Kaya project specifications

What's in a nucleus?

- The basic facilities of a modern OS:
  - the ability to start up and shut down the system
  - the "process" abstraction
  - a way to manage exceptions and I/O, and to notify these events to higher Kaya levels for management
  - some synchronization primitives
  - a way of sharing the CPU among different processes

Kaya upper levels would use these facilities for VM, I/O and user process management (generic processes in Phase 2 will become resource managers and user processes in Phase 3)

Kaya nucleus goals:

- The nucleus should implement the following features:
  - basic initialization of the system after boot
  - creation and termination of asynchronous sequential processes
  - CPU scheduler and nucleus pseudo-clock
  - service request management (SYSCALLs)
  - exception and interrupt handlers
  - P, V and other synchronization primitives
  - "pass up" of requests and events to upper levels
  - system shutdown when no more processes exist

Will use the modules developed in Kaya Phase 1
Some definitions and specifications:
The nucleus should:
- use physical memory addresses
- guarantee finite progress to processes (no starvation)
- preserve process state (e.g., should keep VM on, Kernel or User mode, etc., if these are set)
Some definitions:
- process count: the total number of processes in the system
- soft block count: the number of processes waiting for I/O to complete
- current process: the process currently using the CPU
- ready queue: the queue of processes waiting to get CPU time
- deadlock: is when all processes are blocked (but not on I/O), so there is no way to guarantee finite progress

Scheduler: the beating heart of the system
Minimum scheduler requirements: round-robin with a 5 milliseconds time slice (implementation of more advanced algorithms gives a bonus)
What to do if the ready queue is empty:
- if the process count is zero, perform normal system shutdown
- if a deadlock is detected (that is, when the process count is higher than zero but the soft-block count is zero), perform emergency shutdown
- if process count and soft-block count are higher than zero, processes are waiting for I/O to complete, so the system should enter in a wait state (that is, waiting until some device completes its operation)

Scheduler specifications (cont’d):
- normal system shutdown: (optionally) print out some meaningful information and call HALT()
- emergency shutdown: (optionally) print out some meaningful information and call PANIC()
- how to handle a wait state: best way is to schedule a process performing an infinite loop with interrupts enabled
Remember: the number of clock ticks in 5 milliseconds depends on CPU speed (check Timescale)

uMPS software development
ROM routines strike again:
unsigned int SYSCALL(unsigned int number, unsigned int arg1, unsigned int arg2, unsigned int arg3): generates a SYSCALL exception
unsigned int STST(state_t * statep): saves the current status of the process

<table>
<thead>
<tr>
<th>Code</th>
<th>ROM Service</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>void LOST(state_t * statep)</td>
<td>LOST</td>
<td></td>
</tr>
<tr>
<td>void FORK(unsigned int entryhi, unsigned int status, unsigned int po, state_t *statep)</td>
<td>FORK</td>
<td></td>
</tr>
<tr>
<td>void PANIC()</td>
<td>PANIC</td>
<td></td>
</tr>
<tr>
<td>void HALT()</td>
<td>HALT</td>
<td></td>
</tr>
</tbody>
</table>
Kaya project specifications

Starting up the system:

Boot ROM will call your `main()`; it should:
- initialize nucleus data structures (e.g., ProcBlk queues and ASLs, but also status variables like process count and so on)
- initialize exception vectors
- create a brand new `test()` process and put it in the ready queue
- call the scheduler

The `test()` process will verify the correctness of the nucleus features (it is provided in `p2test.c`)

After start-up, the only way to re-enter nucleus should be by using exceptions and interrupts

Some implementation details:

Remember that exception vectors have fixed addresses in memory

An easy way to initialize most process state fields is to use `STST()`

To set up a correct process state:
- PC should be set to the address of the function representing the starting point of the process (or of the exception handling function)
- $sp$ should point to somewhere in high memory (near RAMTOP)
- $t9$ should be set to the same value as PC, to correctly access the GOT

It is a very sensible measure to assign different stack areas (that is, different $sp$s) to different processes or exception handlers; as a general rule, 1 KByte is enough for most processes and handlers

System memory information is in the bus register area; nucleus memory map is in the `.aout` header

Some implementation details (cont’d):

`test()` process should start in Kernel mode, interrupts on, VM off (remember to set `CP0.Status` register flags), and with a fresh stack frame (set up $sp$ accordingly)

To set up the PC and other registers to `test()` starting address, an easy way is to use it like a pointer with a cast, that is:

```
...PC = (memaddr) test;
```

Where `memaddr` is a type (`unsigned int`) defined in `types.h` to simplify definition of memory addresses

Since `test()` function is defined in `p2test.c`, you have to declare a:

```
extern void test();
```

In your code to allow linking with `p2test.o`

After the start-up:

`test()` will start testing nucleus features

At some point, it will require access to nucleus services: how?

By raising a SYSCALL exception:

```
test() current process state will be saved into SYSCALL/BREAK “old” area
the exception cause will be saved in the `Cause.ExcCode` in the old area
the process state in the corresponding “new” area will be loaded by ROM routines (it will be your responsibility to set it)
SYSCALL parameters will be passed through `$a0...$a3` registers in the old area; `$a0` will contain the SYSCALL number
```

From a process’ point of view, service requests will be like function calls
Enter the Sys/Bp exception handler... It should:

detect the exact cause of the exception (check Cause.ExcCode)

if it is a request the nucleus can service (a SYSCALL numbered 1 to 8) and
the process requesting it is in Kernel mode, service the request

if a SYSCALL 1-8 is requested by a process in User mode, simulate a
PgmTrap exception (see further on)

if the request is a SYSCALL numbered 9-up or a BREAK, the handler should
check if it has to be "passed up"

to "pass up" a request = load a new process state able to service the request;
the requesting process should have already defined which process will be
able to do it (by using a specific SYSCALL)

if the pass up is not possible, terminate the process (and its progeny)

after servicing the request, return an exit code (through $v0) to the requesting
process, if required

Nucleus services:

Create Process (SYS1):

a new process (said to be the progeny of the requesting
process) should be created

$\texttt{a1}$ will contain the physical address of the process state area
descriving the new process

the process requesting the SYS1 service continues to exist

if all goes well, return an exit code of 0 (through $\texttt{v0}$) to the
requesting process

if the new process cannot be created due to lack of resources
(eg. no more free ProcBlks), return an exit code of -1

Terminate Process (SYS2):

the calling process and all its progeny should be terminated in
a recursive way

this SYSCALL should end only when all relevant processes are
terminated

there are no exit codes

Verhogen (SYS3):

$\texttt{a1}$ will contain the physical address of the integer
representing the semaphore

the nucleus should handle all the consequences of a V
operation on that semaphore

there are no exit codes
Kaya project specifications

Nucleus services (cont’d):

Passeren (SYS4):

- \$a1 will contain the physical address of the integer representing the semaphore
- the nucleus should handle all the consequences of a P operation on that semaphore
- there are no exit codes

Specify Exception State Vector (SYS5):

- this operation allows a process to specify his own exception state vector for specific exception types
- this vector is a process state to be loaded if the during the execution of the specifying process an exception of that type is generated; this is more useful than it seems (eg. virtual memory management and pass-up is performed this way)
- \$a1 will define the type of the exception to be managed:
  - 0: TLB exceptions
  - 1: program traps
  - 2: SYSCALL and BREAK exceptions
- interrupts need not to be managed (nucleus will do it)

Specify Exception State Vector (SYS5) continued:

- \$a2 will contain the physical address of the "old" area, where the state of the process who has generated the exception will be saved
- \$a3 will contain the physical address of the "new" area, from where to load the process state which will handle the exception upon this request, the nucleus should:
  - check if a previous SYS5 request for the same exception type has already been made by this process (if so, terminate the process and its progeny)
  - if not, keep track of the exception type and of the two process state areas for this process

"pass-up" of SYSCALL numbered 9-up is done this way

Remember:

- SYSCALL 1-8 are reserved to processes in Kernel mode
- BREAK 0..3 are reserved to ROM routines
Kaya project specifications

Nucleus services (cont’d):

Get CPU Time (SYS6):
returns the CPU time (in microseconds) used up by the calling process through $v0
Remember: clock ticks are not microseconds
You have to decide which process is “charged” when servicing SYSCALLs, interrupts and so on

Wait For Clock (SYS7):
The nucleus maintains a pseudo-clock (similar to a semaphore V-ed every 100 milliseconds by the nucleus itself)
SYS7 allows the calling process to do a P operation on this semaphore, thus blocking the process until it is V-ed by the nucleus
Remember:
all processes waiting on the pseudo-clock have to be unblocked when it is V-ed
a process should always be blocked for a while when calling SYS7
pseudo-clock time accounting should be quite precise

Wait For IO Device (SYS8):
the idea is to have processes starting I/O operations (= writing commands in device registers), then “go to sleep” until the I/O operation completes (that is, no busy waiting)
SYS8 allow processes to request the nucleus to be “put to sleep”; it is equivalent to perform a P on a semaphore associated to the specific device
when the I/O operation is completed, the semaphore is V-ed and the process will continue
the device is identified by interrupt line and device numbers; terminal receivers and transmitters have to be managed like separate devices (a flag will allow to identify which one)

Wait For IO Device (SYS8) continued:
SYS8 parameters:
intlNo: interrupt line number (in $a1)
dnum: device number (in $a2)
waitForTermRead flag (TRUE for receiver, FALSE for transmitter) in $a3
SYS8 exit code (through $v0): the device register STATUS upon I/O operation completion
Remember:
CPU scheduling could make the SYS8 be called after the I/O operation completes; the STATUS has to be recorded and provided to the calling process (which will not block)
**Kaya project specifications**

**Nucleus services (cont’d):**

About process blocking (and unblocking):
- some SYSCALLs require to perform operations on a semaphore:
  - remember of the ASL data structure implemented in Phase1 to block a process on a semaphore
  - set semaphore values accordingly to P and V logic
  - when a process is unblocked, put it in the ready queue

---

**Kaya project specifications**

**Exception and I/O Management:**

The nucleus should provide exception handlers for all uMPS exception types:
- Program Traps (PgmTrap)
- TLB Management (TLB)
- SYSCALL/Breakpoint (SYS/Bp)
- Interrupts (Ints)

TLB and PgmTrap should be handled depending on the fact the offending process has performed a SYS5 for these exceptions (*"pass up"* the exception if it has, terminate the process if not)

SYS/Bp will be the entry point for nucleus services’ requests

---

**Kaya project specifications**

**Exception and I/O Management (cont’d):**

How to perform exception management for processes in User mode requesting SYSCALL 1-8:
- SYSCALL 1-8 are reserved to processes in Kernel mode
- if a process in User mode has requested such a SYSCALL, it has to be managed like a PgmTrap exception, that is:
  - copy the processor state from SYS/Bp "old" area to PgmTrap "old" area
  - set the corresponding `Cause.ExcCode` to RI (Reserved Instruction) exception code
  - invoke the nucleus PgmTrap handler routine

---

**Kaya project specifications**

**Exception and I/O Management (cont’d):**

Process termination issues:
- an exception will require the termination of the offending process in many cases (also when SYS2 is requested)
- process termination will require that:
  - the offending process and its subtree has to be "detached" from the process tree
  - all process progeny has to be terminated too
  - if a process was blocked on a synchronisation semaphore (SYS4), the semaphore value has to be adjusted
  - nucleus internal variables (e.g. process count) have to be adjusted
Exception and I/O Management (cont'd):

How to handle I/O interrupts:

- Interrupts are handled by the nucleus on behalf of processes.
- Interrupt priority is defined by interrupt line and device number (higher number = lower priority).
- A terminal receiver has lower priority than its corresponding transmitter.
- The nucleus should service one interrupt at a time: the highest priority one (look at Cause.IP and Interrupting Devices bitmap).
- Multiple pending interrupts will raise a sequence of exceptions.
- I/O device interrupts have to be acknowledged by the nucleus and handled according to SYS8 specifications (store the result if no process is waiting, V the semaphore unblocking the process, and so on).

Exception and I/O Management (cont'd):

How to handle Interval Timer interrupts:

- Interval Timer is the highest priority device.
- Interrupt is raised on underflow, that is, on 0x0000.0000 → 0xFFFF.FFFF transition.
- Interval Timer may be used to maintain the nucleus scheduler time slice and the pseudo-clock.
- To acknowledge the interrupt, load a new value in the Interval Timer.

Some implementation details:

- Performing exception management requires processor states to be saved and loaded.
- Remember that:
  - CP0.Status VM, KU, and IE bits are pushed when an exception occurs, and popped when returning from an exception (set processor state accordingly when starting a new process).
  - The processor state should be restored (if possible) after an exception as it was before the exception, but in one case has to be changed: when servicing a SYSCALL exception, the PC has to be incremented by 4 to avoid a loop.

Some observations on Phase 2:

- Specifications may not look clear: you have to think there is a still bigger picture.
- Some nucleus features are less critical than others (e.g., CPU time accounting): start working on basic, critical features you will have to decide how to handle a number of situations.
- Writing the code:
  - Work on one main feature at a time.
  - Reuse phase 1 code; do not fear to improve it.
  - Keep it simple.
  - Check for error conditions (plan for the unforeseen).
  - Analyze and understand p2test.c.
Kaya project specifications

Errata corrigae:

*uMPS Principles of Operation*, p. 26:
- DBE exception code is 6
- IBE exception code is 7

*Student Guide to the Kaya Operating System Project*, p. 33:
- BREAK exceptions handled by ROM are numbered 0..3