Communications

How do processes communicate?

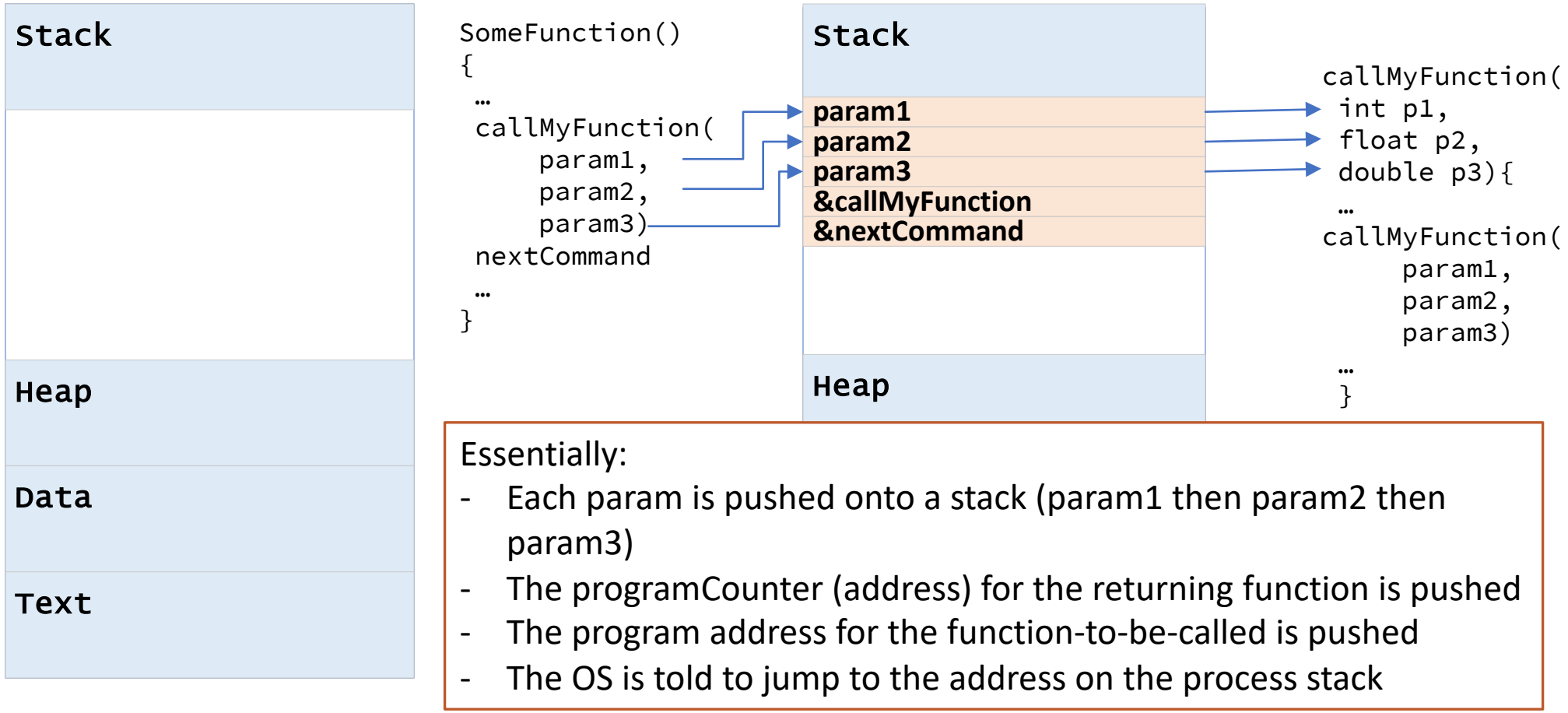
Within a process

- Shared data
- Internal function call
- Can be used in single threaded
- Can be used in multi-threaded (refer to OS slides)

Between processes

- IPC (Interprocess communication)

Basic function calls … WITHIN a process



Essentially:

- Each param is pushed onto a stack (param1 then param2 then param3)
- The programCounter (address) for the returning function is pushed
- The program address for the function-to-be-called is pushed
- The OS is told to jump to the address on the process stack

And so …

This is how communications works within a single application

- Given you have factored your code into multiple pieces (classes, functions, …)
  - You can ‘call’ each piece with a function call or something equivalent
- Application memory is used to keep track of data and addresses using Stack memory
- Question:
  - What happens when you get a stack overflow? (Yeah, it’s not just a place online to find code snippets)…
- Too many parameters, callback/ return addresses
  - Often from infinite loops, recursion

Multiple Processes

This does NOT work?

Why?

- Each process is in it’s own address space
- The Data Stack is NOT shared
- The process addresses are not known from Application A to Application B.

- There is a failure to communicate! (Apologies to Cool Hand Luke)
- And so … we need IPC

IPC examples

- File system
- Sockets
- Pipes
- Shared memory
- Message Queues

It is (really) an API i.e. a ‘contract’ of how to communicate with/ between software

But it is unstructured and untyped

- All manual documentation to define the interface
- Not compile time enforced
- Can have runtime error checks for format
- Example: Configuration files (think .ini files; think registry; think UNIX .rc or .cfg files)
- Example: Windows Message Queues (look it up)

File System

Example

- One process creates a file and deposits in a known location

- Second process picks up the file and reads the instructions

Common usage: Unix

Shared memory

A global shared memory location is established

Each process can read/ write to that location and communicate

Similar to global variables, except outside the process memory space

Message Queues

Like shared memory, but structured and with controls on send/ receive

Global memory assigned for incoming messages

Processes place a message in the queue

Other process(es) read the message and remove from the queue as they use it

Like a mailbox for software

Examples: Windows Message Queues

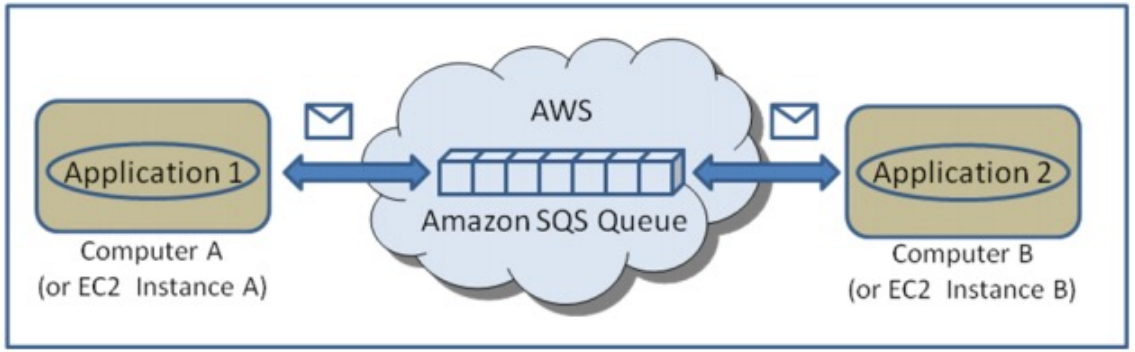


Figure 1: Two distributed applications communicating asynchronously by passing messages through an Amazon SQS queue.

http://sqs-public-images.s3.amazonaws.com/Building_Scalabale_EC2_applications_with_SQS2.pdf

Pipes

Type of message queue

Specifically between two processes (client/ server or peer-peer)

Acts like a FIFO message list

Push a message in, receiver pulls the message out

Named pipes:

Try this in Linux:

> mkfifo named_pipe
> echo "Hi" > named_pipe &
> cat named_pipe

Scan all open ports and list only occurrences of port 22:

```
sudo netstat -tnlp | grep :22
```

Look @ pipe
> ls –al named_pipe

Remove pipe
> rm named_pipe

Sockets

Like a pipe, but uses network socket vs. global memory

Local processes can also listen on a socket so this is still IPC

Example: Localhost

Specifics: When you run a webserver on your PC (for debug), the server is ‘listening’ on Port 8001 (for example) for HTTP messages to process

So now…

You have seen different ways to create an application, communicate between applications

How do you know which mechanism to use?

This will come down to requirements

Reminder: ASR = Architecturally significant requirements

Things that affect

- How you decompose

- How you communicate

- How you have an efficient and reliable program

And now – you know how an application runs – let's put it in practice

Activity:

Investigate application communications, using a 'producer-consumer' pattern. (So far we have seen the layered architecture pattern in OS. We'll see more of these later)

Producer-consumer:

Concept:

Basically, an application that creates some output is the producer. An application that consumes (uses) some item (input) is a consumer. (See activity document)

The two applications need *some* way to exchange information (Hint: Think IPC!)

- In this activity you will write up your approach, and we will discuss in class.
- Download the activity document from the website. You will submit your work to myCourses