Tabla de contenido

[1 RUST 3](#_Toc50301452)

[1.1 Ownership o Propiedad 3](#_Toc50301453)

[1.1.1 Valores duplicables y Copy 4](#_Toc50301454)

[1.2 Borrowing o Préstamo 4](#_Toc50301455)

[1.2.1 Referencia Mutable 4](#_Toc50301456)

[1.2.2 Referencia Compartida 5](#_Toc50301457)

[1.3 Lifetimes o Tiempos de vida 5](#_Toc50301458)

[1.4 Unsafe 6](#_Toc50301459)

[2 SIMULACIÓN 8](#_Toc50301460)

[2.1 Conceptos de Simulación Computacional 8](#_Toc50301461)

[2.2 Ventajas 8](#_Toc50301462)

[2.3 Desventajas 9](#_Toc50301463)

[2.4 Áreas de Aplicación 10](#_Toc50301464)

[2.5 Clasificación de modelos de simulación 10](#_Toc50301465)

[2.5.1 Estáticos y dinámicos 10](#_Toc50301466)

[2.5.2 Determinísticos y estocásticos 10](#_Toc50301467)

[2.5.3 Continuos y discretos 10](#_Toc50301468)

[3 SIMULACIÓN POR EVENTOS DISCRETOS 12](#_Toc50301469)

[3.1 Representación de Tiempo. 12](#_Toc50301470)

[3.2 Estructuras 12](#_Toc50301471)

[3.3 Distribuciones Probabilísticas 12](#_Toc50301472)

[4 SIMULACIÓN DISTRIBUIDA Y PARALELA DE EVENTOS DISCRETOS 13](#_Toc50301473)

[4.1 Protocolos de simulación paralela 13](#_Toc50301474)

[4.1.1 Algoritmos conservativos de simulación 13](#_Toc50301475)

[4.1.2 Algoritmos optimistas de simulación 13](#_Toc50301476)

[4.1.3 Algoritmos aproximados 13](#_Toc50301477)

[5 PROCESOS, HILOS Y CORRUTINAS 14](#_Toc50301478)

[5.1 Procesos 14](#_Toc50301479)

[5.2 Hilos 14](#_Toc50301480)

[5.3 Corrutinas 14](#_Toc50301481)

[5.4 Comparativa 15](#_Toc50301482)

# RUST

Rust es un lenguaje de programación de sistemas joven que tiene como objetivo llenar la brecha entre los lenguajes de alto nivel, que brinda sólidas garantías estáticas como la seguridad de la memoria y subprocesos o hilos, y los lenguajes de bajo nivel, que brindan al programador un control detallado sobre el diseño de datos y la administración de memoria [4].

El principio clave del sistema de tipos de Rust es el siguiente principio de exclusión, los datos se pueden mutar exclusivamente a través de un puntero único o se pueden compartir de manera inmutable entre muchas partes, pero no ambas al mismo tiempo [4]. Este comportamiento se puede definir en los conceptos de “Ownership” y “Borrowing” (Propiedad y Préstamo) los cuales son apoyados por el concepto de “Lifetimes” (Tiempos de vida).

Este comportamiento se puede definir en las siguientes reglas conocidas como “Ownership” [5].

## Ownership o Propiedad

Las reglas de Propiedad se pueden definir en las siguientes [5].

* Cada valor en Rust tiene una variable que es llamada su dueño.
* Puede solo haber un dueño en un momento dado.
* Cuando el dueño termina su “scope” el valor será eliminado.

fn main(){

{ // s no es válido aquí, aun no ha sido declarado

let s = "hello"; // s es válido desde este punto en adelante

// hacer algo con s

} // Esta scope se ha terminado, y s ya no es valido

Figura .1 Ejemplo de alcance en Rust [5]

En Rust el comportamiento por defecto a la hora de realizar asignaciones es que una variable le entrega la propiedad de su valor a otra variable dicha entrega es llamada un movimiento (el valor de una variable es movido a otra variable), por ejemplo, tenemos una variable ‘a’ la cual es dueña de un valor y se crea una nueva variable ‘b’ a la cual le asignamos ‘a’ de manera que *let b = a;* causa que ‘b’ sea el nuevo dueño del valor de ‘a’ y ‘a’ deja de existir causando que cualquier acceso que se realice hacia ‘a’ sea detectado en tiempo de compilación gracias al análisis estático.

La propiedad de valor exclusiva es un mecanismo simple para asegurar libertad de carrera de datos. Sin embargo, es también bastante restrictiva [4]. Cuando se desea ver o modificar los valores de una variable sin necesariamente tomar la propiedad del valor, Rust provee el mecanismo llamado referencias aplicando el concepto de “Borrowing” o Préstamo. el cual permite obtener una vista exclusiva o compartida del valor de la variable sin necesariamente transferir la propiedad del valor.

### Valores duplicables y Copy

En Rust las propiedades de un tipo tales como “duplicable” son expresadas usando *marker traits*. Los Trait son una forma de typeclasses y son el mecanismo primario de abstracción del lenguaje [4]. Un Trait o Rasgo es como Rust define su concepto de Interfaz, es decir, un Trait define métodos que un tipo puede utilizar si este decide implementarlo. Un tipo puede implementar un Trait usando la palabra clave *impl* de la forma *impl Trait for Type* y definiendo los métodos del trait. Un tipo especial de trait es el “marker trait” el cual no posee operaciones que los tipos puede realizar, en cambio, simplemente clasifica los tipos en aquellos que implementan el trait y aquellos que no [4]. Un caso de este es el trait Copy el cual no posee métodos y puede ser implementado por todos los tipos que implementen Clone, la capacidad de crear una copia profunda de su data y entregar dicha copia. Previamente se especifico que los valores por defecto son movidos de una variable a otra, aquellos tipos que implementen Copy cambian este defecto a una copia profunda del valor no causando así el movimiento del valor.

## Borrowing o Préstamo

Las reglas de préstamo se pueden resumir en las siguientes [5]:

* En cualquier momento dado, solo puede existir una referencia mutable o cualquier número de referencias inmutables.
* Referencias siempre deben ser válidas.

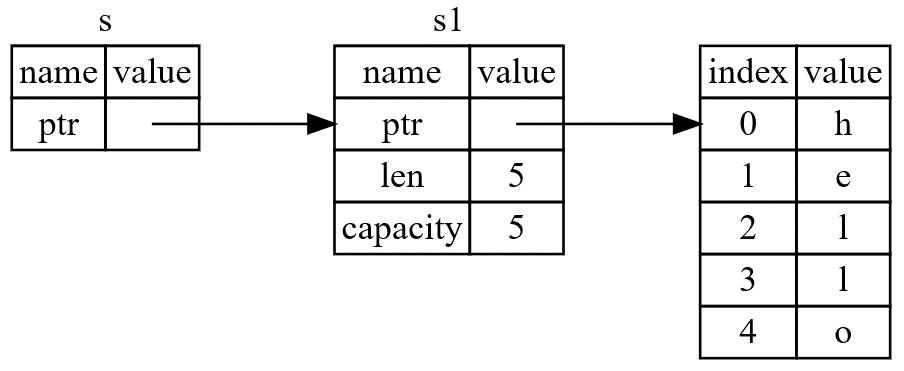


Figura 2 Diagrama de un &String s apuntando a un String s1 [5]

### Referencia Mutable

Una referencia mutable permite acceso exclusivo temporal al valor referenciado. otra forma de explicar una referencia mutable es definirla como un puntero único [4]. Significando que no puede haber otra parte en el programa que tenga acceso al dato referenciado mientras la referencia siga valida, incluyendo al dueño original del dato. Las variables en Rust son inmutables por defecto, para poder crear una referencia mutable a una variable es necesario que dicha variable halla sido declarada de forma mutable con la palabra clave *mut*. Ejemplo: *let mut variable = … ;*

### Referencia Compartida

Una referencia compartida permite acceso compartido temporal al valor referenciado. Tambien conocida como referencia inmutable debido a que, contrario a la referencia mutable, este tipo de referencia no permite modificar el contenido del valor referenciado, pero en cambio, permite la vista al dato a varias partes del programa.

## Lifetimes o Tiempos de vida

Toda referencia en Rust tiene un tiempo de vida asociado, el cual es su alcance para el cual la referencia valida [5].

Los punteros flotantes son un problema de memoria que plagan los programas de lenguajes de bajo nivel como C o C++. Rust en su esfuerzo para eliminar este tipo de problemas implementa el concepto de tiempos de vida el cual es aplicado a todas las referencias que existan en el programa. Previamente se explica que la referencia mutable y compartida entregan un acceso temporal al valor referenciado y los tiempos de vida son utilizados para determinar el alcance de ese temporal.

Para asegurar que las referencias sean utilizadas correctamente el compilador aplica las siguientes dos restricciones [4].

* La referencia solo puede ser utilizada mientras su tiempo de vida sea valido
* El referente original no puede ser utilizado de ninguna manera (para referencias mutables) o no puede ser modificado (para referencias compartidas) hasta que la vida de las referencias creadas haya expirado.

Manteniendo un único dueño y no permitiendo la mutación con el múltiple acceso, Rust efectivamente elimina varios tipos de problemas con el manejo de memoria ya que un programa que presente estos problemas no podrá compilar en primer lugar. Este beneficio se extiende a la programación paralela y concurrente causando que, si bien el tiempo y esfuerzo en lograr un programa funcionar será mucho mayor, la certeza de su buen funcionamiento también lo será.

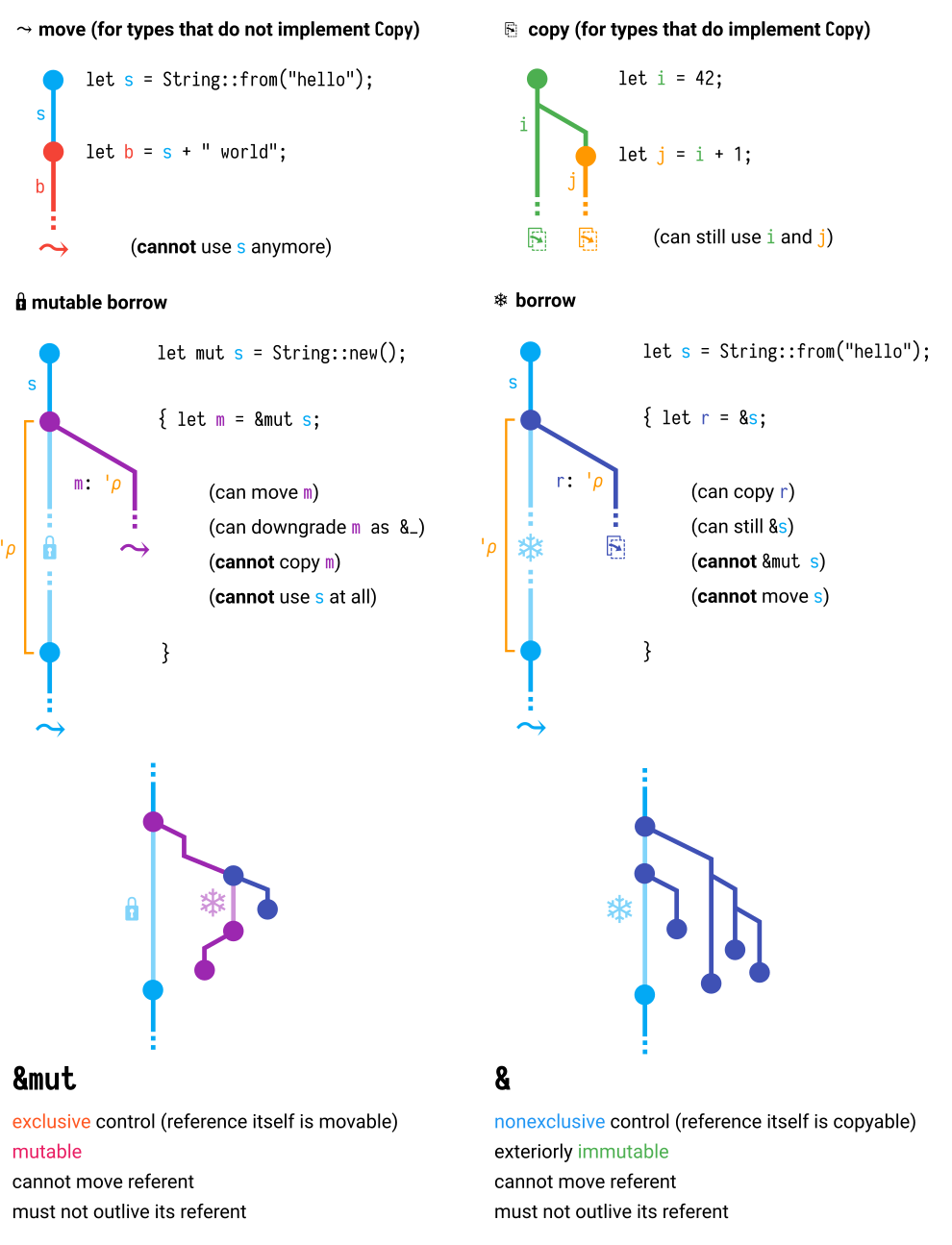


Figura 2.1. Ownership, Borrowing y Lifetimes en Rust [6]

## Unsafe

Hay veces donde todas estas reglas resultan ser muy estrictas, tal que terminan rechazando un programa totalmente seguro debido a que el compilador no puede determinar que lo es, pero el programador puede asegurar su correctitud. En tales casos existe la palabra clave “*unsafe”* el cual da acceso solo a las siguientes funcionalidades [8].

* De-referenciar raw pointers.
* Llamar funciones marcadas como unsafe (incluyendo funciones del lenguaje C, intrínsecos del compilador y el asignador en bruto).
* Implementar Traits marcados como unsafe.
* Mutar (modificar) estáticos.
* Acceder al campo de ‘unión’.

Cuando un programador desea tener un control mas detallado de la memoria pero que el compilador no puede asegurar su correcto uso, se puede utilizar unsafe el cual da acceso a extra funcionalidades no seguras que proveen un mayor control y riesgo. Unsafe marca el alcance sintáctico en el cual operaciones peligrosas pueden ocurrir [4].

Rust como lenguaje se puede pensar como la combinación de dos lenguajes Safe Rust y Unsafe Rust los cuales son iguales solo que unsafe rust tiene ciertas funcionalidades extras que lo hacen no tan seguro. Uno se podrá preguntar como es que unsafe no viola las promesas de seguridad que clama ofrecer Rust y la respuesta esta en el poder de las *“safe abstractions”* o abstracciones seguras [4]. En donde la sección de código unsafe se realizan las operaciones peligrosas y después estas se abstraen en funciones seguras las cuales realizan todo tipo de chequeos para asegurar su correcto uso, en este sentido, el comportamiento peligroso se encapsula y se ofrece una interfaz segura a la implementación insegura.

# SIMULACIÓN

“La simulación es la reproducción del comportamiento dinámico de un sistema de interés con el objetivo de obtener conclusiones que puedan ser aplicadas al sistema” [2]. La gran capacidad predictiva y rapidez de estas simulaciones incentivan su uso en la toma de decisiones. Cuando uno habla de simulación normalmente se refiere a la simulación computacional, es decir, un programa computacional encargado de imitar el funcionamiento de un sistema o proceso del mundo real.

## Conceptos de Simulación Computacional

La simulación está compuesta de varios conceptos conocidos como sistema y modelo, variables de estado del sistema, entidades y atributos, lista de procesamiento, actividades y retrasos [1].

Un sistema es una entidad natural o artificial, real o abstracta que también puede ser vista como una lista ordenada de objetos relacionados que evolucionan a través de diferentes actividades. Un modelo es la representación actual de un sistema (abstracta y entendible) y se utiliza para entender mejor el sistema [1] [2].

Sin mucho detalle una simulación se puede conformar con los siguientes conceptos:

* Sistema: Objeto de estudio, conjunto de objetos que interactúan y se desea conocer mejor.
* Modelo: Representación del sistema con suficiente detalle para su posterior imitación.
* Evento: Actividad que causa un cambio en las variables de estado de la simulación.
* Variables de estado del sistema: Toda la información necesaria para definir que ocurre en el sistema con suficiente nivel (para conseguir una salida de datos deseada).
* Entidades: Un objeto que interactúa con el resto del sistema ya sea de manera dinámica o estática, es decir, se mueve en el sistema u ofrece servicio a otras entidades.
* Recursos: Es una entidad que provee servicios a entidades dinámicas, estas solicitan recursos y los ocupan por periodos de tiempo, si no pueden acceder pueden esperar a que estén disponible o realizan otra acción.

## Ventajas

La simulación computacional posee varias ventajas tales como [1]

* *Elegir correctamente.* Permitir testear un aspecto propuesto sin comprometer recursos para su adquisición, es decir, analizar diseños y decisiones sin preocuparse por los costos de instalación.
* *Modificar el tiempo.* Permitir disminuir o acelerar el paso del tiempo para así poder estudiar un fenómeno de manera profunda.
* *Entender por qué.* Es posible determinar el “por qué” ocurre un fenómeno estudiando a fondo lo sucedido en la simulación, como interactúan las variables, que eventos ocurren, entre otros análisis.
* *Diagnosticar problemas.* Si las interacciones en un sistema son muy complejas la simulación permite entender la relación entre las variables del sistema para determinar su efecto en la ejecución general del sistema.
* *Identificar limitaciones.* Es posible realizar análisis a simulaciones con fines específicos para identificar limitaciones del sistema. Por ejemplo, rendimiento en pruebas de estrés.
* *Entendimiento de desarrollo.* Estudios de simulación ayudan en proveer entendimiento acerca de como realmente funciona un sistema mas que predicciones de como alguien piensa como un sistema opera. Proveen un punto de vista más objetivo permitiendo unificar el proceso de desarrollo.
* *Construir consenso.* Usar simulación para para presentar cambios de diseño proporciona una opinión objetiva.
* *Preparación para el cambio.* El futuro es incierto, la simulación permite responder todas las preguntas “y que pasaría si” que a uno se le ocurra lo cual resulta útil para tanto diseñar nuevos sistemas
* *Entrenar al equipo.* Los modelos de simulación pueden proveer excelente entrenamiento cuando son diseñados para ese propósito.
* *Especificar requerimientos.* Es posible que la especificación para un particular tipo de sistema complejo logre una meta sea desconocido. Al Estimular diferentes capacidades del sistema en la simulación los requerimientos pueden ser establecidos.

## Desventajas

Mientras que entre las desventajas se encuentran [1]

* *La construcción de modelos requiere entrenamiento especial.* Es un arte que se aprende con el tiempo y experiencia.
* *Los resultados de la simulación pueden ser difíciles de interpretar.* Debido a que la mayoría de los resultados de una simulación son esencialmente variables aleatorias (que usualmente se basan en entradas aleatorias), puede resultar difícil determinar si una observación es un resultado de las interrelaciones del sistema o aleatoriedad.
* *El modelamiento de simulaciones y análisis pueden llevar mucho tiempo y ser costoso.* Ahorrar recursos en el desarrollo del modelo y análisis pueden resultar en un modelo de simulación y/o análisis que no es suficiente para la labor.
* *La simulación puede ser utilizada inapropiadamente.* La simulación en algunos casos es usada cuando un análisis analítico es posible o incluso preferible.

## Áreas de Aplicación

La simulación ha sido utilizada como una potente herramienta en las siguientes áreas [1] [9].

* Entretenimiento
* Industria
* Predicciones meteorológicas
* Catástrofes naturales
* Aplicaciones militares
* Sistemas de salud
* Servicios Públicos
* Transporte
* Rendimiento de sistemas de computadora
* Sistemas de comunicación
* Recursos naturales

## Clasificación de modelos de simulación

Dependiendo de la naturaleza del sistema que se desee simular, los modelos resultantes pueden ser clasificados en las siguientes categorías [9] (Ver Fig. 3.1).

### Estáticos y dinámicos

Un sistema que no cambia su comportamiento respecto al tiempo o el tiempo no es un parámetro relevante en su evolución entonces es considerado estático, es decir, un modelo estático puede ser definido como la representación de un sistema en una fracción especifica de tiempo. En cambio, los modelos dinámicos cambian su comportamiento respecto al tiempo por lo que, un modelo de simulación dinámico representa la evolución de un sistema a través del tiempo.

### Determinísticos y estocásticos

Un sistema es considerado determinístico cuando no existe aleatoriedad o incertidumbre que afecten el comportamiento del sistema, es decir, dada una determinada entrada la salida siempre será la misma, en este sentido, los modelos determinísticos son como funciones puras en la programación o simplemente funciones en la matemática. Por el contrario, los sistemas estocásticos poseen un comportamiento probabilístico y existe algún elemento de aleatoriedad o incertidumbre. Este tipo de simulaciones incluyen de manera intrínseca perturbaciones aleatorias (normalmente siguiendo alguna distribución probabilística) que provocan estados futuros impredecibles.

### Continuos y discretos

Un modelo de simulación es considerado continuo si la evolución del sistema ocurre de forma ininterrumpida en el tiempo. Generalmente, los modelos basados en ecuaciones diferenciales describen comportamiento en función del tiempo. En cambio, un modelo de simulación es considerado discreto cuando la evolución del sistema ocurre en puntos distintos de tiempo simulado.

Vale aclarar que existen mas clasificaciones aparte de las definidas anteriormente y que es incluso posible combinar estas categorías, por ejemplo, crear un modelo que sea continuo y discreto a la vez, lo cual se podría decir que es también su propia categoría.

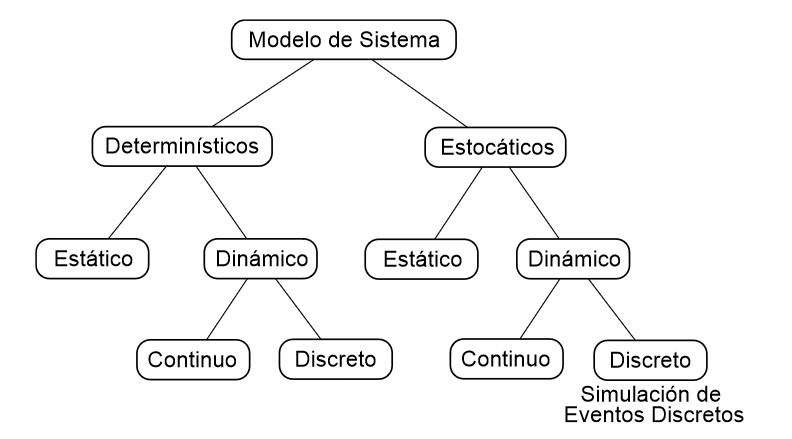


Figura 3.1. Clasificación de modelos de simulación computacional [9]

Aquella simulación que sea estocástica, dinámica y discreta se le conoce como simulación por eventos discretos.

# SIMULACIÓN POR EVENTOS DISCRETOS

La simulación de sistemas por eventos discretos o DES (del inglés Discretee Event Simulación) consiste en modelar sistemas donde el estado cambia únicamente en puntos discretos de tiempo, por ejemplo: 5 segundos después de iniciar la simulación, cada 12 segundos, después de que X evento ocurra. Estos modelos son “ejecutados” creando una historia artificial del sistema generada a partir de los supuestos del modelo donde se recopilan observaciones para analizarlas y estimar las verdaderas medidas de rendimiento del sistema. Esta ejecución se puede realizar de forma manual, una persona con su mente o con lápiz y papel sin embargo esta labor es normalmente realizada por una computadora. [Discrete-Event System Simulation Fourth Edition. John Carson Barry. Page 12]

En una DES las entidades son activadas por eventos y a su vez generan más eventos. Los eventos emitidos por las entidades tienen un tiempo asociado, igual o mayor al tiempo actual, estos eventos deben ser guardados ya que con ellos se calcula el siguiente estado de la simulación. Para ejecutar una simulación los eventos deben ser evaluados del mas cercano al tiempo actual de la simulación al mas lejano, para esta labor se introduce la Lista de Eventos Futuros o FEL (del inglés *Future Event List*).

## Ciclo de Vida

Una DES [Colocar una sección donde defino DES como Discretee Event List] blablablá ciclo de vida.

Ciclo de Vida de una simulación por eventos discretos:

1. Crear las entidades y recursos.
2. Insertar eventos asociados a las entidades en la FEL
3. Extraer el evento mas cercano al tiempo actual

El evento extraído puede causar varias acciones, para simplificar asumamos que el evento causa que otra entidad sea activada (incluyendo la que emitió el evento)

1. Aplicar los cambios de la entidad
2. Insertar el nuevo evento emitido por la entidad en el tiempo definido
3. Repetir paso 3 hasta que se cumpla la condición de termino. Ejemplos: FEL vacía, tiempo limite alcanzado, limite de eventos emitidos alcanzados.

## Enfoques

Las DES se dividen en 3 enfoques: Orientación a eventos – Orientación a actividades – Orientación a procesos. Este trabajo centrara su atención en la simulación por eventos discretos orientada a procesos.

# SIMULACIÓN ORIENTADA A PROCESOS

[ESCRIBIR SOBRE EL ENFOQUE ORIENTADO A PROCESO]

In the process-oriented approach, the simulation programmer composes a set of process descriptions. Each process description serves as a model of one kind of active entity in the simulated system. An active instance of a process description will be called a process. in a simulation system, there is a process management facility which allows processes to become active, to operate in the simulated environment, and to eventually terminate. The process management facility can manage many active processes so that they each appear to be active at the same time. This "pseudo parallelism" for simultaneously active processes is a very important feature of process-oriented simulation.

A real system is modeled, using the process-oriented approach, as collection of processes, each competing for the resources of the system.

Profe Alonso, [2022/09/30 17:51]

[ File : Introduction to Discrete Event Systems.pdf ]

página 609

Profe Alonso, [2022/09/30 17:51]

[ File : Discrete\_Event\_System\_Simulation\_Jerry\_Banks,\_John\_Carson,\_Barry.pdf ]

página 452

Profe Alonso, [2022/09/30 18:08]

[ File : 328885.328963.pdf ]

Este era el paper

# PROCESOS, HILOS Y CORRUTINAS

Un programa de computador es ejecutado en un proceso el cual mantiene su espacio de direcciones virtuales y la ruta de instrucciones a ejecutar. Por lo que se puede presentar un proceso como un poseedor de dos características: Propiedad de recursos y Planificación/ejecución Estas dos características pueden y son utilizadas para definir un proceso sin embargo estas características son sin embargo independientes y para distinguirlas se desarrolla el concepto de hilo o proceso ligero a la unidad que se activa y ejecuta instrucciones mientras que a la unidad de propiedad de recursos se le denomina proceso o tarea [3].

## Procesos

Siguiendo con la definición anterior aquella unidad encargada del control de los recursos, un programa es ejecutado en el contexto de un proceso y en este están presente los hilos que son utilizados para llevar a cabo la ejecución de las instrucciones del programa. Cambiar entre procesos es una actividad con un alto costo, asignar el espacio de memorias correspondiente y cambiar el contexto entre procesos no es como se consigue paralelismo y concurrencia en la actualidad.

## Hilos

Un hilo comparte la memoria del proceso al que pertenece con el resto de los hilos del proceso y todos los hilos son parte de un proceso, logrando que el cambio de contexto entre un hilo y otro sea una acción con un costo menor comparado al de cambiar de proceso, ya que comparten recursos no así los procesos que operan de forma independiente unos de otros, es por esto por lo que también se les conoce como procesos ligeros. Con el uso de estos hilos es posible lograr ejecutar instrucciones de forma paralela.

Agrupar varias instrucciones en un conjunto es una labor realizada a menudo en la programación, dichos conjuntos normalmente denominados rutinas o funciones son paralelizados al ser ejecutados al mismo tiempo en varios hilos, sin embargo, la cantidad de hilos que uno puede emplear está limitado al hardware que ejecute el programa es por lo que un nuevo concepto es introducido conocido como corrutinas o hilos verdes (“green-threads” en inglés).

## Corrutinas

La palabra corrutina está compuesta de dos palabras “*co*” (cooperativa) y “*rutinas*” (funciones). Una corrutina es conceptualmente diferente a un hilo, un programa con varios hilos es capaz de ejecutar varias instrucciones de forma paralela mientras que un programa con varias corrutinas es capaz de ejecutar dichas instrucciones de forma concurrente. Es decir, una y solo una corrutina está ejecutando en un momento dado y es capaz de pausar su ejecución para dejar a otra corrutina continuar la suya.

Los hilos son manejados por el sistema operativo mientras que las corrutinas son manejadas por el usuario, o más específicamente por el programa ejecutado por el usuario, al igual que los hilos existen en un solo proceso las corrutinas existen dentro de un hilo, aunque pueden ejecutarse en varios hilos (solo que no al mismo tiempo). Es posible combinar hilos y corrutinas para permitir la ejecución de instrucciones de forma paralela y concurrente. Como las corrutinas deben existir en el ámbito de un hilo existen diferentes estrategias empleadas en su creación y uso como el modelo 1:1 donde cada hilo es asignado una y solo una corrutina a ejecutar, el modelo N:1 donde todas las corrutinas son asignadas a un solo hilo y el modelo N:M donde las corrutinas son multiplexadas a los hilos, este es el modelo con mayor flexibilidad (relación paralelismo – concurrencia) comparado al resto pero es también el con mayor dificultad de implementar correctamente. De los tres modelos seguiremos elaborando en el modelo N:M debido a es el más flexible a la hora de proporcionar mecanismos de paralelismo y concurrencia. Además, las corrutinas son manejadas a nivel de usuario lo que permite a los programas que las usan un mayor control sobre el orden de la ejecución de sus instrucciones.

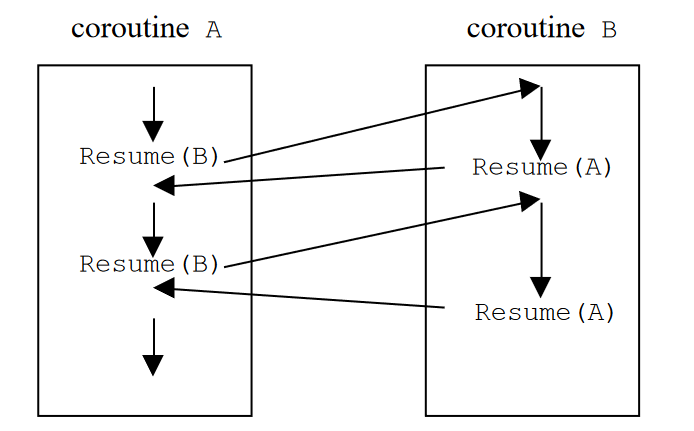


Figura 4. Cooperación entre dos corrutinas [7]

## Comparativa

Los procesos son la unidad encargada del manejo de los recursos y cambiar de procesos es una operación costosa debido a esto nace el concepto de hilo encargado de la ejecución de las instrucciones lo cual reduce el costo de forma significativa, sin embargo, existe un límite definido por el hardware de cuantos hilos es posible usar y por ello nace el concepto de corrutina o hilo verde el cual permite la concurrencia como una forma de optimizar la ejecución de un programa.

Tanto los procesos como los hilos son manejados por el sistema operativo y poseen una naturaleza preventiva, es decir, tienen prioridades y quien tenga la mayor prioridad será ejecutado aun si ya había otra unidad con menor prioridad ejecutándose. En cambio, las corrutinas son de naturaleza cooperativa, es decir una corrutina pausa su propia ejecución para permitirle a otra ejecutarse. En la perspectiva de costos, cambiar el contexto de procesos es el método con el mayor gasto general, los hilos tienen un menor gasto general en el cambio de contexto comparado a los procesos, pero en la actualidad tal costo sigue siendo significativo, las corrutinas por su parte son las que menor costo general provocan, sin embargo, no son capaces de proveer verdadero paralelismo sin la ayuda de los hilos, pero en cambio proveen la capacidad de la ejecución concurrente.

Los procesos e hilos son manejados por el sistema operativo a nivel de Kernel, el levantar un hilo implica realizar una llamada al sistema operativo para que este le asigne los recursos al hilo y nos lo entregue para su posterior uso, en cambio las corrutinas son manejadas por un “Runtime” definido en el nivel de usuario por lo que uno es capaz de decidir cómo controlar sus corrutinas son ejecutadas disminuyendo el contacto con el sistema operativo.

Referencias:

1. Handbook of Simulation
2. Wainer, G. (2009). Discrete-Event Modeling and Simulation: a practitioner approach. CRC Press.Taylor and Francis.
3. Sistemas Operativos - William Stallings página. 158
4. <https://people.mpi-sws.org/~jung/phd/thesis-screen.pdf>
5. <https://doc.rust-lang.org/stable/book/>
6. <https://www.educative.io/edpresso/what-is-a-coroutine>
7. Helsgaun, Keld.A Portable C++ Library for Coroutine Sequencing. 2001
8. <https://doc.rust-lang.org/stable/nomicon/index.html>
9. Solar, R. (2012). Particionamiento y Balance de Carga en Simulaciones Distribuidas de Bancos de Peces. Ph.D. thesis, Universitat Autònoma de Barcelona