Algorithms and Operating Systems Monsoon 2020 - *Project*

On the Duality of Operating Systems Structures

(Paper authors: Lauer and Needham)

Team Name - Enigma
Gowri Lekshmy - 20171053
Shivani Chepuri - 2018122004

Main paper link: https://dl.acm.org/doi/pdf/10.1145/850657.850658
pdf can be found here http://cseweb.ucsd.edu/classes/fa08/cse221/papers/lauer78.pdf

Aim:

- Studying and understanding the dual nature of "**Message-oriented System**" and "**Procedure-oriented System**" with the help of (the research paper) "**On the Duality of Operating" by Hugh C. Lauer AND Roger M. Needham Systems Structures**".
- Literature Survey within the scope of this paper.

Literature Survey Outline and Scope:

In regard to the topics and insights mentioned in the main research paper, we want to explore more papers related to Duality in OS, keeping the scope limited to the ideas in the main research paper.
(survey papers/links are mentioned in the References section)

Problem Statement:

Operating system designs are usually **categorized on the basis of their implementation and the notions of process and synchronization.** In this paper, it is demonstrated that the two categories of Operating systems - **"Message-oriented System" (**Event Based Systems**)** and **"Procedure-oriented System"** (Thread-Based Systems) are duals of each other and that a system which is constructed according to one model has a direct counterpart in the other.

Motivation:

- Different classes of operating systems have varying properties involving **synchronization, processes or interprocess communication** occurring within these systems and among their clients.
- Considering the existence of two broad categories of Operating systems, Event-Based Systems (i.e, **Message-oriented System**) and Thread-based systems (i.e, **Procedure-oriented System**), there is a certain bias favouring one system over the other.
- We have looked into various aspects and topics provided and this duality caught our eye. We want to explore the tradeoffs and the reason behind this duality.
- To acknowledge this highly controversial belief the authors have empirically analysed the two systems and demonstrated that both are essentially duals of each other.

Index/Workflow:

Project Objectives:

- Brief study of Event based and Thread based systems
- Extensive Study of these two systems
- To understand how they are logically equivalent and duals of each other, as proposed.
- To study the characteristics of both the models in terms of duality , similarity of programs and performance are presented.
- To empirically demonstrate with the help of an experimental OS, the Cambridge CAP Computer, by taking help from the paper.

Important Terms:

- **Procedure-oriented System**: characterized by a large, rapidly changing number of small processes and a process synchronization mechanism based on shared data.
- **Message-oriented System:** characterized by a relatively small, static number of processes with an explicit message system for communicating among them.
- **Message:** A message is a data structure meant for sending information from one process to another which contains a small, fixed area for data which is passed by value and space for a pointer to larger data structures which must be passed by reference.
- **Message Identifier**: A message identifier is a handle by which a particular message can be identified.
- **Message port**: A message port is a queue that is capable of holding messages of a certain class or type which might be received by a particular process.

- **Message channel**: A message channel is an abstract structure which identifies the destination of a message. Each message channel must be bound to a particular message port before it can be used. A message port may have more than one message channel bound to it.
- **Procedures:** A procedure is a piece of Mesa text containing algorithms, local data, parameters, and results. It always operates in the scope of a Mesa module and may access any global data declared in that module (as well as in any containing procedures).
- **Process declarations**: A process (or more precisely, a process template) consists of local data and algorithms, defines certain message ports, and refers to (i.e., sends messages to) certain message channels representing other processes.
- **Modules**: A module is the primitive Mesa unit of compilation and consists of a collection of procedures and data. The scope rules of the language determine which of these. procedures and data are accessible or callable from outside the module.
- **Monitors:** A monitor is a special kind of Mesa module which has associated with it a lock to prevent more than one process from executing inside of it at any one time.

A Brief Study on Event Based Systems:

- Message-oriented Systems are Event-based systems.
- Eg. Nginx, Redis and Node (node.js server)
- Event : A significant change in state [10]. Refer to instances like a new incoming connection, ready for read, ready for write, etc.
- In "event driven" runtimes, when a request comes in, the event is dispatched and the **event handler** will pick it up.
- **Note:** All the handlers are called in the same thread. The event loop doesn't have a thread pool to use, it only has one thread.

One question that is intriguing is, How are these Operating Systems (with a single thread) just as fast (or even better) as the Thread based systems, which have all the advantages of parallelism and multiprogramming?

Primary Components: Callbacks, An event loop , A queue

Working:

- The unit of work, or task, is a callback.
- Only one callback is ever executing at a time. There are no locking issues.
- They have an event loop that will listen for an event saying that an asynchronous operation (IO) has completed and then execute the callback that was registered when the Async operation started. Rinse, then repeat.
- In simple terms, it uses a single threaded event loop blocking on resource-emitting events and dispatches them to corresponding handlers and callbacks.
- There is no need to block on I/O, as long as handlers and callbacks for events are registered to take care of them.

- It never waits for anything, which means that the single thread can go hell-for-leather just running code — which makes it really fast.
- **Note:** Handlers/callbacks may utilize a thread pool in multi-core environments.

Handling Multiple Requests:

If a handler takes a very long time to finish (say by having a computationally intensive for loop inside), no other request will be handled during that time, because the event loop will not invoke the next handler before the current one completes. Therefore, multi-programming is lower compared to Thread-based systems.

Measuring RPS (Requests Per Second):

For an event-based server, with a single cpu, (and asynchronous) we can skip the waiting time(t_wait) and consider only CPU time (t_cpu).

RPS = (1000 / t_cpu)

(look at RPS in Thread Based systems for clearer explanation)

Advantages:

- No struggles of multithreading programming.
- No locking issues
- Loose coupling: Services do not need to be dependent on each other.
- Scalability as a consequence of loose coupling
- Asynchronicity: Since services are no longer dependent on a result being returned synchronously, a fire and forget model can be used, which can greatly speed up a process.
- Point in time recovery: If events are backed by a queue or maintaining some kind of history, it is possible to replay events, or even go back in time and recover state.
- Does not waste much CPU cycles compared to thread architecture. The (only one) thread returns back to the thread pool before the response is complete and is ready to serve some other request.
- Lower thread waiting compared to Thread Based (ref images below)

Disadvantages:

- Inconsistencies — Because processes now rely on eventual consistency, it is not typical to support ACID (atomicity, consistency, isolation, durability) transactions, so handling of duplications, or out of sequence events can make service code more complicated, and harder to test and debug all situations.
- Over-engineering of processes — Sometimes a simple call from one service to another is enough. If a process uses event based architecture, it usually requires much more infrastructure to support it, which will add costs (as it will need a queueing system)

A Brief Study on Thread-Based Systems:

- Procedure-oriented Systems are Thread-based systems.
- In "thread driven" runtimes, when a request comes in, a new thread is created and all the handling is done in that thread.No real operating system fits into one category in all aspects and properties. Most Operating systems have constituent subsystems where some of them belong to one system and the rest belong to the other.

Working:

- Thread-based applications essentially divide work up in hardware.
- Eg: All multithreaded servers,
- Each work unit has its own thread, and will block if it needs to (like when it's waiting for IO), the CPU will suspend that thread and start running another that is waiting, etc.
- Every time that happens there is quite a hefty context switch, including moving (about 2MB of data around on an average). In effect the hardware has an upper hand in deciding when to yield control.

Handling Multiple Requests:

If the handler takes a lot of time to complete, it won't hurt other threads much, because they can run at the same time almost independently.

Measuring RPS (Requests Per Second):

Consider a single core server. Suppose that a request to a server takes t_cpu CPU milliseconds and takes t_wait time waiting for external stuff,

Total response time is t_cpu + t_wait milliseconds.

Say, the server can handle **threadNum** threads before performance degrades

min((threadNum * (1000 / (t_wait + t_cpu)), (1000 / t_cpu))

Here, We have to consider the smallest number among CPU bound and Thread bound scenarios because even if we increase the number of threads we are running on a physically limited CPU resource (thread doesn't mean CPU core).

Advantages:

- Writing synchronous code for a multithreaded environment is a lot easier than writing asynchronous code for a single threaded environment. It is getting better though, by some emerging patterns and rules in the community.
    - The pattern for Async functions is that the last argument is always the callback, and the pattern for callbacks is that errors are always the first argument (with results after that). This standardisation makes composition really easy.

Advantages compared to Process based systems:

- Faster context switching
- Simpler System Global Area allocation routine, because it does not require use of shared memory
- Faster spawning of new connections, because threads are created more quickly than processes
- Decreased memory usage, because threads share more data structures than processes

Disadvantages:

- A lot of multithreading programming is involved.
- Extensive locking strategies to avoid concurrent data access and deadlocks
- Linking Task Parallel Library (TPL) with event processing libraries like Reactive Extensions (Rx) is very extensive and heavy, especially in web applications. The new *await* keyword helps make this task easier though.
- Thread-based server wastes cpu cycles. If we need to call an external service to resolve the request the thread has to wait for the response
- Higher thread waiting compared to Event Based (ref images below)

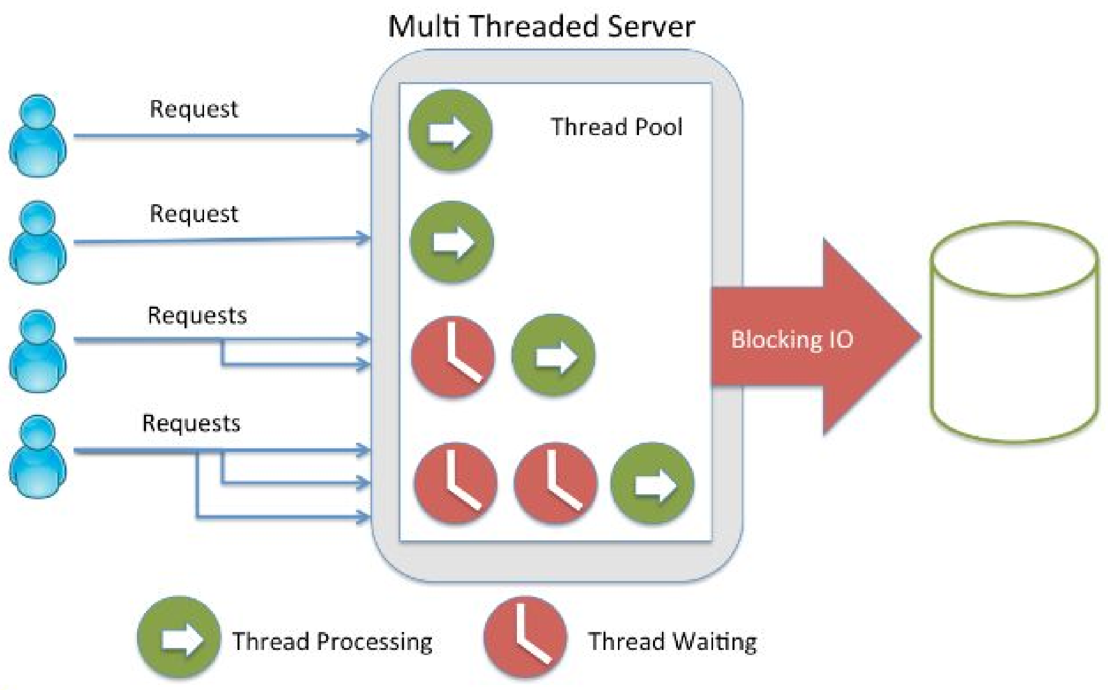
Main differences between thread-based and event-driven server architectures:

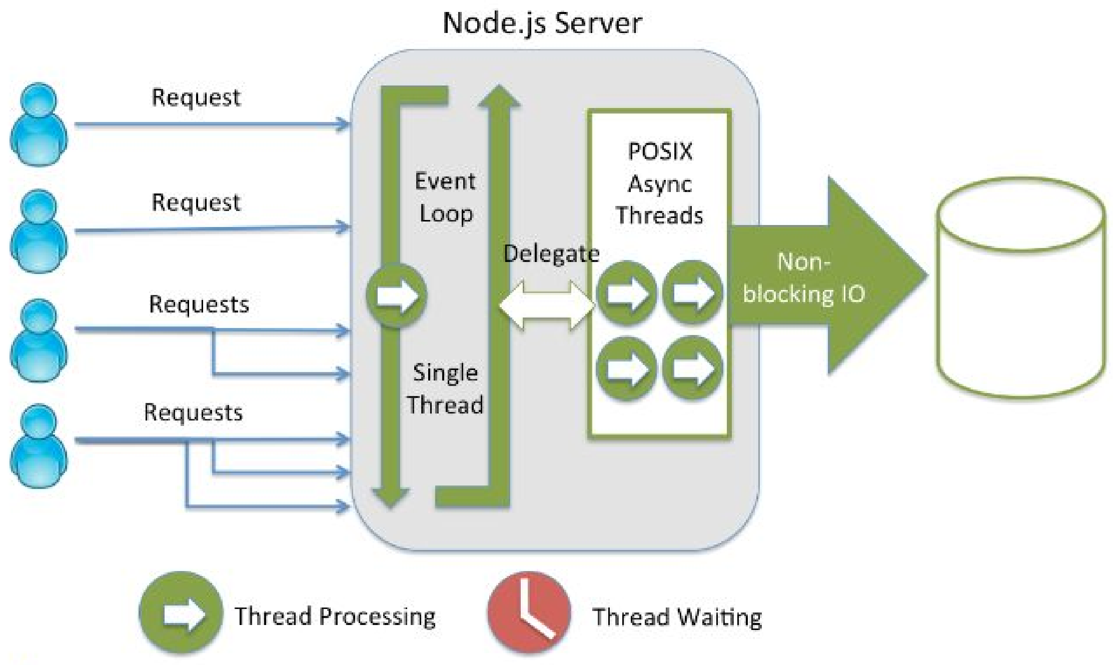
Differences	**Thread-based**	**Event-driven**
connection/request state	thread context	state machine/continuation
main I/O model	synchronous/blocking	asynchronous/non-blocking
activity flow	thread-per-connection	events and associated handlers
primary scheduling strategy	preemptive (OS)	cooperative
scheduling component	Scheduler (OS)	event loop
calling semantics	blocking	dispatching/awaiting events

Use Case - Application Server (with external call):

A classic application, where we require access to a database or the file system, we can suppose a CPU time of 2 ms and waiting time for the external resource of 98 ms.

thread-based -> 250.0 event-based -> 500.0
This is a very common use case, the event-based server can handle more users per seconds.





Ref [13]

Let us now look into the two models of OS based on these two architectures respectively, as described in the paper and then arrive at the Duality Argument.

Overview of the Two Models of Operating Systems

- Both the models have primitive operations that perform differently for managing processes and synchronisation.
- When operations have to be performed by multiple processes on the **same data**:
    - Data is **shared using common address space** for all processes with locking mechanisms for “procedure oriented systems”.
    - Data is **divided and process-specific** for “message oriented systems”.

A **canonical model** has been developed for each form of Operating System to prove that neither system is inherently better than each other.

1. Message-oriented System

The Message-oriented model performs by **passing data encoded by messages/events among processes**.

In simple terms, This system uses a small number of processes that use explicit messaging, resembling event-driven systems.

- They are characterized by a small, **static number of processes** since deletion, creation and changing of connections are difficult due to the queuing mechanism.
- Have an **explicit set of message channels** between them.
- Synchronisation among processes is managed with the help of queues. Messages are queued at the destination (message ports) until they can be acted upon and pre-emption can occur due to high priority messages.
- Data used by multiple processes is passed by reference and processes rarely share data in memory.
- Communication between **peripheral devices** is done by message-passing by treating them as **virtual processes.**

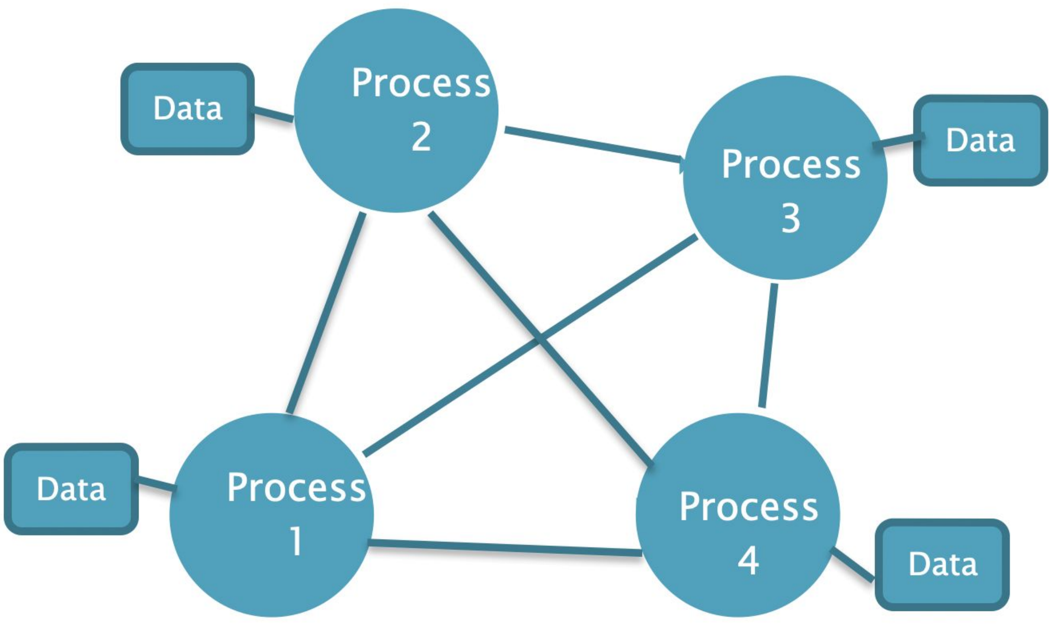
Examples: GEC 4080, IBM OS/360

Operations

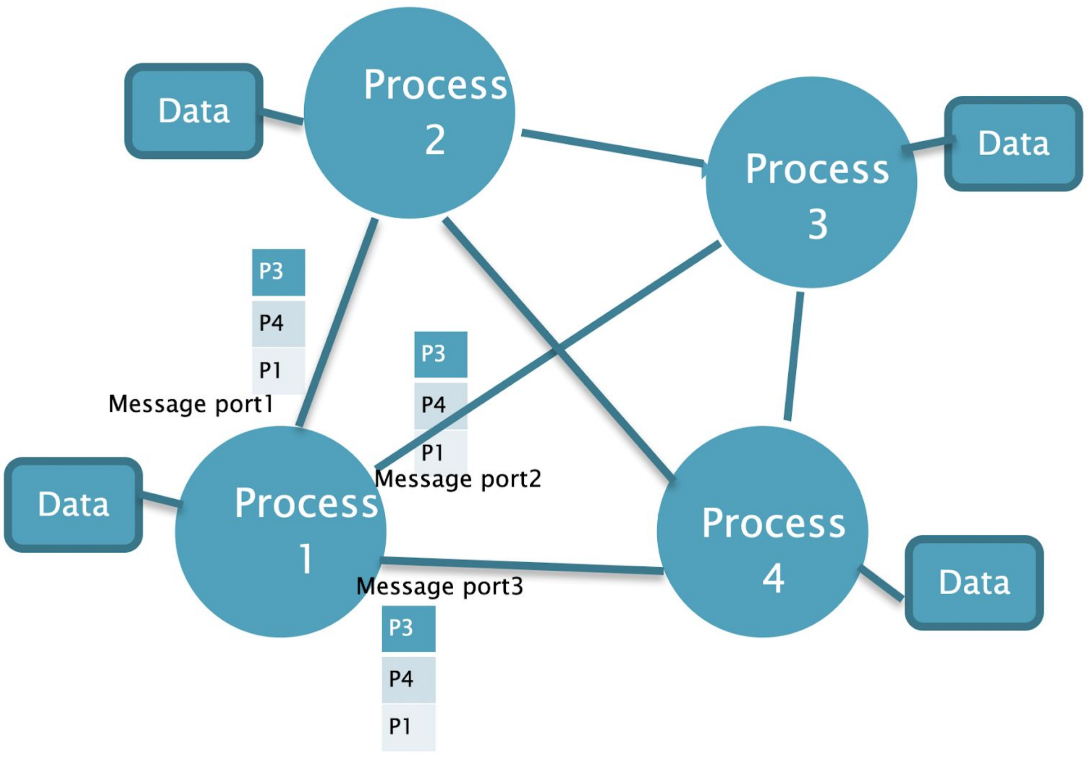
- Four Primitive Operations for Message Transmission
    - **[Send Message]**
    - **[AwaitReply]**
    - **[WaitForMessage]**
    - **[SendReply]**
- **[CreateProcess] -** This operation creates an instance of a process and binds the message channel to its message port
- There is no **DeleteProcess** operation for this model

An example of Process Interaction

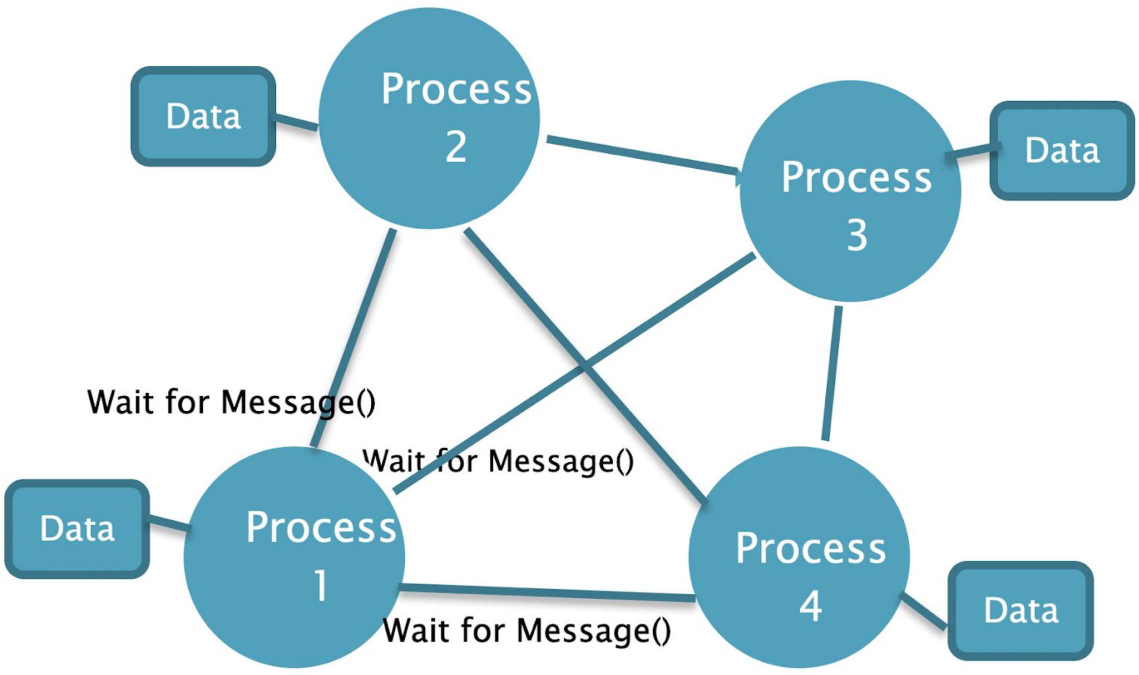
1) Each process has its own data. A process operates on data contained by another process by means of explicit message passing mechanism



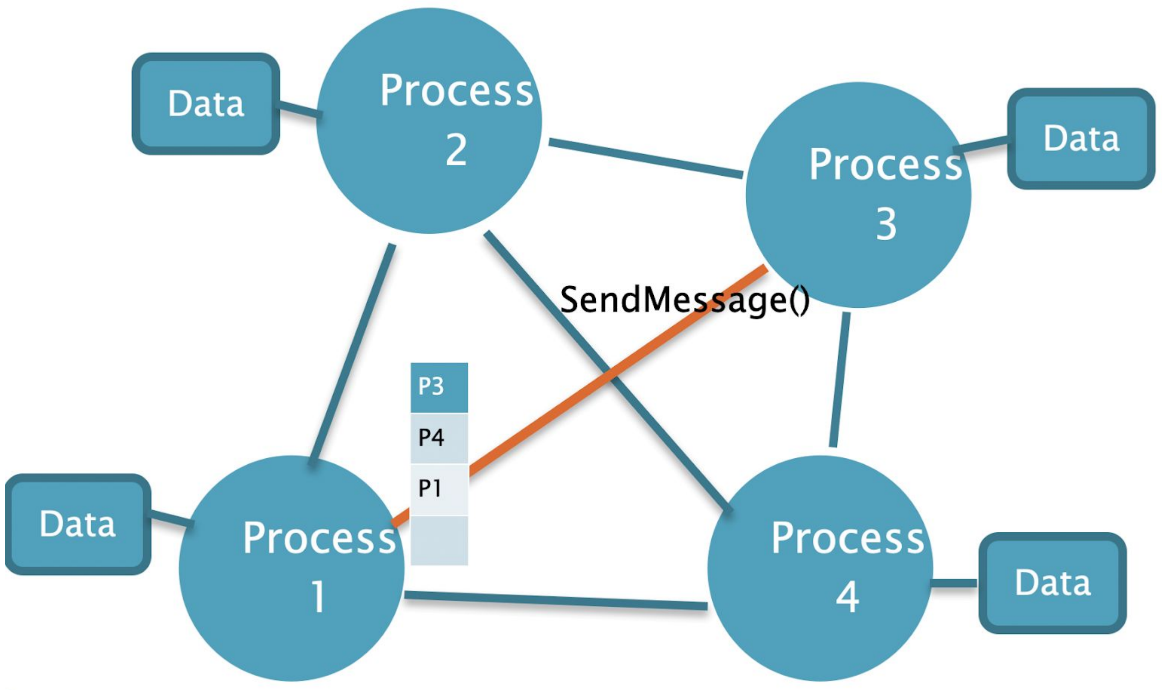
2) Processes have message ports to receive messages in queue



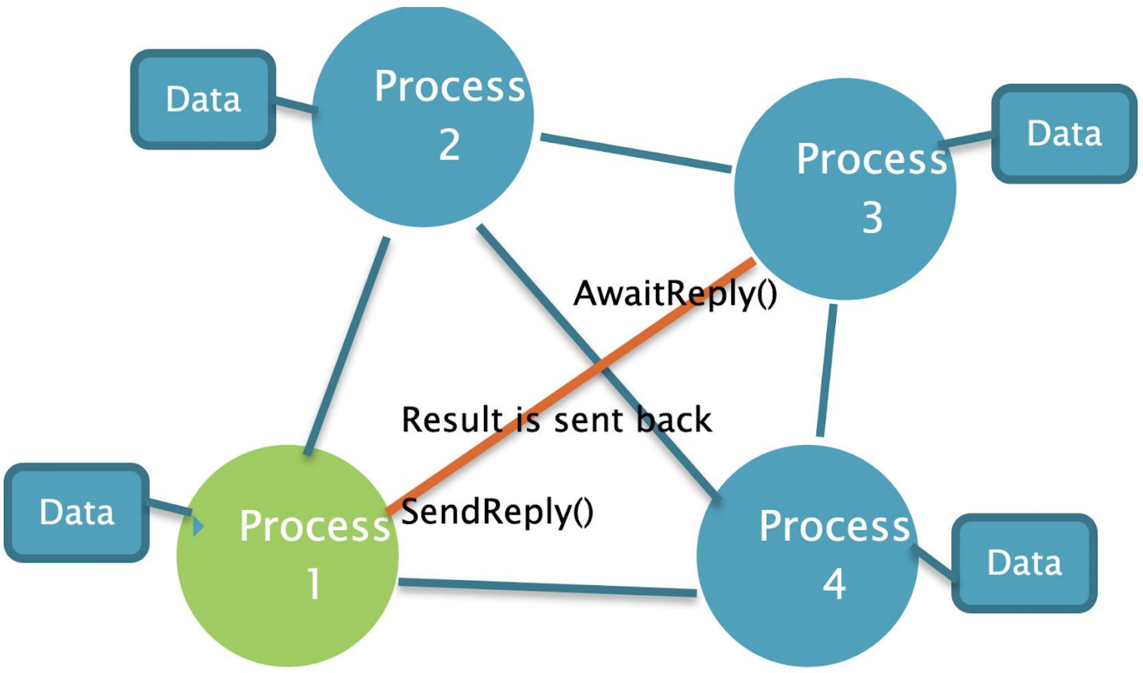
3) **All processes wait for requests from other processes in all its ports. The process containing the data receives a request in the form of a message.**



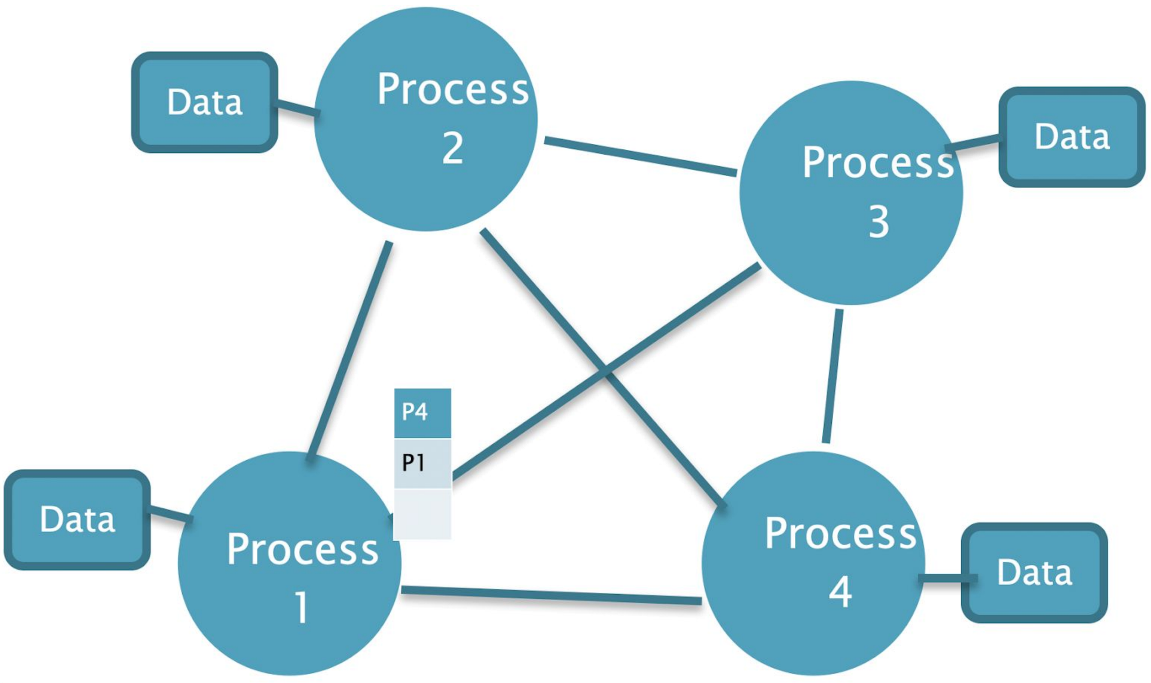
4) **Process 3 sends a request for Process 1 Data. The request is queued in the message port of process1.**



5) **The data is operated by the algorithm defined for that port which received the message. The requesting process can either block (synchronous)/ resume its operation (asynchronous).**



6) **Process 1 processes the next message in the queue. Similar to the Event handler mechanism. All processes process only one message at a time.**



- Communication between components of this system is done exclusively through messages.
- There is no need for any locking mechanisms since the processes are controlled **serially with no simultaneous access or updates.**
- These systems are usually employed for **Real-time systems** and process control with the applications encoded in message blocks.

2. Procedure-oriented System

The procedure-oriented model is characterized by **procedural call facilities** which can take a process very rapidly from one context to another along with **synchronisation using locks, semaphores and monitors**.

In simple terms, this system uses a large number of small processes using shared data, resembling thread-based systems.

- This system has a **large number of small processes** (threads) with easy creation and deletion of processes as there are no communication channels.
- Communication between processes is by means of direct sharing and locking of data. There is little or no direct communication.
- Processes **share memory in kernel space** and are similar to a multithreaded environment.
- Global data is both protected and efficiently accessed by using procedural interfaces which do perform synchronization and manipulation in controlled ways.
- In this kind of system, a process first attempts to claim a lock, and may be forced to wait on a queue until some other process releases it. **Pre-emption of the process** occurs when a release operation is performed on a lock which a **'higher priority'** process is attempting to claim.
- **Process creation is easy** since no communication channels have to be set up with existing processes and the deletion of a process is also easy as long as it is not holding any locks.
- Global naming scheme is being used in this system

Examples: HYDRA, the Plessey System 250, GEC 4080

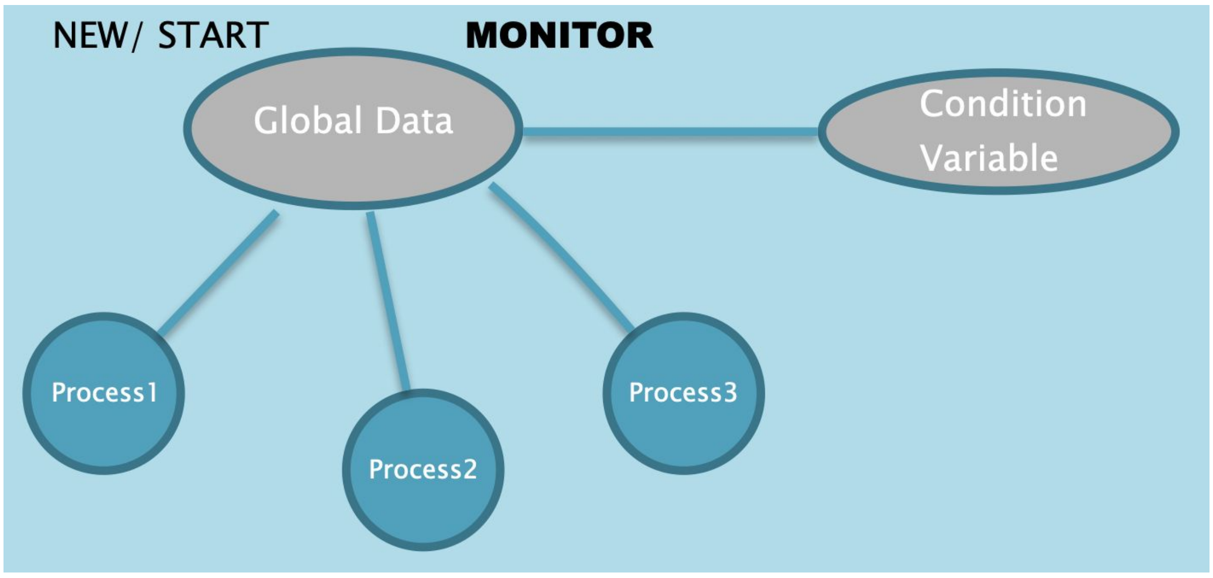
Canonical Model

- In the paper, the canonical model for the procedure-oriented system was built by providing a **programming environment similar to Mesa** (Pascal like language) with hypothetical features.
- **Processes have dynamic priorities** and priorities are associated with the locks or data structures and correspond to the timing requirements of the resources they represent.
- **Role of Procedures**
  - **Procedure:** A procedure is a piece of Mesa text containing algorithms, local data, parameters, and results.
  - There are two types of procedure call facilities:
    - **Synchronous** - This mechanism returns results and performs similar to Mesa procedural calls
    - **Asynchronous** - This mechanism performs similar to Algol and Pascal procedural calls with FORK and JOIN statements.

- The canonical model for this system uses **modules and monitor instances** which are comparatively more difficult to handle than processes.
- **Monitors and Modules**
  - Module is a primitive unit of compilation containing procedures and data.
  - A monitor is a special kind of module that has private data and procedures, protected with a lock.
  - Processes must acquire the lock when they call an entry procedure (a procedure which can be called from outside the monitor).
  - Only one process can operate inside the monitor at a time.
  - Can be **instantiated using NEW and START** statements.
- **Condition variables** are used to control waiting and to have flexible synchronisation among the processes than mutual exclusion.
  - wait operation
  - signal operation
- Finally , procedure-oriented OS uses purely procedural based communication calls between its components and performs concurrent activities with the help of asynchronous calls controlled by monitor locks and condition variables.

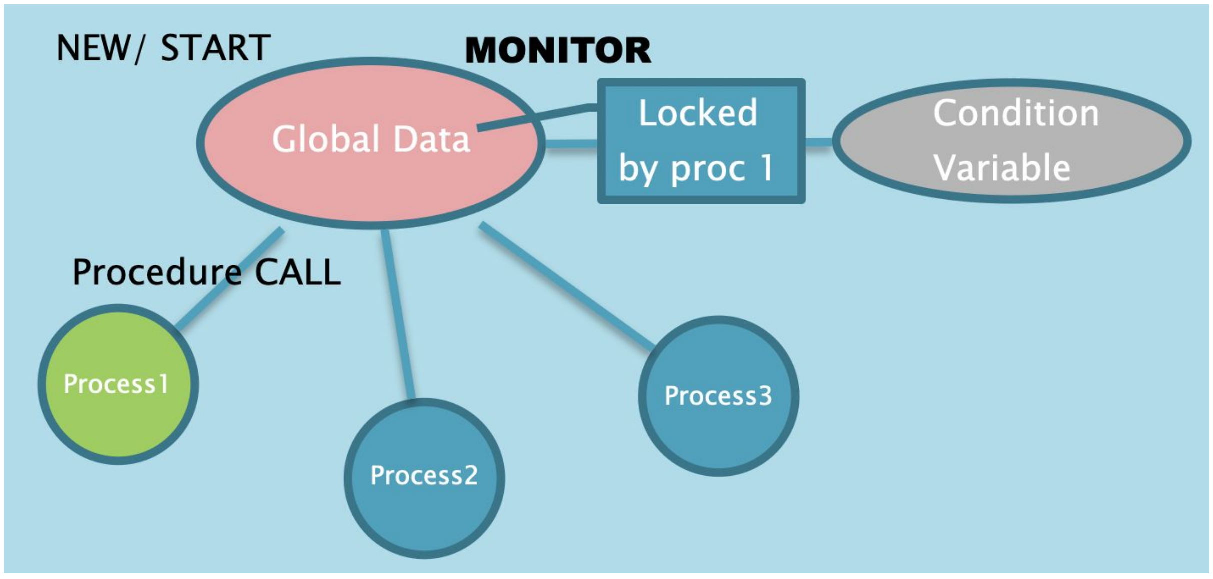
An example of Process Interaction

1) **Global Data is shared across multiple processes. Monitor encapsulates the data and methods and provides synchronized data access.**

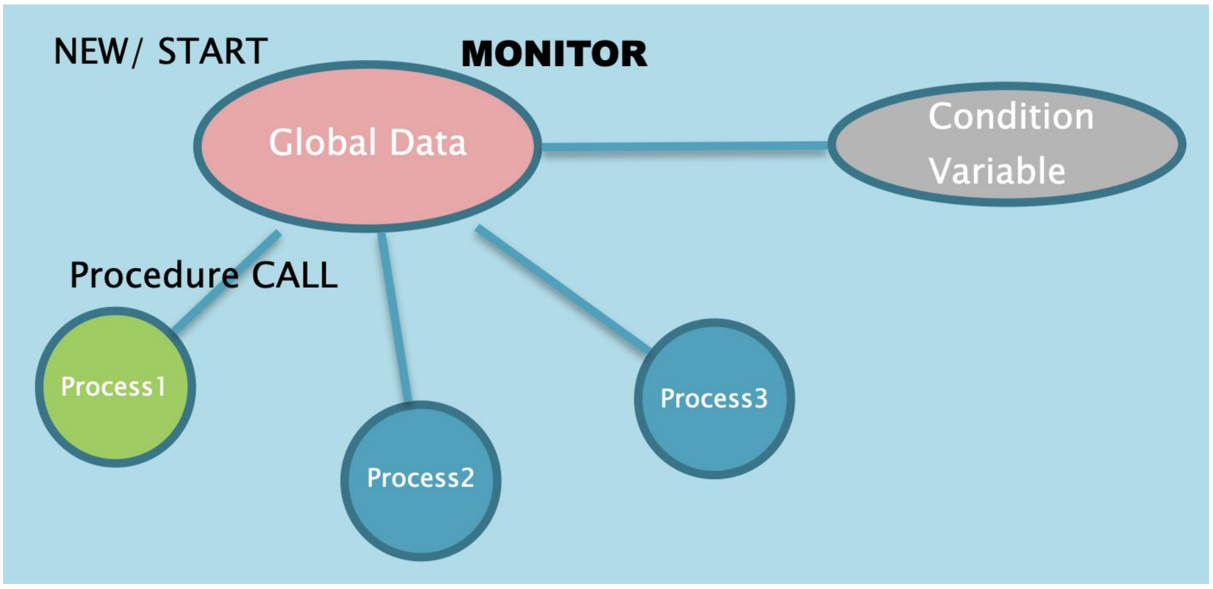


2) **A process should first acquire mutex for data, and then perform its operations. Any other process requesting for mutex is queued. No other process is allowed to access the data if it's locked. Procedure can wait (synchronous) or proceed with**

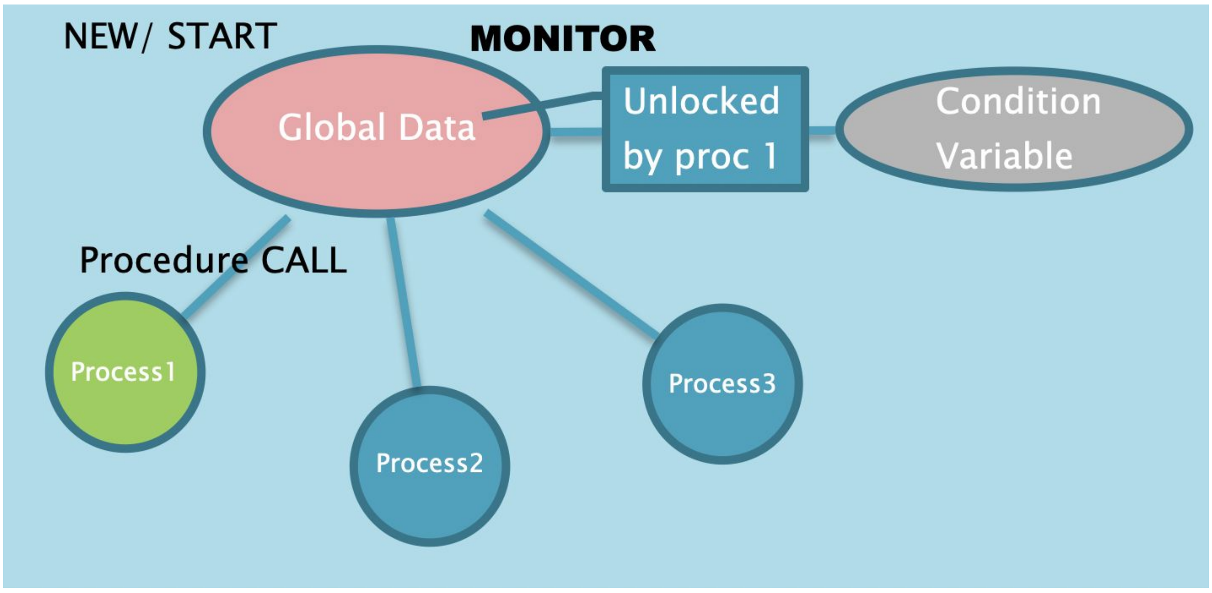
its operations (asynchronous).



3) Algorithm for Procedure 1:
   a) if Resource is exhausted then WAIT C.
   b) the process waits until some other process does SIGNAL C.
   c) Procedure 1 wakes up when data is ready and then finishes.



4) Process 1 releases the mutex. Next process in the queue is scheduled. The data is operated by one process at a time. Thus Synchronization is achieved.



- The only means of interaction in this system among its components is procedural.
- This system uses **asynchronous calls to stimulate concurrent activity.**
- It depends upon monitor locks and condition variables to keep out of the way of each other. Thus no process can be associated with a single address space unless that space be the whole system.

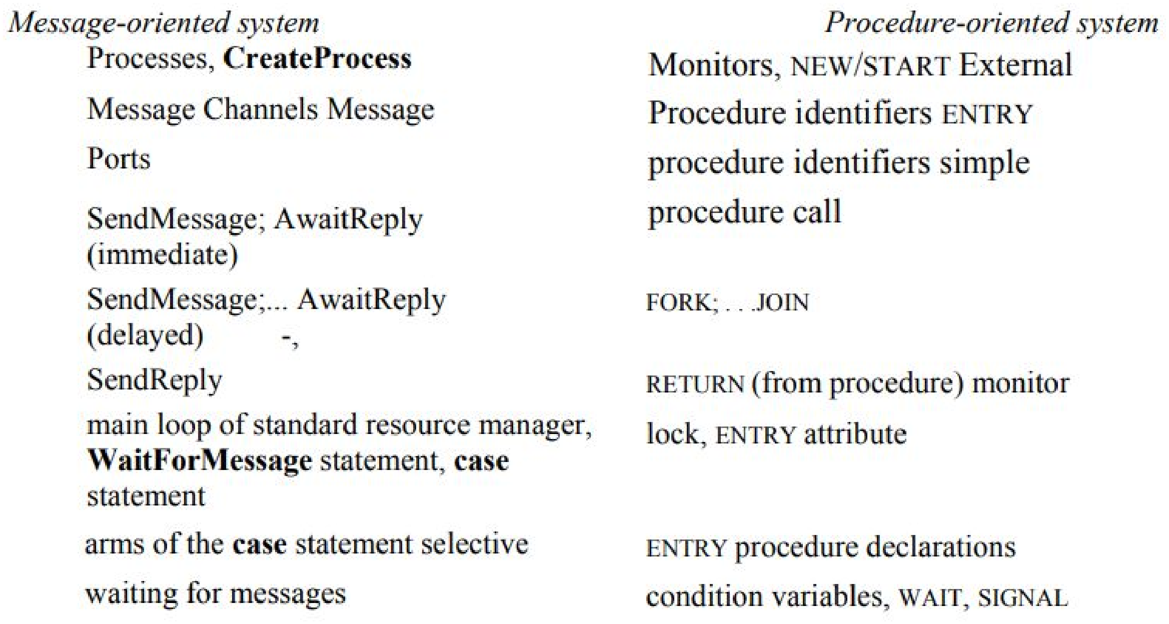
<u>Characteristics of the Models</u>

The Duality Argument:

- **Duality Principle:**
  - (Math) Boolean Algebra remains unchanged when the dual pairs are interchanged.
  - When values and operations can be paired up in a way that leaves everything important unchanged when all pairs are switched simultaneously, we call the members of each pair dual to each other.
  - (Our Context)The primitive operations of these two systems can be swapped simultaneously preserving the logic and performance of a program.
- The flow of control in an application using either one of the models generates a unique graph that contains certain yielding or blocking nodes (e.g. awaiting a reply resp. a procedure return).
- The edges between such nodes represent code that is executed when traversing the graph. According to the duality argument, both thread-based and event-driven programs yield the same blocking points, when equivalent logic is implemented and the programs are thus duals. This graph representation is called a **blocking graph**.

Thread-based	**Event-driven**
monitor	event handler
scheduling	event loop
exported functions	event types accepted by event handler
returning from a procedure	dispatching a reply
executing a blocking procedure call	dispatching a message, awaiting a reply
waiting on condition variables	awaiting messages

Correspondence between the basic system facilities and canonical styles for resource managers:



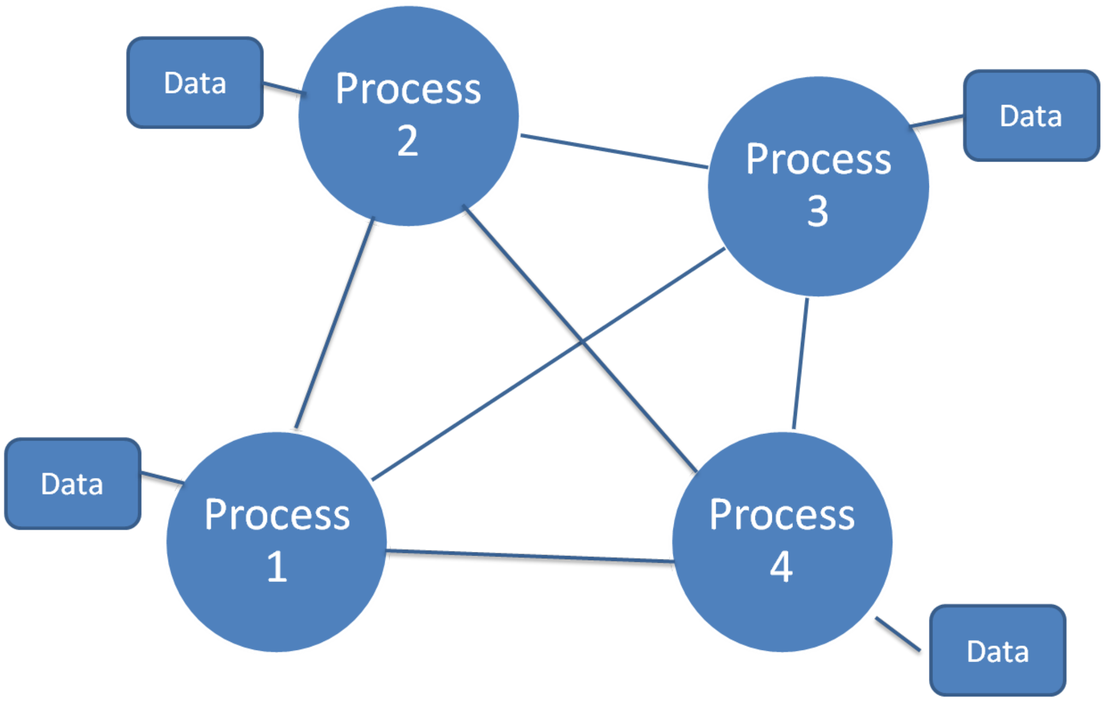
Message-oriented system	*Procedure-oriented system*
Processes, **CreateProcess**	Monitors, NEW/START External
Message Channels Message	Procedure identifiers ENTRY
Ports	procedure identifiers simple
SendMessage; AwaitReply (immediate)	procedure call
SendMessage;... AwaitReply (delayed) -,	FORK; . . .JOIN
SendReply	RETURN (from procedure) monitor
main loop of standard resource manager, **WaitForMessage** statement, **case** statement	lock, ENTRY attribute
arms of the **case** statement selective	ENTRY procedure declarations
waiting for messages	condition variables, WAIT, SIGNAL

(from the paper)

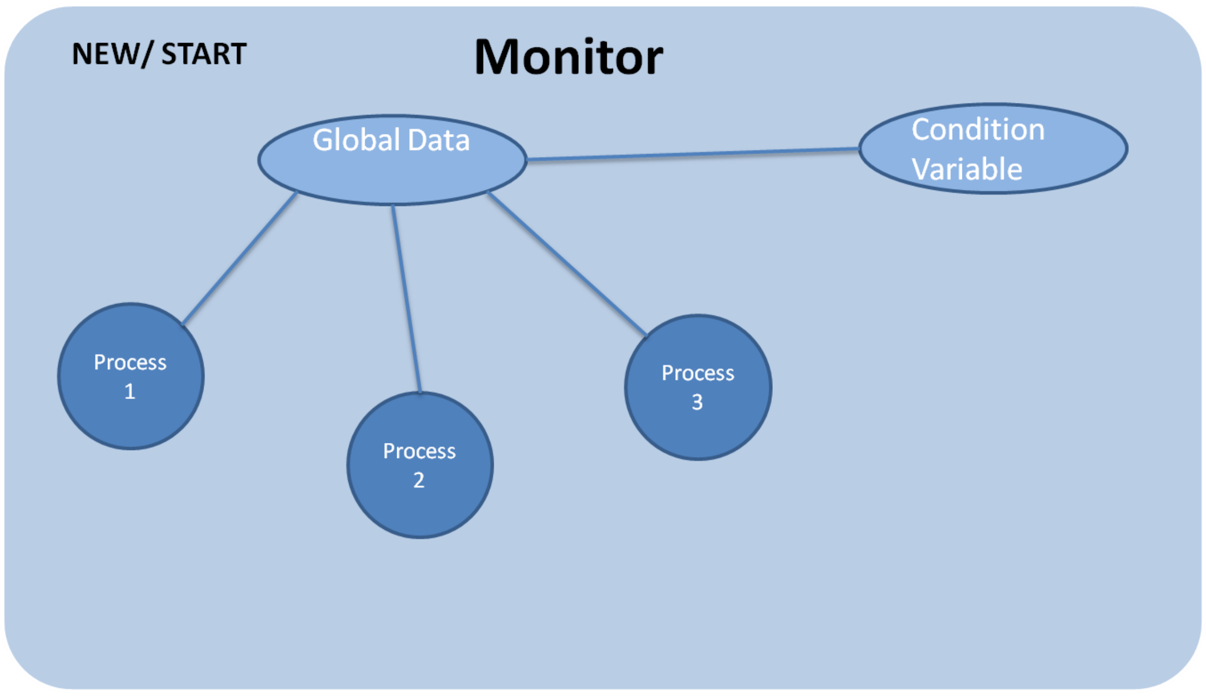
The two figures below can be used to understand the contrast (compare) in the two systems as provided above. They are mentioned to provide a difference in the architecture.

For example, when CreateProcess is called in Message Oriented System, New/Start is called in Procedure Oriented Systems, so on.. Basically, the counterparts are mapped as shown above.

Message Oriented System:



Procedure Oriented Systems:



Similarity of Programs:

- A system constructed with the primitives defined by one model can be mapped directly into a dual system, which fits the other model.
- By replacing concepts and transforming a program from one model into the other, the logic is not affected and the semantic content of the code is thus invariant.
    - None of the important parts are touched or rearranged
    - The semantic component is invariant
    - Only the syntax of primitives varies
- As a result, we can say that both conceptual models are equivalent

Preservation of Performance:

- The dynamic behaviour of a system of program can be characterised by the following three components. These are the elements that affect performance.
    - The execution times of the programs themselves
    - The computational overhead/load of the of the primitive system operations they call
    - The queuing and waiting times which reflect the congestion and sharing of resources, dependence upon external events, and scheduling decisions.

- The performance of both models is the same, given a proper execution environment.
- The duality mapping doesn't modify the main bodies of the programs
- Speed of execution of a single program is same
- Number of additions, multiplications, comparisons are the same. So the amount of computing power is same
- In a suite, the way in which the executions of those programs interact with others is same
- Process wait time is same
- Life time of computation is same

As a consequence, we can say that the choice of the right model depends on the actual application, and neither model is preferable in general.

Related Work - Inferences from Literature Survey

Historically, there is strong criticism and support for both the models.

Issues of Thread-Driven systems

- Developing correct concurrent code with threads is considered extremely difficult. [5]
- Due to nondeterminism and preemptive scheduling, there is a lack of understandability and predictability in the multi-threaded code.
- Multithreading can be error-prone and very difficult to debug.
- Possible **state explosion** from **Interleavings** and **Memory Consistency Errors** of multiple threads.

- This makes execution analysis of concurrent code virtually impossible. This is primarily caused by the unpredictability of preemptive scheduling.
- Lee [8] argues that threads are precisely not a good abstraction for concurrent flows of execution. Extensive context switching affects performance.
- Huge numbers of threads require large amounts of memory due to their thread stacks.
- The usage of locks yields an additional overhead.
- People who support threads argue that a couple of the drawbacks mentioned are actually the result of poor threading library implementations and the essence of preemptive scheduling.

Issues of Event-Driven systems

- The most common reason why event-driven systems are rejected is their programming style. The idea of an event loop and registered event handlers yields an inversion of control. Instead of sequential operations, code is organized as a fragmented set of event handlers and callbacks.
- In non-trivial applications, this leads to heavy chaining of callbacks.
- Gustafsson refers to the notion of an event loop sequentially executing callbacks on events as a form of "delayed GOTO" .
- Compared to threads, that provide higher abstractions, event-driven systems hence appear as a step backwards. [6]

Dual Nature

- Despite these remarks, Lauer and Needham[1] argue that by replacing concepts and transforming a program from one model into the other, the logic is not affected and the semantic content of the code is thus invariant.
- As a result, they claim that both conceptual models are equivalent and even the performance of both models is the same, given a proper execution environment.; which is the crux of this report.

An empirical model

The **Cambridge CAP computer** was the **first successful experimental computer** that demonstrated the use of security capabilities, both in hardware and software.

- It was based on capabilities implemented in hardware, under M. Wilkes and R. Needham with D. Wheeler responsible for the implementation.
- The CAP project has included the design and construction of a computer with an unusual and very detailed structure of memory protection, and subsequently the development of an operating system which fully exploits the protection facilities.[16]

- The architecture of the CAP[3], is implemented partly by hard logic and partly by microprogram.
- It is designed to support a very fine-grained system of memory protection.

- The intention is that each module of the program which is executed on the CAP shall have access to exactly and only that data which are required for correct functioning of the program.
- Access to a particular area of memory should never imply access to any other.
- This requirement led to the design of a non-hierarchic protection architecture.
- The CAP also supports a hierarchical structure of processes. The position of a process in the hierarchy determines the resources available to it.
- Emphasis is on
    - The representation of very detailed Protection Environments
    - Rapid environment switching

Unification of Module Interfaces

B. J. Stroustrup proposed unifying module interfaces [2] and created a **uniform interface mechanism** for activating different kinds of modules, e.g. processes, monitors, and procedures.

A uniform interface to both local and remote processes allows one to write a network interface process which runs unchanged either on the main computer or on a closely attached "NIP" computer. The usefulness of such an interface in the design of modules and in the tuning of a software to specific hardware configurations and jobloads without noticeable overheads was discussed.

New Classification model - Levels of Management

Adya et al. propose the **separation of management concepts** to argue purposefully for the most **convenient form of concurrent programming**, **aside from the coarse thread vs. event** driven model.

- Different management concepts that the programming styles of thread-based and event-based make use of for concurrency are derived.
- However, they argue that these concepts are often conflated and also confused with the actual programming styles themselves.
- Adya et al.[7] states that this makes it harder to reason about appropriate approaches towards concurrent programming. The separation of concepts yields five distinct concepts, most of them orthogonal to each other.
    - Task Management
    - Stack Management
    - I/O Management
    - Conflict Management
    - Data Partitioning

Challenges in the work:

- This form of analysis is **highly controversial** in the community.
- The observations about the similarity of program logic, code, and performance are particularly hard to accept when the universe of discourse is not one of naturally occurring objects but man-made ones.
- Like those of any empirical science, these conclusions cannot be accepted without a lot of thought and supporting experiments.
- High computational complexity incurred to build reliable systems with higher degrees of freedom, so only lower order system were built
- It is not very easy to change the structure of most operating systems in a way which would reflect the dual nature. The underlying addressing structures, use-of global data, and styles of communication are usually so bound to the design and implementation that performing the transformation to a dual version would be a major exercise, not justified by the second order gains.
- While designing the system, processes and synchronisation consume high amounts of energy compared to other parts, leading to organisational difficulties.

Drawbacks in the paper:

- Little empirical support
- No attempt is made to rigorously prove the assertions
- **Higher degrees of freedom are not considered** while analysing. Concluding that both the systems are similar is solely based on the first-order; preference for a particular system can be found in higher orders.
- The message-passing systems described in this paper do not correspond precisely to modern event systems in their full generality.
- Cooperative scheduling for event-based systems, which is an important part of many event-driven systems today, is ignored in this paper.
- They did not allow any kind of shared memory in their mapping. But many event-driven systems indeed use shared memory in a few places.
- Limited **scalability issues** in both multi-threaded and multi-process systems.
- Though the paper concludes similar performance of thread and event based systems, in **highly concurrent applications** such as Internet servers and transaction processing databases it is observed that thread-based systems are more suitable [4].

Critical Review and Comments:

- The reasoning can be considered as informal.
- No formal proof
    - Only one citable case: Cambridge CAP computer
    - Originally message-oriented
- Switched to process-oriented, with little change
- Mixed acceptance among a (biased) peer community

Observations in today's Operating Systems:

- Most systems are biased toward one of these two types.

- No real operating system exactly fits either model in all respects. Many operating systems have some subsystems that fit one model, and others that fit the other.

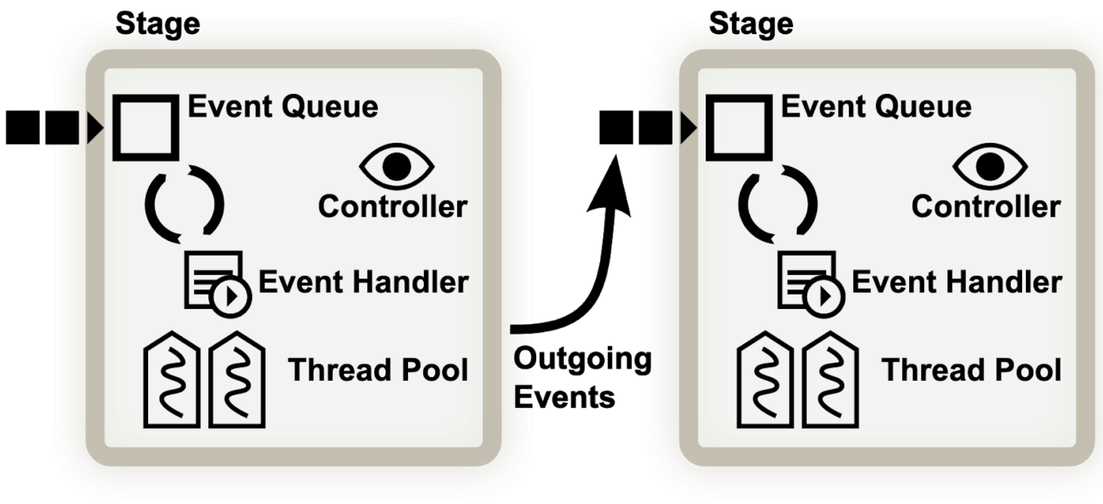
Inferences and 'Take Away' from the paper:

- The duality argument motivates us to question the implementation of the models instead of the models themselves when it comes to performance and scalability.
- Choose model based on
    - Machine Architecture and hardware (the environment in which the OS is running)
    - Programming Environment
    - Not on the software that will be running on the OS
- Neither system is inherently better than the other.
- Eliminates some degrees of freedom in the design process
- This property of equivalence between two categories of operating system can be used in effective implementation of OS on a particular architecture based on performance calculation beforehand.

Scope for Improvement - Recent Developments and Proposed Changes:

- **Hybrid/ Special Purpose Libraries**
    - Hybrid libraries, combining threads and events, have been developed. A combined model for Haskell has been implemented based on concurrency monads.
    - The programming language Scala also provides event-driven and multi-threaded concurrency, that can be combined for server implementations.

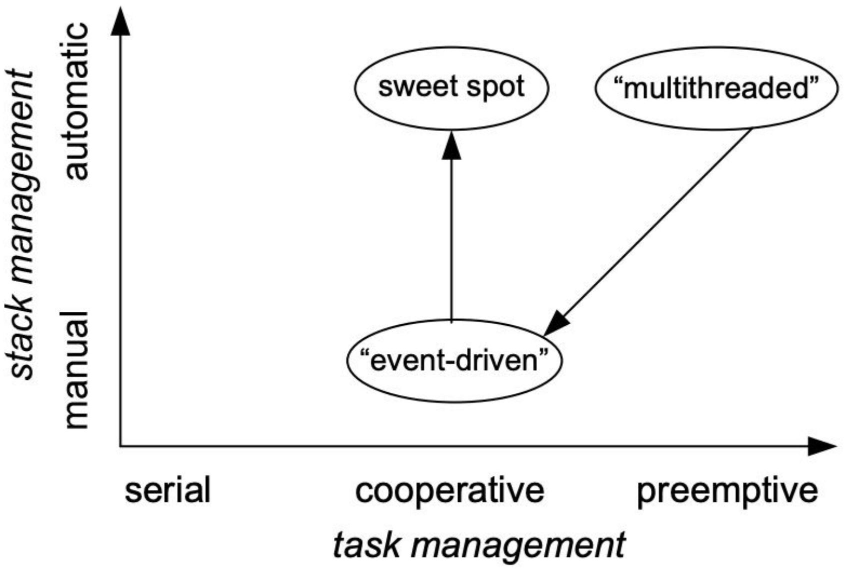
- **To improve scalability - Staged Event-Driven Architecture (SEDA)**
    - To handle the drawbacks of both general models and scalability issues, this architecture is developed by incorporating features of both models (threads and events) and is called SEDA.[9]
    - The concept of SEDA includes dividing the server logic into a series of well-defined stages. Here there are two stages, each with a queue for incoming events, an event handler backed by thread pool and a controller that monitors resources. The only interaction between stages is the passing of events to the next stage in the pipeline.



- Since separation favors modularity, the pipeline of stages can be changed and extended easily.
- Another important feature of the SEDA design is the **resource awareness and explicit control of load**. The size of the enqueued items per stage and the workload of the thread pool per stage gives insights about the overall load factor. In case of an overload situation, a server can adjust the scheduling parameters or thread pool sizes.
- From a concurrency perspective, SEDA represents a **hybrid approach between thread-per-connection multithreading and event-based concurrency**. Having a thread (or a thread pool) dequeuing and processing elements resembles an event-driven approach. The usage of multiple stages with independent threads effectively utilize multiple CPUs or cores and tends to a multi-threaded environment.
- The drawbacks of SEDA are the increased latencies due to queue and stage traversal even in case of minimal load. A better solution is to group multiple stages together with a common thread pool. This decreases context switches and improves response times.
- Eg: Apache MINA, Mule ESB

- **The Sweet Spot**
  - Adya et al.[7] proposed the **separation of management concepts** to argue purposefully for the most convenient form of concurrent programming, aside from the **coarse thread vs. event** debate. They pay special attention to the first two management principles.
  - While traditional event-based systems are mostly based on cooperative task management and manual stack management requiring stack ripping,

thread-based systems often use preemptive task management and automatic stack management.
    - Eventually, they favor a model that makes use of cooperative task management, but releases the developer from the burden of stack management. Such a model eases concurrency reflections, requires minimal conflict management and harmonizes with both I/O management models.



<u>Conclusion:</u>

- The two styles of system design are duals of each other.
- A program written in one model can be mapped directly to an equivalent program based on the other model.
    - The transformation is direct, i.e, on the programs themselves, exchanging the primitive operations and data structures of one style for those of the other.
- As a result of this mapping, the logic of the programs in the dual systems is invariant, although they use diverging concepts and provide a different syntax.
- The performance of the system can be preserved across the mapping.

<u>References</u>

[1] Hugh C. Lauer, Roger M. Needham. "**On the Duality of Operating System Structures.**" in *Proceedings of the Second International Symposium on Operating Systems*, IRIA, October 1978, reprinted in Operating Systems Review, 13, 2, pp. 3-19. April 1979.

[2] B. J. Stroustrup, "**On unifying module interfaces**," *Operating System Review*, 12, 1, pp. 90-98, January 1978.

[3] R. M. Needham and R. D. H. Walker, "**The Cambridge CAP Computer and its protection system**," *Proceedings of the Sixth Symposium on Operating System Principles*, Purdue University, Lafayette, Indiana, November 1977.

[4] VON BEHREN, Rob; CONDIT, Jeremy BREWER, Eric: **Why events are a bad idea (for high-concurrency servers)**, : *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, USENIX Association, Berkeley, CA, USA, 4-4

[5] OUSTERHOUT, John: **Why Threads are a Bad Idea (for most purposes)**, : *USENIX Winter Technical Conference*

[6] GUSTAFSSON, Andreas: **Threads without the Pain**. *Queue* (2005), 3: 34-41

[7] ADYA, Atul; HOWELL, Jon; THEIMER, Marvin; BOLOSKY, William J. DOUCEUR, John R.: **Cooperative Task Management Without Manual Stack Management**, : *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, 289-302

[8] LEE, Edward A.: **The Problem with Threads**. *Computer* (2006), 39: 33-42

[9] WELSH, Matt; CULLER, David BREWER, Eric: **SEDA: an architecture for well-conditioned, scalable internet services,** : *Proceedings of the eighteenth ACM symposium on Operating systems principles,* SOSP '01, ACM, New York, NY, USA, 230-243

[10] https://en.wikipedia.org/wiki/Event-driven_architecture

[11] https://www.queery.io/paper_response/2018/12/26/duality.html

[12] https://berb.github.io/diploma-thesis/original/043_threadsevents.html

[13] https://strongloop.com/strongblog/node-js-is-faster-than-java/

[14] https://berb.github.io/diploma-thesis/original/042_serverarch.html

[15] https://en.wikipedia.org/wiki/Boolean_algebra#Duality_principle

[16] https://en.wikipedia.org/wiki/CAP_computer