
shrehanrajsingh
Inter Process Communication and DBus

KOSS-Task

FOLDERS: KOSS-Task

- Overview
  - overview.md
- IPC
  - what_is_ipc.md
  - models.md
  - naming.md
  - synchronization.md
  - sync-example.c
  - DBus
    - what_is_dbus.md
    - messages.md
    - calling.md
    - signals.md
- BIBLIOGRAPHY.md

PREVIEW overview.md

Overview

Ever wondered how two different programs running on your computer share information between each other?

One solution is to use dedicated, temporary files to store information.
One program writes to the file, and the other reads from it likely by implementing a kind of a “file watcher”, something I did years ago when I was trying to make a UI library (in C) for a programming language (not C).

PREVIEW overview.md

Overview

What are the drawbacks for the mentioned approach?

- Writing to files is expensive (opening file stream, writing to it, closing file stream).
- Implementing a file watcher is expensive.
- Slow approach if multiple messages are needed to be sent continuously.

In the UI library I worked on, every state of the window (components clicked, size changes, window repaints) had to be transmitted to the programming language (we'll call it language X). This involved storing a lot of data in files. Over time, the temporary files grew in size.

What, then, is the solution to this dilemma?

FOLDERS: KOSS-Task
Overview
overview.md
IPC
what_is_ipc.md
models.md
naming.md
synchronization.md
sync-example.c
DBus
what_is_dbus.md
messages.md
calling.md
signals.md
BIBLIOGRAPHY.md

PREVIEW what_is_ipc.md

Inter Process Communication (IPC)

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.
A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing
- Computation speedup
- Modularity
- Convenience

FOLDERS: KOSS-Task
Overview
overview.md
IPC
what_is_ipc.md
models.md
naming.md
synchronization.md
sync-example.c
DBus
what_is_dbus.md
messages.md
calling.md
signals.md
BIBLIOGRAPHY.md

PREVIEW models.md

Different communication models used in IPC

There are two fundamental models for interprocess communication: (1) **shared memory** and (2) **message passing**.

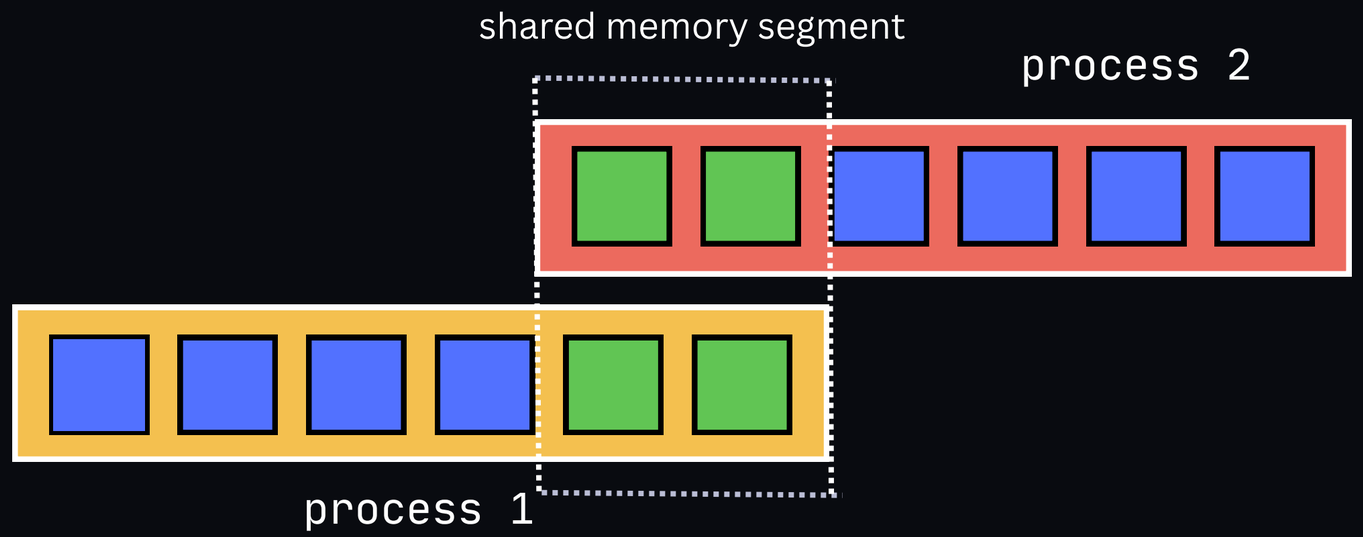
Shared memory

In the shared-memory model, a region of memory that is shared by the cooperating processes if established. Processes can then exchange information by reading and writing data to the shared region (similar to files but here the “file” is a shared memory region).
Shared memory allows maximum speed and convenience of communication. After shared memory is established, all accesses are treated as regular memory accesses (like accessing a variable).

PREVIEW models.md

Shared memory (continued)

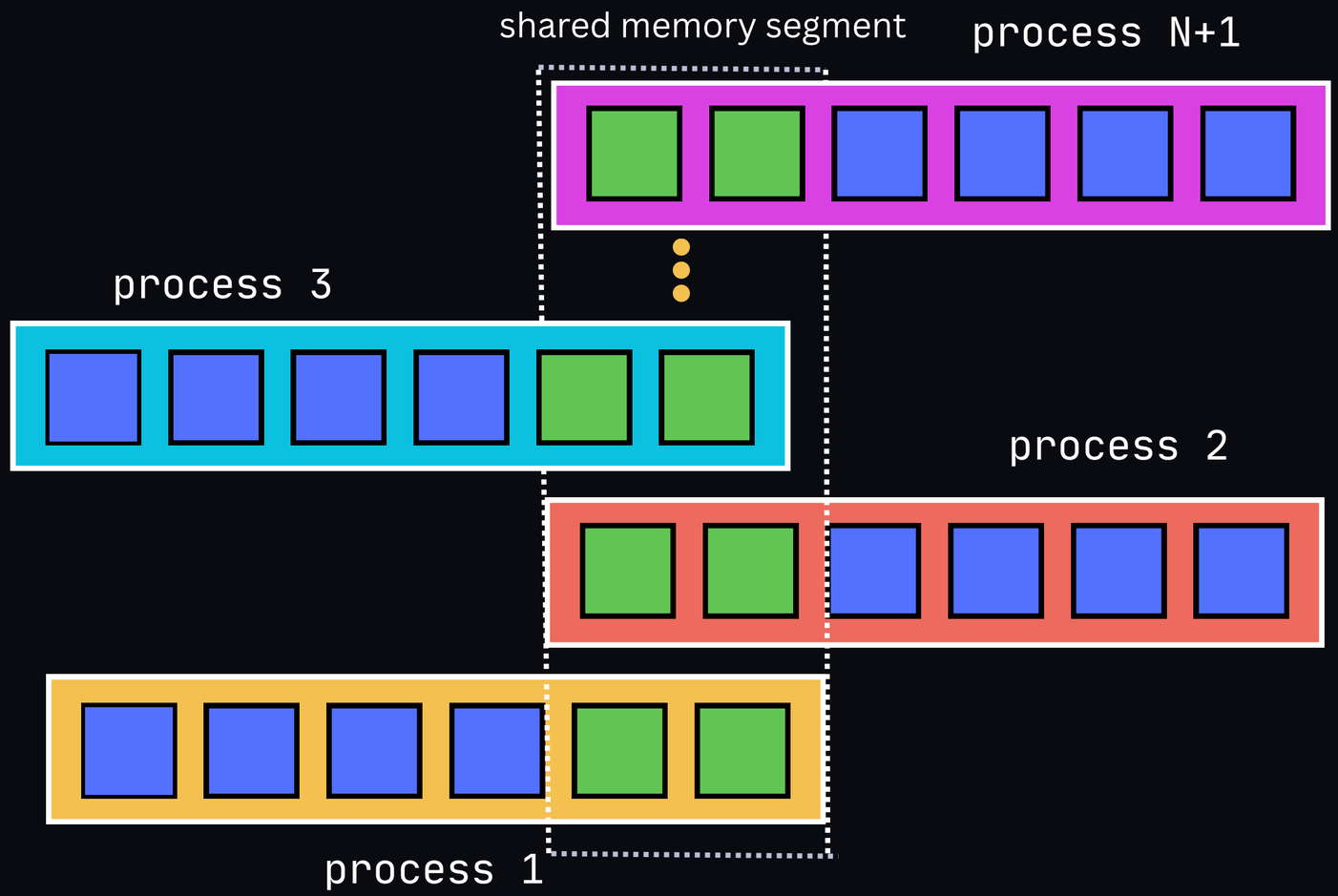
In the shared-memory model, a region of memory that is shared by the cooperating processes if established. Processes can then exchange information by reading and writing data to the shared region (similar to files but the here the “file” is a shared memory region).
Shared memory allows maximum speed and convenience of communication. After shared memory is established, all accesses are treated as routine memory accesses.



PREVIEW models.md

Shared memory (continued)

Multiple processes can share the same memory segment (provided they comply by the necessary rules since the operating system does not ideally allow processes to access addresses of other processes).



PREVIEW models.md

Message passing

In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.
It is useful for exchanging smaller amounts of data, because no conflicts need be avoided. Message passing is also easier to implement than is shared memory for inter-computer communication.

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on a different computers connected by a network.

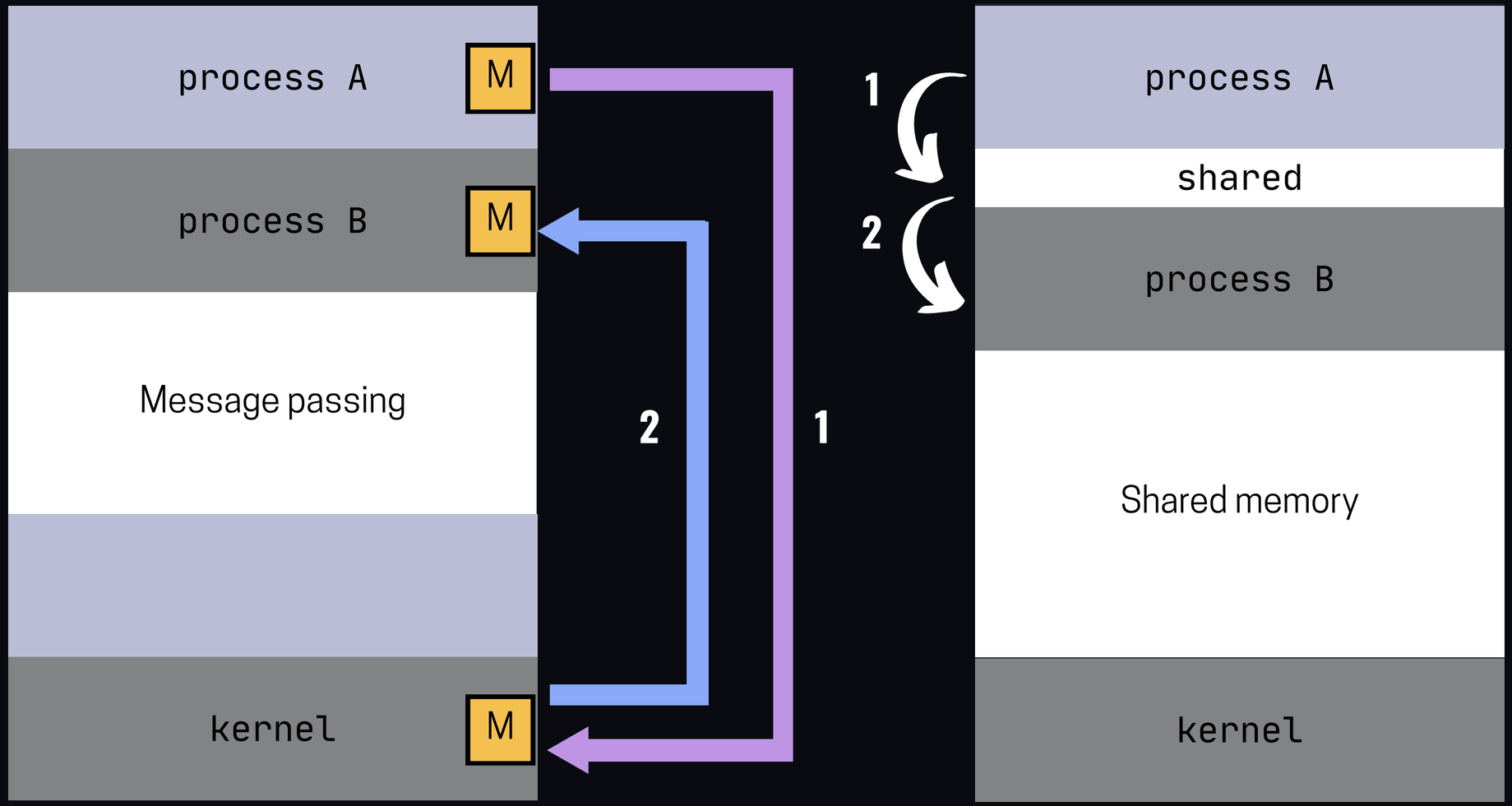
KOSS-Task

FOLDERS: KOSS-Task

- Overview
  - overview.md
- IPC
  - what_is_ipc.md
  - models.md
  - naming.md
  - synchronization.md
  - sync-example.c
  - DBus
    - what_is_dbus.md
    - messages.md
    - calling.md
    - signals.md
- BIBLIOGRAPHY.md

PREVIEW models.md

Communication models



FOLDERS: KOSS-Task
Overview
overview.md
IPC
what_is_ipc.md
models.md
naming.md
synchronization.md
sync-example.c
DBus
what_is_dbus.md
messages.md
calling.md
signals.md
BIBLIOGRAPHY.md

Communication models

Message passing

- Processes communicate through a message passing facility provided for by the operating system.
- Useful for exchanging smaller amounts of data.
- Slower than shared memory, because messages passed require the use of system calls.
- Example: DBus, Mach operating system, LPC (Windows XP)

Shared memory

- Processes agree on a common ground to store information (memory space). This is not under the control of the OS.
- Faster than message passing, since transmission and receiving operations are mere memory accesses.
- Faster than message passing, as system calls are only required to establish shared-memory regions.
- Example: POSIX

PREVIEW naming.md

Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication.

- send (P, message) – Send a message to process P.
- receive (Q)* – Receive a message from process Q.

With indirect communication, the messages are sent to and received from mailboxes, or ports. A mailbox is an object into which messages can be placed by processes and from which messages can be removed. POSIX message queues use an integer value to identify a mailbox.

- send (A, message) – Send a message to mailbox A.
- receive (A)* – Receive a message from mailbox A.

* Certain texts define receive(Q) as receive(Q, message), where message is a buffer to store the incoming message.

PREVIEW synchronization.md

Synchronization

Communication between processes takes place through calls to send and receive() primitives. There are different design options for implementing each primitive. Message passing may either be blocking or nonblocking – also known as synchronous and asynchronous.

- Blocking send – The sending process is blocked until the message is received by the receiving process or by the mailbox.
- Nonblocking send – The sending process sends the message and resumes operation.
- Blocking receive – The receiver blocks until a message is available.
- Nonblocking receive – The receiver retrieves either a valid message or a null.

When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver.

KOSS-Task

FOLDERS: KOSS-Task

- Overview
  - overview.md
- IPC
  - what_is_ipc.md
  - models.md
  - naming.md
  - synchronization.md
  - sync-example.c
  - DBus
    - what_is_dbus.md
    - messages.md
    - calling.md
    - signals.md
- BIBLIOGRAPHY.md

sync-example.c

```c
#define CAPACITY 10
typedef char *str_t;

str_t queue[CAPACITY];
int l = 0, r = 0;

/* blocking send */
void
send_b (str_t msg)
{
 /* empty/partially empty queue */
 if (l == r && ((r + 1) % CAPACITY != l))
 {
     queue[r] = msg;
     r = (r + 1) % CAPACITY;
     return;
 }

 /* block until message(s) is/are received */
 while ((r + 1) % CAPACITY == l)
     ;
}

/* blocking receive */
void
receive_b (str_t *buf)
{
 /* block until message is available */
 while (l == r)
     ;

 *buf = queue[l];
 l = (l + 1) % CAPACITY;
}
```

sync-example.c

```c
/* non-blocking send */
int
send_nb (str_t msg)
{
 /* queue is full */
 if ((r + 1) % CAPACITY == l)
     return 0; /* NULL */

 queue[r] = msg;
 r = (r + 1) % CAPACITY;
 return 1;
}

/* non-blocking receive */
int
receive_nb (str_t *buf)
{
 /* empty queue */
 if (l == r)
     return 0; /* NULL */

 *buf = queue[l];
 l = (l + 1) % CAPACITY;
 return 1;
}
```

FOLDERS: KOSS-Task
Overview
overview.md
IPC
what_is_ipc.md
models.md
naming.md
synchronization.md
sync-example.c
DBus
what_is_dbus.md
messages.md
calling.md
signals.md
BIBLIOGRAPHY.md

PREVIEW what_is_dbus.md

Desktop Bus (D-Bus)

D-Bus (Desktop Bus) is an inter-process communication (IPC) system that allows different processes running on the same machine to communicate with each other. It is commonly used in Linux and Unix-like operating systems to enable communication between system components, applications, and services.

- D-Bus follows a message-based architecture where clients and services communicate through a bus daemon.
- D-Bus implements two buses – (1) **System bus** and (2) **Session bus**.
    - System bus – Used for system-wide communication (like notifying when a device is connected).
    - Session bus – Used for user session communication (like between GUI applications).

KOSS-Task

FOLDERS: KOSS-Task

- Overview
  - overview.md
- IPC
  - what_is_ipc.md
  - models.md
  - naming.md
  - synchronization.md
  - sync-example.c
  - DBus
    - what_is_dbus.md
    - messages.md
    - calling.md
    - signals.md
- BIBLIOGRAPHY.md

PREVIEW what_is_dbus.md

Desktop Bus (D-Bus)

A typical message request looks like this:

```
dbus_message_new_method_call(
    "org.freedesktop.login1",        // Service name
    "/org/freedesktop/login1",       // Object path
    "org.freedesktop.DBus.Properties", // Interface
    "Get"                            // Method name
);
```

Signature

```
DBusMessage* dbus_message_new_method_call(
    const char *bus_name,
    const char *path,
    const char *interface,
    const char *method
);
```

Service name

- bus name, either unique connection name (like :34-107) or well known name (like org.freedesktop.login1).
- bus names are never reused for the lifecycle of the program.

Object path

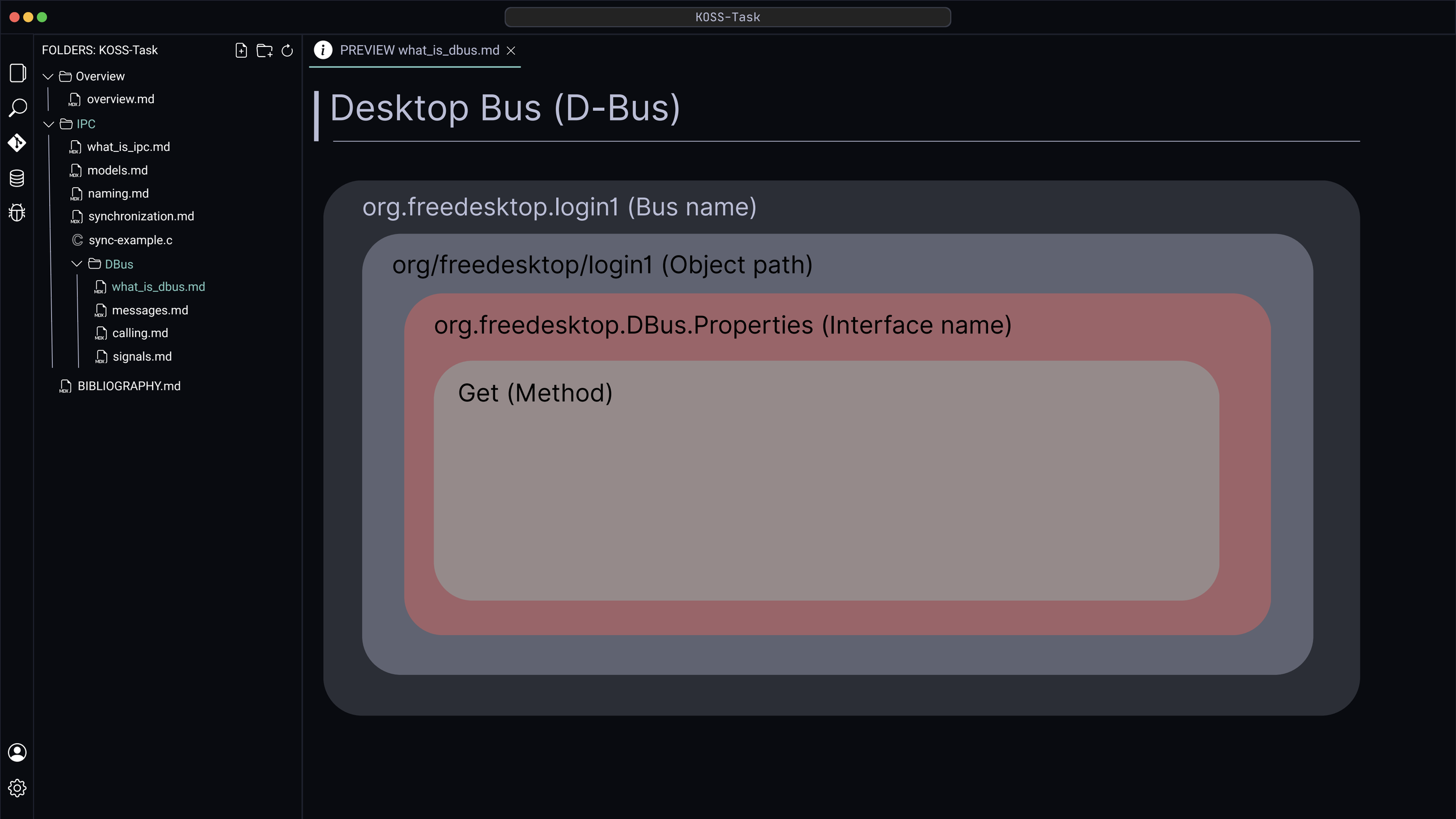
- Objects allow programs to expose different functionalities to other programs. Other programs can refer to object paths to access those functionalities provided by the said program.

Interface

- Namespaces which map different methods to their corresponding features.
- Other programs would want to call methods from the program to access information. Such methods are stored and segregated in interfaces. Example, get_window_title() could be mapped under the interface WindowProperties.

Method name

- It is the name of the method which other programs would call, provided it exists in the interface.


KOSS-Task
FOLDERS: KOSS-Task
Overview
overview.md
IPC
what_is_ipc.md
models.md
naming.md
synchronization.md
sync-example.c
DBus
what_is_dbus.md
messages.md
calling.md
signals.md
BIBLIOGRAPHY.md
PREVIEW what_is_dbus.md
Desktop Bus (D-Bus)
org.freedesktop.login1 (Bus name)
org/freedesktop/login1 (Object path)
org.freedesktop.DBus.Properties (Interface name)
Get (Method)

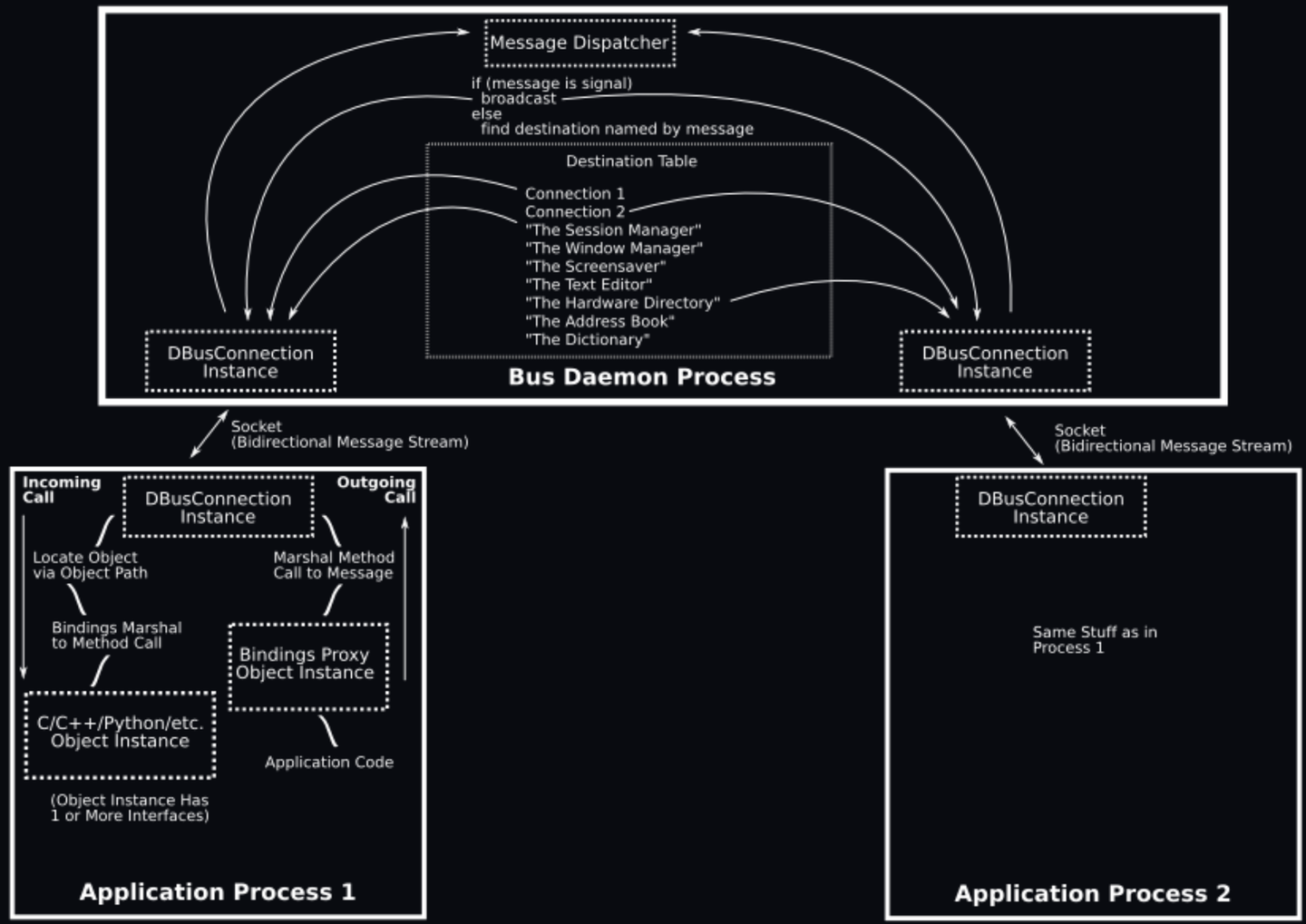
KOSS-Task

FOLDERS: KOSS-Task

- Overview
  - overview.md
- IPC
  - what_is_ipc.md
  - models.md
  - naming.md
  - synchronization.md
  - sync-example.c
  - DBus
    - what_is_dbus.md
    - messages.md
    - calling.md
    - signals.md
- BIBLIOGRAPHY.md

PREVIEW what_is_dbus.md

Desktop Bus (D-Bus) Architecture



PREVIEW messages.md

Messages

D-Bus works by sending messages between processes.

There are 4 message types:

- Method call – messages ask to invoke a method on an object.
- Method return – messages return the results of invoking a method.
- Error messages return an exception caused by invoking a method.
- Signal messages are notifications that a given signal has been emitted (that an event has occurred). You could also think of these as "event" messages.

A method call maps very simply to messages: you send a method call message, and receive either a method return message or an error message in reply.

Each message has a header, including fields, and a body, including arguments. You can think of the header as the routing information for the message, and the body as the payload. Header fields might include the sender bus name, destination bus name, method or signal name, and so forth. One of the header fields is a type signature describing the values found in the body. For example, the letter "i" means "32-bit integer" so the signature "ii" means the payload has two 32-bit integers.

KOSS-Task

FOLDERS: KOSS-Task

- Overview
  - overview.md
- IPC
  - what_is_ipc.md
  - models.md
  - naming.md
  - synchronization.md
  - sync-example.c
  - DBus
    - what_is_dbus.md
    - messages.md
    - calling.md
    - signals.md
- BIBLIOGRAPHY.md

PREVIEW calling.md

Calling a method

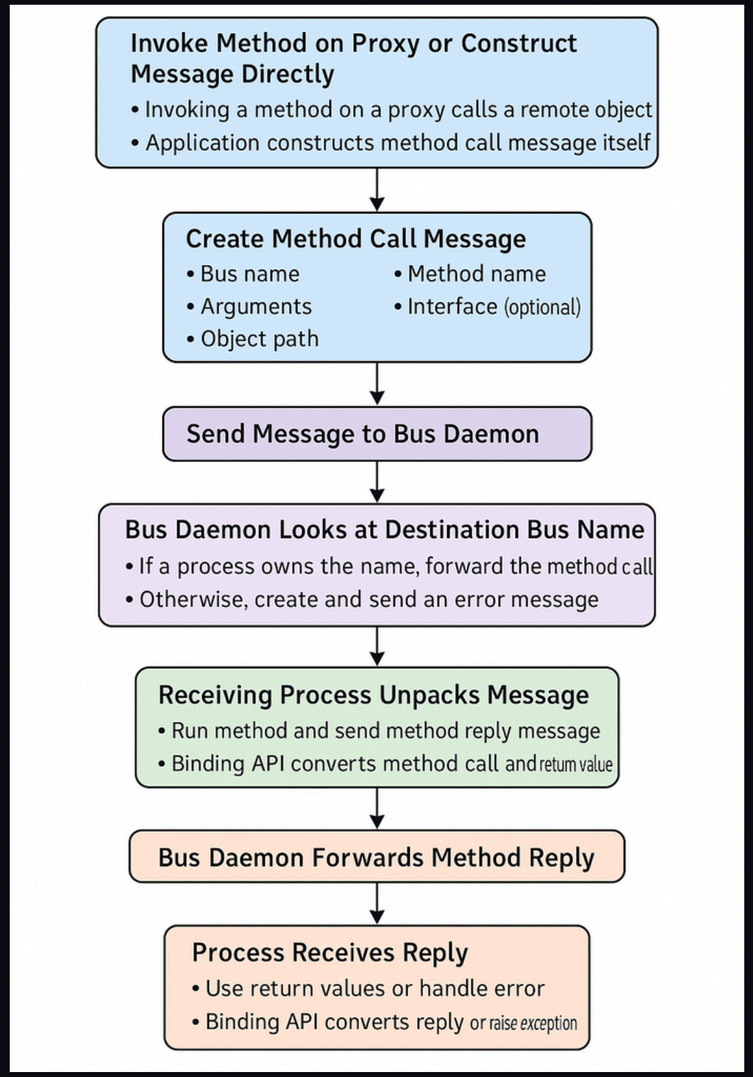
A method call in DBus consists of two messages: a method call message sent from process A to process B, and a matching method reply message sent from process B to process A.
Both the call and the reply messages are routed through the bus daemon. The caller includes a different serial number in each call message, and the reply message includes this number to allow the caller to match replies to calls.
The call message will contain any arguments to the method. The reply message may indicate an error, or may contain data returned by the method.

A method invocation in DBus happens as follows:

- A method call is invoked (with or without proxy).
- The method call message contains: a bus name belonging to the remote process, the name of the method, the arguments to the method, an object path inside the remote process, and optionally the name of the interface that specifies the method.
- The method call message is sent to the bus daemon.
- The bus daemon looks at the destination bus name. If a process owns that name, the bus daemon forwards the method call to that process. Otherwise, the bus daemon creates an error message and sends it back as the reply to the method call message.
- The bus daemon receives the method reply message and sends it to the process that made the method call.
- Kindly refer to the flowchart in the next page.

PREVIEW calling.md

Calling a method



Note – This flowchart is generated using AI.

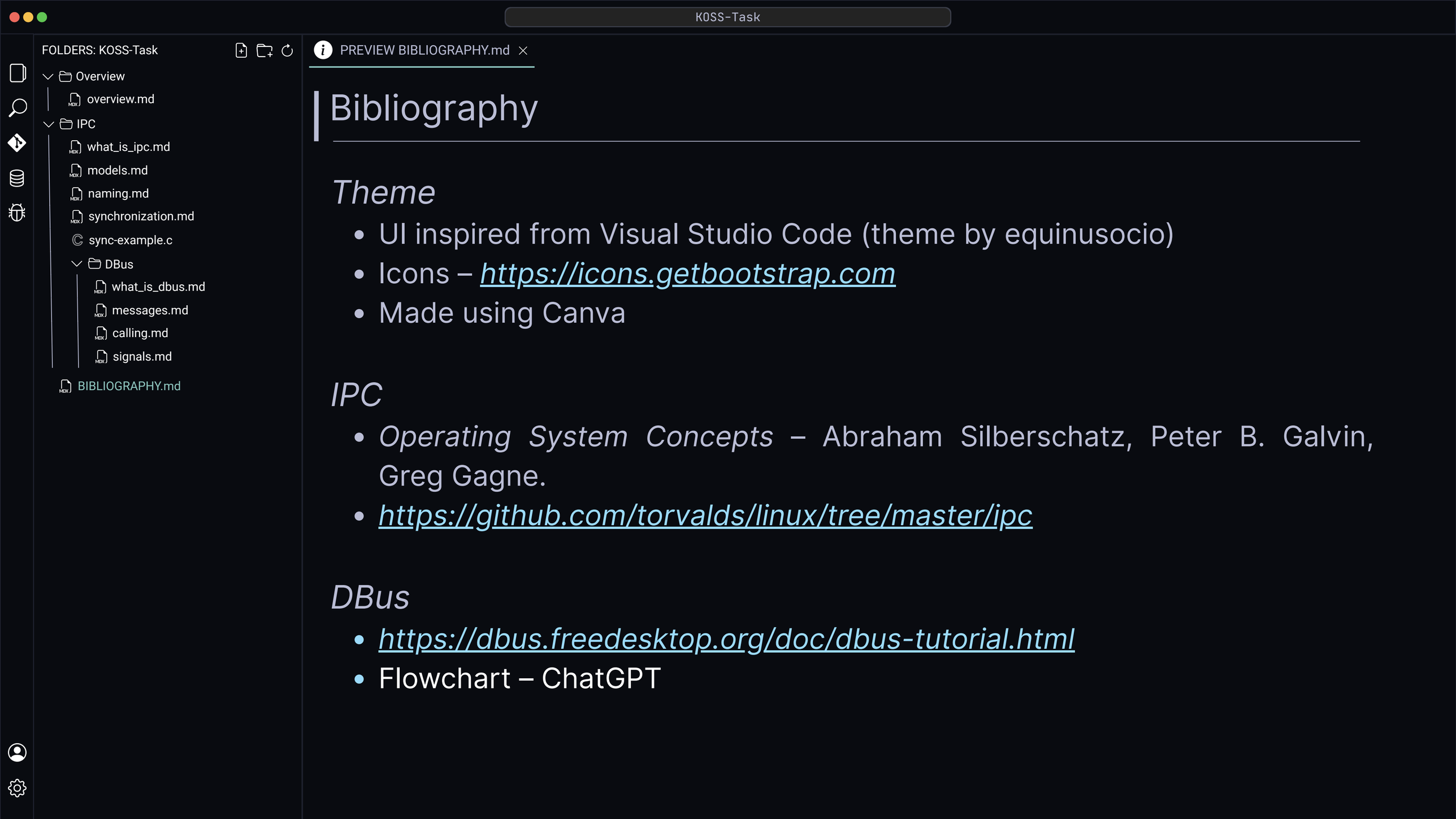
PREVIEW signals.md

Signals

A signal in DBus consists of a single message, sent by one process to any number of other processes. That is, a signal is a unidirectional broadcast.
The signal may contain arguments (a data payload), but because it is a broadcast, it never has a "return value."

A signal in DBus happens as follows

- A signal message is created and sent to the bus daemon.
- The signal message contains the name of the interface that specifies the signal; the name of the signal; the bus name of the process sending the signal; and any arguments
- Any process on the message bus can register "match rules" indicating which signals it is interested in. The bus has a list of registered match rules.
- The bus daemon examines the signal and determines which processes are interested in it. It sends the signal message to these processes.
- Each process receiving the signal decides what to do with it. If using a binding, the binding may choose to emit a native signal on a proxy object. If using the low-level API, the process may just look at the signal sender and name and decide what to do based on that.


KOSS-Task
FOLDERS: KOSS-Task
Overview
overview.md
IPC
what_is_ipc.md
models.md
naming.md
synchronization.md
sync-example.c
DBus
what_is_dbus.md
messages.md
calling.md
signals.md
BIBLIOGRAPHY.md
PREVIEW BIBLIOGRAPHY.md
Bibliography
Theme
UI inspired from Visual Studio Code (theme by equinusocio)
Icons – https://icons.getbootstrap.com
Made using Canva
IPC
Operating System Concepts – Abraham Silberschatz, Peter B. Galvin, Greg Gagne.
https://github.com/torvalds/linux/tree/master/ipc
DBus
https://dbus.freedesktop.org/doc/dbus-tutorial.html
Flowchart – ChatGPT