CS 551 - OPERATING SYSTEMS AND DESIGN IMPLEMENTATION

Project #2 INTERPROCESS COMMUNICATION

DESIGN DOCUMENT OF THE PROJECT2

SUBMITTED AT: ILLINOIS INSTITUTE OF TECHNOLOGY, CHICAGO

DATED: 10-25-2013

GROUP INFORMATION:

GAURAV BHARADWAJA: A20301453 (LIVE SECTION)

LOKESH VENKATESAN: A20310836 (LIVE SECTION)

SOWJANYA REDDY: A20288821 (INTERNET SECTION)

1. Primary purpose of the project:

The purpose of this project is to implement an (IPC) Inter Process Communication.

To implement this we make use of a set of library function calls (API) that utilize a set of system calls for sending/receiving messages through a common Data Structure maintained by the Process Manager in Kernel Space. The system calls implemented help us achieve a robust IPC that implements an interest group infrastructure where processes can communicate via a common medium. The idea is that any process may create an interest group and any other process can subscribe to the interest group to be able to receive messages from the group publisher.

2. System Calls:

To achieve the Inter Process Communication between different processes we make use of some system calls and provide API access to them via C library calls. For our implementation we chose to have each group be implemented by a structure with the following members: group name, publisher id, messages buffer (5 max), subscribers (256 max), total number of subscribers, and number of times messages are read. For simplicity we put a hard limit on the number of groups that may be created to 256. This constraint can certainly be relaxed and the groups can be implemented dynamically as well. The following system calls were implemented to realize the IPC:

- *create()*
- *subscribe()*
- *publish()*
- *retrieve()*
- *delete_()*
- *ListMembers()*
- *ListGroups()*

2.1 int *create(char* group_name, int pid):*

Function Name	Create

Function Description	The create() function is a C posix library call that is used to invoke the do_group_create() system call in kernel space via a call to Minix _syscall. This function has a dual purpose, in the event that this function is called for the first time it initializes the Groups Data Structures that will serve to hold the group information.	
Input Parameters	**Argument**	**Description**
	char* group_name int pid	The name of the group to be created. The pid of the process creating the group.
Returns Codes	Returns 0 {Interest Group was Created Successfully} Returns 1 {Err: The Interest Group Already Exists} Returns 2 {Group Limit of MAX_GROUPS 256 reached}	

2.2 int *subscribe(char* group_name, int pid):*

Function Name	Subscribe	
Function Description	The subscribe() function is a C posix library call that is used to invoke the do_group_subscribe() system call in kernel space via a call to Minix _syscall.	
Input Parameters	**Argument**	**Description**
	char* group_name int pid	The name of the group to join. The pid or the process joining the interest group.
Returns Codes	Returns 0 {Process Successfully Subscribed to the Group} Returns 1 {Err: The Interest Group Does not Exist} Returns 2 {Err: The process is already a member of the group} Returns 3 {Err: The process is already the group publisher/owner}	

2.3 int *publish(char* group_name, int pid, char* message):*

Function Name	Publish	
Function Description	The publish() function is a C posix library call that is used to invoke the do_group_publish() system call in kernel space via a call to Minix _syscall.	
Input Parameters	**Argument**	**Description**
	char* group_name int pid char* message	The name of the group the message shall be published to. The pid of the calling process wishing to publish to the interest group.

Returns Codes	Returns 0 {Message Successfully Published to the Group} Returns 1 {Err: The Interest Group Does not Exist} Returns 2 {Err: Calling Process is not the Group Publisher/Owner} Returns 3 {Err: The Message Buffer is Full (5 messages)}

2.4 char* *retrieve(char* group_name, int pid, int* ret):*

Function Name	Retrieve	
Function Description	The retrieve() function is a C posix library call that is used to invoke the do_group_retrieve() system call in kernel space via a call to Minix _syscall.	
Input Parameters	**Argument**	**Description**
	char* group_name int pid int* ret	The group name of the of the group we want to retrieve messages from. The pid of the subscriber wishing to retrieve messages. The third parameter ret is used to return the error code to the calling API function by reference.
Returns Codes	Returns 0 {Messages Successfully Retrieved} Returns 1 {Err: The Interest Group Does not Exist} Returns 2 {Err: Calling Process is not a Subscriber of the Group} Returns 3 {Err: There are no messages available for the Subscriber}	

2.5 int *delete_(char* group_name, int pid):*

Function Name	Delete	
Function Description	The delete_() function is a C posix library call that is used to invoke the do_group_delete() system call in kernel space via a call to Minix _syscall.	
Input Parameters	**Argument**	**Description**
	char* group_name int pid.	The name of the group to be deleted. The pid of the process wanting to delete the interest group.
Returns Codes	Returns 0 {Group Successfully Deleted} Returns 1 {Err: The Interest Group Does not Exist} Returns 2 {Err: Not the group Owner trying to Delete}	

2.6 char* ListMembers*(char* group_name, int pid, int*ret):*

Function Name	ListMembers	
Function Description	The ListMembers() function is a C posix library call that is used to invoke the do_list_messages() system call in kernel space via a call to Minix _syscall. The purpose of the function is for admin purposes to determine what subscribers are currently registered with the interest group.	
	Argument	**Description**

Input Parameters	char* group_name int pid	The name of the group to List the Members of. The pid of the group owner process.
Returns Codes	Returns 0 {Group Member PID's Successfully Listed} Returns 1 {Err: Interest Group Not Found} Returns 2 {Err: Not The Group Owner} Returns 3 {Err: No Group Subscribers Found}	

2.7 char* ListGroups*(int*ret):*

Function Name	ListGroups	
Function Description	The ListGroups() function is a C posix library call that is used to invoke the do_list_groups() system call in kernel space via a call to Minix _syscall. This function is meant to be informative for Subscribers wishing to know what Interest Groups are Available prior to joining them.	
Input Parameters	**Argument**	**Description**
	int* ret	It is used to return the error code of the function by reference.
Returns Codes	Returns 0 {Group Names Successfully Listed} Returns 1 {Err: No Interest Groups Found}	

3. REGISTERING THE SYSTEM CALLS:

Step 1:

We created our own C header file and Implementation files called: cs551_ipc.h and cs551_ipc.h. These are the files that define and implement our do_* system calls respectfully referenced above. Following the implementation of our solution we added: the C file cs551_ipc.c and header file cs551_iopc.h to the PM folder located at: /usr/src/servers/pm.

Step 2:

We populated the file table.c in the PM folder with an entry for each system call we implemented. This table is called call_vec and it is indexed by Minix Syscall in order to know which system call to execute.

Even though our implementation was only in the PM directory we also have to update the table.c file located in: /usr/src/servers/vfs folder. This tells the file system that a system call with that index already exists in the PM folder and that it is reserved.

Step 3:

Entry of the system handler file should be made in the /usr/src/servers/pm/Makefile.

Step 4:

Compile your pm server:

#cd /usr/src/servers/pm

#make

#cd /usr/src/servers

#make install

Step 5:

Now create posix library function calls and corresponding ASM library calls:

/usr/src/lib/libc/posix/*	/usr/src/lib/libc/syscall/*
_create.c	*create.S*
_subscribe.c	*subscribe.S*
_publish.c	*publish.S*
_retrieve.c	*retrieve.S*
delete.c	*delete_.S*
_ListMembers.c	*ListMemebers.S*
_ListGroups.c	*ListGroups.S*

Step 6:

Add entries of the C lib files above to /usr/src/lib/libc/posix/Makefile.inc. Add entries of the ASM files to /usr/src/lib/libc/syscall! Then Build the Libraries:

#cd/usr/src/lib:

#make depend

#make

#make install

#cd /usr/src/

#make includes

#make libraries

Step 7:

The next step is to make the new boot image using updated servers and library and for that go to /usr/src/tools and issue command "make hdboot".Now we have the system calls ready to use. Be sure to boot he image file generated by doing a shutdown:

C0d0>image=/boot/image/3.1.8rX

Cod0>boot

4. Exception Handling:

4.1 Group Not Found Exception:

If a process issues a system call to the Process Manager in order to perform any kind of action on a group that doesn't exist an exception is raised. For example if a process tries to subscribe, retrieve, publish, delete or list members of a group that doesn't exist a return code of 1 is asserted alerting the process that the group does not exist.

4.2 Not Group Owner Exception:

Typically a process will try to execute commands that only a group owner or administrator can issue. Thus we implemented an exception for the event that a process other that the group owner tries to execute commands that require group owner privileges. The group owner is identified by their unique pid and thus no username and passwords are required. In the event that a process tries to delete, publish, list members of a group for which they are not the owner this exception is raised and control is returned to the calling process unsuccessfully.

4.3 Not Subscriber Exception:

Typically processes that are not already subscribers of an interest group will try to perform system calls that require subscriber status. This exception is reserved for the retrieve system call. In the event that a non-group subscriber calls retrieve this exception is raised alerting the process that they are not a group member!

4.4 Already a Subscriber Exception:

If a process ties to subscribe to a group of which they are already a member of this exception is raised alerting the user of their active status within a particular group. A process may join as many groups as they want.

4.5 Publisher Role Exception:

When a Publisher/Creator tries to subscribe to the interest group they have created this exception is raised. This exception is raised to highlight that it is redundant for the Group Publisher to also be a subscriber of the group because they already know all the published messages.

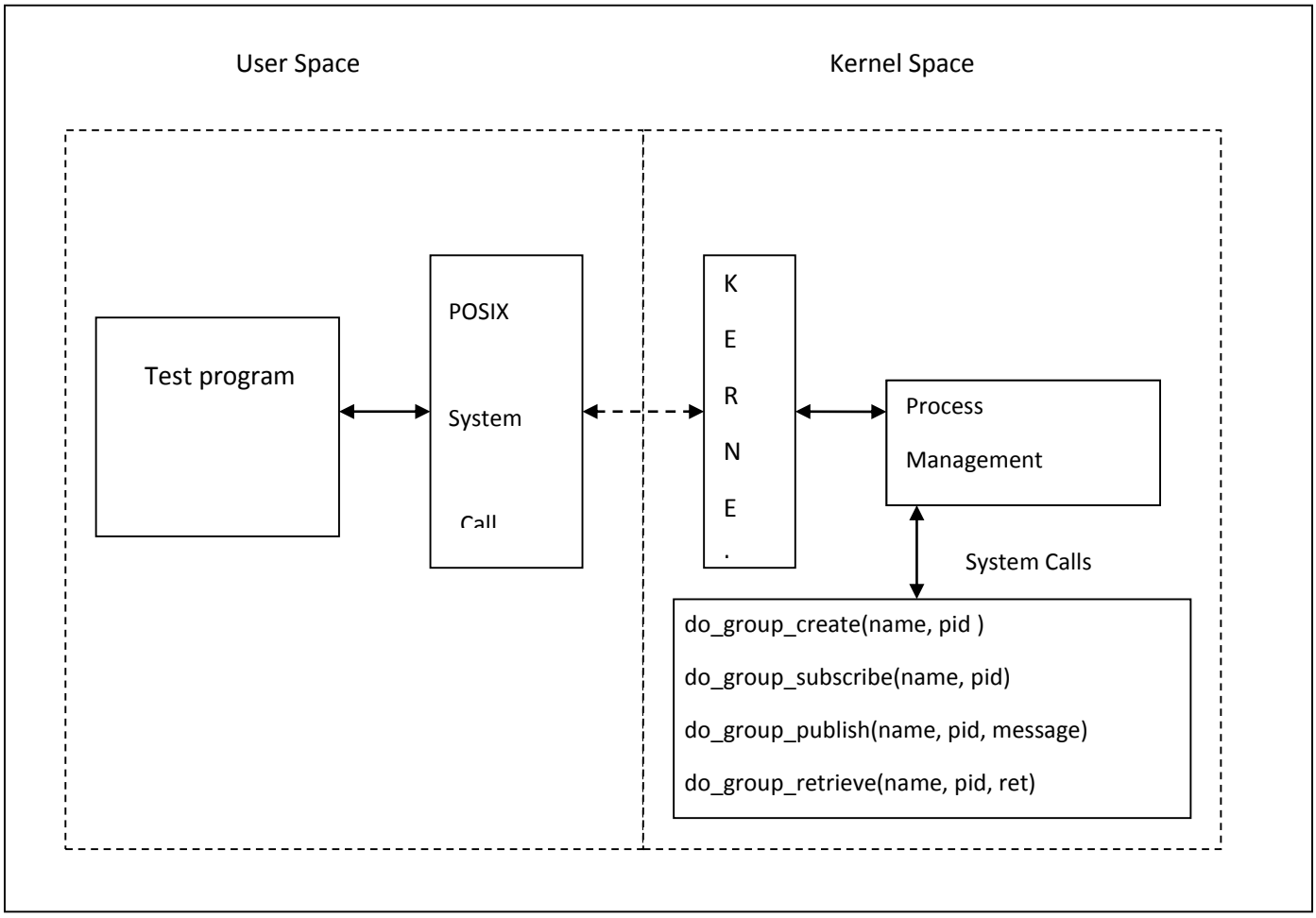
4.6 Retrieve a message 2x and Buffer Empty exception:

After a process retrieves their messages as far as that process is concerned the Buffer is empty thus if subsequent retrieves without intermediate publish are met with a "No Messages Available Exception". In the event that a Subscriber tries to retrieve messages from an empty buffer they are also met with a "No Messages Available Exception".

4.7 Buffer Full Exception:

If the group publisher is trying to publish more than 5 messages before the group subscribers have had a chance to retrieve their messages then this exception is raised. It alerts the publisher process that the Buffer is full and they may not write until some subscribers have read data off the IPC.

5 Diagram:



6 Conclusion and Challenges:

The biggest challenge we faced when implementing this project was having to copy data from user space to kernel space and vise-versa. This is a major challenge because User Space and Kernel Space are addressed differently and further more are implemented differently. We found that when passing strings via a pointer they were garbled on the kernel side. We were able to solve this challenge by utilizing the "sys_datacopy" MiNix system call. This function is a subset of "sys_vircopy" and it performs a copy of virtual bytes from the user space to the physical mapping on the kernel space and vise-versa. Only after utilizing the help of this important function were we able to successfully copy data streams to and from kernel space.

Because this project is implemented in kernel space its actually very similar to a shared memory implementation where processes communicate with the same memory segment protected by a shared secret or key. In our situation we never faced a possibility of deadlock because at any given time one and only one process is writing to the critical region of a group which is the messages buffer. Thus many processes

can read simultaneously but one and only one process will ever write to an interest group. This ensures that deadlock cannot occur. One possibility that can cause data ambiguity is if a process is reading at the same time a write is occurring. If the process is reading before the write is completed then there is a possibility that the reading process will miss one cycle of message but will receive them on the next iteration. In our opinion this is not an issue and the robust testing performed on this implementation supports it.