ICSI412 4 – Messages

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. Especially ".class" files.

You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

Introduction

Processes are like little boxes that the program runs in – they are self-contained. Much of the time, that is perfectly fine. Our video game doesn't need to coordinate with or talk to our spreadsheet. But sometimes applications **do** need to communicate/coordinate. Let's look at a few ways that operating systems designers have solved this problem.

Signals are a 32-bit integer held in the PCB (KernelandProcess for us). A process can set a specified bit in another process. Programs can set up handlers for specified bits; when their bit is set, the handler is run. This is much like hardware-based interrupts. This is very simple, very space efficient, but it has limited usefulness.

Pipes are a unidirectional data transfer mechanism. They can be named or unnamed. Unnamed pipes are only useful within a process hierarchy. They are created by the BASH shell, for example:

```
ls | more
```

The shell is a process. It creates the pipe, then creates two new processes, one for more and one for ls. Because these processes fork (inherit) from the shell, they inherit all of the file descriptors, including the new pipe. Named pipes create a name in the file system. Unrelated processes can connect to the pipe by name.

Sockets are familiar from networking. They are bi-directional – once a connection is established, either side can send and receive data much like reading/writing to a device.

Shared memory uses the kernel and memory management to make the same area of physical memory available to two (or more) processes at once. The processes have to be careful about how they interact with the memory.

Messages are bundles of data that are transmitted between two processes, much like passing notes in school.

Ignoring signals because of their simplicity, there is a lot of overlap between these different forms of communication. Sockets, pipes, and messages all have processes sending data to other processes having the kernel intervene. Shared memory only has the kernel intervene for setup. The power of the API is the difference. Pipes require a pre-known name or a parent launching process. Sockets are between two and only two processes and have an **ugly** set of functions.

Messages are "cleaner" and can be general – you can send a message to any process from any process. This can be very powerful if it is ubiquitous in the operating system. Windows, for example, uses message passing extensively.

Details

Let's start with making the kernel message object. It should hold both the sender pid and the target pid. It should have an integer indicating "what" this message is; the sender uses it to indicate what the message is "for" and the receiver can switch on it to know how to handle the message. Finally, we will have a byte array of data – this can be whatever the applications want it to be. Create a copy constructor – a constructor that accepts a Kernel message and makes a copy of it. We do this because the sender will create the message, but we want to make a new copy for the recipient to get. Otherwise, the two processes are holding a reference to the same memory and that breaks the "wall" of processes. Finally, add a useful "ToString" – format doesn't matter so long as it is useful for debugging.

I added a few simple methods to OS, kernel and scheduler:

```
int GetPid() – returns the current process' pid
int GetPidByName(String) – returns the pid of a process with that name.
```

To implement GetPidByName, I added a name member to KernelandProcess. I get the name of the UserlandProcess using Java: ulp.getClass().getSimpleName(). While we are in that code, add a LinkedList of KernelMessage as a message queue.

Next, we will add our two new OS and kernel methods:

```
void SendMessage(KernelMessage km)
KernelMessage WaitForMessage()
```

SendMessage() should use the copy constructor to make a copy of the original message. It should populate the sender's pid. Two reasons for that – one is security (so the sender can't lie about who they are) and for performance – GetPid() is a kernel call that takes time. Here the kernel is doing two things at once. Next, it should find the target's KernelandProcess. You could loop through all of the queues, but I added another data structure – a HashMap with pid as the key and KernelandProcess as the value. This is a space-time tradeoff that I thought made sense. Of course, I had to add code to CreateProcess and in the process termination part of SwitchProcess() to populate and remove entries. If we find our target pid, add this message to the message queue and finally, if this KernelandProcess is waiting for a message (see below), restore it to its proper runnable queue (like we did with Sleep).

A process that is waiting for a message should not run until a message is sent to it. To implement WaitForMessage, first check to see if the current process has a message; if so take it off of the queue and return it. If not, we are going to de-schedule ourselves (similar to what we did for Sleep()) and add ourselves to a new data structure to hold processes that are waiting. I used a HashMap of pid→KernelandProcess; there are many other choices that one could use.

You should test this using a Ping-Pong example (you may have done something similar in ICSI333). Create two new userland processes, once called Ping and one called Pong. They find each other (using GetPidByName). One sends a message to the other and that one responds with a message which causes another message to be sent… They should print something so that you can see what is going on. Mine looked like this:

I am PING, pong = 3
I am PONG, ping = 2
  PONG: from: 2 to: 3 what: 0
  PING: from: 3 to: 2 what: 0
  PONG: from: 2 to: 3 what: 1
  PING: from: 3 to: 2 what: 1
  PONG: from: 2 to: 3 what: 2

I chose to increment “what” by one each time, just so I could make sure that the messages weren’t repeated, but that is not a requirement.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
KernelMessage	None (0)	Exists (3)	Exists, most methods and members (6)	Correct (10)
GetPid	None(0)	Exists in OS/Kernel/Scheduler as appropriate (3)		Exists in OS/Kernel/Scheduler as appropriate and returns correct value (5)
GetPidByName	None(0)	Exists in OS/Kernel/Scheduler as appropriate (3)	Exists in OS/Kernel/Scheduler as appropriate and returns correct value (5)	Exists in OS/Kernel/Scheduler as appropriate and returns correct value, populates name correctly in KLP(10)
SendMessage	None(0)	Two of: Copies kernel message, sets sending pid, adds to message queue (6)	Copies kernel message, sets sending pid, adds to message queue (10)	Copies kernel message, sets sending pid, adds to message queue, removes process from wait (20)
WaitForMessage	None (0)			Returns right away if message is waiting, puts process into wait state if not (20)
Ping/Pong		Two of: Both processes exists, use SendMessage and WaitForMessage, find each other by name, and print messages from each other. (6)	Three of: Both processes exists, use SendMessage and WaitForMessage, find each other by name, and print messages from each other. (9)	Both processes exists, use SendMessage and WaitForMessage, find each other by name, and print messages from each other. (15)
Testing	None (0)		Partially tested (6)	Test processes that show all functionality working - ping, pong and other processes are running as well (hello/goodbye), they exhibit the ping-pong effect – printing one message, then the other runs (10)