mikro - Efficient Inter Process Communication

Julien Freche & Victor Apercé

julien.freche@lse.epita.fr
viaxxx@lse.epita.fr
http://lse.epita.fr/

Outline I

Introduction

Communication is the key

Most of the processes want to interact with each others.

- Several ways to do it
- Very important in a micro-kernel

Inter Process Communication

Choose your weapon

There is several ways to interact with other processes:

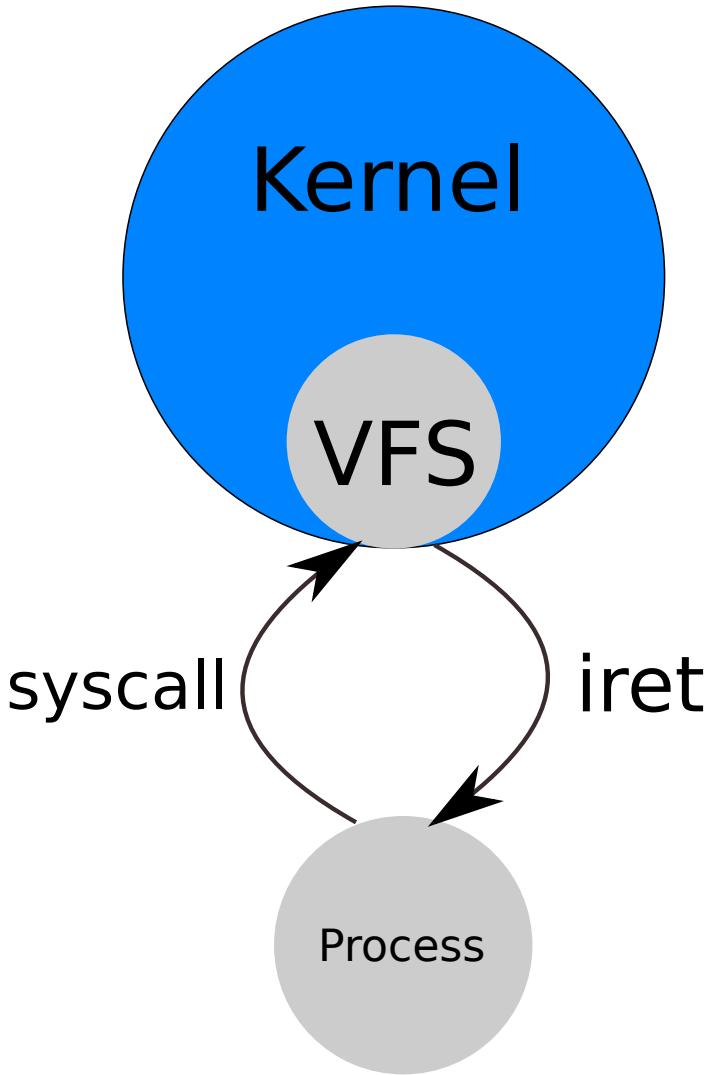
- File
- Signal
- Socket
- Message queue
- Pipe
- Shared memory
- ...

Communication is the key

IPC purposes:

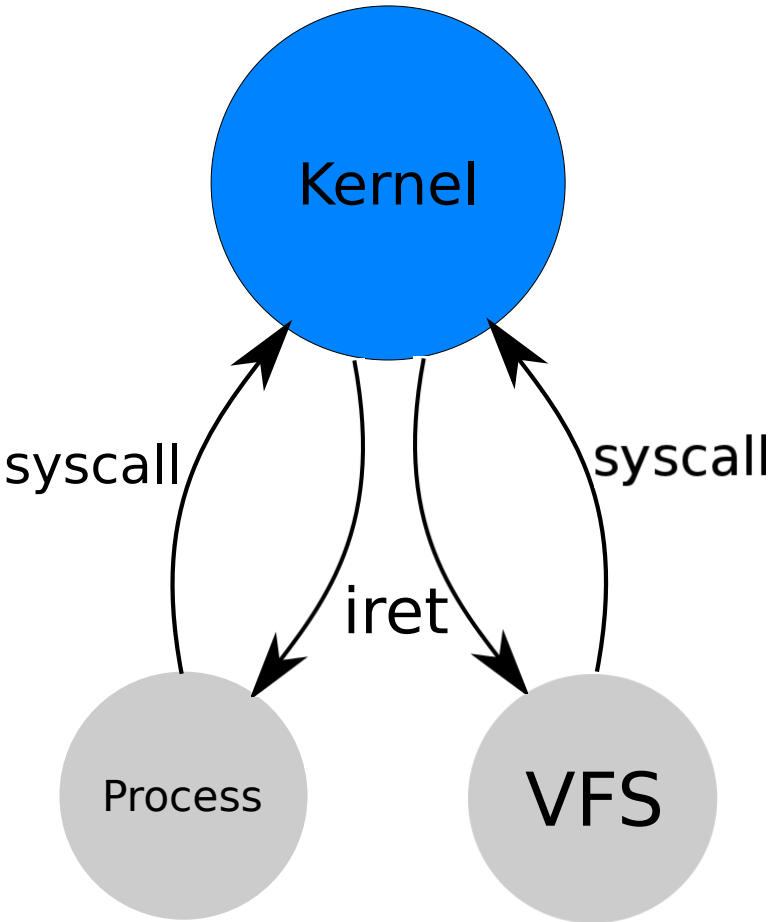
- Information sharing
- Modularity
- Convenience
- Privilege Separation

monolithic



1. The process performs a syscall
2. The kernel handles the request
3. Execution of the process can continue

micro kernel



1. The process performs a syscall: send to the service
2. The kernel handles the request and send the message to the service
3. The service handles the request and performs a syscall: reply
4. The kernel transfer the answer to the process
5. Execution of the process can continue

Implementation

Take my message, now !

There is two kind of message passing:

- Synchronous
- Asynchronous

Synchronous IPC are often considered faster:

- No copy from user to kernel required

Linux

Queue that message

There is two IPC implementations on Linux

- System V IPC
- POSIX IPC

Each implementation contains:

- Message queues
- Semaphores
- Shared Memory

We will discuss about messages queues.

System V Message queues

- Older than POSIX IPCs
- Implemented using queues
- Use RCU: Read-Copy-Update
- Similar to a pipe but you have to send messages, not bytes
- Each message has a type, so you can filter messages
- Very portable

POSIX Message queues

- Implemented as a special file system
- Each queue is a special file, you have to open it by its name
- Each message has a priority
- Message oriented

Mach

Many ports

- Port oriented
- Message transfer is asynchronous
- Similar to BSD sockets
- Rights are associated with ports
- Message oriented (header+body+trailer)
- Slow implementation ?

L4

Depends on the size

There is two types of synchronous IPC:

- Fast IPC
  - Very fast, use only CPU registers
  - During the syscall registers are preserved and execution is transfered to the receiver
  - Can transfer only a limited amount of data
- Long IPC: two types
  - Using shared memory
  - Copy data from an address space to another

seL4

Depends on the size

- Endpoint oriented
- endpoint = small kernel object with a list of threads
- n receiver(s), n sender(s)
- Threads do not send a message to another thread but to an endpoint
- Make synchronous transfer when possible

mikro - Implementation

Existing enhancements

Too slow

- IPC in micro kernel are too slow
- Reducing the IPC time is hard
- mikro current implementation is too simple
- It will be changed soon to the new design introduced in these slides

Spotting the issue

What is reducing IPC speed?

1. Context switch is slow:
   - by itself
   - because it invalidates TLB
2. Copying message is slow
3. Algorithms in send/receive system calls can be too slow
4. Having a lot of intermediates between the sender and the final receiver

Make context switch faster

- Context switch time is quiet impossible to reduce
- For x86 processor: sysenter/sysexit (Intel) or syscall/sysret (AMD) are faster than standard interrupt handling
- This is the only thing that can be done

No copy/paste

Copying message can be avoided by 2 means:

Shared pages

Sharing pages between the sender and receiver avoids any copy.

- Cannot be used for all IPC because it will starve memory
- Synchronisation between process is more complex
- Should be used only for large messages

Synchronous IPC

Synchronous IPC avoids to copy the message in kernel and then to its final destination.

Algorithms rule computers

- Our point of view is that's what can be improved
- That's quiet logical it's the only thing that is really under our control

mikro innovation

O(log(n))

- IPC in L4 family (excluding seL4) have a log(n) cost, where n is the number of processes
- IPC are sent to a particular process, so finding the task struct of this process is log(n)
- mikro will do this in O(1)!

O(log(n))

- IPC in L4 family (excluding seL4) have a log(n) cost, where n is the number of processes
- IPC are sent to a particular process, so finding the task struct of this process is log(n)
- mikro will do this in O(1)!

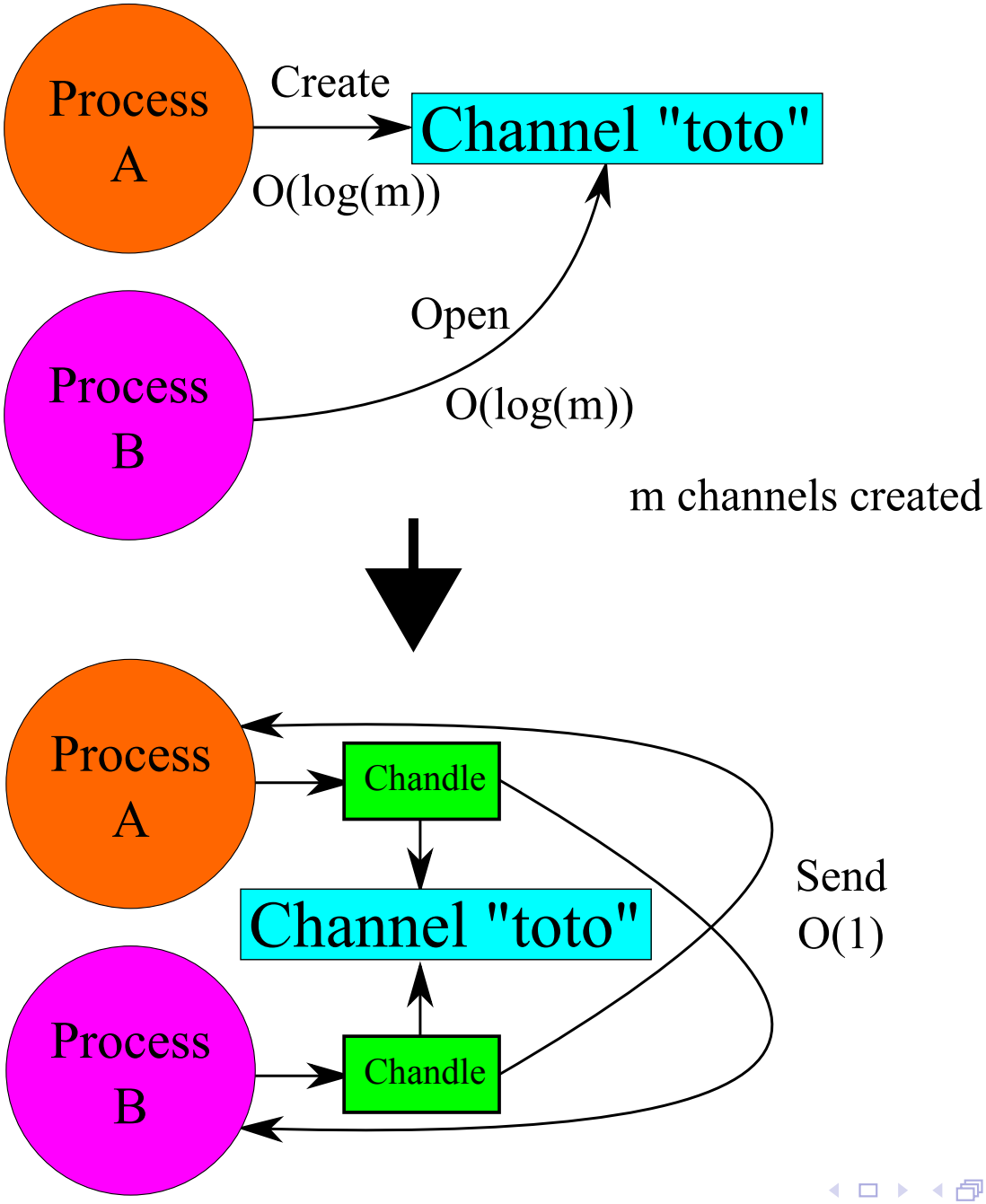
Channel Handle

- mikro IPC will have a connection support, named channel
- Every connection is represented by a channel handle, named chandle
- A chandle is just like a file descriptor, an integer
- There's a limited number of chandles by process
- Chandle associated structures are stored in a fixed-size array in every process task structure

Then why O(1)?

- Opening the connection costs O(log(n)) to find the receiver
- Chandle associated structure contains a pointer to this receiver
- Finding a chandle is looking in a fixed-size array
- Then send cost is O(1)

A little drawing



Ok I lied a bit

- This is perfectly working for 1 to 1 connections
- But when the connection is client-server oriented (1 to N), this does not work
- Because we cannot suppose the fixed number of possible connection is enough for the server
- But we have a solution too!

Server Solution

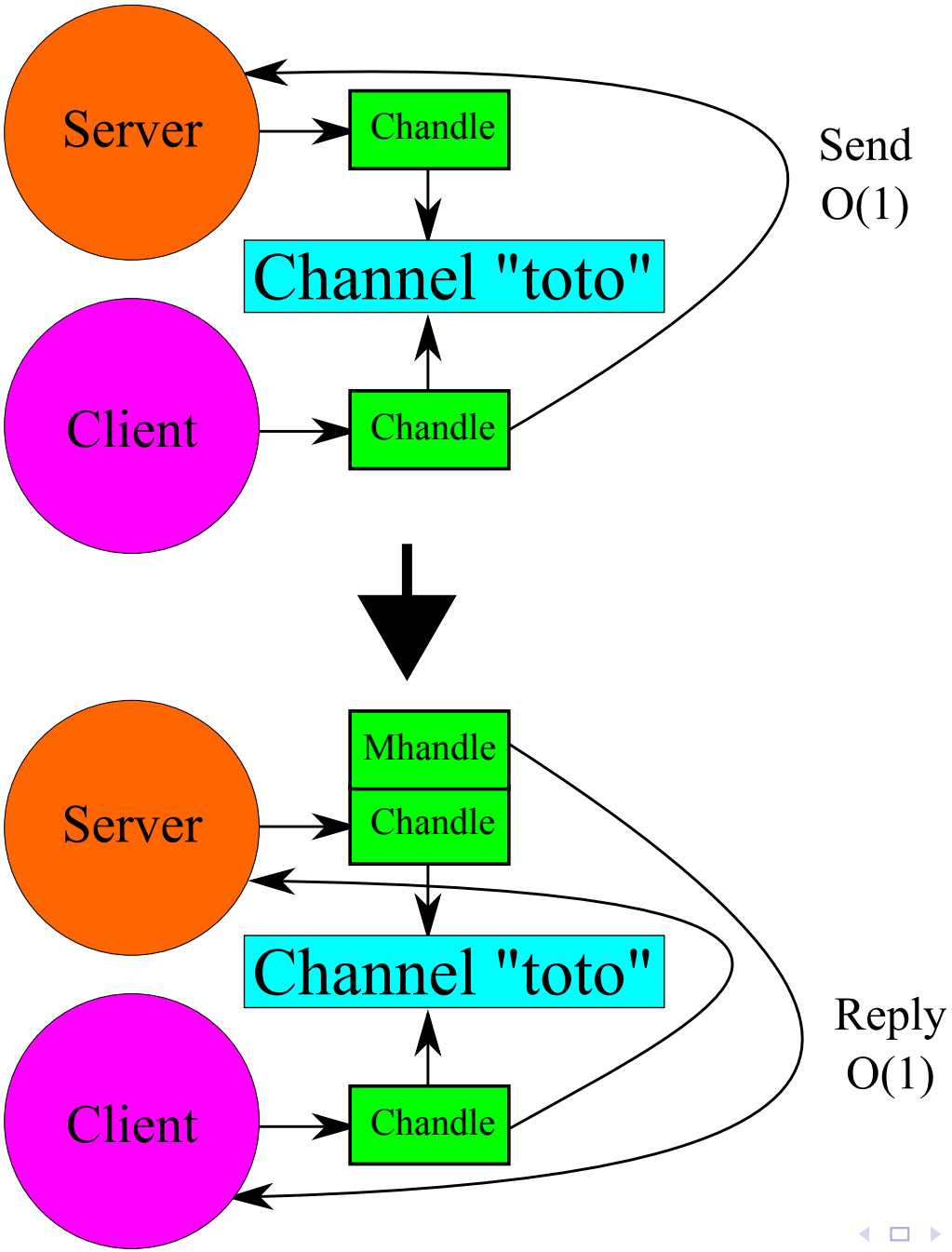
In 1 to N:

- Creating the channel is almost the same as before
- But opening the channel is different:
  - No other chandle is created for the server
  - Client gets its chandle as usual
  - Server keeps track of these connections in its chandle associated structure

Sending in O(1) to a server

- A client can then still send in O(1)
- When the server receives, a special chandle is created, named message handle or mhandle
- This mhandle is a temporary chandle that points to the sending client in order to reply
- The server must then reply to the client using this mhandle or close it

A little drawing



This isn’t perfect

- That way, client send in O(1) and server reply in O(1)
- But when a server want to send to a client directly:
  It costs O(log(n)) where n is the connection number to the server
- This isn’t perfect but we think that kind of behaviour is not that much used

Yet an other optimisation

- mikro will also keep a small data for each client connected to a server channel
- When a client sends a message to the server, this data is provided to the server
- That way server can store a pointer in that data to get faster its own data associated to this client
- It avoids the server to do an extra log(n) to find the data associated with the client PID

Conclusion

This looks good on paper

- As said before, we haven't yet tested all this
- We'll do benchmarks when it'll be implemented to check that model
- So let's see you in winter to check if this idea was the good one

Contacts

Julien Freche

- julien.freche@lse.epita.fr
- @JulienFreche

Victor Apercé

- viaxxx@lse.epita.fr

Feel free to contact us. We will be happy to answer.

Thank you for your attention