Instructions For Databar Exercise 10: Interprocess Communication

1 Introduction

During the work on this exercise, you will implement interprocess communication IPC using message passing.
Before starting work on the tasks, you should read section 2.3.8 in Tanenbaum and Bos book.
It is assumed that you are familiar with processes, threads and how to add system calls to the kernel. You do not need to have implemented memory management or thread synchronization.
These instructions are quite dense. There is a lot of important information in the text. Go back to the instructions and re-read them if you get stuck! You can reuse similar design choices as for the previous exercise on semaphores, using arrays to maintain ports. Fundamentally you will be faced with similar problems.

Read all of these instructions before starting to work!

2 Goals of the task

The task addresses the following learning objectives:

- You can explain in your own words and in text and with your own commented examples, how message passing can be used to coordinate processes.
- Given a skeleton operating system, you can implement a inter-process communication system that allows processes to communicate via messages.
- Assume a thread can be in three different states: running, blocked, ready. You can draw a diagram of your own showing the transitions between states and explain in text and in your own words what each transition means.
- Given reference literature, you can write down in text and in your own words definitions or explanations of the following concepts: Interprocess communication; Message passsing.

3 Report and rules

DTU has a zero tolerance policy on cheating and plagiarism. This also extends to the reports and indeed all your work. For example, to copy text passages without clearly and properly citing your source is considered plagiarism and so is copying source code. See the study hand book for further detail.

4 Setting up the environment

There is a new skeleton on DTU Inside. Download it into the VirtualBox image handed out at the start of the course.

Copy the video.c, kernel customization.c and kernel customization.h files you have developed into the new skeleton. Take care not to lose any work you have done.

Go over the code to spot the differences from the previous version of the skeleton.

5 Working on the exercise

In this exercise, you will implement three system calls for message passing, findport, send and receive. Messages are sent and received via ports. A port is the same thing as a mailbox. Ports are globally identified with a tuple of two values: (1) an identifier which is an integer and (2) the process that owns the port. Only the process that owns the port is allowed to receive from it! It is not defined what happens if two threads from the same process try to receive from the same port at the same time. However, it is allowed for multiple threads from multiple processes to send messages to the same mailbox at the same time. In this case, all messages must be delivered to the receiver!
Messages have a header portion and a payload portion. The header holds information such as the the message type. In the project, messages are sent via ports so the header holds information about the port to which the message is to be sent.
Messages are passed via the message data-structure defined in sysdefines.h. The payload part of the message is 32 bytes large since the message structure has eight member fields each of which are four bytes long. The message header is passed in registers.
All processes are to have one port when they are created. This port has an identifier of 0. You must update the createprocess system call and the kernel initialization code so that it reserves a port with identifier 0 and associated it with each created process. You must also design the data structures needed to manage the ports.
Ports are managed with one system call: findport. It takes two parameters: the identifier of a port in edi and the identity of a process in esi. It returns, in eax, the identity of a matching port if it exists or ERROR otherwise. The identities of processes are integers assumed to start at 0 so that the process created by the kernel at start is process 0 and each created process gets a successive number. Hint: you can set things up so that the process id is its index into process_table.
Messages are transferred between threads through a rendezvous scheme. This means that a sender will block until some other thread performs a receive on the same port. Vice versa, a thread that performs a receive primitive will block until some other thread sends a message to the same port.

You must add code to the terminate system call so that all ports owned by a process are freed. Any associated threads blocked waiting must be released when ports are freed. You must also make sure the eax register of the released threads is set to ERROR, so that the calling processes can detect that messages have not been transferred.

The send system call sends a message via a port. The sender will block until a receiver picks up the message. The identity of the port is passed in the edi register. This can be for example used as an index into an array in the kernel. A pointer to the message is passed in the esi register. You must typecast the contents of the esi register into a local variable of the appropriate type. Hint: look the message parameter of the send function in scwrapper.h.

Messages consist of 32 bytes and are passed using the fields in the message payload structure. The send system call identifier is defined by SYSCALL_SEND and it should return ALL_OK in eax if the message could be delivered or ERROR otherwise.

The receive system call receives a message from a port. The operation will block until a message is sent to the port. The index of the port is passed in the edi register and a pointer to the message is passed in the esi register. You must typecast the contents of the esi register into a local variable of the appropriate type. The receive system call identifier is defined by SYSCALL_RECEIVE and it should return ALL_OK in eax if a message could be received and ERROR otherwise. If successful, the system call returns the received data in a message payload structure. It also returns the sender's process id in the edi register. An overview of the use of registers in the send and receive system calls is given in Table 1.

To transfer messages you need to copy the message between message structures belonging to two different threads. One of the message structures is from the currently running thread, the other is from a blocked thread.

Table 1. Overview of register use in the send and receive system calls.

Task 5B			
Syscall\Register	**eax**	**edi**	**esi**
send	out: status code	port index	message pointer
receive	out: status code	in: port index out: process id of sender	message pointer

6 Test cases

The skeleton code has a built-in test case. This consists of three programs that exchange messages in a request-reply fashion. Process 0 acts as a server and responds with replies to requests from processes 1 and 2. In each step the integrity of message data is checked.

All the processes run in never ending loops and will continue sending messages to each other so you might want to insert “while (1);” statements to stop execution of a process when debugging.

7 Reflection questions

In the third report answer the following questions. The expected length of the answer is one and a half A4 page.

1. The message passing scheme implemented in this task is rendezvous. What might be the motivation behind this design? Are there any drawbacks?
2. Processes in an operating system should be isolated from each other to prevent one badly written or malicious process from destabilizing the whole system. Any process can communicate on a port as long as it knows its index. Briefly outline a scheme in which each process is given private port identifiers, i.e., identifiers or handles which can only be used by the process that obtained them.

Good Luck!