Project 3, Nucleus Message Passing, README
ZIHAO XING-SUID:255153189
ZHEN XIA-SUID:399588462

PURPOSE:
This is our work for Project3, “Nucleus Message Passing”, CIS657-Operating System

COMPILE:
(a) cd JNachos_Project1/src
(b) javac jnachos/*.java jnachos/*/*.java jnachos/*/*/*.java -d ../build/classes

notes: if directory “../build/classes” does not exist, then create one.

RUN:
java jnachos/Main -x ./test/sender,./test/receiver

IDE:
IntelliJ, Eclipse

CHECKLIST:
1, Implement four system calls as described in the papers, to be called by user program
-----------------------------------(DONE)

- int SendMsg(char* receiver, char* msg): An asynchronized system call. Sender process specify the name of the receiver process, and delivery it a message via a system buffer. This syscall returns the buffer ID;
- int WaitMsg(char* sender, char* msg): A synchronized system call. Receiver process specify the name of the sender process, and receive the message sended by the in-out argument, “char* msg”.
- int SendAnswer(char* answer, int BufferId): An asynchronized system call. This syscall is called by receiver program. The receiver replies the sender with an “answer” via the system buffer, which is assigned by “BufferId”. If answer is successfully delivered, return 1 to indicate success, else return 0.
- int WaitAnswer(char* answer, int BufferId): A synchronized system call. Sender waits the answer sent by the receiver or the system arrives at the buffer identified by “BufferId”. If answer sent by system, return 0, else return 1;

2, Implement the buffers correct in the Kernel. ----------------------------------(DONE)
We implemented a buffer pool in JNachos’s kern package, now with constant size of 1024, and with each process a buffer queue, which used to receive the buffer sent by other processes.

3, Processes should be able to communicate with any arbitrary process running in the system, but you need to work out some way of identifying them. --------------(DONE)

Currently, we figured out this problem by given each process the name followed by the source file's name without extensions, thus, each process know other's name at compilation time.

4, The same message buffer should be used for the same two processes to communicate. --------------(DONE).
The message buffer is specify by the buffer id and one message passing process will only use one message buffer.

5, If a process terminates (and you are required to show an example of this happening). This can be done by invoking the Exit system call mid-conversation. The system should send a dummy message out to the waiting process. --------------------(DONE)
We write a user program to test this case, details is in the "TEST" section below.

6, If a process dies while it has sent messages, then those messages should still be receivable by their co-communicators. --------------------(DONE)
See "TEST" section below.

7, A limit on the total number of messages sent by the process should be configurable and enforceable. -------------------- (DONE)
See "TEST" section below.

8, No other process should be able to interfere with communication (save ids). ------(DONE)
See "TEST" section below.

TEST:
We've done several tested user program, the source files are available at **"./test/src/"**. Here are the descriptions of each file and its purpose.

(1) sender.c & receiver.c (Demostrate the basic usages)
This two programs demonstrate the right usages of the system call. Sender program sends a message to receiver and waits for the answer. Receiver program wait for the message sent by sender and reply with answer. Each program will return '114' by default.

RUN:
-x ./test/sender,./receiver
RETURN:
sender returns with 114, receiver returns with 114;

(2) sender.c, sender1.c & receiver.c (Demonstrate Requirement #8)
These three programs demonstrate one of the edge cases of the system call. Sender program sends a message to receiver and waits for the answer. Receiver program wait for the message sent by *another* sender and reply with answer. In this case, the sender1 process returned with exit code "-1", since the answer will be a "dummy answer" sent by the system, and the receiver program and the sender program will return 114, which means success.

RUN:
-x ./test/sender1,./receiver
RETURN:
sender returns with 114, sender1 returns with -1, receiver returns with 114. (sender1 cannot interrupt the conversation between sender and receiver)

(3) sender2 & receiver1 (Demonstrate Requirement #7)
This two programs demonstrate requirement #7. This time, the sender sends 3 messages to receiver and wait for the answers. To demonstrate, we set the maximum messages a process can send to 2. The receiver replies the sender with the same amount of answers. Each program will return ‘114’ if success.

RUN:
-x ./test/sender2,./receive1
RETURN:
sender2 returns with -1, and receiver returns with -1(extra messages cannot be received).

(4) fork1.c (Demonstarte usages in fork case)
This program demonstrate the usage of the syscall under the “fork” case. Parent process send message to its child and the child replies its parent with answer.

RUN:
-x ./test/fork1
RETURN:
Child and parent both return with code 114.

(5) fork2.c (Demonstrate Requirement #6)
This program demonstrate one of the edge cases. Parent process sends message to its child and quit the conversation immediately before the execution of “WaitAnswer” syscall. Also as fork1, the child process waits for parent’s message and replies the parent with answer. In this case, for sender(parent), it will return with code “-1” by default; for receiver(child), the “SendAnswer” syscall cannot rightly return because of the absent of the sender.

RUN:
-x ./test/fork2
RETURN:
Child and parent both return with code -1.

(5) fork3.c (Demonstrate Requirement #5)
When the receiver quits conversation after receiving the message, then the sender will be reply with a dummy message sent by system.

RUN:
-x ./test/fork3

RETURN:
Child and parent both return with code -1.

LICENSE
BSD 3.0