitTaskDaemon,(void*)p,TASK_DAEMON_TIMESLICE,TASK_DAEMON_STACK))
        kPanic("Cannot create TaskDaemon");
#ifdef KERNEL_DEBUG
    if(pid!=0)
        kPanic("TaskDaemon got strange pid!");
#endif
    kStartProcess(pid);
}

/*!
 * \brief Prints coredump and halts kernel, in case of emergency.
 * \param msg message to be printed, which indicates the reason of the core dump. Can be NULL.
 */
void kPanic(const char* msg){
    kDisableInterrupt();
    kStopTimer();
    DEBUG_ON(DBLED_PANIC);
    for(volatile int i=0;i<1000000;i++);

    // It is possible that the heap is not initialized when kPanic is called.
    // In that case, printf cannot be used. So, use _puts here to make sure that the
    // reason of the panic is printed properly.
    _puts("\n\n *** Kernel PANIC! *** \n * Reason: ");
    if(msg){
        _puts(msg);
        _puts("\n");
    }

#ifdef KERNEL_DEBUG
    struct timeval tv;
    gettimeofday(&tv,NULL);
    printk(" * Current time: %u.%03u s after reset.\n",(unsigned int)tv.tv_sec,(unsigned int)
(tv.tv_usec/1000));
#endif
    kCoreDump();
    printk(" * Dumping heaps\n");
    malloc_dump((HEAP*)kernel_context->heap);
    malloc_dump(_fifomem);
    printk(" *** End of core dump, halt processor %d ***\n",GetProcID());

    // Flush DCache to allow stack trace etc. from XMD
    FlushDCache();

    DisableDCache();
    DisableICache();
    kReset();
    while(true);
}

/*!
 * \brief Shuts down kernel and return to bootloader.
 * \details Not gracefull, no process termination, just reboot immediately.
 * \returns EPERM when another process than the task daemon tries to shutdown, otherwise never
 */
int kShutdown(){
    if(kGetPid()!=0)
        return EPERM;

    // wait for our time slot to print to console
    TDEBUG("Shutdown");
```

```
    kDisableInterrupt();
    DisableDCache();
    DisableICache();
    kReset();
    while(true);
}

//////////////////////////////////////////////////////
// Process lifetime
//////////////////////////////////////////////////////

/*!
 * \brief Returns the process id of the current process
 */
pid_t kGetPid(){
    return current_proc->pid;
}

/*!
 * \brief Creates a new process.
 * \details By default, a process is created detached.
 * \details Only the task daemon is allowed to start new processes.
 * \param pid the new process id will be written to the pointer supplied
 * \param main the main function of this process
 * \param arg an optional argument to main, can be NULL
 * \param slicelength the length of the allocated time slice in ms
 * \param stack the size of the stack in words
 * \return 0 on success, otherwise an errno
 */
int kCreateProcess(pid_t &pid,void* (*main)(void*),void* arg,timems_t slicelength,unsigned int
stack){
    proc_table_t* pt=NULL;

    // only the TaskDaemon is allowed to start processes
    if(kGetPid()!=0)
        return EPERM;
    if(slicelength==0)
        return EINVAL;
    // find free PID
    for(pid=0;pid<MAX_PROCESSES && (pt=&kernel_context->proc_table[pid])-
>state!=NOPROC;pid++);
    // no free PIDs
    if(pid==MAX_PROCESSES)
        return EAGAIN;

    DEBUG_ON(DBLED_NEWPROC);
//  DEBUG("Creating process: pid %d, main %p, arg %p, time %d, stack
0x%x",pid,main,arg,slicelength,(unsigned int)stack);

#ifdef KERNEL_DEBUG
    // stack overflow detection field
    stack+=sizeof(unsigned int);
#endif
    if(!(pt->stack=(char*)omalloc(stack,pid)))
        return ENOMEM;
#ifdef TRAP_ALL_DMEM_ACCESS
    pt->stack_size=stack;
#endif
#ifdef KERNEL_DEBUG
    *(unsigned int*)pt->stack=STACK_PROTECTION_VALUE;
#endif
#ifdef MEASURE_STACK_USAGE
    for(unsigned int s=0;s<stack/sizeof(unsigned int);s++)
        ((unsigned int*)pt->stack)[s]=STACK_PROTECTION_VALUE;
#endif
```

```
    if(!(pt->impure_data=(struct _reent *__ATTRIBUTE_IMPURE_PTR__)omalloc(sizeof(struct _reent
__ATTRIBUTE_IMPURE_PTR__),pid)))
        return ENOMEM;

    _REENT_INIT_PTR(pt->impure_data);

    if(!(pt->proc_local=(proc_local_t*)omalloc(sizeof(proc_local_t),pid)))
        return ENOMEM;
    memset(pt->proc_local,0,sizeof(proc_local_t));

    kInitContext(&pt->context,main,arg,pt->stack+stack);
    pt->state=NEW;
    pt->main=main;
    pt->channels=NULL;
    pt->num_channels=0;
    pt->flags=PROC_FLAG_DETACHED;
    pt->next_entry=pt;
    pt->prev_entry=pt;
#ifdef KERNEL_SIGNALS
    kInitSignalContext(&pt->sig_context,&pt->context,&pt->sig_triggered[0],&pt-
>sig_havetriggered);
    InitSignalList(&pt->sig_action[0]);
    sigfillset(&pt->sig_enabled);
    sigemptyset((sigset_t*)&pt->sig_mask);
    memset(&pt->sig_triggered,0,sizeof(pt->sig_triggered));
    pt->sig_havetriggered=0;
    sigemptyset((sigset_t*)&pt->sig_sticky);
    pt->sig_ret=0;
    pt->time_yielded=0;
#endif
#ifdef KERNEL_SYNC_EVENTS
    memset(&pt->sync_event[0],0,sizeof(pt->sync_event));
#endif
#ifdef KERNEL_STATS
    pt->slack=0;
    pt->time_alloc=0;
    pt->min_stack=(int)stack;
    pt->context_switches=0;
    pt->yields=0;
#endif
    int res;
    if((res=kSetProcessTimeslice(pid,slicelength))){
        kDestroyProcess(pid,NULL);
        return res;
    }

    if(sch.pthread_sch)
        mc_apply(sch.pthread_sch->processes,++);

//  DEBUG("proc table at %p, context at %p, stack at %p, thread_local at %p, impure data at
%p",pt,&pt->context,pt->stack,pt->proc_local->user,pt->impure_data);
#ifdef KERNEL_SIGNALS
//  DEBUG("New process %d (context: %p/%p)",pid,&pt->context,&pt->sig_context);
#else
//  DEBUG("New process %d (context: %p)",pid,&pt->context);
#endif

    return ESUCCESS;
}

/*!
 * \brief Entry point of a process.
 * \details Initializes some stuff and calls the entry point as specified using
 kCreateProcess().
 */
void* ProcessEntry(void* (*main)(void*),void* arg){
    int res;
```

```c
    if((res=pthread_thread_init())){
        DEBUG("pthread thread init failed: %d",res);
        return (void*)res;
    }
    atexit((void (*)(void))pthread_thread_cleanup);
#ifdef ENABLE_TRACING
    atexit(TraceExitProc);
    TracePhaseChange(TRACE_PHASE_RUNNING);
#endif
    if((res=atexit(do_gracefull_exit_funs)))
        DEBUG("Cannot register atexit() of process %d: %d",kGetPid(),res);
    return main(arg);
}

/*!
 * \brief Sets specific process flags.
 * \details It is not safe to set the flags and join/destroy the process concurrent.
 * \details So, do process management of a single process in a safe manner.
 * \return 0 on success, otherwise an errno
 */
int kSetProcessFlags(pid_t pid,int flags){
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* pt=&kernel_context->proc_table[pid];
    if(pt->state==NOPROC)
        return ESRCH;
    pt->flags=flags;

    // pthread_detach() is called after pthread_create().
    // So, when the process terminates quickly, the process could be
    // in JOINABLE state before it can be detached.
    // Hence, set to ZOMBIE in case it was already JOINABLE.
    barrier();
    if(flags&PROC_FLAG_DETACHED&&pt->state==JOINABLE)
        pt->state=ZOMBIE;

    return ESUCCESS;
}

/*!
 * \brief Gets the process flags.
 * \param pid the process to retrieve the flags of
 * \param flags will receive the flags of the process, cannot be NULL
 * \return 0 on success, otherwise an errno
 */
int kGetProcessFlags(pid_t pid,int *flags){
    if(!flags)
        return EINVAL;
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* pt=&kernel_context->proc_table[pid];
    if(pt->state==NOPROC)
        return ESRCH;
    *flags=pt->flags;
    return ESUCCESS;
}

/*!
 * \brief Returns the state of a process.
 * \param pid the process to retrieve the state of, -1 means myself
 * \param state will receive the state of the process
 * \param sighandling will set to true when the process is currently handling signals
 * \return 0 on success, otherwise an errno
 */
int kGetProcessState(pid_t pid,proc_state_t *state,bool *sighandling){
    proc_table_t* pt;
    if(unlikely(pid==-1)){
```

```
        pid=kGetPid();
        pt=current_proc;
    }else{
        if(pid<0||pid>=MAX_PROCESSES)
            return ESRCH;
        pt=&kernel_context->proc_table[pid];
        if(pt->state==NOPROC)
            return ESRCH;
    }
    if(state)
        *state=pt->state;
    if(sighandling)
        *sighandling=kIsInSigHandler(pt);
    return ESUCCESS;
}

/*!
 * \brief Changes the length of the timeslice of an existing process.
 * \details When pid is the current process, the new time is effective after the next context
switch
 * \param pid the process to change the timeslice of, which must exist
 * \param slicelength the length of the allocated time slice in ms
 * \return 0 on success, otherwise an errno
 */
int kSetProcessTimeslice(pid_t pid,timems_t slicelength){
    proc_table_t* pt;

    if(slicelength==0)
        return EINVAL;
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    pt=&kernel_context->proc_table[pid];
    if(pt->state==NOPROC)
        return EINVAL;

    timeslice_t ticks_in_ms=GetClockFreq()/1000;
    if(slicelength>(timems_t)(((unsigned int)-1)/ticks_in_ms))
        // slicelength out of range for the timer
        return EINVAL;

    // time in clock cycles
    pt->time=(timeslice_t)slicelength*ticks_in_ms;

    // trace change in process
    kTraceProcChange(TRACE,pid,pt->time,(void*)pt->main);

    return ESUCCESS;
}

/*!
 * \brief Returns the remaining clock ticks before the end of the current timeslice.
 * \details When interrupted by the timer, race conditions can occur, which result in a
reported timeslice that is too high.
 * \details Call from user space, does not need a syscall.
 * \return the number of remaining clock cycles.
 */
timeslice_t GetRemainingTimeslice(){
    unsigned int timer=kGetTimer();
    if((int)timer==-1)timer=0;
#ifndef SCHEDULE_SLOTS
    return timer;
#else
    return timer+(sch.remaining_slots-1)*SCHEDULE_SLOTS+current_proc->time_yielded;
#endif
}

/*!
```

```c
 * \brief Starts a process that has been created using kCreateProcess().
 * \details Only the task daemon is allowed to start processes.
 * \param pid the process to start
 * \return 0 on success, otherwise an errno
 */
int kStartProcess(pid_t pid){
    // only the TaskDaemon is allowed to start processes
    if(kGetPid()!=0)
        return EPERM;
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;

    proc_table_t* pt=&kernel_context->proc_table[pid];
    if(pt->state!=NEW)
        return EINVAL;

    // enable process
    pt->state=RUNNING;

    // update schedule
    pt->prev_entry=current_proc;
    pt->next_entry=current_proc->next_entry;
    if(pt->next_entry!=td_proc)
        pt->next_entry->prev_entry=pt;
    barrier();
    // the next line will put the process in the schedule sequence
    current_proc->next_entry=pt;
    // make sure a yield will have effect
    sch.just_did_work=DID_WORK_YES;

    DEBUG_OFF(DBLED_NEWPROC);
    return ESUCCESS;
}

/*!
 * \brief Forcably ends the timeslice of the current process.
 * \todo fix, several race conditions exist!
 * \param havework set to false when no work has been done and none is expected soon; this
prevents rapid rescheduling
 * \return 0 on success, otherwise an errno
 */
int kYield(bool havework){
    if(havework)
        sch.just_did_work=DID_WORK_YES;
    else if(sch.just_did_work==DID_WORK_MAYBE)
        sch.just_did_work=DID_WORK_NO;
    if(kGetPid()==0){
        if(sch.have_work==DID_WORK_NO&&sch.just_did_work<=DID_WORK_MAYBE){
            // a yield will context switch to myself, so don't do that
            if(!sch.was_idle&&sch.pthread_sch){
                mc_entry_wou(sch.pthread_sch->idle);
                sch.pthread_sch->idle=sch.was_idle=1;
                mc_exit_wou(sch.pthread_sch->idle);
            }
            return ESUCCESS;
        }else if(sch.was_idle&&sch.pthread_sch){
            mc_entry_wou(sch.pthread_sch->idle);
            sch.pthread_sch->idle=sch.was_idle=0;
            mc_exit_wou(sch.pthread_sch->idle);
        }
    }

#ifdef KERNEL_STATS
    current_proc->yields++;
#endif

    // read rest of timeslice/slot
```

```
    timeslice_t yielded=kGetTimer();
#ifdef SCHEDULE_SLOTS
    // calculate actual yielded time
    unsigned int remaining_slots=sch.remaining_slots;
    ASSERT(remaining_slots>0,"Running beyond timeslice boundaries");
    yielded+=(remaining_slots-1)*SCHEDULE_SLOTS;
#endif
#ifdef KERNEL_STATS
    current_proc->slack+=yielded;
#endif
#ifdef SCHEDULE_SLOTS
    yielded+=current_proc->time_yielded;
#endif

    // Trace this yield
    kTraceYield(TRACE,yielded);

    // substract penalty of doing context switches
    if(yielded<CONTEXT_SWITCH_CYCLE_PANELTY*2)
        yielded=0;
    else
        yielded-=CONTEXT_SWITCH_CYCLE_PANELTY;

    // store time that can be reused
#ifndef SCHEDULE_SLOTS
    current_proc->time_yielded=yielded;
    // trigger interrupt
    kSetTimer(2);   // setting to 1 might be a bit funky...
#else
    lastart((int*)&current_proc->time_yielded);
    sch.remaining_slots=1;
    // 1) When interrupted here, the process will loose the rest of its slice.
    // This is not really a problem, the process yielded anyway.
    barrier();
    laend((int*)&current_proc->time_yielded,yielded<SCHEDULE_SLOTS?0:yielded);
    // 2) When interrupted here without interrupting at 1), everything is OK
    // When also at 1) an interrupt occurred, the process's timeslice is seriously extended,
    // therefore, it is guared by lastart/laend.
    barrier();
    // 3) When interrupted here, everything is OK, but the next call will shorten
    // the process's timeslice by the length of the schedule slot. That's not a problem.
    kSetTimer(2,SCHEDULE_SLOTS);
#endif
    return ESUCCESS;
}

/*!
 * \brief Suspends current process until one of the non-masked signals occur.
 * \details Cannot be called from within a signal handler
 * \param mask ignore all signals specified in this set (NULL=no mask)
 * \param wait_sticky keep suspended until one of the wait_sticky sticky signals were
triggered (NULL or none set=don't care; continue on any signal)
 */
int kSuspend(const sigset_t *mask,const sigset_t* wait_sticky){
#ifdef KERNEL_SIGNALS
    if(kIsInSigHandler(current_proc))
        return ENOTSUP;

    sigset_t newmask=mask?*mask:0,oldmask;
    int res=kMaskSignal(SIG_BLOCK,&newmask,&oldmask);
    if(res)
        return res;

    res=ESUCCESS;

    const int wait_for_sticky=
        // take all non-masked signals
```

```
		(mask?~(*mask):-1) &
		// and all signals to wait for
		(wait_sticky?*wait_sticky:0) &
		// check stickyness
		SIG_STICKY;

	sigset_t sticky_fired=0;
	do{
		// set to SUSPENDED when no sticky signals have been set
		if(!(lcondset_mask((int*)&current_proc->sig_sticky,wait_for_sticky,(int*)
(void*)&current_proc->state,SUSPENDED,RUNNING)&wait_for_sticky))
			// no sticky signals have been set, state is now SUSPENDED, give up timeslice
			res=kYield(false);

		// at this point, state is RUNNING (again)
		ASSERT(current_proc->state==RUNNING,"Still suspended after trigger");
		// clear sticky signals
		if(wait_for_sticky)
			sticky_fired=(sigset_t)launset((int*)&current_proc-
>sig_sticky,wait_for_sticky)&wait_for_sticky;

		// if waiting for sticky signals, keep suspended until one of them is fired
	}while(wait_for_sticky!=0 && sticky_fired==0);

	kMaskSignal(SIG_SETMASK,&oldmask,NULL);
	// done
	return res?res:EINTR;
#else
	kYield(false);
	return ENOSYS;
#endif
}

/*!
 * \brief Ends the current process.
 * \details Will be called automatically when the main function of a process returns.
 *          The timeslice will always be ended by a call to kYield().
 * \param ret the value returned by the main of the process, which can be retrieved using
kJoinProcess() and kDestroyProcess()
 * \return never
 */
void kEndProcess(void* ret){
#ifdef MEASURE_STACK_USAGE
	unsigned int s;
	for(s=0;s<current_proc->min_stack/sizeof(unsigned int) && ((unsigned int*)current_proc-
>stack)[s]==STACK_PROTECTION_VALUE;s++);
	current_proc->min_stack=s*sizeof(unsigned int);
#endif

#ifdef KERNEL_DEBUG
	_impure_ptr=_kernel_impure_ptr;
	if(kGetPid()==0)
		kPanic("TaskDaemon terminated!");
	if(ret)
		printk("Process %d@%d exited with %p (context: %p, main:
%p)\n",kGetPid(),GetProcID(),ret,current_context,current_proc->main);
#endif

	// stop this process
	current_proc->ret=ret;
	barrier();
	if(current_proc->flags&PROC_FLAG_JOINABLE)
		current_proc->state=JOINABLE;
	else
		current_proc->state=ZOMBIE;
	kYield();
	while(true);
```

```c
}

/*!
 * \brief Joins a #JOINABLE process.
 *
 * Gets the return value of the process to join.
 * Never blocks, so wrap in a loop when it is intended to wait until the process finishes.
 * A #JOINABLE process will become a #ZOMBIE when the join is successful.
 * The function is not thread-safe.
 *
 * \param pid process to join
 * \param ret pointer to store the return value of the process to join, can be NULL
 * \return 0 on success, EAGAIN when pid was not #JOINABLE, otherwise an appropiate errno
 */
int kJoinProcess(pid_t pid,void** ret){
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* pt=&kernel_context->proc_table[pid];
    if(pt->state==NOPROC)
        return ESRCH;
    if(!(pt->flags&PROC_FLAG_JOINABLE))
        return EINVAL;
    if(pt->state!=JOINABLE)
        return EAGAIN;
    // join
    if(ret)
        *ret=pt->ret;

    // Since the join can be executed by any process and the default task daemon
    // reacts on zombie processes, make sure that everything as been cleaned up
    // before changing this state. Otherwise, nasty things can happen when an
    // context switch occures and a new process is created using this pt.
    barrier();
    pt->state=ZOMBIE;
    return ESUCCESS;
}

/*!
 * \brief Destroys and clean up a process.
 *
 * Only the task daemon is allowed to destroy processes.
 * All allocated memory will be freeed.
 * Process statistics is not deleted, so after destroying a process, kGetProcessStats() can be
called.
 * A process cannot destroy itself.
 *
 * \param pid process to destroy
 * \param ret a pointer to store the return value of the process, can be NULL, holds unknown
data when a non-#ZOMBIE process is destroyed
 * \return 0 on success, otherwise an errno
 */
int kDestroyProcess(pid_t pid,void** ret){
    if(kGetPid()!=0)
        return EPERM;
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    if(kGetPid()==pid)
        return EINVAL;
    proc_table_t* pt=&kernel_context->proc_table[pid];
    if(pt->state==NOPROC)
        return ESRCH;

    // make sure that it won't be scheduled before the memory is freed.
    pt->state=ZOMBIE;
    barrier();

    freeAll(pid);
```

```
    ffreeAll(pid);
    if(ret)
        *ret=pt->ret;

    // update schedule
#ifdef SCHEDULE_SLOTS
    if(pt->next_entry!=td_proc)
        // prev_entry of the task daemon is used for other purposes; do no overwrite in that
case
#endif
        pt->next_entry->prev_entry=pt->prev_entry;
    pt->prev_entry->next_entry=pt->next_entry;
    barrier();
    pt->state=NOPROC;

    if(sch.pthread_sch)
        mc_apply(sch.pthread_sch->processes,--);

    // trace the destruction of this process
    kTraceProcChange(TRACE,pid,0,(void*)NULL);

    return ESUCCESS;
}

/*!
 * \brief Looks for a #ZOMBIE process and destroys it.
 *
 * Walks over all processes and cleans up the first #ZOMBIE process it can find.
 * Only the task daemon is allowed to call the function.
 *
 * \param pid the process that has been cleaned up, can be NULL
 * \param ret pointer to store the return value of the process in, can be NULL
 * \return 0 on success, EAGAIN when no ZOMBIEs has been found, otherwise an appropriate errno
 */
int kGetZombie(pid_t *pid,void** ret){
    pid_t zpid;
    // only the TaskDaemon is allowed to modify the process table
    if(kGetPid()!=0)
        return EPERM;
    // find first zombie
    for(zpid=0;zpid<MAX_PROCESSES && kernel_context->proc_table[zpid].state!=ZOMBIE;zpid++);
    if(zpid==MAX_PROCESSES)
        return EAGAIN;
    if(pid)
        *pid=zpid;
    // clean up zombie process
    return kDestroyProcess(zpid,ret);
}

/*!
 * \brief Collects statistics of the specified process.
 * \param pid the process to collect statistics of (can be an non-existing process)
 * \param stats pointer to struct to write statistics to
 * \returns 0 on success, otherwise an errno
 */
int kGetProcessStats(pid_t pid,proc_stats_t* stats){
    if(stats==NULL)
        return EINVAL;
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* pt=&kernel_context->proc_table[pid];
    stats->pid=pid;
    stats->state=pt->state;
    stats->insighandler=kIsInSigHandler(pt);
    stats->context=&pt->context;
    stats->main=pt->main;
    stats->flags=pt->flags;
```

```c
    timeslice_t ticks_in_ms=GetClockFreq()/1000;
    stats->slice_length=(timems_t)(pt->time/ticks_in_ms);
#ifdef KERNEL_STATS
    stats->slice_slack=(timems_t)(pt->slack/ticks_in_ms);
#ifdef SCHEDULE_SLOTS
    stats->slice_used=(timems_t)((pt->time_alloc*SCHEDULE_SLOTS-pt->slack)/ticks_in_ms);
#else
    stats->slice_used=(timems_t)((pt->time_alloc-pt->slack)/ticks_in_ms);
#endif
#endif

#ifdef KERNEL_STATS
    stats->min_stack=pt->min_stack;
    stats->context_switches=pt->context_switches;
    stats->yields=pt->yields;
#endif
    return ESUCCESS;
}

/*!
 * \brief Prints complete state of the current process and the kernel.
 * \return 0 on success, otherwise an errno
 */
int kCoreDump(){
    printk(" * Core dump of state during last interrupt\n");
    printk("current_proc    : %p @ processor %d\n",current_proc,GetProcID());
    printk("os state        : %d\n",GetOSState());
    printk("kernel_context  : %p\n",kernel_context);
    printk("current_context : %p\n",current_context);
    printk("current process : %d\n",current_proc->pid);
    printk("state           : %d\n",current_proc->state);
    printk("slice           : %u cycles@%dMHz\n",current_proc->time,GetClockFreq()/1000000);
    printk("yielded         : %u cycles\n",current_proc->time_yielded);
    printk("&main           : %p\n",current_proc->main);
    printk("stack           : %p\n",current_proc->stack);
    for(int reg=0;reg<8;reg++){
        printk("r%-2d : 0x%08X     ",reg,current_context->regs[reg]);
        printk("r%-2d : 0x%08X     ",reg+8,current_context->regs[reg+8]);
        printk("r%-2d : 0x%08X     ",reg+16,current_context->regs[reg+16]);
        printk("r%-2d : 0x%08X\n",reg+24,current_context->regs[reg+24]);
    }
    printk("MSR : 0x%08X\n",current_context->msr);
    printk("FSR : 0x%08X\n",current_context->fsr);
    printk("PC  : 0x%08X (=r14)\n",current_context->regs[14]);
    printk("SP  : 0x%08X (=r1)\n",current_context->regs[1]);
    printk("[SP]: 0x%08X\n",*((unsigned int *)(current_context->regs[1])));

    printk("\nProcess table:\n");
    for(int i=0;i<MAX_PROCESSES;i++){
        proc_table_t *p=&kernel_context->proc_table[i];
#ifdef KERNEL_STATS
        printk(" %2d (%10p): state=%d flags=%d minstack=0x%04x\n",i,p,p->state,p->flags,p-
        >min_stack);
#else
        printk(" %2d (%10p): state=%d flags=%d\n",i,p,p->state,p->flags);
#endif
    }
    return ESUCCESS;
}

int kGetProcLocal(proc_local_t** p){
    if(p){
        *p=current_proc->proc_local;
        return ESUCCESS;
    }else{
        return EINVAL;
```

```
    }
}

/////////////////////////////////////////////////////
// Process I/O
/////////////////////////////////////////////////////

/*!
 * \brief Returns the number of allocated channels.
 * \details This does not indicate whether channels are assigned or in use.
 * \param pid the process to get the information of
 * \return The channel count or -1 when an invalid pid is supplied
 */
int kGetChannelCount(pid_t pid){
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* proc=&kernel_context->proc_table[pid];
    if(proc->state==NOPROC)
        return -1;
    if(proc->channels==NULL)
        return 0;
    else
        return proc->num_channels;
}

/*!
 * \brief Allocates memory for channels.
 * \details When called multiple times, the allocated memory will be resized.
 * \param pid process to allocate channel memory for
 * \param num_chan number of channels to allocate, will be initialized to NULL
 * \return 0 on success, otherwise an errno
 */
int kAllocChannels(pid_t pid, int num_chan){
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* proc=&kernel_context->proc_table[pid];
    // allocate memory for channels
    channel_t* newchans=(channel_t*)orealloc((void*)proc-
>channels,num_chan*sizeof(channel_t),pid);
    if(num_chan!=0 && newchans==NULL)
        return ENOMEM;
    // initialize all newly allocated channels to NULL
    for(int i=proc->num_channels;i<num_chan;i++){
        newchans[i].channel=NULL;
        newchans[i].label=0;
    }
    // update process table
    proc->channels=(volatile channel_t*)newchans;
    barrier();
    proc->num_channels=num_chan;
    return ESUCCESS;
}

/*!
 * \brief Assigns a pointer to a channel.
 * \param pid process to assign a channel to
 * \param chan_id channel to assign the pointer to, must be less than num_chan of a previous
kAllocChannels() call, when set to -1, a new channel is allocated at the end of the list
 * \param channel to store in the channel list
 * \param label an alternative label of the channel (set to 0 for no label, all negative
labels are reserved by the kernel)
 * \return 0 on success, otherwise an errno
 */
int kAssignChannel(pid_t pid, int chan_id, void* channel, int label){
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* proc=&kernel_context->proc_table[pid];
```

```c
    if(chan_id<0){
        int ret;
        chan_id=proc->num_channels;
        if((ret=kAllocChannels(pid,proc->num_channels+1)))
            return ret;
    }

    if(proc->channels==NULL)
        return EAGAIN;

    if(chan_id>=(int)proc->num_channels)
        return EINVAL;
    proc->channels[chan_id].channel=channel;
    proc->channels[chan_id].label=label;
    sch.just_did_work=DID_WORK_YES;
    return ESUCCESS;
}

/*!
 * \brief Sets a new label to an existing channel.
 * \param pid process that has the channel to change the label of
 * \param chan_id the channel to change the label of
 * \param label the new label, where 0 means 'no label'
 * \return 0 on success, otherwise an errno
 */
int kSetChannelLabel(pid_t pid, int chan_id, int label){
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* proc=&kernel_context->proc_table[pid];
    if(proc->channels==NULL || chan_id>=(int)proc->num_channels)
        return EINVAL;
    proc->channels[chan_id].label=label;
    return ESUCCESS;
}

/*!
 * \brief Looks for a channel with the specified label.
 * \param label the label to look for
 * \return the channel id of the channel with the specified label, or -1 when not found
 */
int kFindChannel(int label){
    if(label==0)
        return -1;

    for(int i=(int)current_proc->num_channels-1;i>=0;i--)
        if(label==current_proc->channels[i].label)
            return i;

    return -1;
}

/*!
 * \brief Returns the pointer corresponding to the given channel of the current process.
 * \details Blocks and kYield()s until the given channel is not NULL.
 *          Note that the specified channel is not checked for validity.
 * \param channel channel to open
 * \return pointer corresponding to the channel, cannot be NULL
 */
void* kOpenChannel(int channel){
    // wait until the channel has been allocated and assigned
    while((int)current_proc->num_channels<=channel || current_proc-
>channels[channel].channel==NULL)
        kYield(false);
    return (void*)current_proc->channels[channel].channel;
}
```

```c
/*!
 * \brief Returns the pointer to the hook table.
 * \details The memory region of the hook table is owned by the TaskDaemon (pid 0).
 * \return cannot be NULL during normal operation (it can be null during kernel
initialization, just before the table is initialized)
 */
hook_t* kGetHookTable(){
    return kernel_context->hooks;
}

int kSetSignalHandler(int signum,const struct sigaction* act){
#ifdef KERNEL_SIGNALS
    if(signum<=0||signum>NSIG||signum==SIGCONT)
        return EINVAL;

    if((1<<signum)&SIG_HANDLEABLE){
        current_proc->sig_action[signum]=*act;
        if(act->sa_handler==SIG_IGN)
            sigdelset(&current_proc->sig_enabled,signum);
        else
            sigaddset(&current_proc->sig_enabled,signum);
        current_proc->sig_triggered[signum]=0;
        return ESUCCESS;
    }else
        return EINVAL;
#else
    return ENOSYS;
#endif
}

int kClearSignal(int signum,struct sigaction* sa){
#ifdef KERNEL_SIGNALS
    if(signum<=0||signum>=NSIG)
        return EINVAL;
    if(!sa)
        return EINVAL;

    *sa=current_proc->sig_action[signum];

    // block masks (should be unblocked by the callee by sa->sa_mask)
    sigset_t mask=sa->sa_mask;
    sigaddset(&mask,signum);
    kMaskSignal(SIG_BLOCK,&mask,&sa->sa_mask);

    // remove trigger flag
    current_proc->sig_triggered[signum]=0;

    // reset handler when needed
    struct sigaction *s=&current_proc->sig_action[signum];
    if(s->sa_flags&SA_RESETHAND){
        s->sa_handler=_sig_def_func[signum];
        s->sa_flags&=~SA_RESETHAND;
        sigaddset(&current_proc->sig_enabled,signum);
    }

    return ESUCCESS;
#else
    return ENOSYS;
#endif
}

/*!
 * \brief Send signal to other process.
 * \details A signal will only be triggered (and thus handled) once.
 * \details When the signal was already triggered, the previous sv will be overwritten.
 * \param pid the process to send to
 * \param sig the signal number to send
```

```
 * \param sv value to pass to the handler by siginfo_t
 * \return ESRCH when the process does not exist (anymore), ENOENT when the signal was
disabled, EAGAIN when masked, 0 on accept, otherwise another errno
 */
int kSendSignal(pid_t pid,int sig,union sigval* sv){
#ifdef KERNEL_SIGNALS
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* proc=&kernel_context->proc_table[pid];

    switch(proc->state){
    case RUNNING:
    case SUSPENDED:
        break;
    default:
        return ESRCH;
    }
    if(sig<=0||sig>=NSIG)
        return EINVAL;

    // make sure a yield will have effect
    sch.just_did_work=DID_WORK_YES;

    // always set sticky signal, even when masked/disabled
    if(SIG_STICKY&(1<<sig))
        // flag this sticky signal
        laset((int*)&proc->sig_sticky,1<<sig);

    if(!sigismember(&proc->sig_enabled,sig))
        return ENOENT;

    // check mask
    if(sigismember(&proc->sig_mask,sig))
        return EAGAIN;

    // register signal
//  DEBUG("sig %d from %d to %d",sig,kGetPid(),pid);
    if(sv)
        proc->sig_action[sig].sv=*sv;

    switch(sig){
    case SIGCONT:
        proc->sig_trig_handling[SIG_TRIG_HANDLING_CONT]=1;
        break;
    default:
        proc->sig_triggered[sig]=1;
        barrier();
        proc->sig_trig_handling[SIG_TRIG_HANDLING_SEND]=1;
    }
    return ESUCCESS;
#else
    return ENOSYS;
#endif
}

int kMaskSignal(int how, const sigset_t* set, sigset_t* oldset){
#ifdef KERNEL_SIGNALS
    if(!set)
        return EINVAL;

    sigset_t old=current_proc->sig_mask;
    sigset_t newset=(*set)&SIG_BLOCKABLE;

    switch(how){
    case SIG_BLOCK:
        sigorset(&current_proc->sig_mask,&current_proc->sig_mask,&newset);
        break;
```

```
    case SIG_UNBLOCK:
        newset=~newset;
        sigandset(&current_proc->sig_mask,&current_proc->sig_mask,&newset);
        break;
    case SIG_SETMASK:
        current_proc->sig_mask=newset;
        break;
    default:
        return EINVAL;
    }

    if(oldset)
        *oldset=old;

    return ESUCCESS;
#else
    return ENOSYS;
#endif
}

#ifdef KERNEL_SYNC_EVENTS
static sync_event_entry_t* kFindEventEntry(proc_table_t* proc,void* event){
retry:
    sync_event_entry_t *se=&proc->sync_event[0],*firstempty=NULL;
    int i;
    for(i=0;i<KERNEL_SYNC_EVENTS;i++,se++){
        if(se->event==event)
            goto found;
        if(!firstempty&&se->event==NULL)
            firstempty=se;
    }
    // no record found, have empty record?
    if(!firstempty)
        return NULL;
    // try to use this record
    se=firstempty;
    if(lcas((int*)(void*)&se->event,0,(int)event)!=0)
        // just changed record
        goto retry;
    // no need to reset other fields
    ASSERT(se->hit==se->wait,"Sync event record invalid: %d != %d",se->hit,se->wait);
    // ok, got record
found:
    return se;
}
#endif

/*!
 * \brief Marks specified event as hit.
 * \details Increments hit counter corresponding to specified event.
 * \details Should only be called from the task daemon.
 * \param pid the process to sent the hit to
 * \param event the event that has been hit
 * \returns ENOMEM when out of event records, 0 on success, otherwise an errno
 */
int kSyncEventHit(pid_t pid,void* event){
#ifdef KERNEL_SYNC_EVENTS
    if(kGetPid()!=0)
        return EPERM;
    if(pid<0||pid>=MAX_PROCESSES)
        return ESRCH;
    proc_table_t* proc=&kernel_context->proc_table[pid];

    switch(proc->state){
    case RUNNING:
#ifdef KERNEL_SIGNALS
    case SUSPENDED:
```

```
#endif
		break;
	default:
		return ESRCH;
	}
	if(event==NULL)
		return EINVAL;

retry:
	sync_event_entry_t* se=kFindEventEntry(proc,event);
	if(!se)
		return ENOMEM;

	if(lcondset((int*)(void*)&se->event,(int)event,(int*)&se->hit,se->hit+1,se-
	>hit)!=(int)event)
		goto retry;

//	DEBUG("Hit %p=%d -> %d (entry: %p)",event,se->hit,pid,se);
	return 0;
#else
	return ENOSYS;
#endif
}

/*!
 * \brief Blocks until a given event has occured.
 * \param event the event to wait for
 * \param count the number of hits to wait for
 * \returns ENOMEM when out of empty event records, 0 on success, otherwise an errno
 */
int kSyncEventWait(void* event,int count){
#ifdef KERNEL_SYNC_EVENTS
	if(event==NULL||count<1)
		return EINVAL;

	sync_event_entry_t* se=kFindEventEntry(current_proc,event);
	if(!se)
		return ENOMEM;
//	DEBUG("Wait %p (entry: %p, pid=%d)",event,se,kGetPid());

	for(;count>0;count--){
		while(se->wait==se->hit)
			// light weight, fast responding; don't suspend, just yield
			kYield(false);
		// got event
		se->wait++;
	}
	// clean record when unused
	lcondset((int*)&se->wait,se->hit,(int*)(void*)&se->event,0,(int)event);
	// got all requested events
	return 0;
#else
	return ENOSYS;
#endif
}

//////////////////////////////////////////////////////
// Scheduling
//////////////////////////////////////////////////////

#undef TRACE
#define TRACE ((trace_t*)kernel_context->hooks[HOOK_TRACE])

/*!
 * \brief Initializes the scheduler and start scheduling.
 * \details Never returns.
 */
```

```
static void kStartScheduling(){
    DEBUG0("Initializing scheduler...");
    sch.just_did_work=DID_WORK_YES;
    sch.have_work=DID_WORK_YES;
    sch.was_idle=0;
    kSetOSState(OS_MULTI);
    // context switches are triggered when an interrupt occures, so trigger one
#ifndef SCHEDULE_SLOTS
    // somehow the MicroBlaze misses sometimes the first interrupt, so repeat this one to make
    sure...
    kSetTimer(100000,100000);
#else
    sch.remaining_slots=1;
    sch.skip_taskdaemon=1;
    td_proc->prev_entry=NULL;
    kSetTimer(SCHEDULE_SLOTS,SCHEDULE_SLOTS);
#endif
    barrier();
    kEnableInterrupt();
    while(true);
}

#undef ASSERT
#define ASSERT(expr,msg)        do{if(unlikely(!(expr)))kPanic(msg);}while(0)

#ifdef KERNEL_STATS
static void kCheckFreeStack() __attribute__((unused));
static void kCheckFreeStack(){
    // -1 means that the signal context has been reset
    if(likely((int)current_context->regs[1]!=-1)){
        int free_stack=(int)current_context->regs[1]-(int)current_proc->stack;
        if(unlikely(free_stack<current_proc->min_stack&&free_stack>=0))
            current_proc->min_stack=free_stack;
    }
    current_proc->context_switches++;
}
#else
#define kCheckFreeStack(...)
#endif

#ifdef KERNEL_DEBUG
static void kCheckStackOverflow() __attribute__((unused));
static void kCheckStackOverflow(){
    _impure_ptr=_kernel_impure_ptr;
    // Checking for stack overflows is not very reliable. When the stack has shrunk before
    // the context switch, it won't be detected. Also, when a stack overflow occurs, it is
    already
    // too late. Better is to give every process its own page using the MMU and trap when it
    accesses
    // memory outside that page.
    if(unlikely(current_context->regs[1]<(unsigned int)current_proc->stack)){
        // not very elegant, but very handy, since printf() doesn't work anymore at this point
        char msg[]="Process's ? stack overflow detected. Heap datastructures destroyed.";
        msg[10]='0'+kGetPid();
        kPanic(msg);
    }
    if(unlikely(*(unsigned int*)current_proc->stack!=STACK_PROTECTION_VALUE))
        kPanic("Possible stack overflow detected. Heap integrity compromised.");
}
#else
#define kCheckStackOverflow(...)
#endif

static void kScheduleProcess(proc_table_t* next, bool continue_slice){
    // we have the next process to context switch to, choose between signal handler and main
    process
    ASSIGN_CONST_PTR(proc_table_t,current_proc,next);
```

```
#ifdef KERNEL_SIGNALS
    next->sig_ret=0;
    next->sig_trig_handling[SIG_TRIG_HANDLING_CONT]=0;
    if(unlikely(next->sig_havetriggered&&!next->sig_trig_handling[SIG_TRIG_HANDLING_ABRT])){
        // sig_trig_handling[SIG_TRIG_HANDLING_ABRT] is set in exit_from_sigh to indicate that
        we have to return to
        // the normal process immediately, because the process is aborted.
        ASSIGN_CONST_PTR(proc_context_t,current_context,&next->sig_context);
        if(!next->sig_trig_handling[SIG_TRIG_HANDLING_HAND])
            // new invocation of signal handler, force to start of handler
            kResetSignalContext(&next->sig_context);
    }else
#endif
        ASSIGN_CONST_PTR(proc_context_t,current_context,&next->context);
    _impure_ptr=next->impure_data;
    ASSIGN_CONST_PTR(void*,thread_local,&next->proc_local->user[0]);
    ASSIGN_CONST_PTR(proc_local_t,proc_local,next->proc_local);

#ifdef STACK_CHECK
    _stack_end=(void*)((int)next->stack+STACK_CHECK); //reserve space for unchecked printf(),
    etc.
#endif

    sch.just_did_work=DID_WORK_MAYBE;

    // set time slice
    timeslice_t timeslice=continue_slice?next->time_yielded:next->time;

#ifndef SCHEDULE_SLOTS
    ASSERT(timeslice>0,"Cannot resume passed timeslice");
    next->time_yielded=0;
#ifdef KERNEL_STATS
    next->time_alloc+=timeslice;
#endif
    kSetTimer(timeslice);
#else
    next->time_yielded=timeslice%SCHEDULE_SLOTS;
    sch.remaining_slots=timeslice/SCHEDULE_SLOTS;
    ASSERT(sch.remaining_slots>0,"Cannot resume passed timeslice");
#ifdef KERNEL_STATS
    next->time_alloc+=sch.remaining_slots;
#endif
    // timer is still running, no need to set
//  kSetTimer(SCHEDULE_SLOTS,SCHEDULE_SLOTS);
#endif

    if(!kTraceSchedule(TRACE,next->pid,continue_slice))
        kTraceClose(TRACE);
}

/*!
 * \brief Determines the next process to schedule.
 * \details This function is called by entry.S after an interrupt.
 */
void kScheduleNext(){
    DEBUG_ON(DBLED_SCHEDULE);

    kCheckFreeStack();
    kCheckStackOverflow();

    // determine next process to switch to
    proc_table_t* next=NULL;
    // should we reset the timeslice?
    bool continue_slice=false;
    // did we do some work during last process?
    if(sch.just_did_work!=DID_WORK_NO)
        sch.have_work=DID_WORK_YES;
```

```
#ifndef SCHEDULE_SLOTS
    if(current_proc==td_proc)
        // end of schedule cycle, reset have_work
        sch.have_work=DID_WORK_NO;
#else
    // are we context switching away from the task daemon?
    const bool switch_from_taskdaemon=current_proc==td_proc;
    // are we context switching away from the task daemon that just interrupted a normal
    process?
    const bool switch_from_interrupting_taskdaemon=switch_from_taskdaemon&&current_proc-
    >prev_entry!=NULL;
    // did we context switch because the task daemon has work to do?
    const bool switch_to_taskdaemon=sch.skip_taskdaemon==0&&*(int*)FIFO_MEM!=0;

    // update timeslice left
    if(sch.remaining_slots>0){
        // current_proc interrupted (without yield)
        current_proc->time_yielded+=sch.remaining_slots*SCHEDULE_SLOTS;
        if(!kTraceScheduleYield(TRACE,current_proc->time_yielded))
            kTraceClose(TRACE);
#ifdef KERNEL_STATS
        current_proc->slack+=sch.remaining_slots*SCHEDULE_SLOTS;
#endif
    }

    if(switch_to_taskdaemon){
        // interrupt current process and switch to task daemon
        next=td_proc;
        // save the process to switch back to
        next->prev_entry=current_proc;
        // resume timeslice
        continue_slice=true;
        goto have_proc;
    }else if(switch_from_interrupting_taskdaemon){
        // switch back to the process we just interrupted
        next=current_proc->prev_entry;
        ASSERT(next!=td_proc,"Interrupted task daemon");
        if(next->time_yielded>SCHEDULE_SLOTS && next->state==RUNNING){
            // resume timeslice
            continue_slice=true;
            goto have_proc;
        }else{
            // out of time, do normal scheduling
            next=next->next_entry;
            continue_slice=false;
        }
    }else
#endif
#ifdef KERNEL_SIGNALS
    if(unlikely(current_proc->sig_ret &&
#ifdef SCHEDULE_SLOTS
        // in kYield() an unimportant race exists in updating time_yielded, such that
        time_yielded can be < SCHEDULE_SLOTS
        current_proc->time_yielded>=SCHEDULE_SLOTS
#else
        current_proc->time_yielded
#endif
        )){
        // we are yielding from a signal handler, return to same process
        next=current_proc;
        // Just returned from a signal handler and we have some time left in this slice.
        // Reschedule this process to finish its slice.
        ASSERT(next->state==RUNNING,"Returning from non-running signal handler.");
        // do not reset the timeslice
        continue_slice=true;
        goto have_proc;
    }else
```

```
#endif
    {
        // normal scheduling sequence
        next=current_proc->next_entry;
        // reset timeslice
        continue_slice=false;
    }

    // We have the first process to check whether we can context switch to,
    // make sure this one is runnable.
    // The loop below is only executed once, unless there are ZOMBIE or JOINABLE processes.
    // So, ZOMBIE or JOINABLE processes take more time for a context switch, but save time
    // because they are not scheduled at all. Concluded, the context switches per period takes
at most
    // (# of non-NOPROC) times (normal execution of context switch) time.
    // Note that it is always a good idea to cleanup all non-RUNNING processes...
    while(likely(next!=NULL)){
        switch(next->state){
#ifdef KERNEL_SIGNALS
        case SUSPENDED:
            if(!next->sig_havetriggered)
                break;
            next->state=RUNNING;
#endif
        case RUNNING:
            goto have_proc;
        default:;
        }
        // The next process in line was not able to continue, choose next one.
        // This means that the slack is lost and the timeslice is reset.
        continue_slice=false;
        next=next->next_entry;
    }
    ASSERT(false,"Broken scheduling sequence");

have_proc:
#ifdef SCHEDULE_SLOTS
    if(next==td_proc && !continue_slice){
        // this is the start of a replenishment interval, replenish timeslice of task daemon
        td_proc->time_yielded=td_proc->time;
        // next process to schedule is the normal one in line
        next->prev_entry=NULL;
    }
    if(switch_from_taskdaemon){
        // do not switch back the task daemon unless there is time left in its timeslice
        sch.skip_taskdaemon=td_proc->time_yielded>SCHEDULE_SLOTS?0:1;

        if(!switch_from_interrupting_taskdaemon)
            // end of schedule cycle, reset have_work
            sch.have_work=DID_WORK_NO;
    }
    if(next==td_proc){
        // do not interrupt the task daemon to execute the task daemon
        sch.skip_taskdaemon=1;
    }
#endif

    kScheduleProcess(next,continue_slice);
    DEBUG_OFF(DBLED_SCHEDULE);
}

#ifdef __cplusplus
} // extern "C"
#endif
```