Practical 2: Inter-process communication

Due date: 5PM, Tuesday 7th October

Your task in this assignment is to implement a message passing facility in the AdelaideOS kernel, based on the following system calls:

- `int send(pid_t to, unsigned int tag, const void *data, size_t size);`

  Sends a message to the specified process. `tag` is an application-defined value indicating the type of the message. `data` and `size` specify the contents and size of the message, respectively. The maximum size is 1024 bytes. `send` returns 0 on success, or -1 on error (setting `errno` appropriately). It should never block.

- `int receive(message *msg, int block);`

  Receives a message sent to the current process. If a message is available immediately, its sender, tag, size, and data are stored in the message structure. Otherwise, the behaviour depends on the value of `block`. If true, the calling process is suspended until a message arrives. If false, the function returns immediately with a result of -1, and `errno` set to `EAGAIN`.

The `message` structure is defined as follows:

```
#define MAX_MESSAGE_SIZE 1024
typedef struct message {
  pid_t from;
  unsigned int tag;
  size_t size;
  char data[MAX_MESSAGE_SIZE];
} message;
```

For this assignment, you should modify version 8 of the kernel, available at:

https://cs.adelaide.edu.au/users/third/os/2008-s2/kernel/

Note: These files have recently been updated; if you downloaded them before 21 August you should get the latest copy. A tar.gz file is also available in this directory containing all of the source files.

In preparing for this assignment you will also need to familiarise yourself with the overall structure of the kernel. The best way to do this is to read through the documentation available at the URL below. You won't need to understand all of this material for the purposes of the assignment, though you are encouraged to read through this to better understand the concepts covered in the course.

https://cs.adelaide.edu.au/users/third/os/2008-s2/handouts/guide.pdf

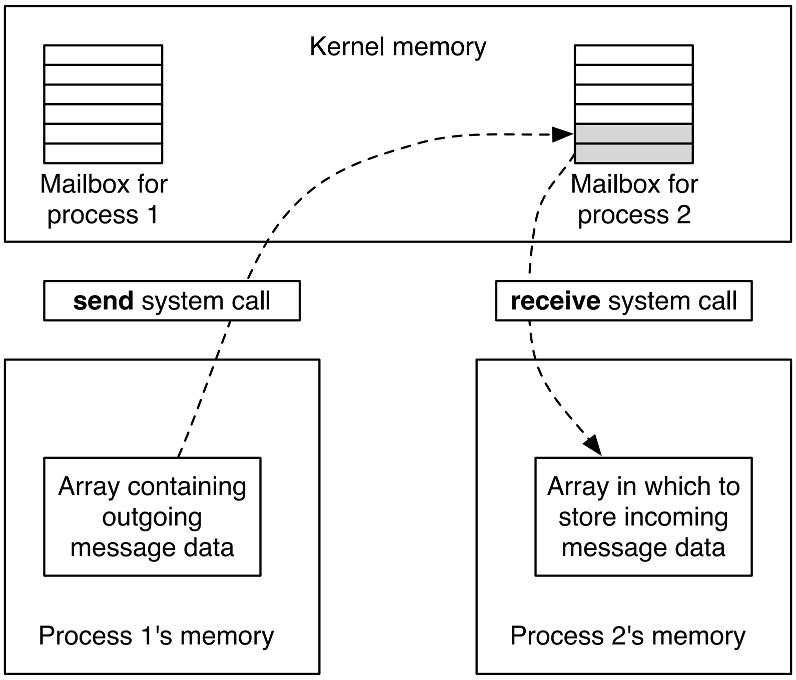

Figure 1: Message transfer

Stage 1: Message queue representation

You will first need to decide how to implement message transmission. The approach you should use is to have a message queue, or *mailbox*, associated with each process. This resides in kernel memory, which is separate from the private memory of each process. Because it is not possible for processes to directly access kernel memory, messages must be copied to and from the mailboxes via the `send` and `receive` system calls. Figure 1 depicts the way in which this should occur.

One or more fields will need to be added to the `process` struct, defined in `kernel.h`. These should point to a data structure holding the messages, each of which is represented by a `message` struct as shown above. You can use whatever data structure you like to store these, e.g. an array or linked list. Your implementation should place a reasonable limit on the maximum number of messages that can be stored in a queue at a time. If the `send` call encounters a full message queue on its destination, it should return with an error (see the section on error handling below for more details).

Stage 2: Send and non-blocking receive (30%)

In this stage, you must implement the `send` and `receive` functions that work in *non-blocking* mode. This means that the functions should return immediately, whether or not they were able to complete successfully. `send` simply needs to append the message to the queue (if there is sufficient space). `receive` needs to either remove the first message from the queue and return it to the process, or return an error code indicating that the queue is empty.

The first few tests supplied as part of the sample code only use non-blocking operation, so you can test this functionality before adding support for blocking.

Stage 3: Blocking receive (30%)

In this stage, when `receive` is invoked with `block` set to true, it should suspend the calling process if no messages are available yet. You will also need to modify the `send` system call so that when it adds a message to the queue, it detects the case where the destination process is blocked on a call to `receive`, and wakes it up if necessary.

Error handling (10%)

Your code must handle the following error conditions:

System call	**Condition**	`errno` **value**
`send`	Invalid message size specified (i.e. greater than `MAX_MESSAGE_SIZE` bytes)	`EINVAL`
`send`	Supplied pointer is outside the calling process's address space	`EFAULT`
`send`	Specified process does not exist	`ESRCH`
`receive`	No message available (non-blocking mode only)	`EAGAIN`
`receive`	Supplied pointer is outside the calling process's address space	`EFAULT`

Testing

The file `mptest.c` contains a number of routines that can be used to test the message passing functionality. When the kernel boots, you can run this from the shell by typing `mptest`. These are the same tests that automark runs when you submit your code, and your marks for the implementation part of this assignment will be based on how many of these produce the correct results.

Report (30%)

You must write a brief report (around 2-3 pages) describing how you have gone about implementing message passing. This should include a justification of the design choices with respect to mailbox representation and blocking. Your report should be well presented, and structured into at least 3 sections including an introduction and conclusion.

The writing should be aimed at a reader who understands the general concepts of IPC and message passing, but is not familiar with this particular assignment or the AdelaideOS kernel. It should therefore explain any implementation-specific details that are relevant to the discussion. The report should also mention at least one way in which the design of the message passing system could be improved.

Your report must be submitted in PDF format. If you are using LaTeX, you can generate PDFs using `pdflatex`, which is available on the Linux machines. OpenOffice under Linux and MS Word for the Mac are also capable of generating PDFs.

Submission

The code and your report should both be checked into your SVN repository, and submitted via the web submission system in the same manner as for assignment 1.

To create a directory in your SVN repository for the prac, type the following command, all on one line (replacing `aXXXXXXX` with your own username):

```
svn mkdir -m "OS prac 2"
  https://version-control.adelaide.edu.au/svn/aXXXXXXX/os-08-s2-prac2/
```

Then type:

```
svn co https://version-control.adelaide.edu.au/svn/aXXXXXXX/os-08-s2-prac2/
```

This will place a checkout of the (empty) `os-08-s2-prac2` directory into your current working directory. Next, copy all of the supplied files from version 8 of the kernel into this directory, and run the following commands:

```
cd os-08-s2-prac2
svn add *
svn commit -m "Initial import of supplied code for assignment 2"
```

Then work on your the code from this directory, and commit as appropriate. To submit the assignment, go to `https://cs.adelaide.edu.au/for/students/automark/` and follow the prompts to get to OS practical 2, then click on "Make A New Submission For This Assignment". Read through the declaration form, and click on "I Agree" to indicate that you understand the policies relating to plagiarism. Your assignment will then be processed by the automated marking system, with the results displayed on the page after some delay. The report will be marked separately.

Peter Kelly & Cruz Izu, August 2008