

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

### **AMIKaya Phase 1 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

---

#### ◆ **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 1 is the first software layer of AMIKaya (below it there are only bare hardware and ROM microcode)
- ◆ Phase 1 requires the development of Thread Queue and Message Queue Managers

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

2

- ◆ **Why to develop Queue Managers?**
  - ◆ because threads, messages and their queues are the basic data structures managed by any microkernel
  - ◆ to provide the required ADTs (*Abstract Data Types*) to AMIKaya's phase2
  - ◆ to gain experience in software development and debugging by using the uMPS simulator

- ◆ **Queue Manager goals:**
  - ◆ Queue managers = a collection of *modules* that will be used in Phase 2
  - ◆ Should implement the following features:
    - ◆ allocation and de-allocation of single ThreadBLK and MessageBLK descriptors
    - ◆ maintenance of ThreadBLK queues
    - ◆ maintenance of ThreadBLK trees
    - ◆ maintenance of MessageBLK queues

## AMIKaya project specifications

---

### ◆ Queue Manager features:

- ◆ Allocation of ThreadBLK and MessageBLK descriptors:
  - ◆ No OS = no heap = no dynamic memory allocation
  - ◆ How to allocate these descriptors?
  - ◆ *static* allocation: define
    - ◆ an array of MAXTHREADS ThreadBLKs
    - ◆ an array of MAXMESSAGES MessageBLKs
  - ◆ and use a *free list* (that is, a list of free elements) for each: `tcbFree` and `msgFree`
  - ◆ MAXTHREADS and MAXMESSAGES will be provided with `p1test`
  - ◆ MAXMESSAGES could be redefined in Phase 2 to handle the number of messages generated by `p2test`

## AMIKaya project specifications

---

### ◆ Thread Control Block definition (ThreadBLK):

```
/* thread control block */
typedef struct tcb_t {
    /* thread queue fields */
    struct tcb_t *t_next, /* pointer to next entry */

    /* thread tree fields */
    *t_parent, /* pointer to parent */
    *t_sibling, /* pointer to next sibling */
    *t_child; /* pointer to 1st child */

    /* thread's message queue */
    struct msg_t *inbox;

    struct state_t proc_state; /* processor state */

    /* Other fields will be added during phase2 development */
} tcb_t;
```

## AMIKaya project specifications

### ◆ Allocation-related functions:

- `void initTcbs(void);`  
Initialize the tcbFree list.  
This method will be called only once during data structure initialization
- `void freeTcb(tcb_t *t);`  
Insert the element pointed to by t onto the tcbFree list
- `tcb_t * allocTcb(void);`  
Return NULL if the tcbFree list is empty, otherwise remove an element from the tcbFree list, provide initial values for all of the ThreadBLK's fields (i.e. NULL and/or 0) and then return a pointer to the removed element; ThreadBLKs get reused, so it is important that no previous value remains in a ThreadBLK when it gets reallocated

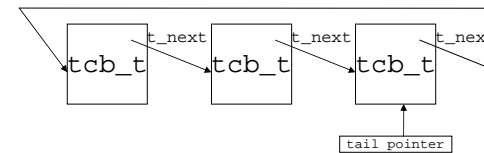
© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

7

## AMIKaya project specifications

### ◆ ThreadBLK Queue features:

- Circular, single-linked, tail pointed; using t\_next
- Efficiency may be improved introducing double-linked queues ( and t\_prev)
- Implementation (and documentation) of a more sophisticated data structure gives a bonus



© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

8

## AMIKaya project specifications

### ◆ Maintenance of ThreadBLK Queues:

```
tcb_t * mkEmptyThreadQ(void);  
Used to initialize a variable to be tail pointer to a thread queue;  
returns a pointer to the tail of an empty thread queue, i.e. NULL  
int emptyThreadQ(tcb_t *tp);  
Returns TRUE if the queue whose tail is pointed to by tp is empty,  
FALSE otherwise.  
void insertThread(tcb_t **tp, tcb_t *t_ptr);  
Insert the ThreadBLK pointed to by t_ptr into the thread queue whose  
tail-pointer is pointed to by tp; note the double indirection through  
tp to allow for the possible updating of the tail pointer as well  
tcb_t * removeThread(tcb_t **tp);  
Remove the first (i.e. head) element from the thread queue whose  
tail-pointer is pointed to by tp. Return NULL if the thread queue was  
initially empty; otherwise return the pointer to the removed element.  
Update the process queue's tail pointer if necessary  
tcb_t * outThread(tcb_t **tp, tcb_t *t_ptr);  
Remove the ThreadBLK pointed to by t_ptr from the queue whose tail-pointer  
is pointed to by tp. Update the queue's tail pointer if necessary.  
If the desired entry is not in the queue (an error condition), return NULL;  
otherwise, return t_ptr. Note: t_ptr can point to any element of the queue  
tcb_t * headThread(tcb_t *tp);  
Return a pointer to the first ThreadBLK from the queue whose tail is  
pointed to by tp. Do not remove the ThreadBLK from the queue.  
Return NULL if the queue is empty
```

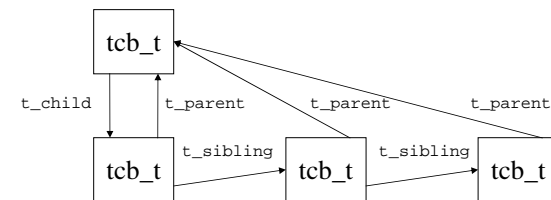
© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

9

## AMIKaya project specifications

### Maintenance of ThreadBLK trees:

- Implemented using `t_parent`, `t_child`, `t_sibling`
- Efficiency may be improved introducing double-linked queues at sibling level (and `t_prev_sibling`)
- Implementation (and documentation) of a more sophisticated data structure gives a bonus



© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

10

## AMIKaya project specifications

---

### ◆ Maintenance of ThreadBLK trees:

```
int emptyChild(tcb_t *t);
Return TRUE if the ThreadBLK pointed to by t has no children,
FALSE otherwise.

void insertChild(tcb_t *prnt, tcb_t *t);
Make the ThreadBLK pointed to by t a child of the ThreadBLK pointed to by
prnt. In other words: insert t in the thread tree as a child of prnt

tcb_t * removeChild(tcb_t *t);
Make the first child of the ThreadBLK pointed by t no longer a child of t.
Return NULL if there are no t's children, otherwise a pointer to the
removed ThreadBLK first child. In other words: if t has children, remove
the first one from the tree and return a pointer to it,
otherwise return NULL

tcb_t * outChild(tcb_t *t);
Make the ThreadBLK pointed to by t no longer the child of its parent. If the
ThreadBLK pointed to by t has no parent, return NULL, otherwise return t.
The element pointed by t need not be the first child of its parent.
In other words: look in the tree for the ThreadBLK pointed by t; if t has
no parent, return NULL; if it does have a parent, remove t from the tree
and return t
```

## AMIKaya project specifications

---

### ◆ Message definition (MessageBLK):

```
/* message block */
typedef struct msg_t{
    struct msg_t *m_next; /* pointer to next entry */

    struct tcb_t *m_sender; /* thread that sent this message */
    unsigned int message; /* the payload of the message */

} msg_t;
```

## AMIKaya project specifications

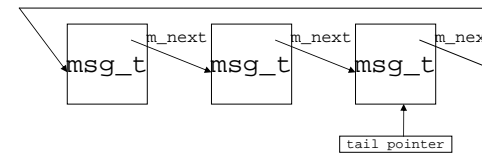
### ◆ Allocation-related functions:

- `void initMsgs(void);`  
Initialize the msgFree list.  
This method will be called only once during data structure initialization
- `void freeMsg(msg_t *m);`  
Insert the element pointed to by `m` onto the msgFree list
- `msg_t *allocMsg(void);`  
Return NULL if the msgFree list is empty, otherwise remove an element from the msgFree list, provide initial values for all of the MessageBLK's fields (i.e. NULL and/or 0) and then return a pointer to the removed element; MessageBLKs get reused, so it is important that no previous value remains in a MessageBLK when it gets reallocated

## AMIKaya project specifications

### ◆ MessageBLK Queue features:

- Circular, single-linked, tail pointed; using `m_next`
- Efficiency may be improved introducing double-linked queues ( and `m_prev`)
- Implementation (and documentation) of a more sophisticated data structure gives a bonus



## AMIKaya project specifications

### ◆ Maintenance of MessageBLK Queues:

```
msg_t * mkEmptyMessageQ(void);
    Used to initialize a variable to be tail pointer to a message queue;
    returns a pointer to the tail of an empty message queue, i.e. NULL
int emptyMessageQ(msg_t *mp);
    Returns TRUE if the queue whose tail is pointed to by mp is empty,
    FALSE otherwise
void insertMessage(msg_t **mp, msg_t *m_ptr);
    Insert the MessageBLK pointed to by m_ptr at the end of the queue whose
    tail-pointer is pointed to by mp; note the double indirection through
    mp to allow for the possible updating of the tail pointer as well
void pushMessage(msg_t **mp, msg_t *m_ptr);
    Insert the MessageBLK pointed to by m_ptr at the head of the queue whose
    tail-pointer is pointed to by mp; note the double indirection through
    mp to allow for the possible updating of the tail pointer as well
msg_t * popMessage(msg_t **mp, tcb_t *t_ptr);
    Remove the first element (starting by the head) from the message queue
    accessed via mp whose sender is t_ptr. If t_ptr is NULL, return the first
    message in the queue. Return NULL if the message queue was empty or if
    no message from t_ptr was found; otherwise return the pointer to the
    removed message. Update the message queue's tail pointer if necessary
msg_t * headMessage(msg_t *mp);
    Return a pointer to the first MessageBLK from the queue whose tail is
    pointed to by mp. Do not remove the MessageBLK from the queue.
    Return NULL if the queue is empty
```

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

15

## AMIKaya project specifications

### ◆ Some observations on Phase 1:

- ◆ Specifications may not look clear:
  - ◆ you have to think there is a bigger picture
  - ◆ focus on module structure and features
- ◆ Writing the code:
  - ◆ start writing stubs; build one step at a time and keep it simple
  - ◆ check for error conditions (plan for the unforeseen)
  - ◆ analyze and understand `pltest.c`
  - ◆ look at the examples provided with uMPS
- ◆ remember `#define HIDDEN static` for module data structures

© 2007 Enrico Cataldi, Mauro Morsiani, Renzo Davoli

16