Design Document

1. Introduction

Inter-process communication (IPC) is a very important concept in operating system because process is the interface between user and the system, and its communication determines the quality of the operating system.

The goal of designing this project is to deeply understand the concept of IPC and how to solve issues about IPC.

2. New system call

Because MINIX IPCs do not allow a user process (thread) to send or receive a message to another, user process oriented system calls must be created to accomplish inter-communications between user processes.

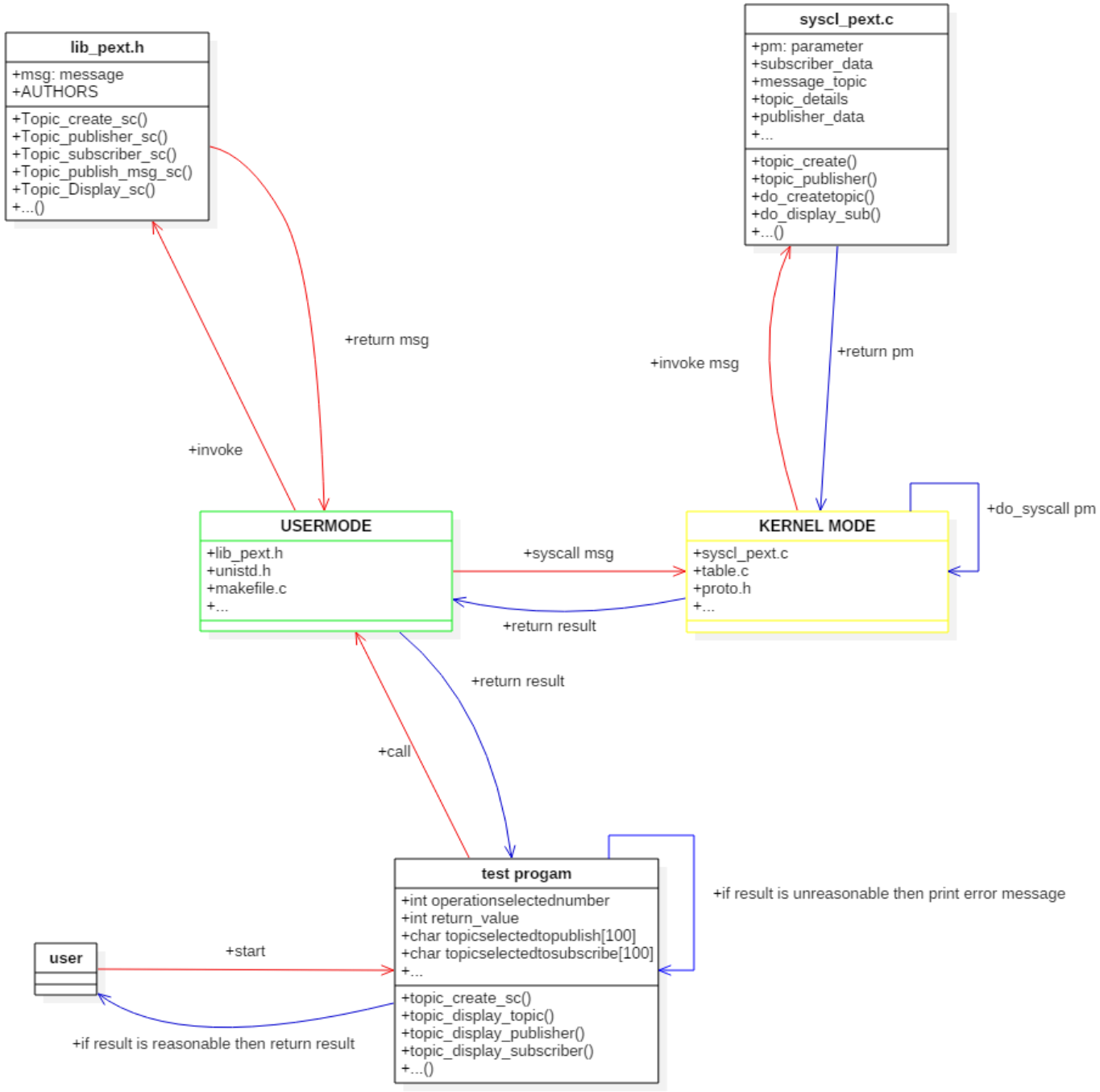
1.1. system call description

New system calls are created as required. To add these system calls, we add some new files: syscl_pext.c, syscl_errdef.h, And we modify the callnr.h, makefile, proto.c and table.c in /servers/pm and makefile.inc in /lib/libc/minix, the detail of each system call is shown in the following:

System call	API used in sycl_pext.c	Input	Output	Description	Exception
Topic_create	int createtopic()	Char *	int	If success return kRETSUCCESS;	Topic exists, return kTOPIC_DOES_NOT_EXIST
Topic_publisher	int Topic_publisher(char name_p[],pid_t current_pid)	Char*(topic name)	Int	Current process register as one publish. If success, return kRETSUCCESS	Topic doesn't exist, return kTOPIC_DOES_NOT_EXIST;
Topic_subscriber	int Topic_subsciber(char name_p[],pid_t current_process)	Char*(topic name)	int	Current process register as one subscriber. If success, return kRETSUCCESS	Topic doesn't exist, return kTOPIC_DOES_NOT_EXIST;
Topic_publish_MSG	int publish_message(char message[],pid_t	Char *(msg content)	Int	Current process publish one msg into	If buffer is full(contains 5 msg), return -1; If current process is not specific

	current_pid) int addNewMessage(int topic_id)			specific topic. If success, return kRETSUCCESS	topic's publisher, return kINVALID_PUBLISHER
Topic_receive_MSG	int get_message(char *cd, pid_t current_pid)	Void	Int	Current process retrieve one msg from interete d topic. If success, return kRETSUCCESS	If current process is not a subscriber, return -2; If no new msg received ,return -3; If no msg received, return -4; If function bugs occur, return -1;
Topic_init	void init()	Void	Int	Init all data,including msg, topic,publisher and subscriber list.	
Topic_Display	void Topic_lookup()	Voic	Int	Print topic list	
Topic_Displa	void	Void	Int	Print publish	

y_publisher	show_subscribe()			er list	
Topic_Display_subscriber	void show_publish()	Void	Int	Print subscriber list	



There are other APIs which are mainly about block and deadlock(parameter's description is in annotation):

// Unblock

```
void try_unblock(mqueue *block_queue, mqueue *unblock_queue, int src);    /* try unblock a process:
                                                                              src is unblocked
                                                                              pid*/

void do_unblock(endpoint_t proc_e, message *msg);            /* unblock a process */
int do_server_unblock(tdata  *g_ptr, int src);               /* unblock process  :g_ptr is
                                                              topic point, */
```

// Deadlock

```
int deadlock(tdata *g_ptr, int call_nr);                    /* valid deadlock :g_ptr is topic
                                                        point, call_nr is the type of
                                                        send/receive command */
void deadlock_addpend(mqueue *proc_q, mqueue *pend_q, int call_nr);  /*recursive detect
                                                                      deadlock */
```

1.2. How to make IPC invoke system call

Firstly, we need to define system call in callnr.h, and implement them(invoke which API) in lib_pext.h. At the same time, we implement API in syscl_pext.c. In order to let user mode invoke these system call, we add lib_pext.h.

In order to implement our system call into IPC, we modify main.c in /servers/PM and rewrite their related function.

3. Exception handling method

Exception handling method is in syscl_errdef.h. More details about Exception handling are described in section 1.1.

4. Blocking method

We use queue and one specific structure to handle blocking:

```
typedef struct{
tdata *topic; /* topic */
endpoint_t sender; /* sender */
endpoint_t receiver; /* receiver */
int call_nr; /* caller_nr: SEND/RECEIVE */
message *msg; /* message */
}top_message;
```

When msg is sent, it will enqueue first. So if another msg want to dequeue, the queue will check whether the buffer is full and whether there exists one mutual exclusion in the queue. Then the queue will decide which one should be dequeued first based on the current situation.

For example:

If P1->P2, P3, P4, then we have 3 top_messages And push them into msg_queue. msg_queue

store processes, and top_message will push into the sender process.

Msg_queue

P1: P1-send->P2, P1-send->P3, P1-send->P4

If P3 -> P4, we will do the same thing, then the structure is :

Msg_queue

P1: P1-send->P2, P1-send->P3, P1-send->P4

P3: P3-send->P4

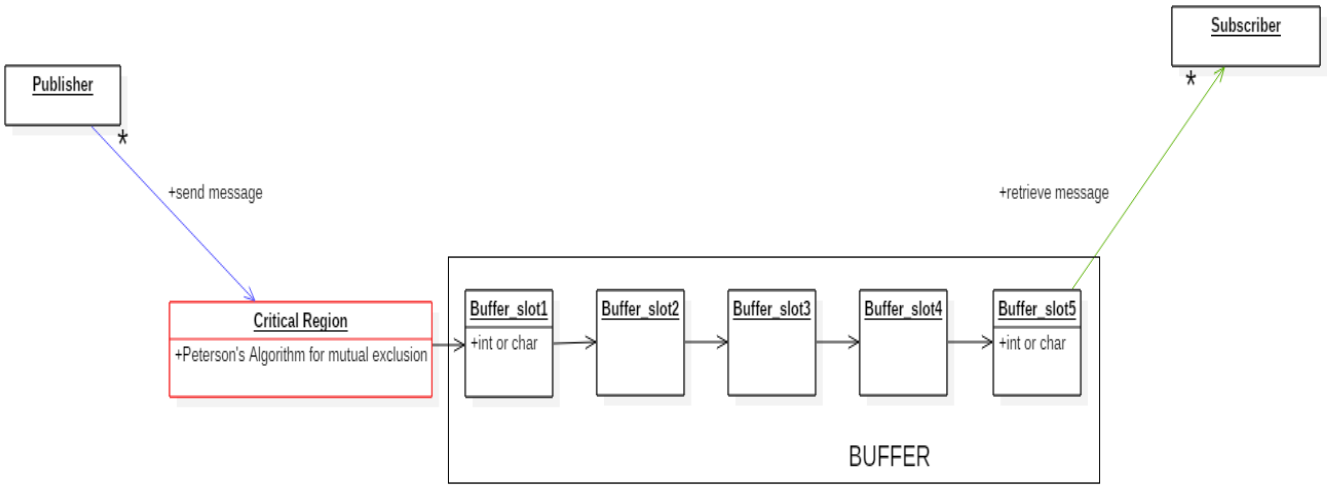
If P4<-receive-P1, we will find P1->send->p4 in P1, and remove it.

Msg_queue

P1: P1-send->P2, P1-send->P3

P3: P3-send->P4

Because we removed P1->send->p4, then unblock_queue->size would be 1, we will try_unblock P1 and P4, however, we find we cannot unblock P1, then we will do_unblock P4 only.



5. Deadlock detection and recovery

We use 2 queue to handle deadlock: valid_q and pend_q.

If p1->p2->p3->p4->p7, we call p4->p1 and p4->p5.

Then, group->pending_q will push p4->p1.

We dequeue from group->pending_q and push receivers p1, p5 into pend_q.

We check p5 and find out that p5 does now send message to other processes, so we push p5 into valid_q.

We check p1 and find p1->p2, then push p1 into valid_q and p2 into pend_q.

Then the structure is:

valid_q: p5, p1

pend_q: p2

we check pending_q iteratively:

valid_q: p5, p1, p2

pend_q: p3

Finally, we find pending_q is empty, and the structure is:

valid_q: p5, p1, p2, p3, p4, p7

pend_q: NULL

Now, we have the sender p4 from group->pending_q and find p4 is already in valid_q. In other words, we meet a deadlock.

Next, we set group->g_stat +=M_DEADLOCK and return ELOCKED.

To recover from deadlock, we set group->g_stat -=M_DEADLOCK and find the original states.

The algorithm is:

```
/*************************************************/
* 1. push receiver A into pend_q.
* 2. interactively check pend_q. e.g. Dequeue A from
     Pend_q and put A into valid_q.
* 3. if the value A send to another process, e.g. BCD,
     then push receivers into pend_q.
* 4. valid B, C, D each. put them into valid_q
* 5. iterative execute step 2,3,4. until the pend_q is
     empty.
* 6. check sender S, if s in valid_q, that means circle
     send/receive [deadlock].
* 7. else, not deadlock.
```

```
* 8. for multiple message, we put them in group->pending_q and valid each from step 1-7.
/**********************************************/
```