



Sistemas Distribuidos I (75.74)

Multithreading y Multiprocessing

Conceptos. Mecanismos de sincronización. IPCs. Problemas clásicos. Paralelización de tareas

Docentes

- Pablo D. Roca
- Ezequiel Torres Feyuk

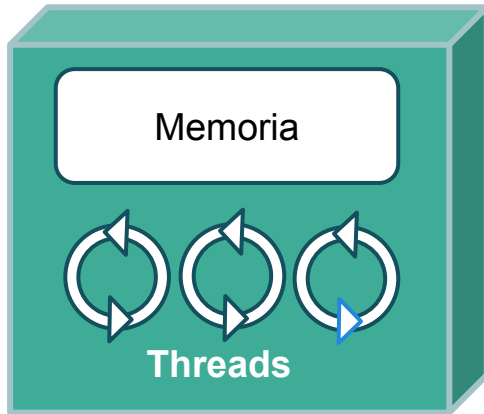


● Conceptos

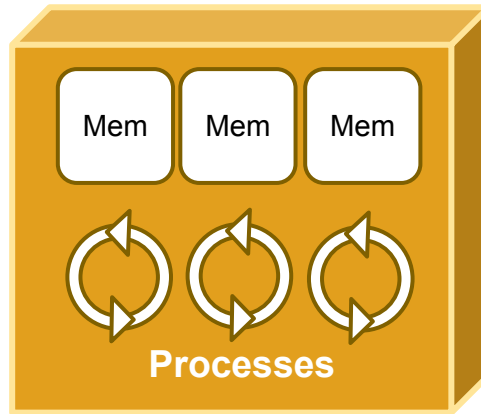
- Mecanismos de sincronización
- IPCs
- Problemas clásicos
- Paralelización de tareas



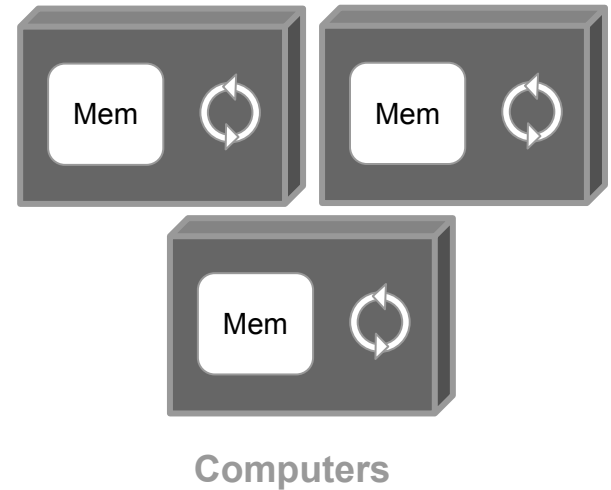
Multi-threading



Multi-processing



Multi-computing





Recursos Compartidos

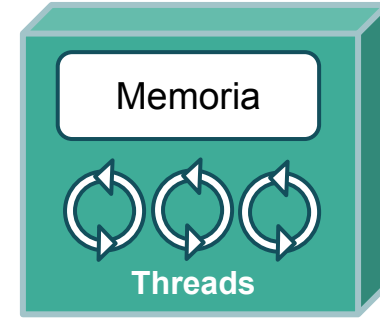
- Heap
- Data Segment
- File Descriptors
- Code Segment (read-only)

Sincronización

- Soporte threading del SO (pthread-mutex, etc)
- Soporte threading del *runtime* (threads Java, .Net, etc)
- Inter Process Communication (IPC)

Características clave

- Sencillo compartir información entre threads.
- Alto acoplamiento entre componentes del sistema.
- Escasa estabilidad => 1 thread defectuoso afecta todo el sistema.
- Escalabilidad muy limitada.



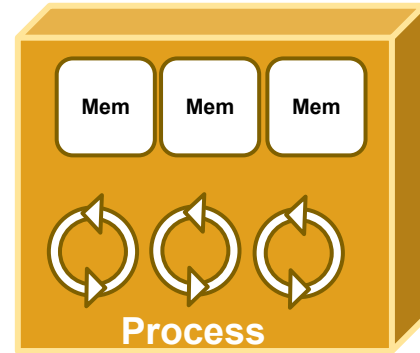


Recursos Compartidos

- Code Segment (read-only)

Sincronización

- InterProcess Communication (IPCs):
 - Signals
 - Pipes / Fifos
 - Queues
 - Shared Memory
 - Semáforos
 - Locks
 - Sockets



Características clave

- No es trivial compartir información entre procesos.
- Componentes separados, en general simples.
- Más escalable y más estable que multi-threading.
- Sin tolerancia a fallos de hardware, sistema operativo, etc.



Safety properties (siempre verdadera)

- Exclusión mutua
- Ausencia de deadlocks

Liveness properties (eventualmente verdadera)

- Ausencia de starvation
- Fairness





Asegurar estado *safety* de propiedades de un sistema se transforma en un pilar de la teoría de concurrencia.

Basada en Algoritmos

- Sin existencia de abstracciones especiales.
- Condiciones lógicas simples para asegurar el cumplimiento de cierta *Critical Section*.

Basada en Abstracciones

- Basada en abstracciones provistas por el SO.
- Permite construir mecanismos compuestos por combinaciones de las mismas.



Conceptos | Basados en Algoritmos

- **Busy-Waiting:** responsable de la mayoría de los problemas de performance en sistemas concurrentes.
- **Spin-lock:** caso más simple de Busy-Wait (`while (flag);`)
- **Algoritmos de espera:** Dekker, Lamport (del panadero), Peterson, etc.

Ej. Algoritmo de Peterson p/dos procesos:

```
bool flag[2] = {false, false};
int turn;
P0:    flag[0] = true;
        turn = 1;
        while (flag[1] && turn == 1);
        /* critical section code */
        flag[0] = false;
```

```
P1:    flag[1] = true;
        turn = 0;
        while (flag[0] && turn == 0) ;
        /* critical section code */
        flag[1] = false;
```




Conceptos | Basados en Algoritmos

- **Operaciones atómicas:** Mecanismos provistos por un lenguaje para actualizar variables/objetos sin utilizar mecanismos de sincronización
 - **Contadores atómicos de tipos POD (int, char, double, etc.)**
 - **CAS (Compare and Swap):** Operación por excelencia para actualizar contenedores de forma segura en ambientes multithreading.

```
CAS (Pseudocódigo):  
function cas(p : *int, old : int, new  
: int) returns bool {  
    if *p ≠ old {  
        return false  
    }  
    *p ← new  
    return true  
}
```

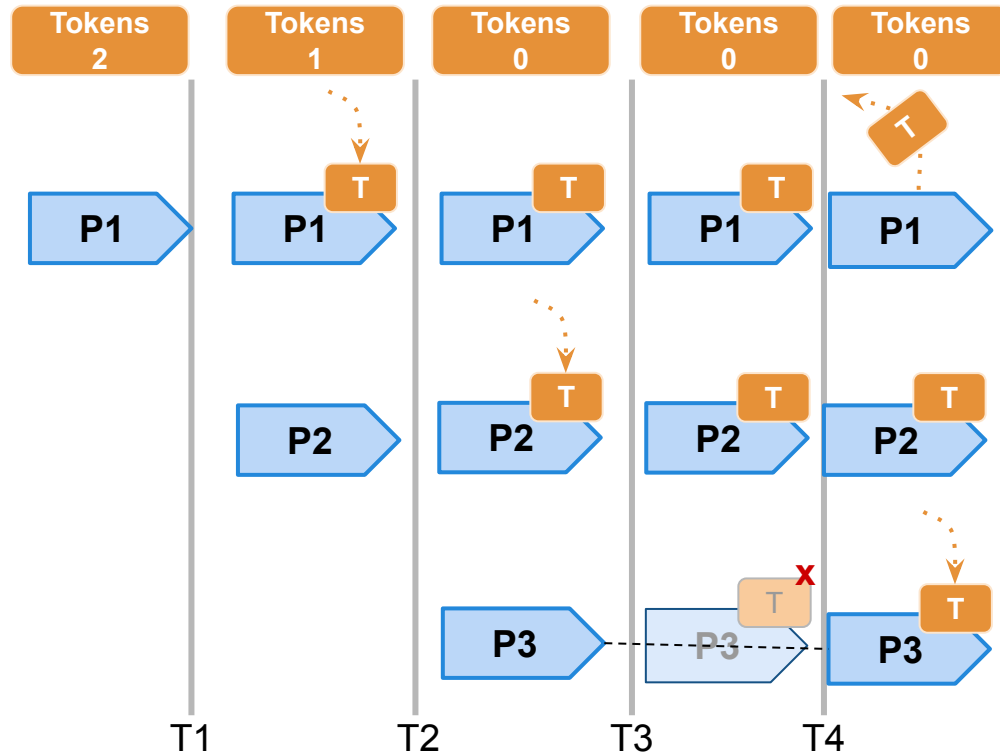
```
Ejemplo: CAS de objetos en Golang  
  
package atomic  
  
func CompareAndSwapUint32(  
    addr *uint32,  
    old uint32,  
    new uint32,  
    ) (swapped bool)
```



- Conceptos
- **Mecanismos de sincronización**
- IPCs
- Problemas clásicos
- Paralelización de tareas



Mecanismos de Sincronización | Semáforos





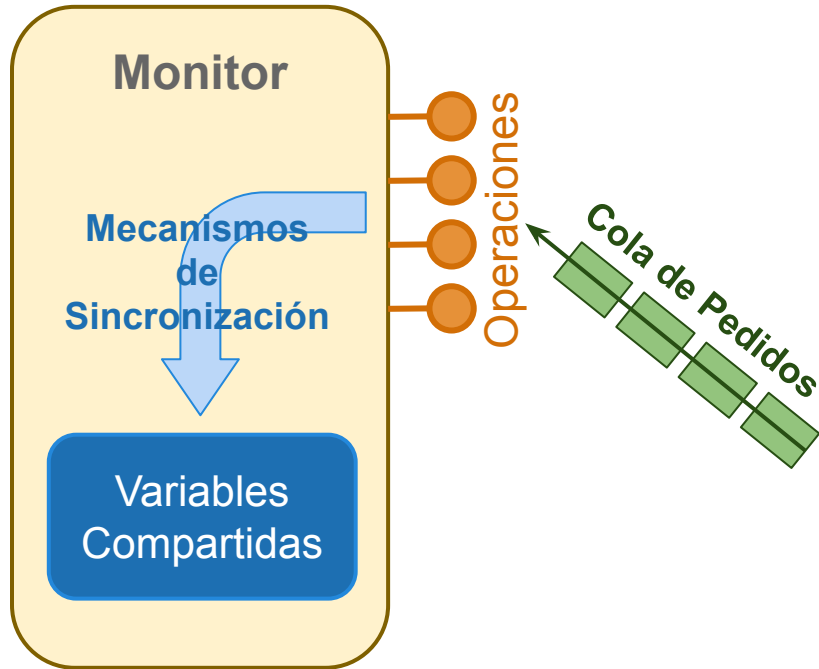
Mecanismos de Sincronización | Semáforos

- Variable entera utilizada para acceder a *recursos compartidos* (e.g. *Shared Mem*)
- El mismo queda definido por los valor que puede adoptar (e.g. $S = \{0,1,2\}$)
- Operaciones válidas:
 - signal (P): Incrementa el valor de S
 - wait (V): Decrementa el valor de S
- Mutex ($S = \{0,1\}$)
 - Utilizado para acceder a *secciones críticas*



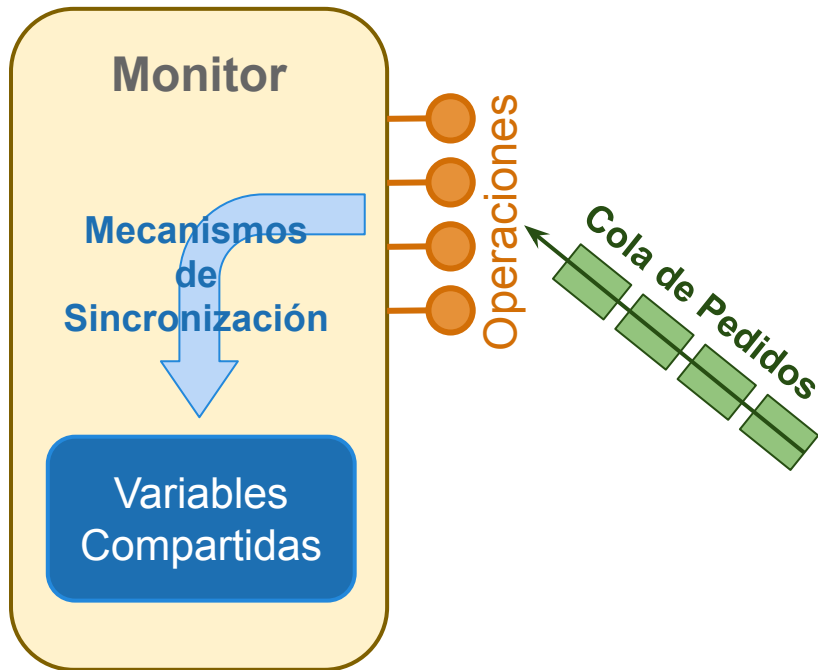


Mecanismos de Sincronización | Monitores





Mecanismos de Sincronización | Monitores



```
monitor class Account:
    private int balance = 0
    public method bool withdraw(int
amount):
    if balance < amount:
        return false
    else:
        balance = balance - amount
        return true

    public method deposit(int amount):
        balance := balance + amount
```



Mecanismos de Sincronización | Cond. Variables

- Ejemplo práctico de un monitor
- Mutex debe ser adquirido antes de realizar una operación
- Operaciones válidas:
 - **wait:** Bloquea al proceso hasta que otro proceso lo despierte
 - **notify / notify_all:** Despierta a *un proceso / todos los procesos* esperando que se cumpla una condición

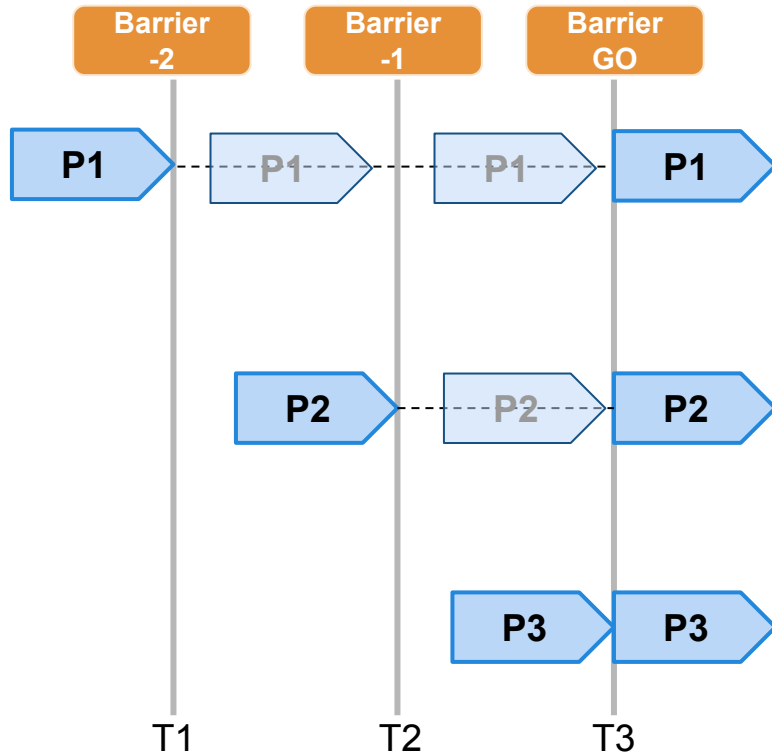
Proceso N°1

```
cv.acquire()
while not an_item_is_available():
    cv.wait()
get_an_available_item()
cv.release()
```

Proceso N°2

```
cv.acquire()
make_an_item_available()
cv.notify() / cv.notify_all()
cv.release()
```


Mecanismos de sincronización | Barrera



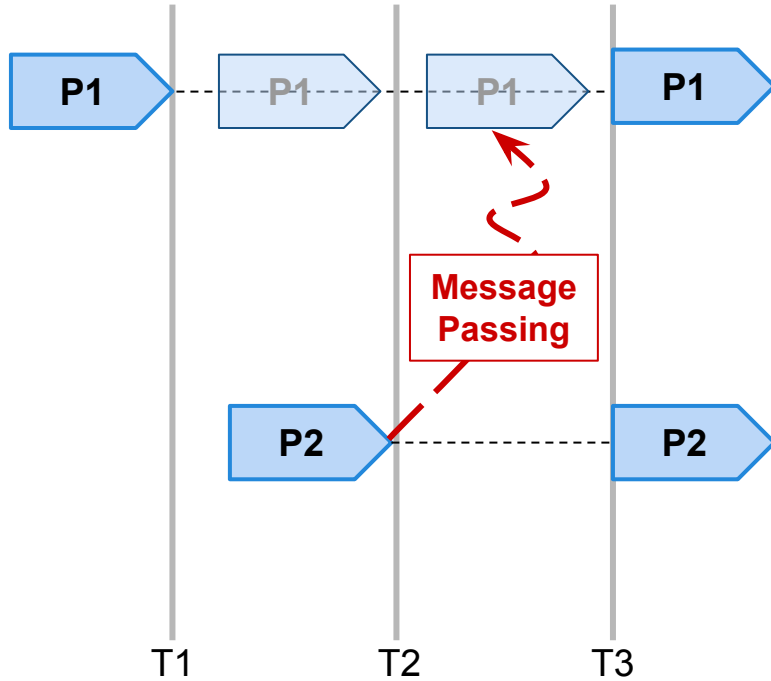


Ejercicio

- N *threads/procesos* deben ejecutar M tareas
- Cada *thread/proceso* ejecuta su tarea y espera a que sus pares terminen de hacer lo mismo
- Cuando todos los *threads/procesos* hayan terminado de ejecutar una ronda de tareas, proceden a ejecutar una nueva ronda



Mecanismos de sincronización | Rendezvous





Ejercicio

- N *threads/procesos* deben ejecutar M tareas
- Cada *thread/proceso* ejecuta su tarea y espera a que sus pares terminen de hacer lo mismo
- Cuando todos los *threads/procesos* hayan terminado de ejecutar una ronda de tareas, proceden a ejecutar una nueva ronda
- Resolver utilizando la abstracción **BlockingQueue**

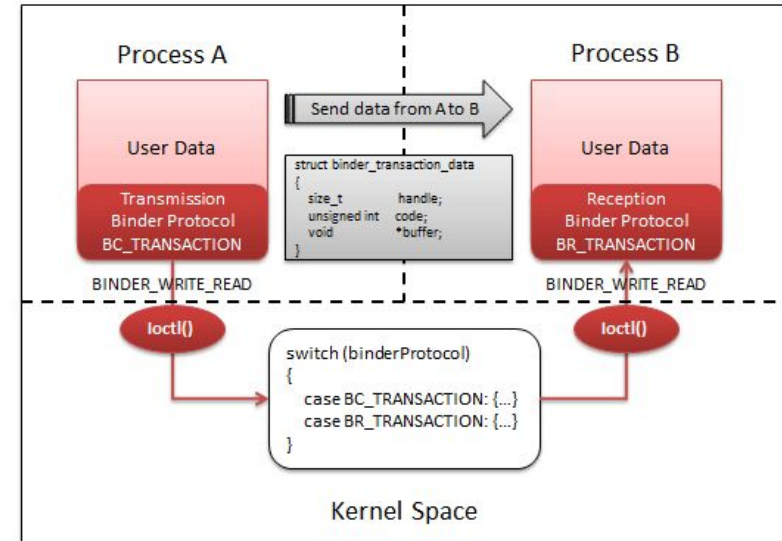
Agenda



- ☐ Conceptos
- ☐ Mecanismos de sincronización
- ☒ **IPCs**
- ☐ Problemas clásicos
- ☐ Paralelización de tareas



- Permiten la comunicación entre dos o más procesos
- Provistos por el SO
- Creación y destrucción exceden la vida del proceso
 - Usuario es responsable de la vida de los mismos
 - Proceso *Launcher* y *Terminator* para administrar la vida de los mismos
- Usualmente identificados por nombre
- En Linux todos los IPCs son vistos como diferentes **tipos de archivos**





<i>Mecanismo de sincronización</i>	<i>IPC</i>
<i>Semáforo</i>	<i>Semáforo</i>
<i>??</i>	<i>Shared Memory</i>
<i>Monitor</i>	<i>File Lock</i>
<i>Barrera</i>	<i>??</i>
<i>Rendezvous</i>	<i>Signal</i>
	<i>Queue</i>
	<i>Pipes / Fifos</i>
	<i>Sockets</i>



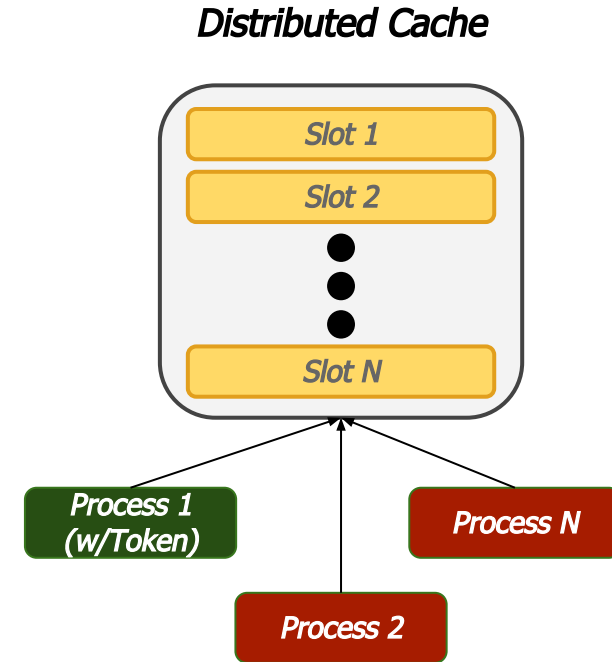
IPCs | Signals

- Existen 31 tipos distintos (kill -l)
- Cada proceso decide cuales handlear (Ej. [libcURL](#) y [SIGALRM](#))
- SIGSTOP y SIGKILL son la excepción
- Ejemplos de signals estándar
 - SIGINT y SIGTERM: Graceful Quit
 - SIGSEGV: Problemas en la memoria
 - SIGABRT: Code assertions
- Propagación de *signals* en *threads*. ([Masks setting](#))



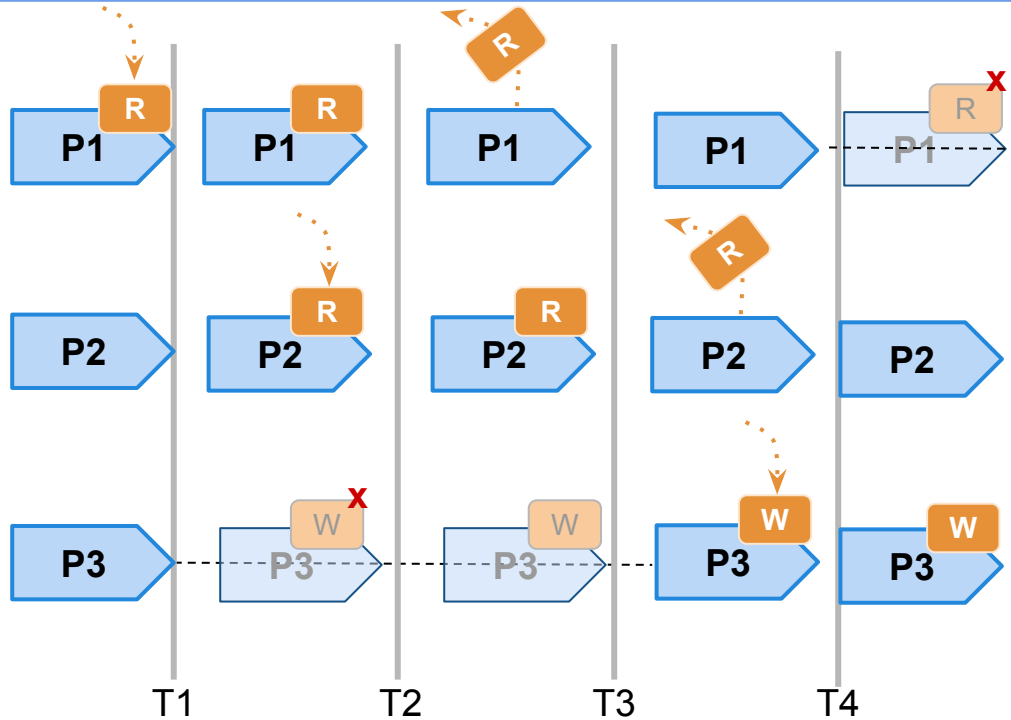


- Mecanismo provisto por el SO (Linux) para compartir recursos
- *Abstracción inexistente en threads*: heap entre dos threads de un mismo proceso es compartido
- Su tamaño se define al ser creada
- Mutex es necesario **solo si** dos procesos no pueden acceder a la memoria al mismo tiempo (e.g. shared counter)





IPCs | File Locks





IPCs | File Locks

- Control de acceso a un file descriptor

```
int flock(int fd, int operation);
```

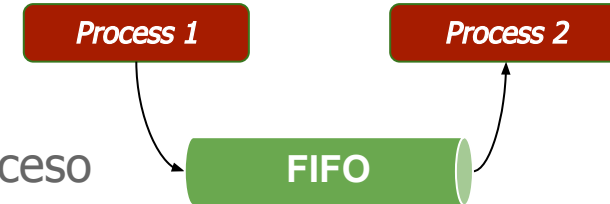
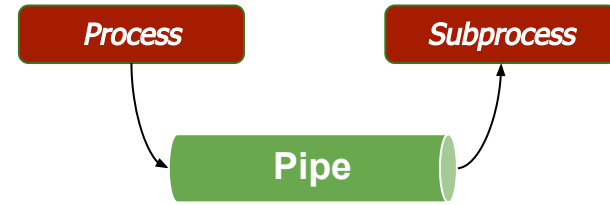
- Existen dos tipos:
 - *Shared lock (R)*: Read only lock.
Múltiples read locks permitidos
 - *Exclusive lock (W)*: RW lock.
Sólo un exclusive lock a la vez por File





IPCs | Pipes y Fifos

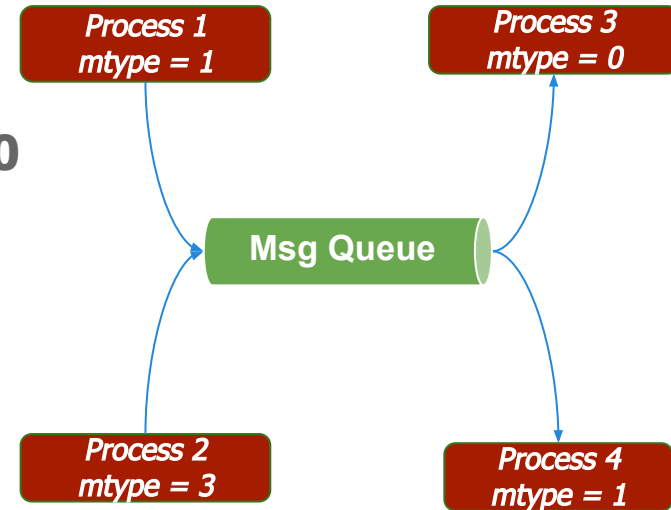
- Pasaje de información directa entre 2 procesos
- Linux: API de un archivo para la escritura/lectura
- Unnamed Pipes (Pipes)
 - Comunicación entre procesos padre e hijo
 - Dejan de existir al finalizar el proceso
- Named Pipes (FIFO)
 - Comunicación entre dos procesos cualesquiera
 - Viven en el SO por lo cual excede la vida del proceso
- ¿Cuál es el tamaño de un pipe?





IPCs | Message Queues (System V)

- Procesos escriben / reciben bloques de bytes
- Campo **mtype**
 - Identifica el tipo de mensaje
 - Sender debe enviar mensajes con **mtype** > 0
 - Receptor con **mtype** = 0 recibe mensajes sin importar el **mtype**
 - Caso esotérico: Receptor con **mtype** < 0
- Mensajes leídos son removidos de la cola
- Buffer size definido durante la creación



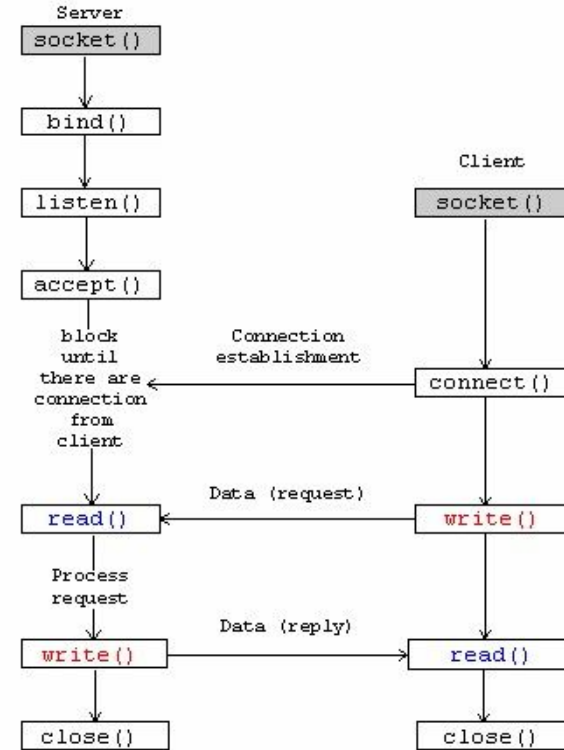


IPCs | Sockets

- Permite comunicar dos procesos a través de un canal de comunicación (endpoint)

```
int socket(int domain, int type, 0);
```

- Domain
 - AF_UNIX - Unix socket
 - AF_INET / AF_INET6 - Network Socket
- Type (Protocolos de comunicación)
 - SOCK_DGRAM => UDP
 - SOCK_STREAM => TCP
 - SOCK_RAW => ??



<https://stackoverflow.com/questions/27014955/socket-connect-vs-bind>



Mecanismos de sincronización | Rendezvous

Ejercicio

- N *threads/procesos* deben generar M mensajes
- Cada *thread/proceso* genera un mensaje y espera a que sus pares terminen de hacer lo mismo
- Cuando todos los *threads/procesos* hayan terminado de generar una ronda de mensajes procederán a generar uno nuevo
- **Resolver utilizando IPCs**
 - Shared Memory / Semaphores
 - Message Queues

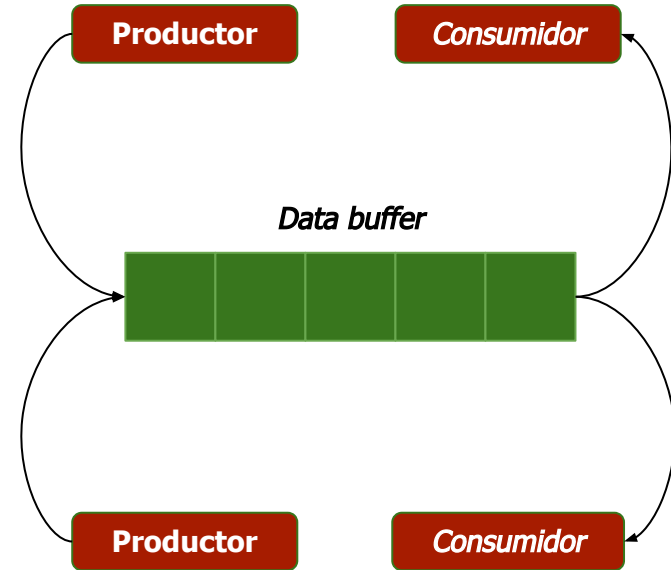


- Conceptos
- Mecanismos de sincronización
- IPCs
- **Problemas clásicos**
- Paralelización de tareas



Problemas clásicos | Productor Consumidor

- Productores agregan paquetes en el buffer
- Consumidores extraen paquetes del buffer
- Situaciones de bloqueo
 - Productor intenta agregar un paquete cuando el buffer está lleno
 - Consumidor intenta extraer un paquete cuando el buffer está vacío
- Acceso al buffer **debe** ser sincronizado
- El buffer es acotado o infinito?

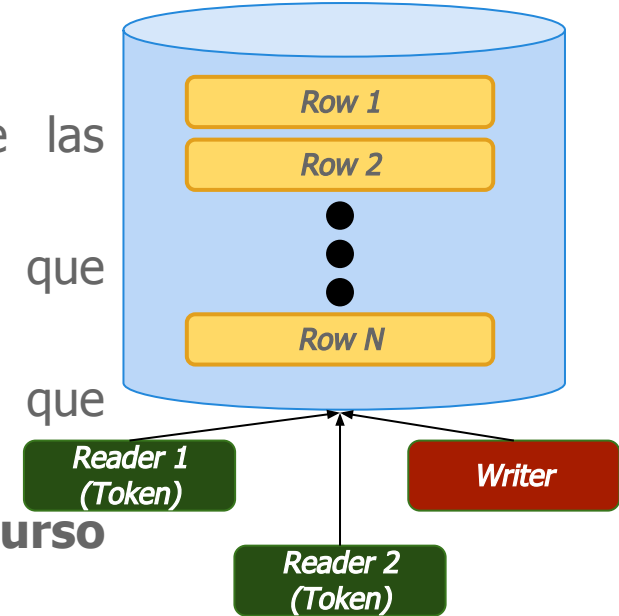




Problemas clásicos | Lectores Escritores

Database Access

- Procesos intentan acceder a una memoria compartida
- Dos tipos de procesos (Lectores y Escritores)
- Tipos de problemas definidos en función de las propiedades *fairness* y *starvation*
 - **Prioridad Lectores:** Escritores esperan a que lectores liberen recurso compartido
 - **Prioridad Escritores:** Lectores esperan a que Escritores liberen recurso compartido
 - Lectores y Escritores acceden a **recurso compartido por tiempo limitado**





- Barbero dormilón
- Filósofos comensales
- Fumadores de cigarrillos

Agenda



- Conceptos
- Mecanismos de sincronización
- IPCs
- Problemas clásicos
- **Paralelización de tareas**



Paralelización de tareas | Introducción

- Objetivos
 - Reducir el tiempo de cómputo de una tarea (**latencia**)
 - Incrementar la cantidad de tareas que se pueden realizar en paralelo (**throughput**)
 - Reducir la potencia consumida al realizar todas las tareas
- **Camino crítico**
 - Máxima longitud de tareas secuenciales a computar
 - Define el mejor rendimiento que se puede obtener al realizar un conjunto de tareas



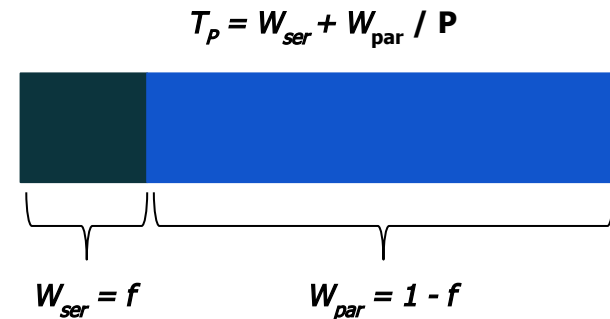
Paralelización de tareas | Ley de Amdahl

"...the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude..."

- Corolario:

$$T_p = W_{ser} + W_{par} / P \quad \text{con } P \text{ unidades de cómputo}$$

- **Speedup**
 - **Ratio de optimización** de una operación

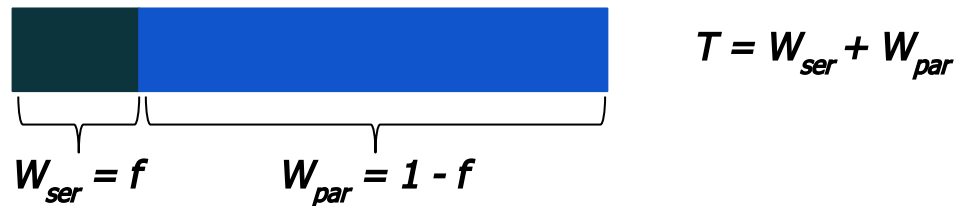




Paralelización de tareas | Ley de Amdahl

"...the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude..."

- Corolario: todo trabajo de cómputo se divide en fracciones Secuenciales y Paralelas:



- Luego, utilizando P unidades de cómputo, el tiempo de ejecución es:

$$T_p = W_{ser} + W_{par} / P$$



Paralelización de tareas | Ley de Amdahl

- Se define **Speedup** como el ratio de optimización de una operación:

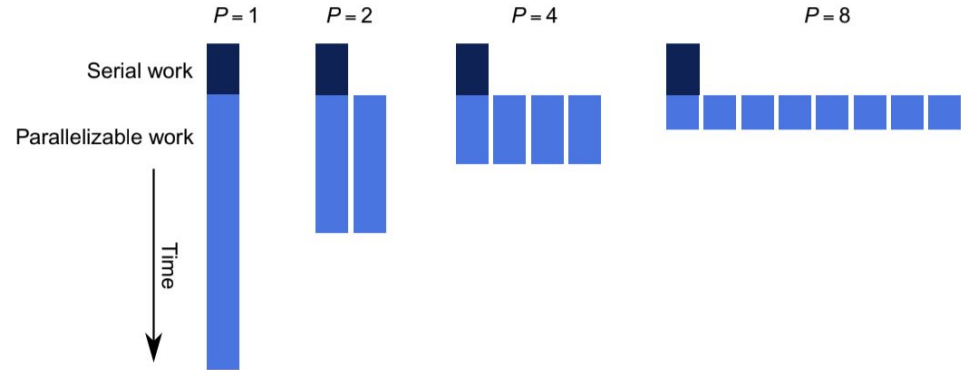
$$S_p = T_1 / T_p$$

- Reemplazando, se obtiene:

$$S_p \leq 1 / (f + (1 - f) / P)$$

- Pero con $P \rightarrow \infty$:

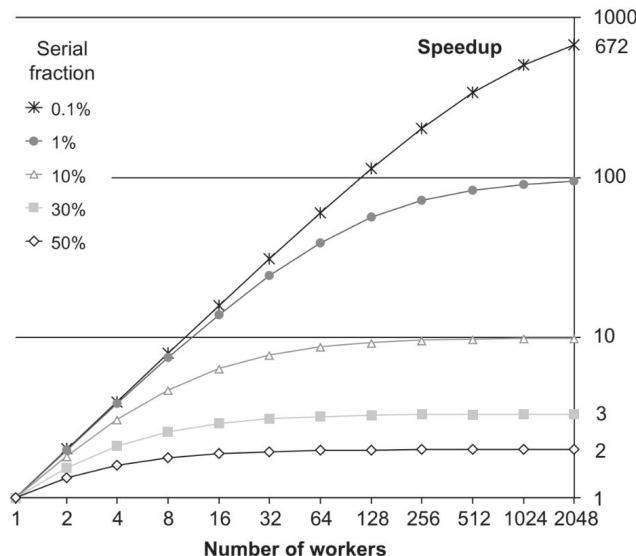
$$S_\infty \leq 1/f$$





Paralelización de tareas | Ley de Amdahl

- *Speedup* máximo se encuentra acotado **por la fracción de tiempo que no puede ser paralelizable**
- Fracción paralelizable distribuida **uniformemente** entre procesadores



Ejemplo:

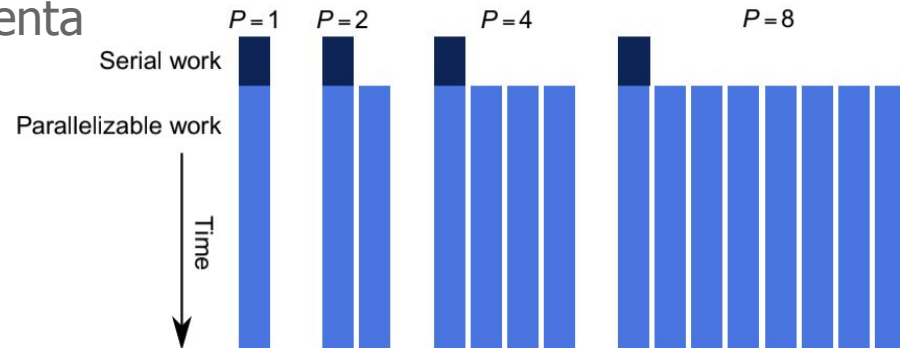
f	Processors	S_p
0.01	100	50
0.01	1000	90
0.01	10000	99
0.01	inf	100



Paralelización de tareas | Ley de Gustafson

*"Speedup should be measured by **scaling the problem** to the number of processors, **not by fixing the problem size**"*

- Corolario: aumentar el paralelismo puede permitir la modificación del problema original para ejecutar más trabajo
- Si el problema crece, caben dos alternativas:
Parte serial disminuye $\Rightarrow$ Speedup aumenta
Paralelismo aumenta $\Rightarrow$ Speedup aumenta





Paralelización de tareas | Modelo Work-Span

- **Características**

- Modelo más cercano a la realidad para estimar optimizaciones que el usado por Amdahl
- Provee una *cota inferior* y una *cota superior* para el *Speedup*

- **Hipótesis**

- *Paralelismo Imperfecto*: No todo el trabajo paralelizable se puede ejecutar al mismo tiempo
- *Greedy scheduling*: proceso disponible => tarea ejecutada
- Tiempo de acceso a memoria despreciable
- Tiempo de comunicación entre procesos despreciable



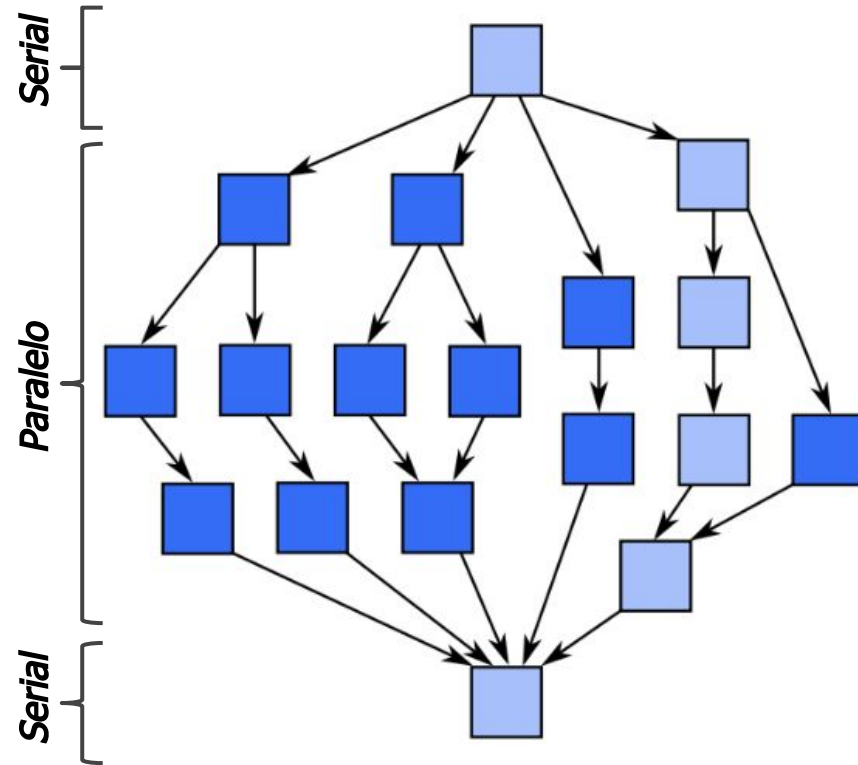
Paralelización de tareas | Modelo Work-Span

- Definiciones
 - $T_1(\text{work})$: Tiempo en ejecutar *operación/algorithm* con 1 sólo proceso
 - $T_{\text{inf}}(\text{span})$: Tiempo en ejecutar el **camino crítico** de la *operación/algorithm*

Cota	Speedup	Consideraciones
Superior	$\min(P, T_1/T_{\text{inf}})$	Se obtiene P en escenarios de Speedup lineal.
Inferior	$(T_1 - T_{\infty})/P + T_{\infty}$	El trabajo se puede dividir en perfecta e imperfectamente paralelizable



Paralelización de tareas | Modelo Work-Span

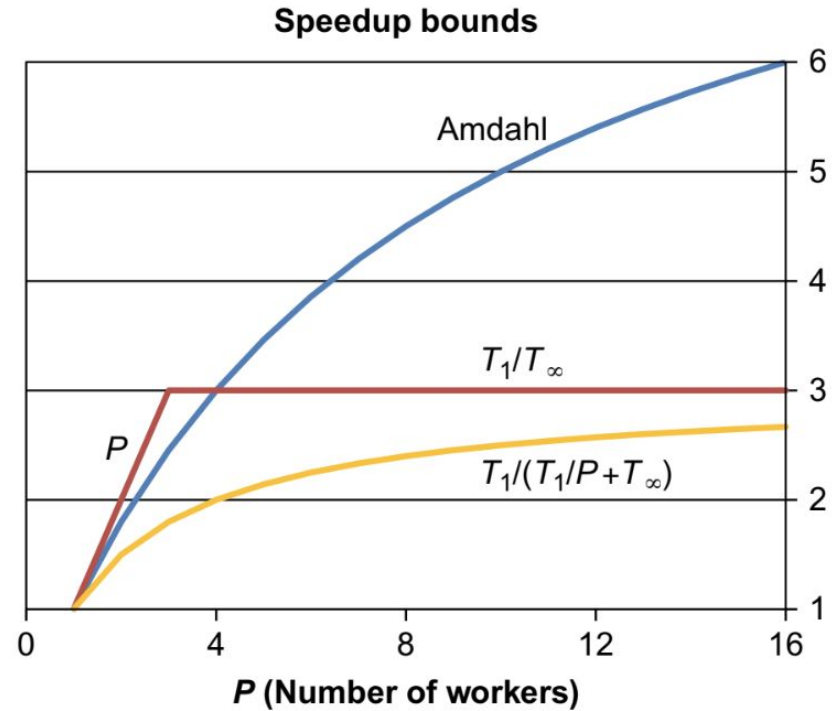
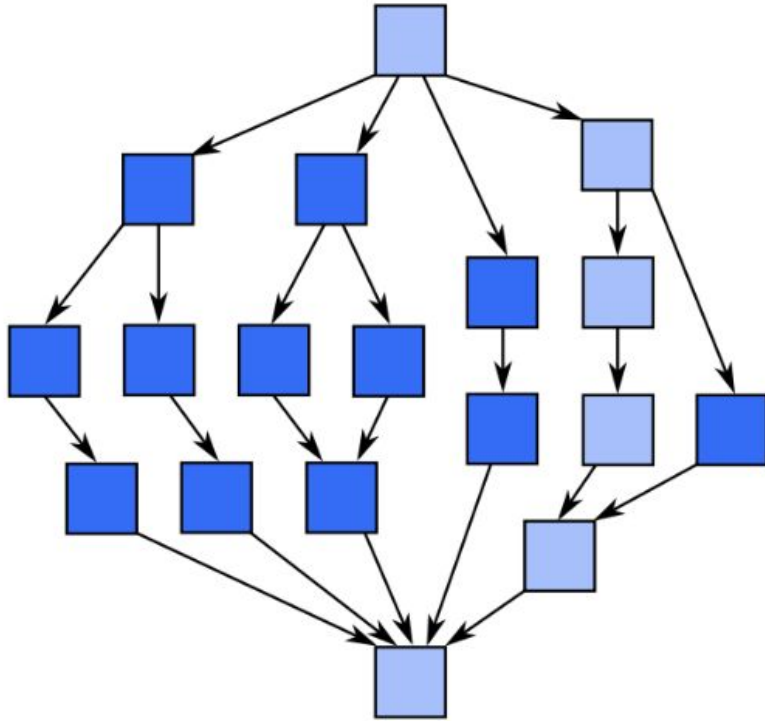


Ejemplo Práctico

- **Work-Span**
 - $T_1 = 18$ (work)
 - $T_{inf} = 6$ (span)
 - $S_p = T_1 / T_{inf} = \mathbf{3}$
- **Amdahl**
 - $f = 2/18 = 1/9$
 - $S_{inf} = 1/f = \mathbf{9}$



Paralelización de tareas | Modelo Work-Span





Estrategias de paralelización

- **Descomposición Funcional**

$\text{foo}(\text{data}) = f(\text{data}) + g(\text{data}) + h(\text{data})$ //1 proceso máximo

VS

$\text{foo}(\text{data}) = \text{go } f(\text{data}) + \text{go } g(\text{data}) + \text{go } h(\text{data})$ //3 procesos máx.

- **Particionamiento de Datos**

$\text{foo}(\text{data}) = f(\text{data})$ //1 proceso máx.

VS

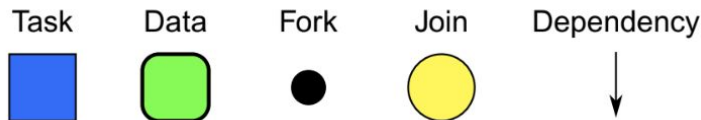
$\text{foo}(\text{data}) = \text{go } f(\text{data}[0:N/P-1]) \& \dots \& \text{go } f(\text{data}[(P-1)*N/P:N-1])$
//P procesos máx. (sólo si $f(x)$ es particionable)



Patrones de Procesamiento | Patrones de Procesamiento

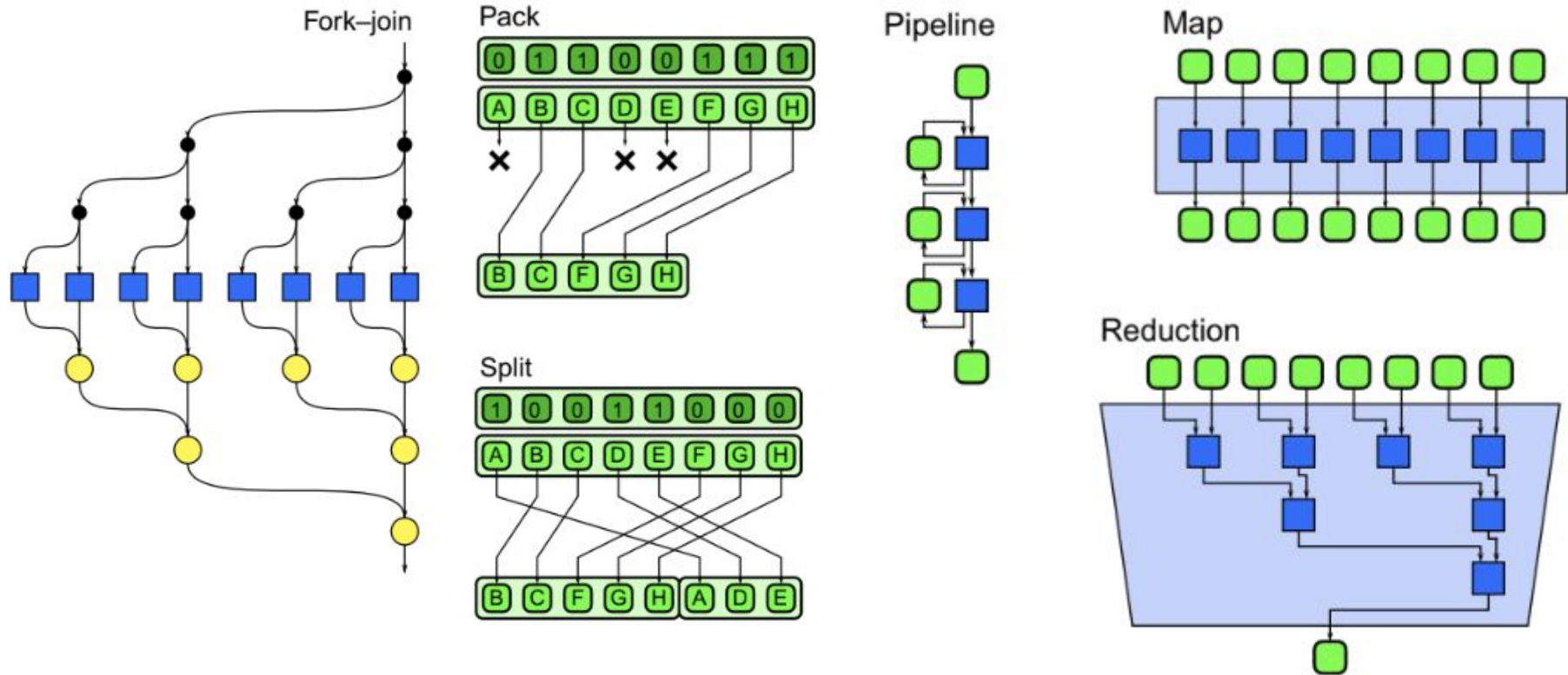
- Basados en algoritmos
 - No tan abstractos como *patrones de diseño*
 - No incluyen detalles de implementación
 - Agnósticos a lenguajes de programación
- Patrones deben poder incluir otros patrones (*nesting*)
- Herramientas básicas de trabajo también en *multi-computing*

Notación:





Paralelización de tareas | Patrones de Procesamiento





- McCool M., Robison A. D., Reinders J., Structured Parallel Programming Patterns for Efficient Computation, 2012, Elsevier-Morgan Kaufmann.
 - Capítulos 1 y 2