

*Laboratorio di Sistemi Operativi*  
*Anno Accademico 2006-2007*

### **AMIKaya Phase 2 Project Specifications**

Enrico Cataldi, Mauro Morsiani, Renzo Davoli

Copyright © 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli.  
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free  
Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no  
Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found at:  
<http://www.gnu.org/licenses/fdl.html#FOCI>

### AMIKaya project specifications

---

#### ◆ **Errata corrige:**

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

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

2

## AMIKaya project specifications

---

- ◆ **AMIKaya is:**
  - ◆ a complete Operating System project specification developed by Enrico Cataldi and based on Kaya and AMIKE
  - ◆ designed as a *microkernel* system, to be developed in a number of *phases*; can be implemented using the uMPS simulator
  - ◆ Phase 2 is the second software layer of AMIKaya (below it there are: bare hardware, ROM microcode and Phase 1 ADTs)

## AMIKaya project specifications

---

- ◆ **AMIKaya Phase 2 requires the development of:**
  - ◆ The OS nucleus, including:
    - ◆ a message passing subsystem
    - ◆ interrupt and exception event handling
  - ◆ The implementation of the System Service Interface (SSI)
  - ◆ What has been developed in Phase 1 has to be considered a starting point, but this doesn't mean that it cannot be modified or extended

## AMIKaya project specifications

---

### ◆ What's in the nucleus?

- ◆ The basic facilities of a modern microkernel OS:
  - ◆ the ability to start up and shut down the system
  - ◆ thread creation, execution and destruction
  - ◆ low level CPU scheduling
  - ◆ thread synchronization primitives based on the message passing paradigm
  - ◆ a way to manage exceptions and I/O, and to notify these events to the SSI and/or to higher AMIKaya levels for management
- ◆ AMIKaya upper levels would use these facilities for VM, I/O and user thread management (generic threads in Phase 2 will become resource managers and user threads in Phase 3)

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

5

## AMIKaya project specifications

---

### ◆ AMIKaya nucleus goals:

- ◆ The nucleus should implement the following features:
  - ◆ basic initialization of the system after boot
  - ◆ creation and termination of *asynchronous sequential threads*
  - ◆ CPU scheduler and nucleus *pseudo-clock*
  - ◆ exception and interrupt handlers
  - ◆ message passing facilities
  - ◆ SSI initialization and start-up
  - ◆ “pass up” of requests and events to upper levels
  - ◆ system shutdown when no more serviceable threads exist
- ◆ Will use the modules developed in AMIKaya Phase 1

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

6

## AMIKaya project specifications

---

- ◆ **The idea behind a microkernel is that:**
  - ◆ A nucleus is required only at the lowest OS level
  - ◆ All other services are demanded to dedicated Kernel “server” threads
  - ◆ Server threads need to have access to the Kernel address space to manage devices, threads, scheduling policies
  - ◆ There should be some communication mechanism to allow threads to synchronize and to get their requests satisfied
  - ◆ The Message Passing paradigm allows to:
    - ◆ synchronize threads
    - ◆ require services

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

7

## AMIKaya project specifications

---

- ◆ **Message Passing in AMIKaya:**
  - ◆ Threads can be referred by the address of their ThreadBLK
  - ◆ Message Passing should be implemented to allow:
    - ◆ *Asynchronous message send*: a thread sending a message should not block its execution waiting for a reply
    - ◆ *Synchronous message receive*: a thread asking to receive a message should block its execution until a message is received
  - ◆ How to implement it:
    - ◆ To send/receive a message = to perform a SYSCALL; that is, an exception managed by the nucleus
    - ◆ The nucleus will exchange messages on behalf of threads
    - ◆ “to block” a thread = put it in a waiting queue

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

8

## AMIKaya project specifications

### ◆ ROM routines strike again:

- ◆ unsigned int SYSCALL(unsigned int number, unsigned int arg1, unsigned int arg2, unsigned int arg3)

Raise a SYSCALL exception, mapping function arguments to processor registers **\$a0..\$a3**

- ◆ unsigned int STST(state\_t \* statep)

Save the current status of the processor

C usage	ROM Service/Instr.
void LDST(state_t *statep)	<b>LDST</b>
void FORK(unsigned int entryhi, unsigned int status, unsigned int pc, state_t *statep)	<b>FORK</b>
void PANIC()	<b>PANIC</b>
void HALT()	<b>HALT</b>

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

9

## AMIKaya project specifications

### ◆ MsgSend:

- ◆ unsigned int MsgSend(unsigned int code, tcb\_t \* dest, unsigned int payload)

- ◆ Thread request to send a message

- ◆ Will be mapped to SYSCALL

- ◆ code = **\$a0** = SEND

- ◆ **\$a1** should contain the receiver id, **\$a2** the payload of the message

- ◆ if all goes well, return an exit code of 0 (through **\$v0**) to the requesting thread

- ◆ if the operation cannot be completed due to lack of resources (eg. no more free MessageBLK), return MSGNOGOOD

- ◆ this operation is *asynchronous*: the thread will not be blocked by the nucleus

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

10

## AMIKaya project specifications

### ◆ **MsgRecv:**

- `tcb_t * MsgRecv(unsigned int code, tcb_t * source, unsigned int * reply)`
  - ◆ Thread request to receive a message
  - ◆ Will be mapped to SYSCALL
  - ◆ code = **\$a0** = `RECV`
  - ◆ **\$a1** should contain the required source id: it could be a specific thread address, or it could be `ANYMESSAGE` to get the first message available in the thread `inbox` queue
  - ◆ **\$a2** should be a pointer to the thread variable where the message `payload` will be copied upon reception
  - ◆ Will return the sender thread address (through **\$v0**) to the requesting thread
  - ◆ This operation is *synchronous*: the thread will be blocked by the nucleus until such a message is received

## AMIKaya project specifications

### ◆ **Some implementation details**

- ◆ `MsgSend/MsgRecv` should be invoked only by threads in Kernel mode
- ◆ `MsgSend/MsgRecv` from threads in User mode, or attempts to exchange messages with non-existent threads should be considered like *traps* and could cause the termination of the calling thread and its *progeny*
- ◆ It will be nucleus' responsibility to set **\$v0** and `reply` values correctly
- ◆ Adjust `MAXMESSAGES` to allow `p2test` to run without getting `MSGNOGOOD` errors
- ◆ "to block" a thread = put it in a waiting queue
- ◆ "to not block" a thread = a scheduling policy decision:
  - ◆ reload the thread state in the processor
  - ◆ put it in a ready queue
- ◆ Remember: to return from a SYSCALL, the PC need to be incremented by one instruction (4 bytes) to avoid a loop

## AMIKaya project specifications

---

### ◆ How to require a service?

- ◆ To require a service = to send a message
- ◆ We can make use of basic Message Passing, but...
  - ◆ There is an *addressing problem*: you have to know the address of the server thread to ask a service
  - ◆ There is a *security problem*: you may want to keep server thread addresses hidden to user threads
  - ◆ There is a *payload problem*: you may have to transport messages larger than the hardware/software interface allows for
- ◆ Or we can make a clever use of Message Passing:
  - ◆ The SSI will provide an efficient and secure link between user threads, server threads and other parts of the system

## AMIKaya project specifications

---

### ◆ About the System Service Interface (SSI):

- ◆ Runs inside the Kernel address space
- ◆ Each relevant system event will become a message managed by the SSI
- ◆ Will be the only server thread which will exchange messages with other threads: for example, a thread which requires to know its accounted CPU time will send a message to the SSI, and wait for the answer
- ◆ The SSI thread should implement the following RPC server algorithm:

```
while (TRUE) {
    receive a request;
    satisfy the received request;
    send back the results;
}
```

## AMIKaya project specifications

---

### ◆ About other trap management threads:

- ◆ The SSI is the only server thread which you must write in Phase 2: some sample trap management threads will be provided as stub in `p2test.c` (the real implementation is demanded to AMIKaya's successive phases)
- ◆ Trap management threads run inside the Kernel address space
- ◆ Trap management threads do not exchange messages directly with other threads
- ◆ Each trap management thread implements the following RPC server algorithm:

```
while (TRUE) {  
    receive a trap management request;  
    manage the request;  
    send back a decision upon continuing the trapped  
    thread or not;  
}
```

## AMIKaya project specifications

---

### ◆ The way a thread makes a service request:

- ◆ `void SSIRequest(unsigned int service, unsigned int payload, unsigned int * reply)`
- ◆ `service` is a mnemonic code identifying the service requested, `payload` contains an argument (if required) for the service, and `reply` will point to the area where the answer (if required) should be stored
- ◆ If `service` does not match any of those provided by the SSI, the SSI should terminate the thread and its progeny
- ◆ When a thread requires a service to the SSI, it *must wait* for the answer
- ◆ `SSIRequest` has to be implemented by you, finding solutions for:
  - ◆ the addressing problem
  - ◆ the security problem
  - ◆ the payload problem

## AMIKaya project specifications

---

### ◆ SSIRequest hints:

- ◆ You have to use a combination of `MsgSend/MsgRecv` to perform the required message exchange
- ◆ You can hide the SSI thread address inside the `SSIRequest` function (simpler but less safe), or use a *magic number* to address the SSI (choose - and document - the magic number wisely to get a bonus)
- ◆ You can use and/or expand `msg_t` elements to solve the payload problem

## AMIKaya project specifications

---

### ◆ Some definitions and specifications:

- ◆ The nucleus should:
  - ◆ use physical memory addresses
  - ◆ guarantee *finite progress* to threads (no starvation)
  - ◆ *preserve* thread state (eg. should keep VM on, Kernel or User mode, etc., if these are set)
- ◆ Some definitions:
  - ◆ *thread count*: the total number of thread in the system
  - ◆ *soft block count*: the number of thread waiting for I/O or completion of a service request by the SSI
  - ◆ *current thread*: the thread currently using the CPU
  - ◆ *ready queue*: the queue of threads waiting to get the CPU
  - ◆ *deadlock*: is when all threads are blocked waiting for a message (but *not* on I/O or service request), 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 and documentation of a more sophisticated scheduler gives a bonus)
- ◆ What to do if the ready queue is empty:
  - ◆ if the thread count = 1 and the SSI is the only thread in the system: *normal system shutdown*
  - ◆ if a deadlock is detected (that is, after system boot, if the thread count is higher than zero but the soft-block count is zero): *emergency shutdown*
  - ◆ if thread count and soft-block count are higher than zero, some threads are waiting for I/O to complete: the system should enter in a *wait state* (that is, waiting until some device completes its operation)

◆ **Scheduler specifications:**

- ◆ *normal system shutdown*: terminate SSI, (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 thread performing an infinite loop with interrupts enabled
- ◆ Remember: the number of clock ticks in 5 milliseconds depends on CPU speed (remember Timescale)

## AMIKaya project specifications

---

### ◆ Some tough scheduling policy issues:

- ◆ What to do when:
  - ◆ a service is requested
  - ◆ a message gets sent
  - ◆ a message is received
  - ◆ an exception is raised
  - ◆ an interrupt is received
  - ◆ ...
- ◆ "Who pays the bill"?
  - ◆ CPU time accounting should be implemented
  - ◆ two choices:
    - ◆ 32-bit: easier to implement
    - ◆ 64-bit: more complex (gives a bonus)
  - ◆ always use TODHI and TODLO together as timestamp

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

21

## AMIKaya project specifications

---

### ◆ Starting up the system:

- ◆ Boot ROM will call *your* `main()`; it should:
  - ◆ initialize nucleus data structures (eg. ThreadBLK and MessageBLK queues, but also status variables like thread count and so on) and *exception vectors*
  - ◆ Initialize SSI thread and put it in the ready queue
  - ◆ create a brand new `test()` thread and put it in the ready queue
  - ◆ call the scheduler
- ◆ the `test()` thread 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

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

22

## AMIKaya project specifications

---

### ◆ Some implementation details:

- ◆ Remember that exception vectors have fixed addresses in memory
- ◆ An easy way to initialize most thread state fields is to use `STST( )`
- ◆ To set up a correct thread state:
  - ◆ PC should be set to the address of the function representing the starting point of the thread (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 `$sps`) to different threads or exception handlers; as a general rule, 1 KByte is enough for most threads and handlers
  - ◆ system memory information is in the bus register area; nucleus memory map is in the `.aout` header

## AMIKaya project specifications

---

### ◆ Some implementation details:

- ◆ SSI requires a special treatment by the scheduler:
  - ◆ if the SSI ever gets terminated: *emergency shutdown*
  - ◆ run with an infinite time slice and with interrupts off, *or*
  - ◆ run with a finite time slice and interrupts on, but protecting its critical sections from interrupts (this more advanced solution gives a bonus)
- ◆ SSI should start in Kernel mode, with VM off (remember to set **CP0.Status** register flags)

## AMIKaya project specifications

### ◆ Some implementation details:

- ◆ `test()` thread should start:
  - ◆ in Kernel mode, with interrupts on, VM off (remember to set **CP0.Status** register flags)
  - ◆ with enough stack area available for all the threads it will create (set up **\$sp** accordingly). Hint: create `test()` *after* exception handlers and SSI
- ◆ 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`

## AMIKaya project specifications

### ◆ After the start-up:

- ◆ `test()` will start testing nucleus features
- ◆ At some point, it will require access to nucleus services: how?
- ◆ By invoking a SYSCALL, thus raising an exception:
  - ◆ `test()` current thread state will be saved into SYSCALL/BREAK “old” area
  - ◆ the exception cause will be saved in the **Cause.ExcCode** in the old area
  - ◆ the thread 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 thread point of view, service requests will be like function calls

- ◆ **Enter the Sys/Bp exception handler... It should:**
  - ◆ Update the processor state of the thread that generated the exception
  - ◆ Detect the exact cause of the exception (check **Cause.ExcCode**)
  - ◆ If it is a request the nucleus can service (a SYSCALL 1-2) *and* the thread requesting it is in Kernel mode, process the message passing request
  - ◆ In any other case, handle the request as a “pass up” attempt to some trap management thread, if defined
  - ◆ If the pass up is not possible, terminate the thread and its progeny
  - ◆ Call the scheduler

- ◆ **Which services need to be implemented?**
  - ◆ “Brother” thread creation
  - ◆ “Child” thread creation
  - ◆ Termination request
  - ◆ PRG Trap Manager specification
  - ◆ TLB Trap Manager specification
  - ◆ SYS Trap Manager specification
  - ◆ Get CPU Time
  - ◆ Freeze until next pseudo-clock tick
  - ◆ Freeze until I/O operation completion

◆ **Nucleus services:**

◆ **CREATEBROTHER**

- ◆ Create a new thread whose initial processor state is passed by reference as service request `payload`
- ◆ The new thread should be inserted in the thread tree as a brother of the caller
- ◆ If all went fine, return the created thread address, otherwise `CREATENOGOOD`

◆ **Nucleus services:**

◆ **CREATESON**

- ◆ Create a new thread whose initial processor state is passed by reference as service request `payload`
- ◆ The new thread should be inserted in the thread tree as a son of the caller
- ◆ The new thread should inherit trap managers from the parent, if defined
- ◆ If all went fine, return the created thread address, otherwise `CREATENOGOOD`

- ◆ Q: why is it necessary to distinguish brothers from sons?

◆ **Nucleus services:**

◆ **TERMINATE**

- ◆ The caller thread and all of its progeny should be terminated in a recursive way
- ◆ Messages sent from the caller thread or from one of its progeny to other threads still have to be delivered, but the SSI and/or the nucleus should discard further requests from the terminated threads
- ◆ `payload` is not used
- ◆ there are no return codes

◆ **Nucleus services:**

◆ **SPECPRGMGR**

- ◆ This request allows a thread to specify his own trap management thread for Program Trap exceptions
- ◆ The trap management thread address is passed as service request `payload`; if the management thread does not exist, the requestor thread and its progeny should be terminated
- ◆ Once assigned or inherited, the trap management thread cannot be redefined: further service requests of this type should cause the termination of the thread and its progeny
- ◆ `reply` is not used (leave unchanged)

◆ **Nucleus services:**

◆ **SPECTLBMGR**

- ◆ This request allows a thread to specify his own trap management thread for TLB Management exceptions
- ◆ The trap management thread address is passed as service request `payload`; if the management thread does not exist, the requestor thread and its progeny should be terminated
- ◆ Once assigned or inherited, the trap management thread cannot be redefined: further service requests of this type should cause the termination of the thread and its progeny
- ◆ `reply` is not used (leave unchanged)

◆ **Nucleus services:**

◆ **SPECSYSMGR**

- ◆ This request allows a thread to specify his own trap management thread for SYSCALL/Break exceptions
- ◆ The trap management thread address is passed as service request `payload`; if the management thread does not exist, the requestor thread and its progeny should be terminated
- ◆ Once assigned or inherited, the trap management thread cannot be redefined: further service requests of this type should cause the termination of the thread and its progeny
- ◆ This trap management thread will be called when “pass up” of SYSCALLs and service requests is attempted
- ◆ `reply` is not used (leave unchanged)

◆ **Nucleus services:**

◆ **GETCPU**

- ◆ Return the CPU time (in microseconds) used by the requesting thread
- ◆ `payload` is not used
- ◆ Remember: clock ticks are *not* microseconds
- ◆ You have to decide which thread is “charged” when servicing SYSCALLs, interrupts and so on

◆ **Nucleus services:**

◆ **WAITFORCLOCK**

- ◆ The nucleus maintains a *pseudo-clock* (a pseudo-device which should raise an interrupt every 100 milliseconds)
- ◆ Upon request, block the requesting thread until the next pseudo-clock tick
- ◆ `payload` is not used
- ◆ `reply` is not used (leave unchanged)
- ◆ Remember:
  - ◆ all threads waiting on the pseudo-clock have to be unblocked when it ticks
  - ◆ pseudo-clock time accounting should be quite precise: at most one interrupt message should be pending in the SSI inbox

◆ **Nucleus services:**

◆ **WAITFORIO**

- ◆ The idea is to have threads starting I/O operations (= writing commands in device registers), then “go to sleep” until the I/O operation completes (that is, no busy waiting)
- ◆ WAITFORIO allow threads to request to be “put to sleep”
- ◆ When the I/O operation is completed, the thread should be able to continue
- ◆ The device is identified by its *device register base address* (that is, the address of its **STATUS** register)
- ◆ Remember: terminal receivers and transmitters have to be managed like separate devices

◆ **Nucleus services:**

◆ **WAITFORIO**

- ◆ Block the requesting thread, until the device specified by `payload` completes its I/O operation
- ◆ If the device does not exist, terminate the thread and its progeny
- ◆ Return the **STATUS** register value of the device upon I/O completion
- ◆ Remember:
  - ◆ because of CPU scheduling, the I/O operation could be completed before the service request could be processed: the **STATUS** upon I/O completion has to be recorded until the thread performs the service request

### ◆ Exception and I/O Management

- ◆ The nucleus should provide basic exception handlers for all uMPS exception types:
  - ◆ *Program Traps* (PgmTrap) & *TLB Management* (TLB)
  - ◆ *SYSCALL/Breakpoint* (SYS/Bp)
  - ◆ *Interrupts* (Ints)
- ◆ Exceptions should be handled depending on the fact the offending thread has requested or inherited a SPECPRGMGR / SPECTLBMGR / SPECSYSMGR:
  - ◆ “pass up” the exception, if it has
  - ◆ terminate the thread and its progeny, if not
- ◆ Interrupts will be managed only by the nucleus

### ◆ Exception and I/O Management

- ◆ **How a trap management thread works:**
  - ◆ upon exception, the current thread is stopped and its state is saved in the old area
  - ◆ the nucleus should update the thread processor state with the one saved in the old area, and dispatch a message to the trap management thread with:
    - ◆ the current thread as sender
    - ◆ the value of the **CAUSE** register as payload
  - ◆ the trap management thread will act upon the request and, at last, will send a message to the thread raising the trap with TRAPCONTINUE or TRAPTERMINATE as payload
  - ◆ the nucleus must intercept this message and cause the thread termination or continuation

- ◆ **Exception and I/O Management**

- ◆ **Thread termination issues:**

- ◆ an exception will require the termination of the offending thread in many cases
- ◆ thread termination will require that:
  - ◆ the offending thread and its subtree has to be "detached" from the thread tree
  - ◆ all thread progeny has to be terminated too
  - ◆ messages sent from the caller thread or one of its progeny to other threads still have to be delivered, but the SSI and/or the nucleus should discard any further requests from the terminated threads
  - ◆ nucleus internal variables (eg. thread count) have to be adjusted

- ◆ **Exception and I/O Management**

- ◆ **How to handle I/O interrupts:**

- ◆ interrupt priority is defined by interrupt line and device number (higher number = lower priority)
- ◆ a terminal receiver has lower priority than its corresponding terminal transmitter
- ◆ 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 nucleus

- ◆ **Exception and I/O Management**

- ◆ **How to handle I/O interrupts:**

- ◆ each interrupt must be translated into a message for the SSI, with:
  - ◆ the base address of the device as sender
  - ◆ the **STATUS** register value as payload
- ◆ this payload value should be provided to the thread which will request the WAITFORIO service for that device
- ◆ you will have to decide how to:
  - ◆ store the relevant data in the SSI and the nucleus
  - ◆ define the associated scheduler policy (eg. prioritize I/O bound processes or not?)

- ◆ **Exception and I/O Management**

- ◆ **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
- ◆ pseudo-clock accounting should be quite precise: at most one pseudo-clock message should be pending in the SSI message queue to be processed

- ◆ **Exception and I/O Management**

- ◆ **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 thread)
  - ◆ 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 (eg. 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`