ReTiS Lab
Scuola Superiore S. Anna
Pisa
kernel details

Paolo Gai
Evidence Srl / ReTiS Lab (SSSUP)
summary

- introduction to scheduling modules
- introduction to resource modules
- general purpose vs. compiling kernel
- kernel internal functions
- design of a scheduling module
- an example: the EDF module
part I

introduction to scheduling modules
objectives

- handle a **Scheduling Algorithm** in a way independent to the other modules in the system
- handle scheduling algorithms that depends on other scheduling algorithms (typically they depends on the queuing policy). For example, an **Aperiodic Server** using the services offered by another module (also called Master Module)
module interface (1)

- Public Functions
  - called by the Generic Kernel to implement the primitives called by the application
  - e.g. creation, dispatch, end, ...

Diagram:
- Application
- Generic Primitives
- Generic Kernel
- Module
- Public f.
module interface (2)

- Private Functions
  - called to implement the internal behavior of a scheduling algorithm
  - or...

Diagram:
- Application
- Generic Primitives
- Generic Kernel
- Module
- Public f.
- Private f.
module interface (3)

- Private Functions
  - called by an Aperiodic Server for a task inserted into the Master Module
  - dispatch, epilogue, insert, extract
module interface (4)

- each task is assigned to a module whose number is written into proc_table[p].task_level
- the task_level is used when calling the Public Functions
- aperiodic servers use the master module
example

the system is composed by

- 2 scheduling modules

<table>
<thead>
<tr>
<th>m0: Master Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1: Aperiodic Server</td>
</tr>
</tbody>
</table>

- 2 tasks
  - task t0: `proc_table[t0].task_level = m0`
  - task t1: `proc_table[t1].task_level = m1`
example (2)

- the Generic Kernel function `schedule()`
  - calls `level_table[m0].public_scheduler()`
  - the level scheduler can schedule the two tasks

- to dispatch the task t0
  `level_table[proc_table[t0].task_level].public_dispatch()`

- to dispatch the task t1
  `level_table[proc_table[t1].task_level].public_dispatch()`
  - that calls `level_table[m0].private_dispatch()`
part II

introduction to resource modules
objectives

- the algorithms that implements resource sharing:
  - should be modular
  - can cohesist at the same time

- the algorithms should, if possible, be independent from the module that owns the task that use the protocols
mutex

- each resource module exports a common interface
- the protocols uses some generic **mutex** that are preallocated by the system
- the link between mutex and protocol is defined at runtime when the mutex is initialized (as in POSIX)
parameters and models

- the protocols require some parameters that have to be given
  - for each mutex that use the protocol
  - for each task that uses a mutex of that type
- the parameters are given to the system using models that are passed
  - at mutex initialization
  - at task creation (e.g., the last parameter of the `task_create()` primitive)
priority inversion

- there exist protocols where the task given by the *scheduling* mechanism is not the same task that is then *dispatched*.

- for example, protocols like that are:
  - priority inheritance
  - priority ceiling
priority inversion (2)

- a medium priority ready task is exchanged with a low priority ready task

Example (PI):

- Critical section
- Normal execution

J1

J2

J3

exchange (push-through blocking)
problem

- in a system organized in levels that solution can not be implemented because
  - there is not a **global** concept of priority
  - to be modular the implementation of each level have to prescind from the implementation of the resource sharing algorithms
a first solution

implement the scheduling modules and the resource modules in a way that

- the resource modules know the implementation details of the scheduling modules
- the resource modules modifies the internal data structures of the scheduling modules

- advantages
  - optimized implementation
- disadvantages
  - the use of these protocols is limited to the data structures and the modules for that they have been written
the S.Ha.R.K. solution

the inheritance mechanisms are supported directly by the Generic Kernel in a way independent from the module implementation
basic idea

- traditional implementations were done inserting tasks out-of-order: the low priority ready task Tb is moved into the high priority position of the blocked task Ta.

The idea is that...

- Tb goes into the position of the high priority task.
- Ta blocks accessing the resource.
- Ta goes into the Blocked queue.
basic idea (2)

- In the S.Ha.R.K. approach it is the high priority blocked task that, staying into the ready queue, indexes another shadow task that have to be executed in substitution at dispatch time.

- Ta blocks accessing a resource.

- Tb does not feel anything.

- Ta index a shadow task remaining into the ready queue.
implementation

- the pointer to the running task (exec) is divided in two:
  - **exec** task given by the scheduler
  - **exec_shadow** task really executed
implementation (2)

- a **shadow** field is added to the task descriptor

- the **shadow** field is the task that have to be executed in substitution
implementation (3)

- at the start, proc_table[p].shadow = p (no substitution)

- the shadow field is set when a task blocks
implementation (4)

- a wait DAG can grow
implementation (5)

- the schedulers of each module index a task to be executed (prescinding from the shadows)
- the function `schedule` stores that task into the `exec` variable
- the dispatch function follows the shadow chain until `shadow = PID`
- `exec_shadow` is set to the last task of the shadow chain
implementation (6)
deadlock

- using the shadow field, deadlocks can be easily detected
- to detect if a deadloc occurred, the system should check if a cycle is inserted into the graph when a shadow field is modified
overhead

- the **dispatch** overhead is quite low (a list with integer indexes have to be examined, and usually that list is formed by 1-2 elements)
- the **blocking** overhead can be high since to block a task more than one shadow field have to be modified (it depends from the protocol used)
part III

two ways of implementing kernel primitives
software interrupts

- general purpose kernels usually provides
  - memory protection
  - separation between user and kernel space
- primitives usually implemented using software interrupts
  - needed to run untrusted applications
- RT applications are typically trusted, and does not need memory protection
interrupt disabling

- for that reason, often RT kernels are compiling kernels:
  - primitives are *functions* that works with interrupts disabled
    - eventually they can change context
  - memory is shared among all the threads and the kernel
  - internal kernel data structures can be visible
- S.Ha.R.K. uses this technique
a (blocking) primitive

- a typical structure is like that:

```c
void a_blocking_primitive(...) {
    disable_interrupts();
    capacity_control();
    record_JET_data();
    Module_internal_Task_Call(...);
    tracer_hook(...);
    scheduler();
    change_context(exec_shadow);
    enable_interrupts();
}
```

is a typical C function!
simply a CLI without changing stack!

now all is ready to choose the next task to be executed.

note that this function will call other 2 module functions:
- `public_scheduler()`: to choose the next task
- `public_dispatch()`: to tell the module that the task is really executed

the module implements its behavior here, AFTER the kernel accounted for its computation time.

example of functions are:
- `public_message`
- `public_block`

a tracer can be attached to every primitive. some functions are used to record task events

account the time spent since the last ctx switch now
part IV

kernel functions

interface
kernel types

- PID  task index
- IQUEUE  queue type
- TASK  = (void *)
- LEVEL  index for scheduling modules
- bandwidth_t  free system bandwidth
typedef struct {
    DWORD task_ID; /* progressive (unique) task number */
    LEVEL task_level; /* task’s module level */
    CONTEXT context; /* CPU Register pointer */
    BYTE *stack; /* task stack pointer */
    TASK (*body)(); /* task body function */
    char name[MAX_TASKNAME]; /* task name (e.g. “Goofy”) */
    WORD status; /* task status (EXE,ready,...) */
    WORD pclass; /* model type (SOFT/HARD,...) */
    WORD group; /* task group (an int number) */
    WORD stacksize; /* task stack size */
    DWORD control; /* status flags (see kernel/model.h) */
}
task descriptor (2)

int frozen_activations;  number of frozen activations
int sigmask;  signals informations
int sigpending;
int sigwaiting;
int avail_time;  task available time (us)
PID shadow;  the shadow field
struct _task_handler_rec *cleanup_stack;  cleanup handler pointer
int errnumber;  task’s errno() value
PID waiting_for_me;  used in task_join()
void *return_value;  used in task_join()
task descriptor (3)

```c
TIME   jet_table[JET_TABLE_DIM];
int    jet_tvalid;
int    jet_curr;
TIME   jet_max;
TIME   jet_sum;
TIME   jet_n;
void *keys[PTHREAD_KEYS_MAX]; // used with condition variables
struct condition_struct *cond_waiting;
int    delay_timer; // a per-task timer
int    wcet;
} proc_des;
```
typedef struct {
    void (*private_insert)(LEVEL l, PID p, TASK_MODEL *m);
    [...]
    PID (*public_scheduler)(LEVEL l);
    int (*public_guarantee)(LEVEL l, bandwidth_t *freebandwidth);
    int (*public_create)(LEVEL l, PID p, TASK_MODEL *m);
    void (*public_end)(LEVEL l, PID p);
    [...]
} level_des;
global variables

- include/kernel/var.h
  - proc_des proc_table[] task descriptor table
  - level_des *level_table[] sched modules descr. table
  - resource_des *resource_table[] res. modules descriptor table

- PID exec task selected for scheduling
  - PID exec_shadow RUNNING TASK

- int cap_timer event n. for capacity control
- struct timespec schedule_time last time schedule() was called
- struct timespec cap_lasttime previous value for schedule_time
global variables (2)

- `int task_counter` number of user tasks
- `int system_counter` number of system tasks
- `int calling_runlevel_func` set when shutdown is in progress
- `IQUEUE freedesc` free descriptor queue
event handling

- events are generated by the system timer and by external interrupts

- timer events
  - int kern_event_post(const struct timespec *time, void (*handler)(void *p), void *par);
  - int kern_event_delete(int index);
  - -1 is the invalid Event index

- to reschedule the system inside an event handler
  (e.g., a task has been activated using an event)
  - void event_need_reschedule(void);
exceptions

- to raise an exception
  - `void kern_raise(int n, PID p);`
- the exceptions are mapped in the real-time signal SIGHEXC
- the signals are handled with interrupts enabled, into the context of a task that have the signal unmasked
kernel memory management

- generic memory
  - void *kern_alloc(DWORD s);
  - void kern_free(void *block, size_t size);
- memory under 1Mb
  - void *DOS_alloc(DWORD size);
  - void DOS_free(void *ptr, DWORD size);
- other functions exists
- the tasks should use malloc() and free()
  - these functions do not disable interrupts!
kernel queues

(these functions have to be called with interrupt disabled)

- void iq_init(IQUEUE *q, IQUEUE *share, int flags);
- Typical usage: iq_init(&myqueue, NULL, 0);

- setting task priorities before insertion
  - struct timespec *iq_query_timespec(PID p, IQUEUE*q);
  - DWORD *iq_query_priority (PID p, IQUEUE *q);

- IQUEUE insertion
  - void iq_priority_insert (PID p, IQUEUE *q);
  - void iq_timespec_insert (PID p, IQUEUE *q);
  - void iq_insertfirst (PID p, IQUEUE *q);
  - void iq_insertlast (PID p, IQUEUE *q);
kernel queues (2)

- **IQUEUE extraction**
  - void iq_extract (PID p, IQUEUE *q);
  - PID iq_getfirst (IQUEUE *q);
  - PID iq_getlast (IQUEUE *q);

- **first/last task into the queue**
  - PID iq_query_first (IQUEUE *q);
  - PID iq_query_last (IQUEUE *q);

- **iterators and miscellaneous**
  - PID iq_query_next (PID p, IQUEUE *q);
  - PID iq_query_prev (PID p, IQUEUE *q);
  - int iq_isempty (IQUEUE *q);
acceptance test

an on-line acceptance test is performed at creation time

- **locally into** `public_create()`
  - e.g., task set of the module has $U_{tot} > 1$

- **globally into** `guarantee()`
  - **that calls** `public_guarantee()`
  - checks if the whole system is schedulable
  - **it uses the** `bandwidth_t` **type**
    - integer type (no floats inside the kernel!)
    - valid ranges into $[0...\text{MAX\_BANDWIDTH}]$
    - 0 means 0.0; MAX\_BANDWIDTH means 1.0
int guarantee() {
    bandwidth_t num=MAX_BANDWIDTH;
    int l;
    for (l =0; l<MAX_SCHED_LEVEL &&
         level_table[l]->public_guarantee; l++)
        if (!level_table[l]->public_guarantee(l,&num))
            return -1;
    return 0; /* OK */
}
part V

tips & tricks
let’s start!

the simplest way to create a new module is:

- look for a similar module that already exists (look in the modules directory)
- copy it in your local directory (2 files .c + .h)

change the prefix of every symbol with a new one

- for example from EDF_public_create to MYMODULE_public_create

add a new task model (if needed) into the .h

add your private data structures

- example: deadlines, periods, ...

finally, modify the source code
OSLib events: periodic tasks

- one OSLib event for each periodic task
  - the event index should be kept in a private data structure
- post the event into `public_activate`
- when the event fires:
  - reactivates the task
  - posts again an event for the next activation
- delete the reactivation event when the task dies
OSLib events: tick scheduler

- one OSLib event for the whole module
  - it can be used to divide the time in slots
- in the module registration function, post a `RUNLEVEL_INIT` initialization function
- the function should post an OSLib event
- when the event fires
  - increments the tick number
  - re-post the event for the next tick
OSLib events: temporal isolation

- use task descriptor’s \texttt{wcet} and \texttt{avail\_time}
- set the flag \texttt{CONTROL\_CAP} into the \texttt{control} field of the task descriptor
- put the available time into \texttt{avail\_time}
- \texttt{public\_epilogue} is called when the capacity is exhausted
  - time can reach negative values
  - check if \texttt{avail\_time} < 0
part VI

design of a scheduling module
strategy

- first, a scheduling strategy is needed
- the strategy defines the behavior related to a set of events
  - synchronous events
    - the user called a system primitive
  - asynchronous events
    - an external event arrived: an external interrupt, or an OSLib eventa (a deadlines, a capacity exhaustion,...)
- **FAST** implementation needed
  (10-30 us for each event)
task lifecycle

- a task is a C function that
  - is created
  - is activated
  - executes one or more instances
    - a task can block on a synchronization primitive
    - a task can end an instance
    - a task can be preempted by another task
  - finishes at the last
synchronous events

synchronous events describe the behavior of a task when it calls a primitive

- **task creation**
  
  ```c
  int (*public_create)(LEVEL l, PID p, TASK_MODEL *m);
  ```

- **task activation**
  
  ```c
  void (*public_activate)(LEVEL l, PID p);
  ```

- **blocking/unblocking at a synchronization point**
  
  ```c
  void (*public_unblock)(LEVEL l, PID p);
  ```
  ```c
  void (*public_block)(LEVEL l, PID p);
  ```

- **endcycle (end of an instance)**
  
  ```c
  int (*public_message)(LEVEL l, PID p, void *m);
  ```

- **end**
  
  ```c
  void (*public_end)(LEVEL l, PID p);
  ```
asynchronous events

A scheduling algorithm can create asynchronous events that interrupts the running task at a specific time:

- to activate periodic/aperiodic tasks
- to control deadlines
- to monitor task execution time
- to implement scheduling slots (e.g., every 1 ms)
other events

A scheduling module has information about:

- when a task is dispatched or preempted
  
  ```c
  void (*public_dispatch)(LEVEL l, PID p, int nostop);
  void (*public_epilogue)(LEVEL l, PID p);
  ```

- when a task has failed its creation
  
  ```c
  void (*public_detach)(LEVEL l, PID p);
  ```

Finally, using all the information collected by the above elements, the scheduling algorithm provides:

- A scheduler
  
  ```c
  PID (*public_scheduler)(LEVEL l);
  ```

- An acceptance test
  
  ```c
  int (*public_guarantee)(LEVEL l, bandwidth_t *freebandwidth);
  ```
an example: SIMPLEEDF

provides:

- hard periodic and aperiodic scheduling using the EDF algorithm (Liu and Layland, 1973)
- on-line guarantee based on the utilization factor
- support for aperiodic servers
- deadline miss and wcet violation exceptions

- the EDF module currently used has been written by Anton Cervin, University of Lund
implementation

- the level_des descriptor have to be redefined like a C++ class
- since C is used, an extension to the descriptor is needed

```c
struct {
    level_des lev;
    /* my data structures */
} SIMPLEXEDF_level_des;
```
- the Generic Kernel is **NOT** modified
- EDF_level_des contains all the data structures **private** to the EDF module
implementation (2)

- think `SIMPLEEDF_level_des` as a C++ class
- the registration function (like a C++ constructor) simply initialize the module and put a pointer to `SIMPLEEDF_level_des` into the first free available pointer into the `level_table`
- there is not a standard level destructor
  - if needed, `sys_atrunlevel` should be used
SIMPLEEDF task states
ready queue management

- the SIMPLEEDF module use a ready queue to store its tasks
- the ready queue is ordered by deadline, stored into the ready queue data structure
- the running task (pointed also by the `exec_shadow` field) is extracted from the ready queue
typedef struct {
    level_des l;
    TIME period[MAX_PROC];
    int deadline_timer[MAX_PROC];
    int flag[MAX_PROC];
    IQUEUE ready;
    int flags;
    bandwidth_t U;
} SIMPLEEDF_level_des;
static PID SIMPLEEDF_public_scheduler(LEVEL l)
{
    SIMPLEEDF_level_des *lev =
        (SIMPLEEDF_level_des *)(level_table[l]);
    return iq_query_first(&lev->ready);
}
static void SIMPLEEDF_public_dispatch(LEVEL l, PID p, int nostop)
{
    SIMPLEEDF_level_des *lev =
        (SIMPLEEDF_level_des *)(level_table[l]);
    iq_extract(p, &lev->ready);
}
task creation

static int SIMPLEEDF_public_create(LEVEL l, PID p, TASK_MODEL *m)
{
    SIMPLEEDF_level_des *lev =
        (EDF_level_des *)(level_table[l]);
    HARD_TASK_MODEL *h;

    if (m->pclass != HARD_PCLASS) return -1;
    if (m->level != 0 && m->level != l) return -1;

    h = (HARD_TASK_MODEL *)m;
    if (!h->wcet || !h->mit) return -1;

    Model Type Identification
    (A Model can be handled if it is of the right pclass and it has the right level number)

    Cast to the right type
    (only after checking the real type!)

    Check if all the data in the Model is filled in the right way
if (lev->flags & SIMPLEEDF_ENABLE_GUARANTEE) {
    bandwidth_t b;
    b = (MAX_BANDWIDTH / h->mit) * h->wcet;
    if (MAX_BANDWIDTH - lev->U > b)
        lev->U += b;
    else
        return -1;
}
lev->period[p] = h->mit;
if (h->periodicity == APERIODIC)
    lev->flag[p] = SIMPLEEDF_FLAG_SPORADIC;
else
    lev->flag[p] = 0;
lev->deadline_timer[p] = -1;

A check is done to see if the tasks allocated to EDF uses a bandwidth >1
The guarantee failed because the Utot of the EDF tasks >1
Here we know that the task can be accepted by the EDF module. We do not know yet if it can be accepted systemwide. To know that the task creation primitive will call guarantee()
Fill Module private data
Enable the monitoring of the execution time of the task. The CONTROL_CAP features are provided by the Generic Kernel. A module can also implement its own temporal isolation, as for example the Polling Server (kernel/modules/ps.c)
guarantee

static int SIMPLEEDF_public_guarantee(LEVEL l,
    bandwidth_t *freebandwidth)
{
    SIMPLEEDF_level_des *lev =
            (SIMPLEEDF_level_des *)(level_table[l]);

    if (*freebandwidth >= lev->U) {
        *freebandwidth -= lev->U;
        return 1;
    }
    else
    
        return 0;
}

There is enough bandwidth
to schedule all the EDF tasks

There is not enough bandwidth
task activation

static void SIMPLEEDF_public_activate(LEVEL l, PID p)
{
    SIMPLEEDF_level_des *lev =
        (SIMPLEEDF_level_des *)(level_table[l]);
    struct timespec *temp;

    if (proc_table[p].status == SIMPLEEDF_WAIT) {
        kern_raise(XACTIVATION,p); return;
    }

    if (proc_table[p].status != SLEEP &&
        proc_table[p].status != SIMPLEEDF_WCET_VIOLATED)
        return;

    Task activated too early

    Task already activated

temp = iq_query_timespec(p, &lev->ready);
kern_gettime(temp);
ADDUSEC2TIMESPEC(lev->period[p], temp);

/* Insert task in the correct position */
proc_table[p].status = SIMPLEEDF_READY;
iq_timespec_insert(p,&lev->ready);

/* Set the deadline timer */
lev->deadline_timer[p] = kern_event_post(temp,
    SIMPLEEDF_timer_deadline, (void *)p);
}
deadline event

```c
static void SIMPLEEDF_timer_deadline(void *par)
{
    PID p = (PID) par;  SIMPLEEDF_level_des *lev;
    struct timespec *temp;
    lev = (SIMPLEEDF_level_des *)
        level_table[proc_table[p].task_level];
    switch (proc_table[p].status) {
    case SIMPLEEDF_ZOMBIE:
        proc_table[p].status = FREE;
        iq_insertfirst(p,&freedesc);
        lev->U -= (MAX_BANDWIDTH/lev->period[p]) * proc_table[p].wcet;
        break;
    }
```

Free the task descriptor if the task ended correctly
case SIMPLEEDF_IDLE:
    temp = iq_query_timespec(p, &lev->ready);
    ADDUSEC2TIMESPEC(lev->period[p], temp);
    proc_table[p].status = SIMPLEEDF_READY;
    iq_timespec_insert(p, &lev->ready);
    lev->deadline_timer[p] = kern_event_post(temp, SIMPLEEDF_timer_deadline, (void *)p);
    event_need_reschedule();
    break;
The task was aperiodic

```c

deadline event (3)

case SIMPLEEDF_WAIT:
    proc_table[p].status = SLEEP;
    break;

default:
    kern_raise(XDEADLINE_MISS, p);

```

What ?!?
preemption or wcet exaustion

static void SIMPLEEDF_public_epilogue(LEVEL l, PID p)
{
    SIMPLEEDF_level_des *lev =
        (SIMPLEEDF_level_des *)(level_table[l]);

    if ((lev->flags & SIMPLEEDF_ENABLE_WCET_CHECK) &&
        proc_table[p].avail_time <= 0) {
        kern_raise(XWCET_VIOLATION,p);
        proc_table[p].status = SIMPLEEDF_WCET_VIOLATED;
    }
    else {
        iq_timespec_insert(p,&lev->ready);
        proc_table[p].status = SIMPLEEDF_READY;
    }
}
The task has already been extracted from the ready queue by public_dispatch().

NOTE: The blocked queue will be handled by the synchronization point function (e.g., sem_wait, not by the scheduling modules!)
synchronization points (2)

static void SIMPLEEDF_public_unblock(LEVEL l, PID p)
{
    SIMPLEEDF_level_des *lev =
        (SIMPLEEDF_level_des *)(level_table[l]);

    proc_table[p].status = SIMPLEEDF_READY;
    iq_timespec_insert(p,&lev->ready);
}

When the task exit from a synchronization point (e.g., sem_post), the task is put again in the ready state.
static int SIMPLEEDF_public_message(LEVEL l, PID p, void *m) {
    SIMPLEEDF_level_des *lev=(SIMPLEEDF_level_des *)(level_table[l]);

    if (!lev->flag[p] & SIMPLEEDF_FLAG_SPORADIC)
        proc_table[p].status = SIMPLEEDF_IDLE;
    else
        proc_table[p].status = SIMPLEEDF_WAIT;

    if (lev->flags & SIMPLEEDF_ENABLE_WCET_CHECK)
        proc_table[p].avail_time = proc_table[p].wcet;
    jet_update_endcycle();
    return 0;
}
task end

static void SIMPLEEDF_public_end(LEVEL l, PID p)
{
    proc_table[p].status = SIMPLEEDF_ZOMBIE;
}
contact info

Paolo Gai

pj@evidence.eu.com

(please use the Shark Forum for SHaRK related questions!)

http://feanor.sssup.it/~pj
contact info

Luca Santinelli

luca.santinelli@sssup.it