Programación Concurrente

Clase 1

Facultad de Informática
UNLP

Metodología del curso

- **Comunicación:** Plataforma IDEAS (ideas.info.unlp.edu.ar).
  - Solicitar inscripción.

- **Bibliografía / material:**
  - ***Libro base:*** Foundations of Multithreaded, Parallel, and Distributed Programming. Gregory Andrews. Addison Wesley. (www.cs.arizona.edu/people/greg/mpdbook).
  - Material de lectura adicional: bibliografía, web.
    - Principles of Concurrent and Distributed Programming, 2/E. Ben-Ari. Addison-Wesley
    - An Introduction to Parallel Computing. Design and Analysis of Algorithms, 2/E. Grama, Gupta, Karypis, Kumar. Pearson Addison Wesley.
    - The little book of semaphores. Downey. http://www.cs.ucr.edu/~kishore/papers/semaphores.pdf.
  - Planteo de temas/ejercicios (recomendado hacerlos).

Objetivos del curso

- Plantear los fundamentos de programación concurrente, estudiando sintaxis y semántica, así como herramientas y lenguajes para la resolución de programas concurrentes.
- Analizar el concepto de sistemas concurrentes que integran la arquitectura de Hardware, el Sistema Operativo y los algoritmos para la resolución de problemas concurrentes.
- Estudiar los conceptos fundamentales de comunicación y sincronización entre procesos, por Memoria Compartida y Pasaje de Mensajes.
- Vincular la concurrencia en software con los conceptos de procesamiento distribuido y paralelo, para lograr soluciones multiprocesador con algoritmos concurrentes.

Temas del curso

- **Conceptos básicos.** Concurrencia y arquitecturas de procesamiento. Multithreading, Procesamiento Distribuido, Procesamiento Paralelo.
- **Concurrencia por memoria compartida.** Procesos y sincronización. Locks y Barreras. Semáforos. Monitores. Resolución de problemas concurrentes con sincronización por MC.
- **Concurrencia por pasaje de mensajes (MP).** Mensajes asincrónicos. Mensajes sincrónicos. Remote Procedure Call (RPC). Rendezvous. Paradigmas de interacción entre procesos.
- **Lenguajes que soportan concurrencia.** Características. Similitudes y diferencias.
- **Introducción a la programación paralela.** Conceptos, herramientas de desarrollo, aplicaciones.

Motivaciones del curso

- ¿Por qué es importante la concurrencia?
- ¿Cuáles son los problemas de concurrencia en los sistemas?
- ¿Cómo se resuelven usualmente esos problemas?
- ¿Cómo se resuelven los problemas de concurrencia a diferentes niveles (hardware, SO, lenguajes, aplicaciones)?
- ¿Cuáles son las herramientas?

Links a los archivos con audio (formato MP4)

Los archivos con las clases con audio están en formato MP4. En los link de abajo están los videos comprimidos en archivos RAR.

- Introducción
  https://drive.google.com/uc?id=1uejkIzGePutyHpDDJNTMYEhgwzPjI_n5&export=download
- Conceptos básicos de concurrencia
  https://drive.google.com/uc?id=1pCmHhrvv_7TxthXU_eumwomju-LuQdZ_&export=download
- Concurrencia a nivel de hardware
  https://drive.google.com/uc?id=1cykQHm4kc249O7j_2H1U1-XBwwI-sI_O&export=download

Introducción

Concurrencia

¿Que es?

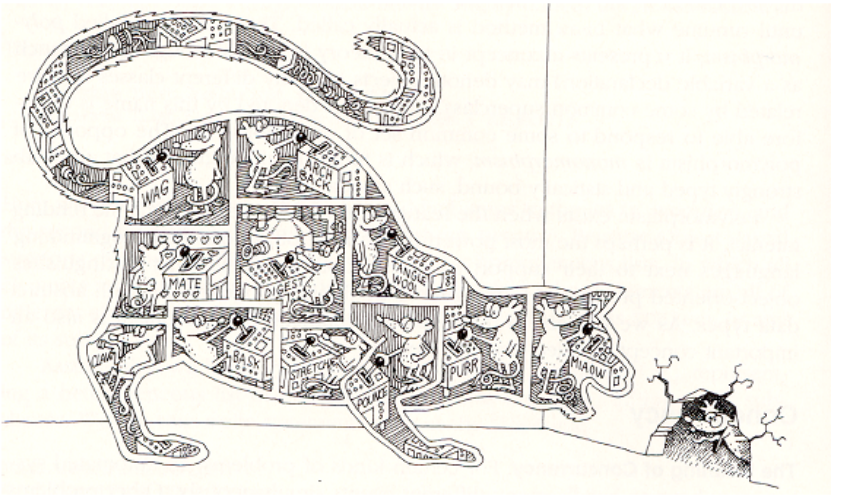
- Concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente.
- Permite a distintos objetos actuar al mismo tiempo.
- Factor relevante para el diseño de hardware, sistemas operativos, multiprocesadores, computación distribuida, programación y diseño.

¿Donde está?

- Navegador Web accediendo una página mientras atiende al usuario.
- Varios navegadores accediendo a la misma página.
- Acceso a disco mientras otras aplicaciones siguen funcionando.
- Impresión de un documento mientras se consulta.
- El teléfono avisa recepción de llamada mientras se habla.
- Varios usuarios conectados al mismo sistema (reserva de pasajes).
- Cualquier objeto más o menos “inteligente” exhibe concurrencia.
- Juegos, automóviles, etc.

Concurrencia

- Los sistemas biológicos suelen ser masivamente concurrentes: comprenden un gran número de células, evolucionando simultáneamente y realizando (independientemente) sus procesos.



- En el mundo biológico los sistemas secuenciales rara vez se encuentran.
- En algunos casos se tiende a pensar en sistemas secuenciales en lugar de concurrentes para simplificar el proceso de diseño. Pero esto va en contra de la necesidad de sistemas de cómputo cada vez más poderosos y flexibles.

Concurrencia “natural”

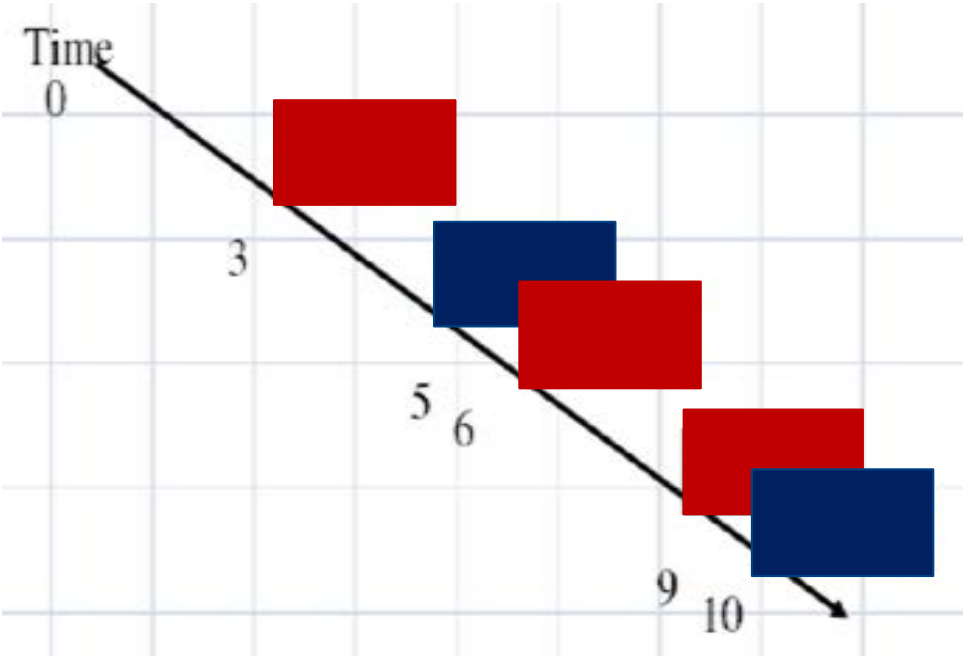
- **Problema:** Desplegar cada 3 segundos un cartel ROJO.

- **Solución secuencial:**

Programa Cartel
Mientras (true)
Demorar (3 seg)
Desplegar cartel
Fin mientras
Fin programa

Concurrencia “natural”

- **Problema:** Desplegar cada 3 segundos un cartel ROJO y cada 5 segundos un cartel AZUL.



Concurrencia “natural”

```
Programa Carteles
    Proximo_Rojo = 3
    Proximo_Azul  = 5
    Actual = 0
    Mientras (true)
        Si (Proximo_Rojo < Proximo_Azul)
                Demorar (Proximo_Rojo – Actual)
                Desplegar cartel ROJO
                Actual = Proximo_Rojo
                Proximo_Rojo = Proximo_Rojo +3
        sino
                Demorar (Proximo_Azul – Actual)
                Desplegar cartel AZUL
                Actual = Proximo_Azul
                Proximo_Azul  = Proximo_Azul  +5
    Fin mientras
Fin programa
```

Concurrencia “natural”

- Obliga a establecer un orden en el despliegue de cada cartel.
- Código más complejo de desarrollar y mantener.
- ¿Que pasa si se tienen más de dos carteles?
- ***Más natural:*** cada cartel es un elemento independiente que actúa concurrentemente con otros → ***es decir, ejecutar dos o más algoritmos simples concurrentemente***.

```
Programa Cartel (color, tiempo)
        Mientras (true)
                Demorar (tiempo segundos)
                Desplegar cartel (color)
        Fin mientras
Fin programa
```

- No hay un orden preestablecido en la ejecución ⇒ ***no determinismo*** (ejecuciones con la misma “entrada” puede generar diferentes “salidas”)

¿Por qué es necesaria la Programación Concurrente?

- No hay más ciclos de reloj → Multicore → ¿por qué? y ¿para qué?

- Aplicaciones con estructura más natural.
  - El mundo no es secuencial.
  - Más apropiado programar múltiples actividades independientes y concurrentes.
  - Reacción a entradas asincrónicas (ej: sensores en un STR).

- Mejora en la respuesta
  - No bloquear la aplicación completa por E/S.
  - Incremento en el rendimiento de la aplicación por mejor uso del hardware (ejecución paralela).

- Sistemas distribuidos
  - Una aplicación en varias máquinas.
  - Sistemas C/S o P2P.

Objetivos de los sistemas concurrentes

***Ajustar el modelo* de arquitectura de hardware y software al problema del mundo real a resolver.**

***Incrementar la performance*, mejorando los tiempos de respuesta de los sistemas de cómputo, a través de un enfoque diferente de la arquitectura física y lógica de las soluciones.**

Algunas ventajas ⇒

- La velocidad de ejecución que se puede alcanzar.
- Mejor utilización de la CPU de cada procesador.
- Explotación de la concurrencia inherente a la mayoría de los problemas reales.

Posibles comportamientos de los procesos

Programa Secuencial: un único flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente.

Por ahora llamaremos "**Proceso**" a un programa secuencial.

Un único hilo o flujo de control
→ programación secuencial, monoprocesador.

Múltiples hilos o flujos de control
→ programa concurrente.
→ programa paralelos.

Los procesos cooperan y compiten...

Posibles comportamientos de los procesos

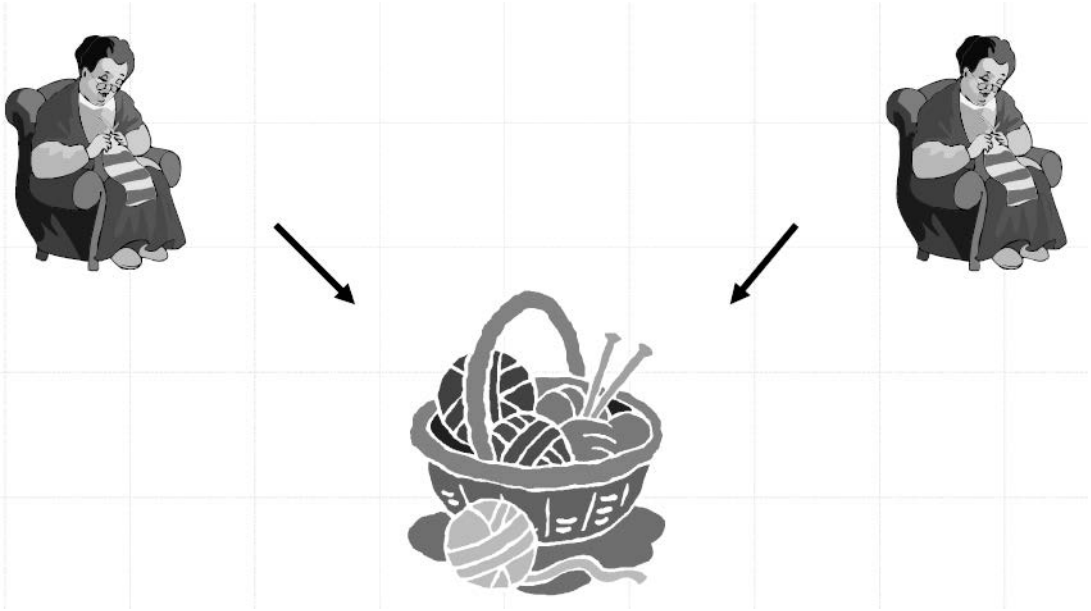
Procesos independientes

- Relativamente raros.
- Poco interesante.

Posibles comportamientos de los procesos

Competencia

- Típico en Sistemas Operativos y Redes, debido a recursos compartidos.

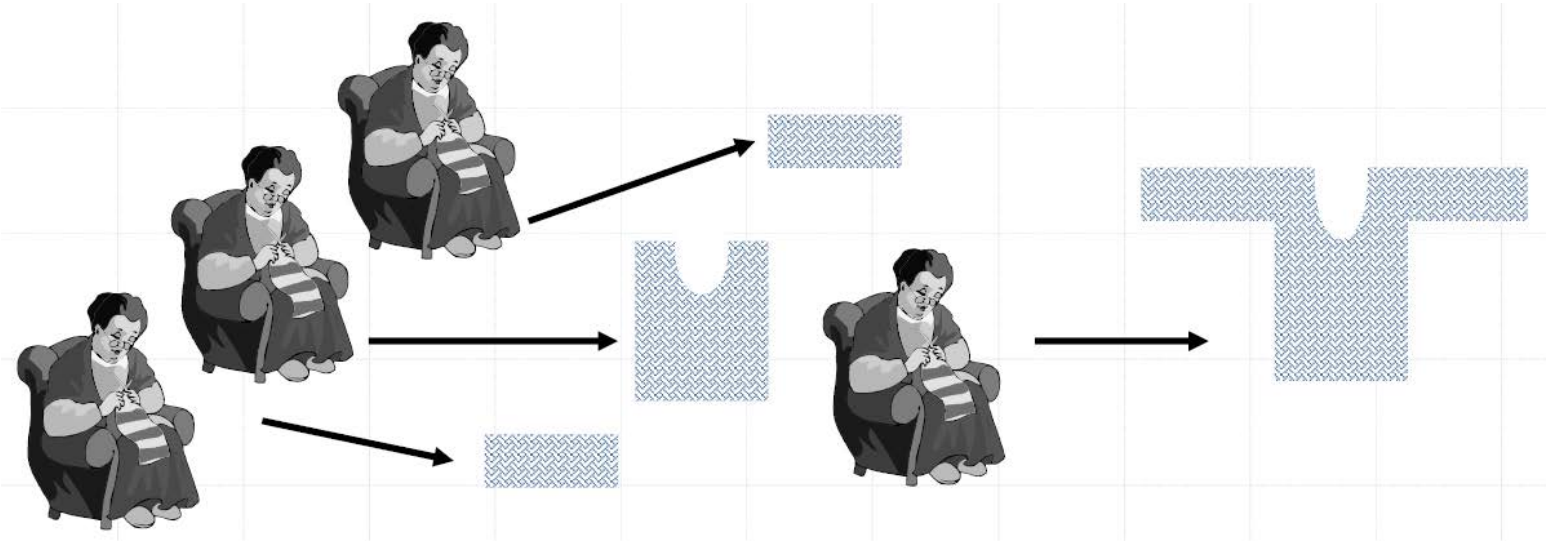


Deadlock Inanición

Posibles comportamientos de los procesos

Cooperación

- Los procesos se combinan para resolver una tarea común.
- Sincronización.



Procesamiento secuencial, concurrente y paralelo

Analicemos la solución ***secuencial*** y monoprocesador (***una máquina***) para fabricar un objeto compuesto por N partes o módulos.

La solución secuencial nos fuerza a establecer un estricto orden temporal.

Al disponer de sólo una máquina, el ensamblado final del objeto se podrá realizar luego de N pasos de procesamiento (la fabricación de cada parte).

Procesamiento secuencial, concurrente y paralelo

Si disponemos de ***N máquinas*** para fabricar el objeto, y no hay dependencia (por ejemplo de la materia prima), cada una puede trabajar ***al mismo tiempo*** en una parte. ***Solución Paralela.***

Consecuencias ⇒

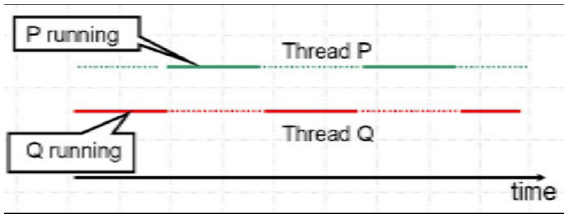
- Menor tiempo para completar el trabajo.
- Menor esfuerzo individual.
- Paralelismo del hardware.

Dificultades ⇒

- Distribución de la carga de trabajo (diferente tamaño o tiempo de fabricación de cada parte, diferentes especializaciones de cada máquina y/o velocidades).
- Necesidad de compartir recursos evitando conflictos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Tratamiento de las fallas.
- Asignación de una de las máquinas para el ensamblado (¿Cual?).

Procesamiento secuencial, concurrente y paralelo

Otro enfoque: ***un sóla máquina*** dedica una parte del tiempo a cada componente del objeto ⇒ Concurrencia sin paralelismo de hardware ⇒ Menor speedup.



Dificultades ⇒

- Distribución de carga de trabajo.
- Necesidad de compartir recursos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Necesidad de recuperar el "estado" de cada proceso al retomarlo.

CONCURRENCIA ⇒ Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.

Procesamiento secuencial, concurrente y paralelo

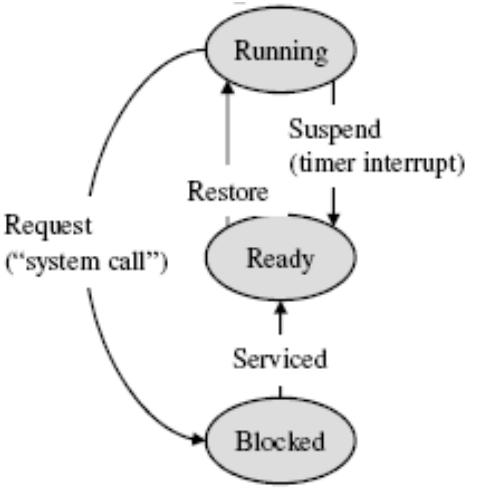
Este último caso sería multiprogramación en un procesador

- El tiempo de CPU es compartido entre varios procesos, por ejemplo por ***time slicing***.
- El sistema operativo controla y planifica procesos: si el slice expiró o el proceso se bloquea el sistema operativo hace ***context (process) switch.***

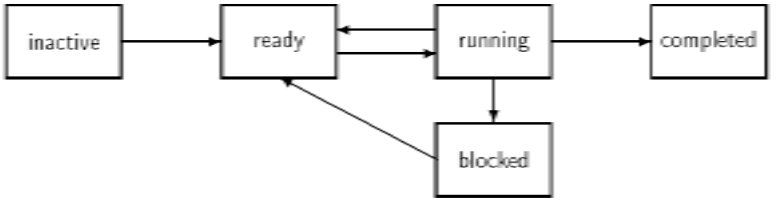
Process switch: suspender el proceso actual y restaurar otro

1. Salvar el estado actual en memoria. Agregar el proceso al final de la cola de ***ready*** o una cola de ***wait***.
2. Sacar un proceso de la cabeza de la cola ***ready***. Restaurar su estado y ponerlo a correr.

Reanudar un proceso bloqueado: mover un proceso de la cola de wait a la de ready.

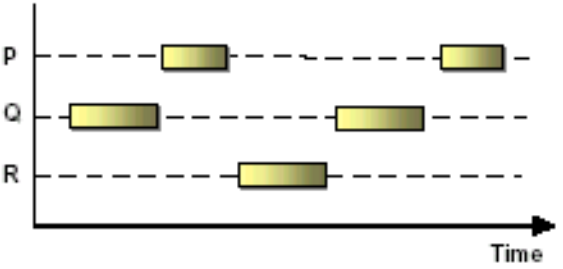


- Estados de los Procesos



Programa Concurrente

Un programa concurrente especifica dos o más "programas secuenciales" que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos.



Un proceso o tarea es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos o tareas, si el hardware lo permite (por ejemplo los TASKs de ADA).

Un programa concurrente puede tener ***N procesos*** habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de ***M procesadores*** cada uno de los cuales puede ejecutar uno o más procesos.

Características importantes:

- interacción
- no determinismo $\Rightarrow$ dificultad para la interpretación y debug

Procesos e Hilos

- ***Procesos:*** Cada proceso tiene su propio espacio de direcciones y recursos.

- ***Procesos livianos, threads o hilos:***
  - Proceso "liviano" que tiene su propio contador de programa y su pila de ejecución, pero no controla el "contexto pesado" (por ejemplo, las tablas de página).
  - Todos los hilos de un proceso comparten el mismo espacio de direcciones y recursos (los del proceso).
  - El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias.
  - La concurrencia puede estar provista por el lenguaje (Java) o por el Sistema Operativo (C/POSIX).

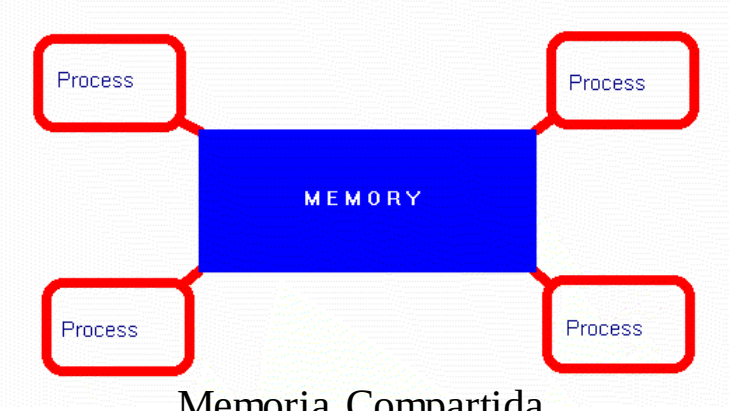
Conceptos básicos de concurrencia

Conceptos básicos de concurrencia

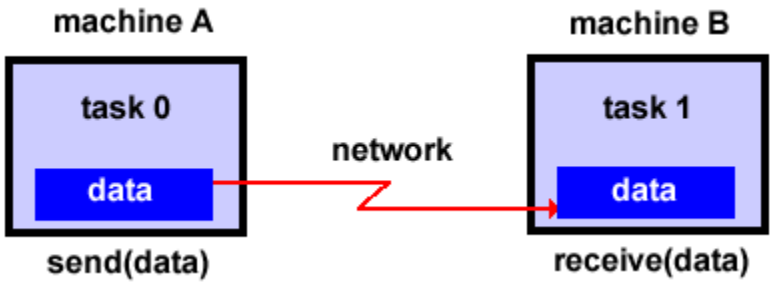
Comunicación entre procesos

La comunicación entre procesos concurrentes indica el modo en que se organizan y trasmiten datos entre tareas concurrentes. Esta organización requiere especificar ***protocolos*** para controlar el progreso y la corrección. Los procesos se **COMUNICAN**:

- Por ***Memoria Compartida***.
- Por ***Pasaje de Mensajes***.



Memoria Compartida



Pasaje de Mensajes

Conceptos básicos de concurrencia

Comunicación entre procesos

- **Memoria compartida**
  - Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella.
  - Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a bloquear y liberar el acceso a la memoria.
  - La solución más elemental es una variable de control tipo "semáforo" que habilite o no el acceso de un proceso a la memoria compartida.

- **Pasaje de mensajes**
  - Es necesario establecer un canal (lógico o físico) para transmitir información entre procesos.
  - También el lenguaje debe proveer un protocolo adecuado.
  - Para que la comunicación sea efectiva los procesos deben "saber" cuándo tienen mensajes para leer y cuando deben trasmitir mensajes.

Conceptos básicos de concurrencia

Sincronización entre procesos

La **sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan:

- Por *exclusión mutua*.
- Por *condición*.

- **Sincronización por exclusión mutua**
  - Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
  - Si el programa tiene ***secciones críticas*** que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
- **Sincronización por condición**
  - Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

Ejemplo de los dos mecanismos de sincronización en un problema de utilización de un área de memoria compartida (buffer limitado con productores y consumidores).

Conceptos básicos de concurrencia

Interferencia

***Interferencia*:** un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

Ejemplo 1: nunca se debería dividir por 0.

```
int x, y, z;

process A1
  { ….
    y = 0;
    ….
  }

process A2
  { …..
    if (y <> 0)   z = x/y;
    …..
  }
```

Ejemplo 2: siempre *Público* debería terminar con valor igual a *E1+E2* .

```
int Público = 0
process B1
  { int E1 = 0;
    for i= 1..100
      {  esperar llegada
        E1 = E1 + 1;
        Público = Público + 1;
       }
  }
process B2
  { int E2 = 0;
    for i= 1..100
      {  esperar llegada
        E2 = E2 + 1;
        Público = Público + 1;
       }
  }
```

Conceptos básicos de concurrencia

Prioridad y granularidad

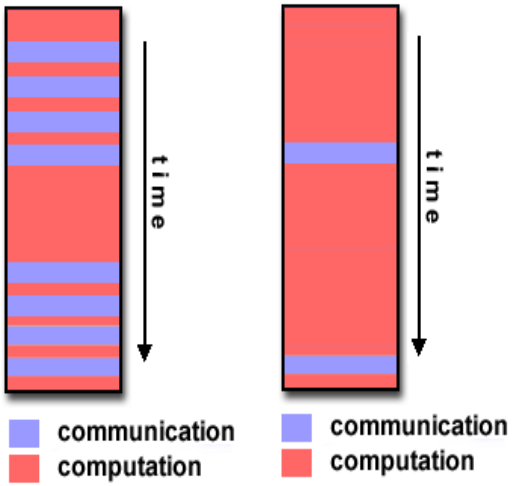
Un proceso que tiene mayor **prioridad** puede causar la suspensión (preemption) de otro proceso concurrente.

Análogamente puede tomar un recurso compartido, obligando a retirarse a otro proceso que lo tenga en un instante dado.

La **granularidad de una aplicación** está dada por la relación entre el cómputo y la comunicación.

Relación y adaptación a la arquitectura.

Grano fino y grano grueso.



Conceptos básicos de concurrencia

Manejo de los recursos

Uno de los temas principales de la programación concurrente es la administración de recursos compartidos:

- Esto incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.
- Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (***fairness***).
- Dos situaciones NO deseadas en los programas concurrentes son la ***inanición*** de un proceso (no logra acceder a los recursos compartidos) y el ***overloading*** de un proceso (la carga asignada excede su capacidad de procesamiento).
- Otro problema importante que se debe evitar es el ***deadlock***.

Conceptos básicos de concurrencia

Problema de deadlock

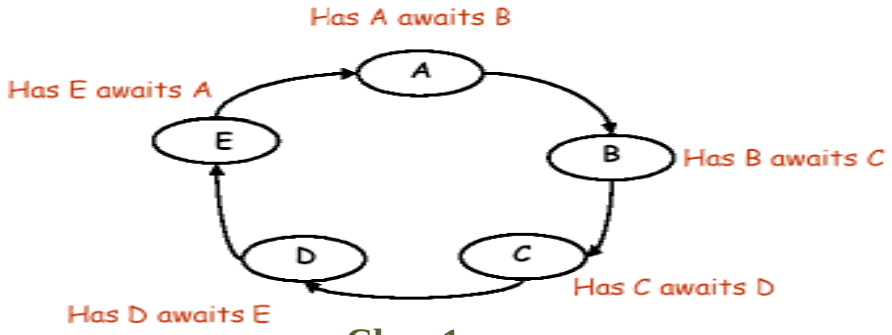
Dos (o más) procesos pueden entrar en ***deadlock*****,** si por error de programación ambos se quedan esperando que el otro libere un recurso compartido. La ausencia de deadlock es una propiedad necesaria en los procesos concurrentes.

Conceptos básicos de concurrencia

Problema de deadlock

4 propiedades necesarias y suficientes para que exista deadlock son:

- Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.
- Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperar adquirir recursos adicionales.
- No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
- Espera cíclica: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.

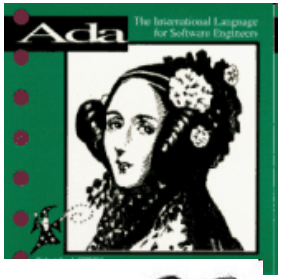


Conceptos básicos de concurrencia

Requerimientos para un lenguaje concurrente

Independientemente del mecanismo de comunicación / sincronización entre procesos, los **lenguajes de programación concurrente** deberán proveer primitivas adecuadas para la especificación e implementación de las mismas.

- **Requerimientos de un lenguaje de programación concurrente:**
  - Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
  - Mecanismos de sincronización.
  - Mecanismos de comunicación entre los procesos.



Problemas asociados con la Programación Concurrente

- Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas.
- Los procesos iniciados dentro de un programa concurrente pueden NO estar "vivos". Esta pérdida de la propiedad de ***liveness*** puede indicar deadlocks o una mala distribución de recursos.
- Hay un no determinismo implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas $\Rightarrow$ ***dificultad para la interpretación y debug.***
- Posible reducción de performance por overhead de context switch, comunicación, sincronización, ...
- Mayor tiempo de desarrollo y puesta a punto. Difícil paralelizar algoritmos secuenciales.
- Necesidad de adaptar el software concurrente al hardware paralelo para mejora real en el rendimiento.

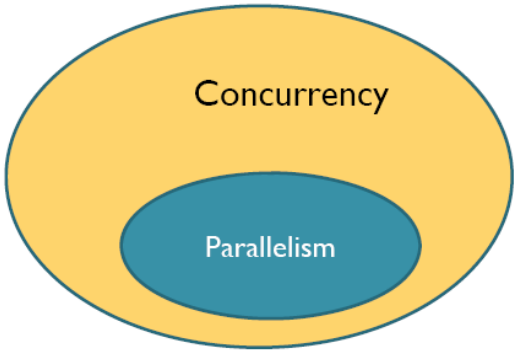
Conceptos básicos de concurrencia

Concurrencia y Paralelismo

CONCURRENCIA ⇒ Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.
Especificar la concurrencia implica especificar los ***procesos concurrentes***, su ***comunicación*** y su ***sincronización***.

PARALELISMO ⇒ Se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.

.

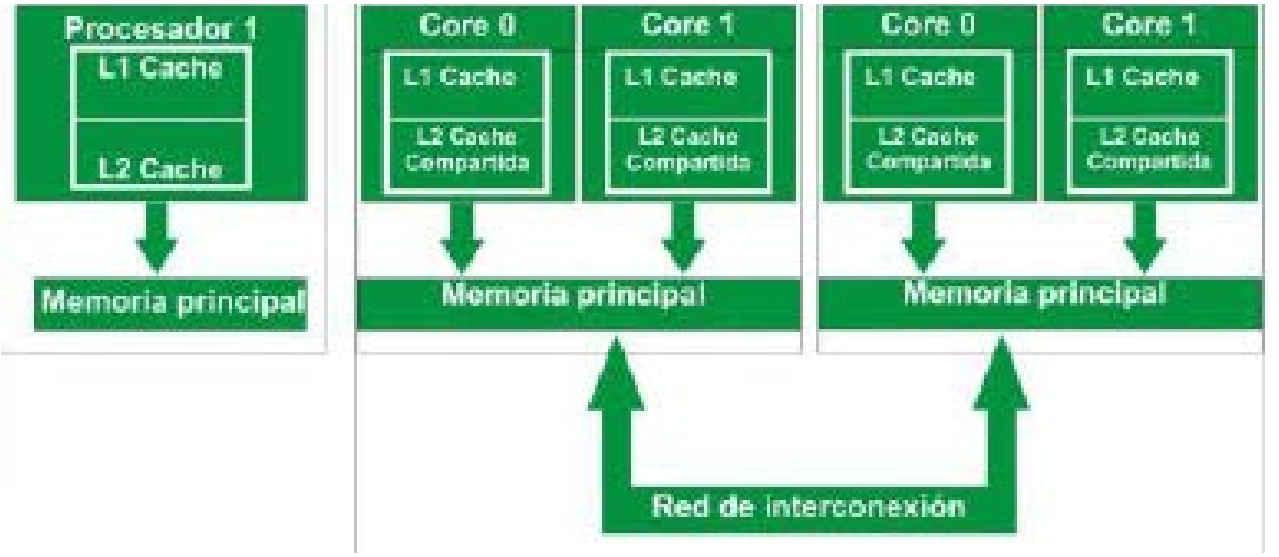


Concurrencia a nivel de hardware

Concurrencia a nivel de hardware

Límite físico en la velocidad de los procesadores

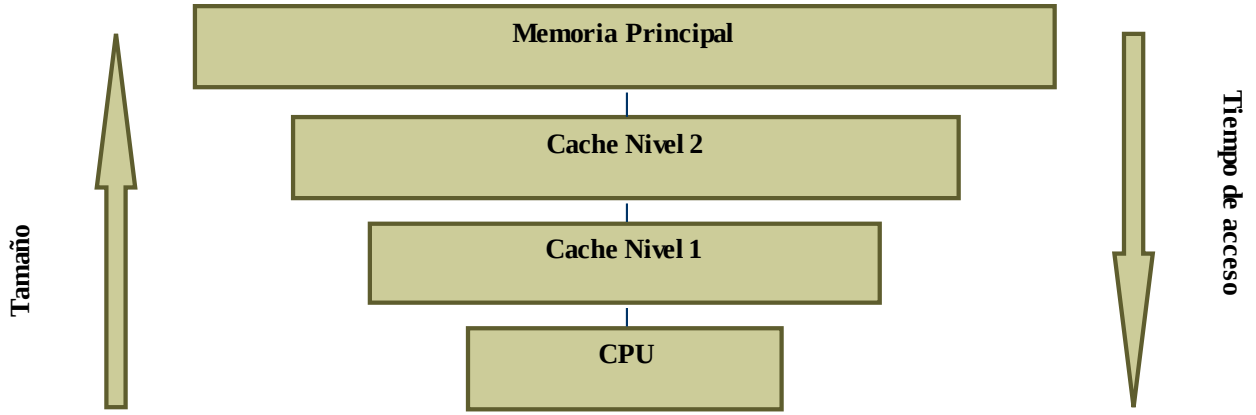
- Máquinas monoprocesador ya no pueden mejorar.
- Más procesadores por chip para mayor potencia de cómputo.
- Multicores → Cluster de multicores → Consumo.
- **Uso eficiente → Programación concurrente y paralela.**



Concurrencia a nivel de hardware

Niveles de memoria.

- Jerarquía de memoria. ¿Consistencia?
- Diferencias de tamaño y tiempo de acceso.
- Localidad temporal y espacial de los datos.

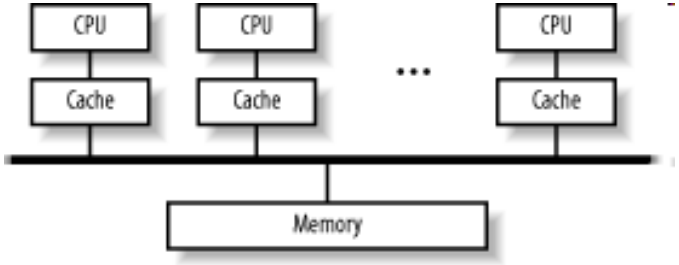


Máquinas de memoria compartida vs memoria distribuida.

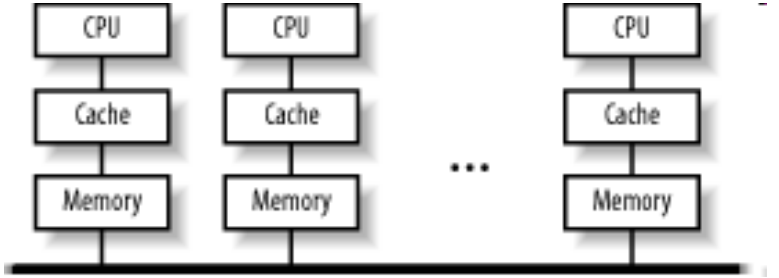
Concurrencia a nivel de hardware

Multiprocesadores de memoria compartida.

- Interacción modificando datos en la memoria compartida.
- Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problemas de sincronización y consistencia.
- Esquemas NUMA para mayor número de procesadores distribuidos.
- Problema de consistencia.



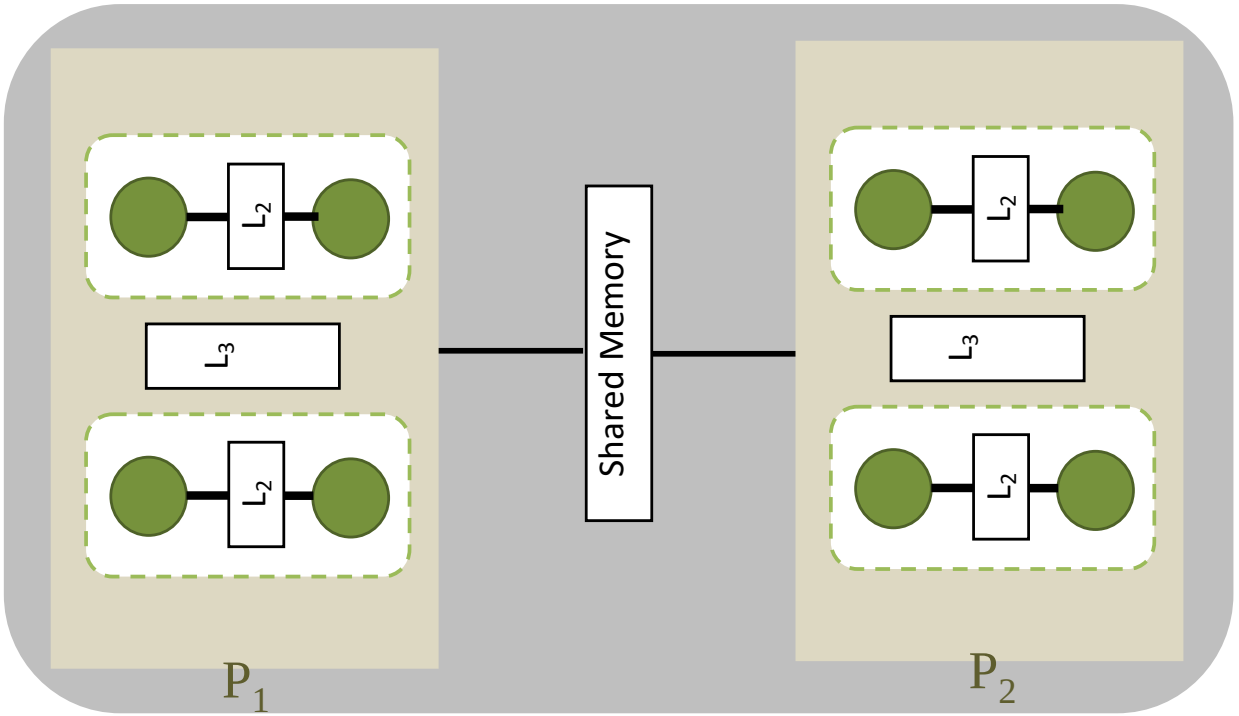
Esquema UMA



Esquema NUMA

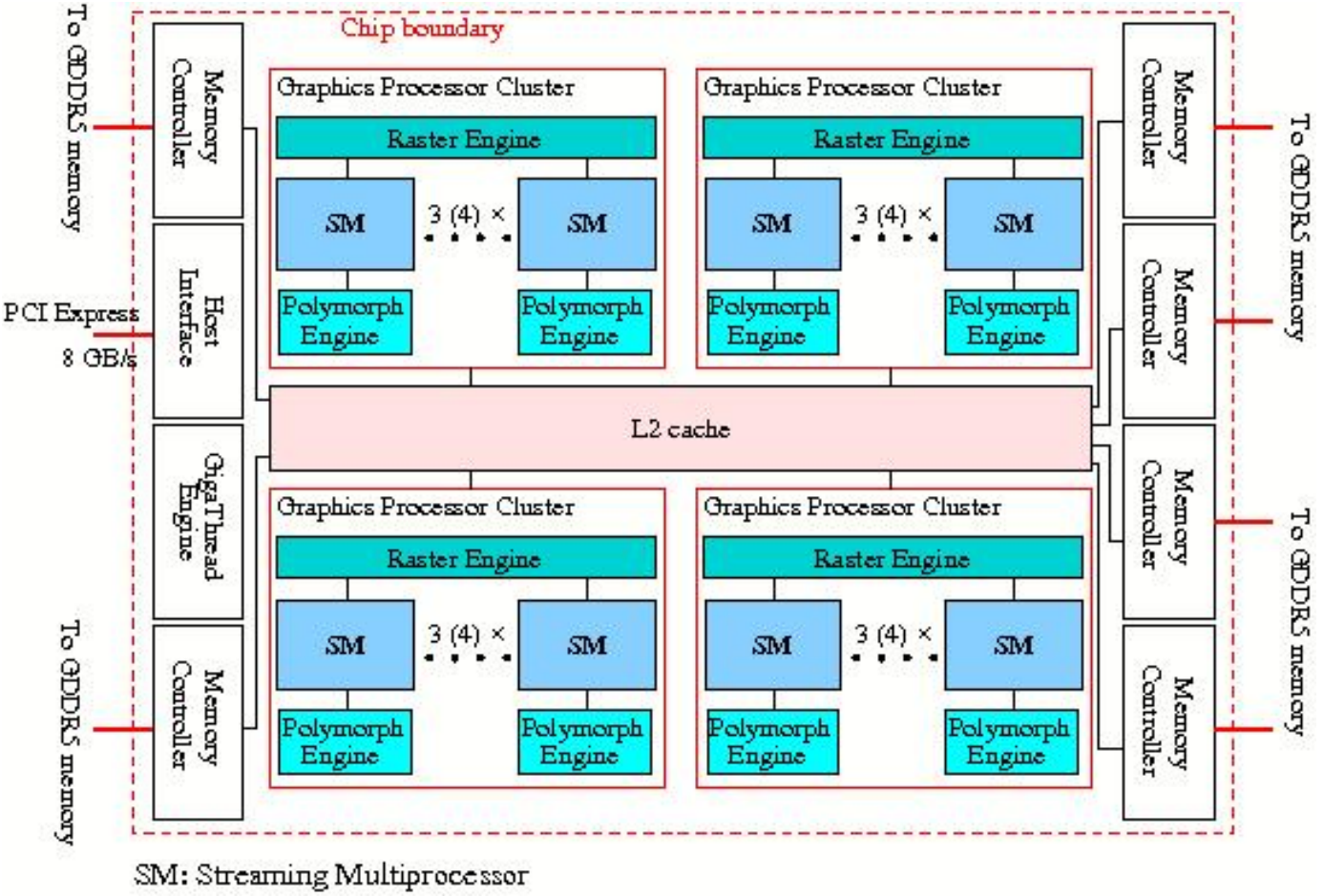
Concurrencia a nivel de hardware

- Ejemplo de multiprocesador de memoria compartida: *multicore de 8 núcleos*.



Concurrencia a nivel de hardware

- Ejemplo de multiprocesador de memoria compartida: *GPU*.



Concurrencia a nivel de hardware

Multiprocesadores con memoria distribuida.

- Procesadores conectados por una red.
- Memoria local (no hay problemas de consistencia).
- Interacción es sólo por pasaje de mensajes.
- Grado de acoplamiento de los procesadores:
  - Multicomputadores (máquinas fuertemente acopladas). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
  - Memoria compartida distribuida.
  - Clusters.
  - Redes (multiprocesador débilmente acoplado).

Un poco de historia

Evolución en respuesta a los cambios tecnológicos → De enfoques ad-hoc iniciales a técnicas generales

- **60's :** Evolución de los SO. Más procesadores por chip para mayor potencia de cómputo.
  - Controladores de dispositivos (canales) independientes permitiendo E/S → Interrupciones. No determinismo. Multiprogramación. Problema de la sección crítica.
- **70's:** Formalización de la concurrencia en los lenguajes.
- **80's**: Redes, procesamiento distribuido.
- **90's**: MPP, Internet, C/S, Web computing.
- **2000's**: SDTR, computación móvil, Cluster y multicluster computing, sistemas colaborativos, computación pervasiva y ubicua, grid computing, virtualización.
- **Hoy:** big data, IA, computación elástica, cloud computing, Green computing, bioinformática, redes de sensores, IoT, banca electrónica, …

Programación Concurrente

Clase 2

Facultad de Informática

UNLP

Links a los archivos con audio (formato MP4)

Los archivos con las clases con audio están en formato MP4. En los link de abajo están los videos comprimidos en archivos RAR.

- Clases de Instrucciones
  https://drive.google.com/uc?id=1bdsNk8uY2MKpA3usLnp8tqZt8nG6pZRU&export=download
- Acciones Atómicas y Sincronización
  https://drive.google.com/uc?id=1DzEl1aKJ-fXW9k3t7tDgy59C9HtdS2vf&export=download
- Propiedades y Fairness
  https://drive.google.com/uc?id=1lxnI0SIV-movMHbamVD2tl6VYmRS4Vij&export=download

Clases de Instrucciones

Clases de instrucciones

Programación secuencial y concurrente

Un programa concurrente esta formado por un conjunto de programas secuenciales.

- La programación secuencial estructurada puede expresarse con 3 clases de instrucciones básicas: **asignación**, **alternativa** (decisión) e **iteración** (repetición con condición).
- Se requiere una clase de instrucción para representar la concurrencia.

DECLARACIONES DE VARIABLES

- Variable simple: **tipo variable = valor . Ej: int x = 8; int z, y;**
- Arreglos: **int a[10]; int c[3:10]**

  int b[10] = ([10] 2)

  int aa[5,5]; int cc[3:10,2:9]

  int bb[5,5] = ([5] ([5] 2))

Clases de instrucciones

Programación secuencial y concurrente

ASIGNACION

- Asignación simple: **x = e**
- Sentencia de asignación compuesta: **x = x + 1; y = y – 1; z = x + y**
  a[3] = 6; aa[2,5] = a[4]
- Llamado a funciones: **x = f(y) + g(6) - 7**
- swap: **v1 :=: v2**
- **skip**: termina inmediatamente y no tiene efecto sobre ninguna variable de programa.

Clases de instrucciones

Programación secuencial y concurrente

ALTERNATIVA

- Sentencias de alternativa simple:

  if B → S

  B expresión booleana. S instrucción simple o compuesta ({}).

  B "guarda" a S pues S no se ejecuta si B no es verdadera.

- Sentencias de alternativa múltiple:

  if B1 → S1

  □ B2 → S2

  .....

  □ Bn → Sn

  fi

  Las guardas se evalúan en algún orden arbitrario.

  Elección no determinística.

  Si ninguna guarda es verdadera el *if* no tiene efecto.

- Otra opción:

  if (cond) S;

  if (cond) S1 else S2;

Ejemplos de *Sentencia Alternativa Múltiple*

Ejemplo 1:

```
if  p > 2 → p = p * 2
 □ p < 2 → p = p * 3
 □ p == 2 → p = 5
fi
```

¿Puede terminar sin tener efecto?

¿Que sucede si *p = 2*?

Ejemplo 2:

```
if  p > 2 → p = p * 2
 □ p < 2 → p = p * 3
fi
```

Ejemplo 3:

```
if  p > 2 → p = p * 2
 □ p < 6 → p = p + 4
 □ p == 4 → p = p / 2
fi
```

¿Que sucede con los siguiente valores de *p = 1, 2, 3, 4, 5, 6, 7*?

Clases de instrucciones

Programación secuencial y concurrente

ITERACIÓN

- Sentencias de alternativa ITERATIVA múltiple:

  do B1 → S1
  □ B2 → S2
  .....
  □ Bn → Sn
  od

  Las sentencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas.
  La elección es no determinística si más de una guarda es verdadera.

- For-all: forma general de repetición e iteración

  fa cuantificadores → Secuencia de Instrucciones af
  Cuantificador ≡ **variable :=** exp_inicial **to** exp_final **st B**
  El cuerpo del *fa* se ejecuta 1 vez por cada combinación de valores de las variables de iteración. Si hay cláusula *such-that* ***(st)***, la variable de iteración toma sólo los valores para los que *B* es true.
  Ejemplo: **fa i := 1 to n, j := i+1 to n st a[i] > a[j] → a[i] :=: a[j] af**

- Otra opción:

  while (cond) S;

  for [i = 1 to n, j = 1 to n st (j mod 2 = 0)] S;

Clases de instrucciones

Programación secuencial y concurrente

Ejemplos de *Sentencia Alternativa Iterativa Múltiple*

Ejemplo 1:

```
do  p > 0 → p = p - 2
 □ p < 0 → p = p + 3
 □ p == 0 → p = random(x)
od
```

¿Cuándo termina?

Ejemplo 2:

```
do  p > 2 → p = p * 2
 □ p < 2 → p = p * 3
od
```

¿Cuándo termina?

Ejemplo 3:

```
do  p > 0 → p = p - 2
 □ p > 3 → p = p + 3
 □ p > 6 → p = p / 2
od
```

¿Cuándo termina?

¿Que sucede con *p = 0, 3, 6, 9*?

Ejemplo 4:

```
do  p == 1 → p = p * 2
 □ p == 2 → p = p + 3
 □ p == 4 → p = p / 2
od
```

¿Cuándo termina?

Clases de instrucciones

Programación secuencial y concurrente

Ejemplos de *For-All*

```
fa i := 1 to n  → a[i] = 0 af
```

Inicialización de un vector

```
fa i := 1 to n, j := i+1 to n → m[i,j] :=: m[j,i] af
```

Trasposición de una matriz

```
fa i := 1 to n, j := i+1 to n st a[i] > a[j] → a[i] :=: a[j]  af
```

Ordenación de menor a mayor de un vector

Clases de instrucciones

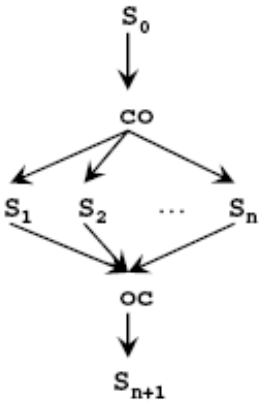
Programación secuencial y concurrente

CONCURRENCIA

- Sentencia ***co***:

  co S1 // ..... // Sn oc → Ejecuta las *Si* tareas concurrentemente.
  La ejecución del *co* termina cuando todas las tareas terminaron.

  Cuantificadores.
  co [i=1 to n] { a[i]=0; b[i]=0 } oc → Crea *n* tareas concurrentes.



- ***Process:*** otra forma de representar concurrencia

  process A {sentencias} → proceso único independiente.

  Cuantificadores.
  process B [i=1 to n] {sentencias} → *n* procesos independientes.

- **Diferencia:** ***process*** ejecuta en ***background***, mientras el código que contiene un ***co*** espera a que el proceso creado por la sentencia ***co*** termine antes de ejecutar la siguiente sentencia.

Clases de instrucciones

Programación secuencial y concurrente

Ejemplo: ¿qué imprime en cada caso? ¿son equivalentes?

```
process imprime10
   {
        for [i=1 to 10]  write(i);
    }
```

```
process imprime1 [i= 1..10]
   {
         write(i);
    }
```

No determinismo….

Acciones Atómicas y Sincronización

Atomicidad de grano fino

- ***Estado*** de un programa concurrente.
- Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más acciones atómicas.
- *Una* ***acción atómica*** hace una transformación de estado indivisibles (estados intermedios invisibles para otros procesos).
- Ejecución de un programa concurrente → ***intercalado*** (*interleaving*) de las acciones atómicas ejecutadas por procesos individuales.
- ***Historia*** de un programa concurrente (*trace*): ejecución de un programa concurrente con un *interleaving* particular. En general el número de posibles historias de un programa concurrente es enorme; pero no todas son válidas.
- ***Interacción*** → determina cuales historias son correctas.

Atomicidad de grano fino

- Algunas historias son válidas y otras no.

```
int buffer;
process 1
 { int x
  while (true)
     p1.1: read(x);
     p1.2: buffer = x;
 }
```

```
process 2
 { int y;
   while (true)
     p2.1: y = buffer;
     p2.2: print(y);
 }
```

Posibles historias:

p11, p12, p21, p22, p11, p12, p21, p22, ... ☑
p11, p12, p21, p11, p22, p12, p21, p22, ... ☑
p11, p21, p12, p22, ... ☒
p21, p11, p12, ... ☒

- Se debe asegurar un orden temporal entre las acciones que ejecutan los procesos → las tareas se intercalan ⇒ deben fijarse restricciones.

La sincronización por condición permite restringir las historias de un programa concurrente para asegurar el orden temporal necesario.

Atomicidad de grano fino

Una acción atómica de ***grano fino*** (fine grained) se debe implementar por hardware.

- ¿La operación de asignación A=B es atómica?

  NO $\Rightarrow$ (i) Load PosMemB, reg
  (ii) Store reg, PosMemA

- ¿Qué sucede con algo del tipo X=X+X?

  (i) Load PosMemX, Acumulador
  (ii) Add PosMemX, Acumulador
  (iii) Store Acumulador, PosMemX

Atomicidad de grano fino

Ejemplo 1: Cuáles son los posibles resultados con 3 procesadores. La lectura y escritura de las variables x, y, z son atómicas.

```
x = 0; y = 4; z=2;
co
  x = y + z        (1)
  // y = 3         (2)
  // z = 4         (3)
oc
```

(1) Puede descomponerse por ejemplo en:
- (1.1) Load PosMemY, Acumulador
- (1.2) Add PosMemZ, Acumulador
- (1.3) Store Acumulador, PosMemX

(2) Se transforma en: Store 3, PosMemY

(3) Se transforma en: Store 4, PosMemZ

- y = 3, z = 4 en todos los casos.
- x puede ser:
  - 6 si ejecuta (1)(2)(3) o (1)(3)(2)
  - 5 si ejecuta (2)(1)(3)
  - 8 si ejecuta (3)(1)(2)
  - 7 si ejecuta (2)(3)(1) o (3)(2)(1)
  - 6 si ejecuta (1.1)(2)(1.2)(1.3)(3)
  - 8 si ejecuta (1.1)(3)(1.2)(1.3)(2)
  - .....

Atomicidad de grano fino

Ejemplo 2: Cuáles son los posibles resultados con 2 procesadores. La lectura y escritura de las variables x, y, z son atómicas.

```
x = 2; y = 2;
co
  z = x + y          (1)
  // x = 3; y = 4;   (2)
oc
```

(1) Puede descomponerse por ejemplo en:

(1.1) Load PosMemX, Acumulador
(1.2) Add PosMemY, Acumulador
(1.3) Store Acumulador, PosMemZ

(2) Se transforma en:

(2.1) Store 3, PosMemX
(2.2) Store 4, PosMemY

x = 3, y = 4 en todos los casos.
z puede ser: 4, 5, 6 o 7.

Nunca podría parar el programa y ver un estado en que x+y = 6, a pesar de que z = x + y si puede tomar ese valor

Atomicidad de grano fino

Ejemplo 3: "Interleaving extremo" (Ben-Ari & Burns)
Dos procesos que realizan (cada uno) *N* iteraciones de la sentencia *X=X+1*.

```
int X = 0

Process P1
{  int i
   for [i=1 to N] → X=X+1
 }

Process P2
{  int i
   fa [i=1 to N] → X=X+1
}
```

¿Cuál puede ser el valor final de *X*?

- *2N*
- entre *N+1* y *2N-1*
- *N*
- *< N* (incluso 2…)

¿Cuando valdrá *2N*?
En cada iteración ….
1. Proceso 1: *Load X*
2. Proceso 1: *Incrementa su copia*
3. Proceso 1: *Store X*
4. Proceso 2: *Load X*
5. Proceso 2: *Incrementa su copia*
6. Proceso 2: *Store X*

¿Cuando valdrá *N*?
En cada iteración ….
1. Proceso 1: *Load X*
2. Proceso 2: *Load X*
3. Proceso 1: *Incrementa su copia*
4. Proceso 2: *Incrementa su copia*
5. Proceso 1: *Store X*
6. Proceso 2: *Store X*

Atomicidad de grano fino

¿Cuando valdrá *2*?

1. Proceso 1: *Load X*
2. Proceso 2: *Hace N-1 iteraciones del loop*
3. Proceso 1: *Incrementa su copia*
4. Proceso 1: *Store X*
5. Proceso 2: *Load X*
6. Proceso 1: *Hace el resto de las iteraciones del loop*
7. Proceso 2: *Incrementa su copia*
8. Proceso 2: *Store X*

… no podemos confiar en la intuición para analizar un programa concurrente…

Atomicidad de grano fino

- En la mayoría de los sistemas el tiempo absoluto no es importante.
- Con frecuencia los sistemas son actualizados con componentes más rápidas. La corrección no debe depender del tiempo absoluto.
- El tiempo se ignora, sólo las secuencias son importantes

- Puede haber distintos ordenes (*interleavings*) en que se ejecutan las instrucciones de los diferentes procesos; los programas deben ser correctos para todos ellos.

Atomicidad de grano fino

En lo que sigue, supondremos máquinas con las siguientes características:

- Los valores de los tipos básicos se almacenan en elementos de memoria leídos y escritos como acciones atómicas.
- Los valores se cargan en registros, se opera sobre ellos, y luego se almacenan los resultados en memoria.
- Cada proceso tiene su propio conjunto de registros (context switching).
- Todo resultado intermedio de evaluar una expresión compleja se almacena en registros o en memoria privada del proceso.

Atomicidad de grano fino

- Si una expresión ***e*** en un proceso no referencia una variable alterada por otro proceso, la evaluación será atómica, aunque requiera ejecutar varias acciones atómicas de grano fino.

- Si una asignación ***x = e*** en un proceso no referencia ninguna variable alterada por otro proceso, la ejecución de la asignación será atómica.

Normalmente los programas concurrentes no son disjuntos ⇒ es necesario establecer algún requerimiento más débil ...

Referencia crítica en una expresión ⇒ referencia a una variable que es modificada por otro proceso.

Asumamos que toda referencia crítica es a una variable simple leída y escrita atómicamente.

Atomicidad de grano fino

Propiedad de *"A lo sumo una vez"*

Una sentencia de asignación ***x*** = ***e*** satisface la propiedad de ***"A lo sumo una vez"*** si:

1) **e** contiene a lo sumo una referencia crítica y ***x*** no es referenciada por otro proceso, o
2) **e** no contiene referencias críticas, en cuyo caso ***x*** puede ser leída por otro proceso.

Una expresiones ***e*** que no está en una sentencia de asignación satisface la propiedad de ***"A lo sumo una vez"*** si no contiene más de una referencia crítica.

Puede haber a lo sumo una variable compartida, y puede ser referenciada a lo sumo una vez

Atomicidad de grano fino

Propiedad de *"A lo sumo una vez"*

Si una sentencia de asignación cumple la propiedad ASV, entonces su ejecución ***parece*** atómica, pues la variable compartida será leída o escrita sólo una vez.

Ejemplos:

- int x=0, y=0;
  co x=x+1 // y=y+1 oc;
  No hay ref. críticas en ningún proceso.
  En todas las historias ***x = 1*** e ***y = 1***

- int x = 0, y = 0;
  co x=y+1 // y=y+1 oc;
  El 1er proceso tiene 1 ref. crítica. El 2do ninguna.
  Siempre ***y = 1*** y ***x = 1 o 2***

- int x = 0, y = 0;
  co x=y+1 // y=x+1 oc;
  Ninguna asignación satisface ASV.
  Posibles resultados: ***x= 1 e y =2*** / ***x = 2*** e ***y = 1***
  Nunca debería ocurrir x = 1 e ***y = 1 → ERROR***

Especificación de la sincronización

- Si una expresión o asignación no satisface ASV con frecuencia es necesario ejecutarla atómicamente.
- En general, es necesario ejecutar secuencias de sentencias como una única acción atómica ***(sincronización por exclusión mutua)*** .

Mecanismo de sincronización para construir una acción atómica *de* ***grano grueso*** (*coarse grained*) como secuencia de acciones atómicas de grano fino (*fine grained*) que aparecen como indivisibles.

⟨**e**⟩ indica que la expresión ***e*** debe ser evaluada atómicamente.

⟨**await (B) S;**⟩ se utiliza para especificar sincronización.

La expresión booleana B especifica una condición de demora.
S es una secuencia de sentencias que se garantiza que termina.
Se garantiza que B es true cuando comienza la ejecución de S.
Ningún estado interno de S es visible para los otros procesos.

Especificación de la sincronización

Sentencia con alto poder expresivo, pero el costo de implementación de la forma general de *await* (exclusión mutua y sincronización por condición) es alto.

- ***Await general*:** ⟨**await (s>0) s=s-1;**⟩

- ***Await para exclusión mutua*:** ⟨**x = x + 1; y = y + 1**⟩

- ***Ejemplo await para sincronización por condición*:** ⟨**await (count > 0)** ⟩

  Si B satisface ASV, puede implementarse como ***busy waiting*** o ***spin loop***

  do (not B) → skip od **(while (not B);)**

Acciones atómicas incondicionales y condicionales

Especificación de la sincronización

Ejemplo: productor/consumidor con buffer de tamaño N.

```
cant: int = 0;
Buffer: cola;

process Productor
 { while (true)
       Generar Elemento
       <await  (cant < N); push(buffer, elemento); cant++ >
 }

process Consumidor
 { while (true)
       <await  (cant > 0); pop(buffer, elemento); cant-- >
       Consumir Elemento
 }
```

Propiedades y Fairness

Propiedades de seguridad y vida

Una *propiedad* de un programa concurrente es un atributo verdadero en cualquiera de las historias de ejecución del mismo

Toda propiedad puede ser formulada en términos de dos clases: seguridad y vida.

- ***seguridad*** (safety)
  - Nada malo le ocurre a un proceso: asegura estados consistentes.
  - Una *falla de seguridad* indica que algo anda mal.
  - Ejemplos de propiedades de seguridad: exclusión mutua, ausencia de interferencia entre procesos, ***partial correctness***.
- ***vida*** (liveness)
  - Eventualmente ocurre algo bueno con una actividad: progresa, no hay deadlocks.
  - Una *falla de vida* indica que las cosas dejan de ejecutar.
  - Ejemplos de vida: ***terminación***, asegurar que un pedido de servicio será atendido, que un mensaje llega a destino, que un proceso eventualmente alcanzará su SC, etc $\Rightarrow$ ***dependen de las políticas de scheduling.***

¿Que pasa con la ***total correctness***?

Fairness y políticas de scheduling

***Fairness*: trata de garantizar que los procesos tengan chance de avanzar, sin importar lo que hagan los demás**

Una acción atómica en un proceso es ***elegible*** si es la próxima acción atómica en el proceso que será ejecutada. Si hay varios procesos ⇒ hay ***varias acciones atómicas elegibles***.
Una ***política de scheduling*** determina cuál será la próxima en ejecutarse.

Ejemplo: Si la política es asignar un procesador a un proceso hasta que termina o se demora. ¿Qué podría suceder en este caso?

```
bool continue = true;
co while (continue); // continue = false; oc
```

Fairness y políticas de scheduling

Fairness Incondicional. Una política de scheduling es incondicionalmente fair si toda acción atómica incondicional que es elegible eventualmente es ejecutada.

En el ejemplo anterior, RR es incondicionalmente fair en monoprocesador, y la ejecución paralela lo es en un multiprocesador.

Fairness Débil. Una política de scheduling es débilmente fair si :
(1) Es incondicionalmente fair y
(2) Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada, asumiendo que su condición se vuelve ***true*** y permanece ***true*** hasta que es vista por el proceso que ejecuta la acción atómica condicional.

No es suficiente para asegurar que cualquier sentencia ***await*** elegible eventualmente se ejecuta: la guarda podría cambiar el valor (de ***false*** a ***true*** y nuevamente a ***false***) mientras un proceso está demorado.

Fairness y políticas de scheduling

Fairness Fuerte. Una política de scheduling es ***fuertemente fair*** si:
(1) Es incondicionalmente fair y
(2) Toda acción atómica condicional que se vuelve elegible eventualmente es ejecutada pues su guarda se convierte en ***true*** con infinita frecuencia.

Ejemplo: ¿Este programa termina?

```
bool continue = true, try = false;
co while (continue) { try = true; try = false; }
 // ⟨await (try) continue = false⟩
oc
```

No es simple tener una política que sea práctica y fuertemente fair. En el ejemplo anterior, con 1 procesador, una política que alterna las acciones de los procesos sería fuertemente fair, pero es impráctica. Round-robin es práctica pero no es fuertemente fair.

Programación Concurrente

Clase 3

Facultad de Informática
UNLP

Links a los archivos con audio (formato MP4)

El archivo con la clase con audio está en formato MP4. En el link de abajo está el video comprimido en archivo RAR.

- Sincronización por Variables Compartidas (Locks – Barreras)
  https://drive.google.com/uc?id=1B_pFlBswRc19QUIz8srb9LJ2MHcDeJgc&export=download

Sincronización por Variables Compartidas

Locks - Barreras

Herramientas para la concurrencia

- **Memoria Compartida**
  - Variables compartidas
  - Semáforos
  - Monitores

- **Memoria distribuida (pasaje de mensajes)**
  - Mensajes asincrónicos
  - Mensajes sincrónicos
  - Remote Procedure Call (RPC)
  - Rendezvous

Locks y barreras

Problema de la Sección Crítica: implementación de acciones atómicas en software (***locks***).

Barrera: punto de sincronización que todos los procesos deben alcanzar para que cualquier proceso pueda continuar.

En la técnica de ***busy waiting*** un proceso chequea repetidamente una condición hasta que sea verdadera:

- Ventaja de implementarse con instrucciones de cualquier procesador.
- Ineficiente en multiprogramación (cuando varios procesos comparten el procesador y la ejecución es intercalada).
- Aceptable si cada proceso ejecuta en su procesador.

El problema de la Sección Crítica

```
process SC[i=1 to n]
  {  while (true)
       {    protocolo de entrada;   ⇔  <
            sección crítica;        ⇔  SC
            protocolo de salida;    ⇔  >
            sección no crítica;
       }
  }
```

Las soluciones a este problema pueden usarse para implementar sentencias ***await*** arbitrarias.

¿Qué propiedades deben satisfacer los protocolos de entrada y salida?.

El problema de la Sección Crítica

Propiedades a cumplir

Exclusión mutua: A lo sumo un proceso está en su SC

Ausencia de Deadlock (Livelock): si 2 o más procesos tratan de entrar a sus SC, al menos uno tendrá éxito.

Ausencia de Demora Innecesaria: si un proceso trata de entrar a su SC y los otros están en sus SNC o terminaron, el primero no está impedido de entrar a su SC.

Eventual Entrada: un proceso que intenta entrar a su SC tiene posibilidades de hacerlo (eventualmente lo hará).

- Solución trivial ⟨SC⟩. Pero, ¿cómo se implementan los ⟨ ⟩?

El problema de la Sección Crítica
Implementación de sentencias *await*

- Cualquier solución al problema de la SC se puede usar para implementar una acción atómica incondicional ⟨ S;⟩ ⇒ **SCEnter ; S; SCExit**

- Para una acción atómica condicional ⟨await (B) S;⟩ ⇒
  SCEnter ; while (not B) {SCExit; SCEnter;} S; SCExit;

- Si ***S*** es *skip*, y ***B*** cumple *ASV*, ⟨await (B);⟩ puede implementarse por medio de ⇒ **while (not B) skip;**

Correcto, pero ***ineficiente***: un proceso está spinning continuamente saliendo y entrando a SC hasta que otro altere una variable referenciada en *B*.

- Para reducir *contención de memoria* ⇒
  SCEnter ; while (not B) {SCExit; Delay; SCEnter;} S; SCExit;

El problema de la Sección Crítica.
Solución hardware: deshabilitar interrupciones

```
process SC[i=1 to n] {
    while (true) {
        deshabilitar interrupciones;        # protocolo de entrada
        sección crítica;
        habilitar interrupciones;           # protocolo de salida
        sección no crítica;
    }
}
```

- Solución correcta para una máquina monoprocesador.
- Durante la SC no se usa la multiprogramación → penalización de performance
- La solución no es correcta en un multiprocesador.

El problema de la Sección Crítica.

Solución de "grano grueso"

bool in1=false, in2=false # MUTEX: ¬(in1 ∧ in2) #

```
process SC1
 {  while (true)
      { in1 = true; # protocolo de entrada
        sección crítica;
        in1 = false; # protocolo de salida
        sección no crítica;
      }
 }
```

```
process SC2
 {  while (true)
      { in2 = true; # protocolo de entrada
        sección crítica;
        in2 = false; # protocolo de salida
        sección no crítica;
      }
 }
```

- No asegura el invariante MUTEX ⇒ solución de "grano grueso"

```
process SC1
 {  while (true)
      {⟨await (not in2) in1 = true;⟩
        sección crítica;
        in1 = false;
        sección no crítica;
      }
 }
```

```
process SC2
 {  while (true)
      { ⟨await (not in1) in2 = true;⟩
        sección crítica;
        in2 = false;
        sección no crítica;
      }
 }
```

- **¿Satisface las 4 propiedades?**

El problema de la Sección Crítica.

Solución de "grano grueso" - ¿Cumple las condiciones?

Exclusión mutua: por construcción, SC1 y SC2 se excluyen en el acceso a la SC.

```
bool in1=false, in2=false   # MUTEX: ¬(in1 ∧ in2)  #
```

```
process SC1
 {  while (true)
      {⟨await (not in2) in1 = true;⟩
        sección crítica;
        in1 = false;
        sección no crítica;
      }
 }
```

```
process SC2
 {  while (true)
      { ⟨await (not in1) in2 = true;⟩
        sección crítica;
        in2 = false;
        sección no crítica;
      }
 }
```

Ausencia de deadlock: si hay deadlock, SC1 y SC2 están bloqueados en su protocolo de entrada ⇒ **in1** e **in2** serían *true* a la vez. Esto NO puede darse ya que ambas son falsas en ese punto (lo son inicialmente, y al salir de SC, cada proceso vuelve a serlo).

Ausencia de demora innecesaria: si SC1 está fuera de su SC o terminó, **in1** es *false*; si SC2 está tratando de entrar a SC y no puede, **in1** es *true*; (¬ **in1** ∧ **in1** = *false*) ⇒ ***no hay demora innecesaria***.

El problema de la Sección Crítica.

Solución de "grano grueso" - ¿Cumple las condiciones?

```
bool in1=false, in2=false   # MUTEX: ¬(in1 ∧ in2)  #
```

```
process SC1
 {  while (true)
      {⟨await (not in2) in1 = true;⟩
        sección crítica;
        in1 = false;
        sección no crítica;
      }
 }
```

```
process SC2
 {  while (true)
      { ⟨await (not in1) in2 = true;⟩
        sección crítica;
        in2 = false;
        sección no crítica;
      }
 }
```

Eventual Entrada:

- Si SC1 está tratando de entrar a su SC y no puede, SC2 está en SC (**in2** es *true*). Un proceso que está en SC eventualmente sale → **in2** será *false* y la guarda de SC1 *true*.
- Análogamente para SC2.
- Si los procesos corren en procesadores iguales y el tiempo de acceso a SC es finito, las guardas son *true* con infinita frecuencia.

Se garantiza la eventual entrada con una política de scheduling fuertemente fair.

El problema de la Sección Crítica.

Solución de "grano grueso"

```
bool in1=false, in2=false   # MUTEX: ¬(in1 ∧ in2)  #
```

```
process SC1
 {  while (true)
      {⟨await (not in2) in1 = true;⟩
        sección crítica;
        in1 = false;
        sección no crítica;
      }
 }
```

```
process SC2
 {  while (true)
      { ⟨await (not in1) in2 = true;⟩
        sección crítica;
        in2 = false;
        sección no crítica;
      }
 }
```

- ¿Si hay ***n*** procesos? → Cambio de variables.

```
bool lock=false;  # lock = in1 v in2 #
```

```
process SC1
 {  while (true)
      {⟨await (not lock) lock= true;⟩
        sección crítica;
        lock = false;
        sección no crítica;
      }
 }
```

```
process SC2
 {  while (true)
      { ⟨await (not lock) lock= true;⟩
        sección crítica;
        lock= false;
        sección no crítica;
      }
 }
```

El problema de la Sección Crítica.

Solución de "grano grueso"

```
bool lock=false;  # lock = in1 v in2 #
```

```
process SC1
 {  while (true)
      {⟨await (not lock) lock= true;⟩
        sección crítica;
        lock = false;
        sección no crítica;
      }
 }
```

```
process SC2
 {  while (true)
      { ⟨await (not lock) lock= true;⟩
        sección crítica;
        lock= false;
        sección no crítica;
      }
 }
```

- Generalizar la solución a *n* procesos

```
process SC [i=1..n]
 {  while (true)
      {⟨await (not lock) lock= true;⟩
        sección crítica;
        lock = false;
        sección no crítica;
      }
 }
```

El problema de la Sección Crítica.
Solución de "grano fino": *Spin Locks*

***Objetivo**:* hacer "atómico" el *await* de grano grueso.

Idea: usar instrucciones como ***Test & Set*** (TS), ***Fetch & Add*** (FA) o ***Compare & Swap***, disponibles en la mayoría de los procesadores.

¿Como funciona ***Test & Set***?

```
bool TS (bool ok);
{  < bool inicial = ok;
      ok = true;
      return inicial; >
}
```

El problema de la Sección Crítica.

Solución de "grano fino": *Spin Locks*

```
bool lock = false;
process SC [i=1..n]
 {  while (true)
      {⟨await (not lock) lock= true;⟩
        sección crítica;
        lock = false;
        sección no crítica;
      }
 }
```

```
bool lock=false;
process SC[i=1 to n]
{  while (true)
     {  while (TS(lock)) skip ;
        sección crítica;
        lock = false;
        sección no crítica;
     }
}
```

Solución tipo ***"spin locks"***: los procesos se quedan iterando (spinning) mientras esperan que se limpie ***lock***.

Cumple las 4 propiedades si el scheduling es fuertemente fair.
Una política débilmente fair es aceptable (rara vez todos los procesos están simultáneamente tratando de entrar a su SC).

El problema de la Sección Crítica.

Solución de "grano fino": *Spin Locks*

TS escribe siempre en *lock* aunque el valor no cambie ⇒ Mejor ***Test-and-Test-and-Set***

```
bool lock=false;
process SC[i=1 to n]
{  while (true)
     {   while (TS(lock)) skip ;
         sección crítica;
         lock = false;
         sección no crítica;
     }
}
```

```
while (lock) skip;
while (TS(lock))
      while (lock) skip;
```

Memory contention se reduce, pero no desaparece. En particular, cuando *lock* pasa a *false* posiblemente todos intenten hacer TS.

Problema de la Sección Crítica.
Solución Fair: algoritmo *Tie-Breaker*

Spin locks ⇒ no controla el orden de los procesos demorados ⇒ es posible que alguno no entre nunca si el scheduling no es fuertemente fair (***race conditions***).

Algoritmo Tie-Breaker *(2 procesos):* protocolo de SC que requiere scheduling sólo débilmente fair y no usa instrucciones especiales ⇒ más complejo.

Usa una variable por cada proceso para indicar que el proceso comenzó a ejecutar su protocolo de entrada a la sección crítica, y una variable adicional para romper empates, indicando qué proceso fue el último en comenzar dicha entrada ⇒ esta última variable es compartida y de acceso protegido.

Demora (quita prioridad) al último en comenzar su *entry protocol.*

Problema de la Sección Crítica.
Solución Fair: algoritmo *Tie-Breaker*

Solución de "*Grano Grueso*" al Algoritmo ***Tie-Breaker***

```
bool in1 = false, in2 = false;
int ultimo  = 1;

process SC1 {
   while (true) {        ultimo  = 1; in1 = true;
                         ⟨await (not in2 or ultimo==2);⟩
                         sección crítica;
                         in1 = false;
                         sección no crítica;
    }
}

process SC2 {
   while (true) {        ultimo  = 2; in2 = true;
                         ⟨await (not in1 or ultimo==1);⟩
                         sección crítica;
                         in2 = false;
                         sección no crítica;
    }
}
```

Problema de la Sección Crítica.
Solución Fair: algoritmo *Tie-Breaker*

Solución de "*Grano Fino*" al Algoritmo ***Tie-Breaker***

```
bool in1 = false, in2 = false;
int ultimo  = 1;

process SC1 {
   while (true) {        in1 = true; ultimo  = 1;
                         while (in2 and ultimo == 1) skip;
                         sección crítica;
                         in1 = false;
                         sección no crítica;
    }
}

process SC2 {
   while (true) {        in2 = true; ultimo  = 2;
                         while (in1 and ultimo == 2) skip;
                         sección crítica;
                         in2 = false;
                         sección no crítica;
    }
}
```

Problema de la Sección Crítica.
Solución Fair: algoritmo *Tie-Breaker*

Generalización a *n* procesos:

- Si hay *n* procesos, el protocolo de entrada en cada uno es un *loop* que itera a través de *n-1* etapas.
- En cada etapa se usan instancias de ***tie-breaker*** para dos procesos para determinar cuáles avanzan a la siguiente etapa.
- Si a lo sumo a un proceso a la vez se le permite ir por las *n-1* etapas $\Rightarrow$ a lo sumo uno a la vez puede estar en la SC.

```
int in[1:n] = ([n] 0), ultimo[1:n] = ([n] 0);
process SC[i = 1 to n] {
   while (true) {
      for [j = 1 to n] {     # protocolo de entrada
            # el proceso i está en la etapa j y es el último
            in[i] = j; ultimo[j] = i;
            for [k = 1 to n st i <> k] {
                # espera si el proceso k está en una etapa más alta
                # y el proceso i fue el último en entrar a la etapa j
                while (in[k] >= in[i] and ultimo[j]==i) skip;
              }
       }
       sección crítica;
       in[i] = 0;
       sección no crítica;
   }
}
```

Problema de la Sección Crítica.

Solución Fair: algoritmo *Ticket*

Tie-Breaker n-proceso ⇒ complejo y costoso en tiempo.

Algoritmo Ticket: se reparten números y se espera a que sea el turno.

Los procesos toman un número mayor que el de cualquier otro que espera ser atendido; luego esperan hasta que todos los procesos con número más chico han sido atendidos.

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

{ TICKET: proximo > 0 ^ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]== proximo) ^
   (turno[i] >0) ⇒ (∀j: 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) )}

process SC [i: 1..n]
{ while (true)
   {  < turno[i] = numero;  numero = numero +1; >
      < await turno[i] == proximo; >
      sección crítica;
      < proximo = proximo + 1; >
      sección no crítica;
   }
}
```

Problema de la Sección Crítica.
Solución Fair: algoritmo *Ticket*

Potencial problema: los valores de *próximo* y *turno* son ilimitados.
En la práctica, podrían resetearse a un valor chico (por ejemplo, 1).

Cumplimiento de las propiedades:

- El predicado ***TICKET*** es un invariante global, pues ***número*** es leído e incrementado en una acción atómica y ***próximo*** es incrementado en una acción atómica $\Rightarrow$ hay a lo sumo un proceso en la SC.
- La ausencia de deadlock y de demora innecesaria resultan de que los valores de ***turno*** son únicos.
- Con scheduling débilmente fair se asegura eventual entrada

El **await** puede implementarse con busy waiting (la expresión booleana referencia una sola variable compartida).

El incremento de ***proximo*** puede ser un load/store normal (a lo sumo un proceso puede estar ejecutando su protocolo de salida)

Problema de la Sección Crítica.

Solución Fair: algoritmo *Ticket*

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

process SC [i: 1..n]
{ while (true)
    {  <turno[i] = numero;  numero = numero +1>
       while (turno[i] <> proximo) skip;
       sección crítica;
       proximo = proximo + 1;
       sección no crítica;
    }
}
```

¿Cómo se implementa la primera acción atómica donde se asigna el número?

- Sea Fetch-and-Add una instrucción con el siguiente efecto:

FA(var,incr): < temp = var; var = var + incr; return(temp) >

```
int numero = 1, proximo = 1, turno[1:n] = ( [n] 0 );

process SC [i: 1..n]
{ while (true)
    {  turno[i] = FA (numero, 1);
       while (turno[i] <> proximo) skip;
       sección crítica;
       proximo = proximo + 1;
       sección no crítica;
    }
}
```

El problema de la Sección Crítica.
Solución Fair: algoritmo *Bakery*

Ticket $\Rightarrow$ si no existe FA se debe simular con una SC y la solución puede no ser fair.

Algoritmo Bakery*:* Cada proceso que trata de ingresar recorre los números de los demás y se auto asigna uno mayor. Luego espera a que su número sea el menor de los que esperan.

Los procesos se chequean entre ellos y no contra un global.

- El algoritmo ***Bakery*** es más complejo, pero es *fair* y no requiere instrucciones especiales.
- No requiere un contador global ***proximo*** que se "entrega" a cada proceso al llegar a la SC.

El problema de la Sección Crítica.

Solución Fair: algoritmo *Bakery*

```
int turno[1:n] = ([n] 0);

{BAKERY: (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{   while (true)
        {     ⟨ turno[i] = max(turno[1:n] + 1; ⟩
              for [j = 1 to n st j <> i] ⟨ await (turno[j] == 0 or turno[i] < turno[j]); ⟩
              sección crítica
              turno[i] = 0;
              sección no crítica
        }
}
```

Esta solución de grano grueso no es implementable directamente:

- La asignación a turno[i] exige calcular el máximo de n valores.
- El await referencia una variable compartida dos veces.

El problema de la Sección Crítica.

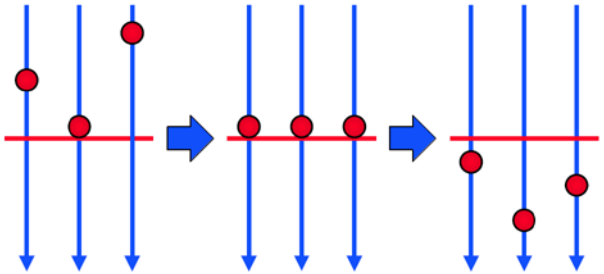
Solución Fair: algoritmo *Bakery*

```
int turno[1:n] = ([n] 0);

{BAKERY: (∀ i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i] > 0) ∧ ( ∀ j : 1 ≤ j ≤ n, j ≠ i:
turno[j] = 0 ∨ turno[i] < turno[j] ) ) }

process SC[i = 1 to n]
{   while (true)
        {  turno[i] = 1;   //indica que comenzó el protocolo de entrada
           turno[i] = max(turno[1:n]) + 1;
           for [j = 1 to n st j != i]        //espera su turno
                 while (turno[j] != 0)  and (  (turno[i],i) > (turno[j],j) ) →  skip;
            sección crítica
            turno[i] = 0;
            sección no crítica
        }
}
```

Sincronización *Barrier*

Sincronización barrier: una barrera es un punto de demora a la que deben llegar todos los procesos antes de permitirles pasar y continuar su ejecución.

Dependiendo de la aplicación las barreras pueden necesitar reutilizarse más de una vez (por ejemplo en algoritmos iterativos).



Sincronización *Barrier*

Contador Compartido

n procesos necesitan encontrarse en una barrera:

- Cada proceso incrementa una variable *Cantidad* al llegar.
- Cuando *Cantidad* es *n* los procesos pueden pasar.

```
int cantidad = 0;
process Worker[i=1 to n]
{ while (true)
        {   código para implementar la tarea i;
            ⟨ cantidad = cantidad + 1; ⟩
            ⟨ await (cantidad == n); ⟩
         }
}
```

- Se puede implementar con:

FA(cantidad,1);
while (cantidad <> n) skip;

Sincronización *Barrier*

Contador Compartido

```
int cantidad = 0;
process Worker[i=1 to n]
{ while (true)
        {  código para implementar la tarea i;
           FA (cantidad, 1);
           while (cantidad <> n) skip;
         }
}
```

¿Cuando se reinicia Cantidad en 0 para la siguiente iteración?

Sincronización *Barrier*

Flags y Coordinadores

- Si no existe FA → Puede distribuirse *Cantidad* usando *n* variables (arreglo *arribo[1..n]*).
- El *await* pasaría a ser:
  ⟨**await (arribo[1] + … + arribo[n] == n);**⟩
- Reintroduce contención de memoria y es ineficiente.

Puede usarse un conjunto de valores adicionales y un proceso más ⇒ ***Cada Worker espera por un único valor***

```
int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);

process Worker[i=1 to n]
{ while (true)
        {   código para implementar la tarea i;
            arribo[i] = 1;
            ⟨ await (continuar[i] == 1); ⟩
            continuar[i] = 0;
        }
}

process Coordinador
{ while (true)
        {   for [i = 1 to n]
                { ⟨ await (arribo[i] == 1); ⟩
                    arribo[i] = 0;
                }
            for [i = 1 to n] continuar[i] = 1;
        }
}
```

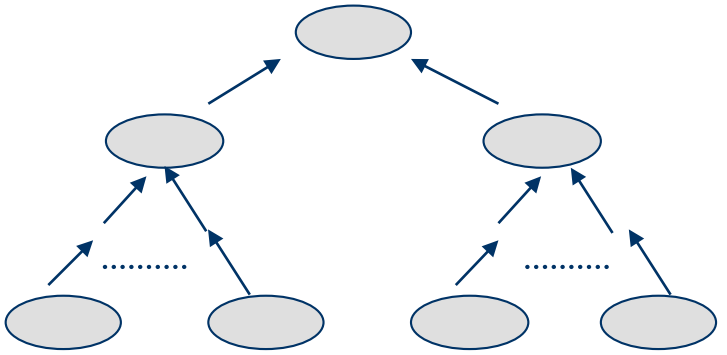
Sincronización *Barrier*

Flags y Coordinadores

```
int arribo[1:n]  = ([n] 0), continuar[1:n]  = ([n] 0);

process Worker[i=1 to n]
{ while (true)
        {     código para implementar  la tarea i;
              arribo[i]  = 1;
              while (continuar[i]  == 0) skip;
              continuar[i]  = 0;
         }
}

process Coordinador
{ while (true)
        {     for [i = 1 to n]
                    { while (arribo[i]  == 0)  skip;
                      arribo[i]  = 0;
                    }
              for [i = 1 to n] continuar[i]  = 1;
         }
}
```

Sincronización *Barrier*
Árboles

- **Problemas:**
  - Requiere un proceso (y procesador) extra.
  - El tiempo de ejecución del coordinador es proporcional a *n*.
- **Posible solución**:
  - Combinar las acciones de *Workers* y *Coordinador*, haciendo que cada *Worker* sea también *Coordinador*.
  - Por ejemplo, *Workers* en forma de árbol: las señales de arribo van hacia arriba en el árbol, y las de continuar hacia abajo ⇒ ***combining tree barrier*** (más eficiente para *n* grande).



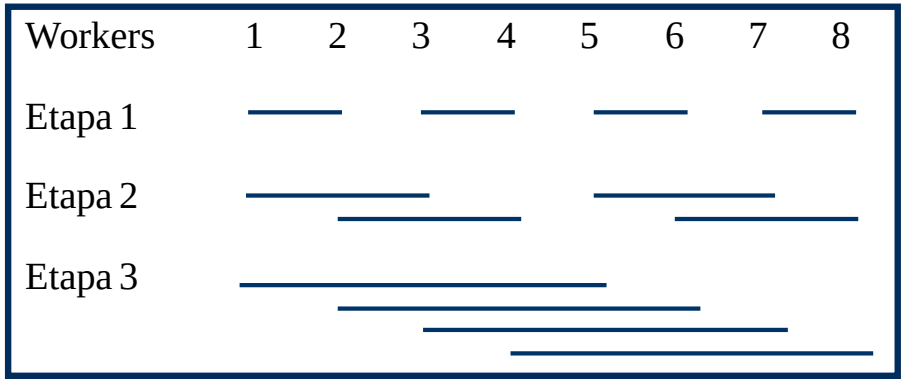
Sincronización *Barrier*

Barreras Simétrica

- En ***combining tree barrier*** los procesos juegan diferentes roles.
- Una ***Barrera Simétrica*** para *n* procesos se construye a partir de pares de barreras simples para dos procesos:

```
W[i]:: ⟨ await (arribo[i] == 0); ⟩        W[j]:: ⟨ await (arribo[j] == 0); ⟩
       arribo[i] = 1;                             arribo[j] = 1;
       ⟨ await (arribo[j] == 1); ⟩                ⟨ await (arribo[i] == 1); ⟩
       arribo[j] = 0;                             arribo[i] = 0;
```

- ¿Cómo se combinan para construir una barrera *n* proceso? *Worker[1:n]* arreglo de procesos. Si *n* es potencia de 2 ⇒ ***Butterfly Barrier.***



- $\boldsymbol{log_2 n}$ etapas: cada *worker* sincroniza con uno distinto en cada etapa.
- En la etapa ***s***, un worker sincroniza con otro a distancia $\boldsymbol{2^{s-1}}$.
- Cuando cada *worker* pasó $\boldsymbol{log_2 n}$ etapas, todos pueden seguir.

Sincronización *Barrier*

Barreras Simétrica – *Butterfly barrier*

```
int E = log(N);
int arribo[1:N] = ([N] 0);

process P[i=1..N]
{  int j;
   while (true)
      {  //Sección de código anterior a la barrera.
         //Inicio de la barrera
         for (etapa = 1; etapa <= E; etapa++)
            {   j =  (i-1) XOR (1<<(etapa-1));    //calcula el proceso con cual sincronizar
                while (arribo[i] == 1) → skip;
                arribo[i] = 1;
                while (arribo[j] == 0) → skip;
                arribo[j] = 0;
             }
         //Fin de la barrera
         //Sección de código posterior a la barrera.
      }
}
```

Defectos de la sincronización por ***busy waiting***

- Protocolos "***busy-waiting***": complejos y sin clara separación entre variables de sincronización y las usadas para computar resultados.
- Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos.
- Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso ***spinning*** puede ser usado de manera más productiva por otro proceso.

Necesidad de herramientas para diseñar protocolos de sincronización.

Programación Concurrente

Clase 4

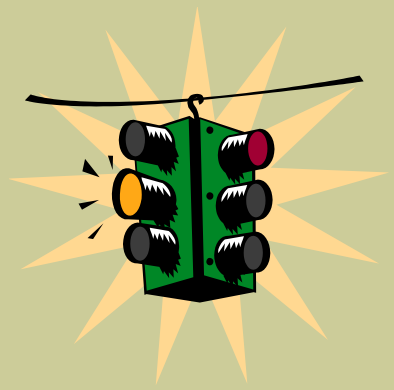
Facultad de Informática
UNLP

Links a los archivos con audio (formato MP4)

El archivo con la clase con audio está en formato MP4. En el link de abajo está el video comprimido en archivo RAR.

- Semáforos
  https://drive.google.com/uc?id=1MjOhte-Wv6nQh0V42bwiEJ-HlO2m7rEJ&export=download

Semáforos



Defectos de la sincronización por *Busy Waiting*

- ***Protocolos "busy-waiting":*** complejos y sin clara separación entre variables de sincronización y las usadas para computar resultados.
- Es difícil diseñar para probar corrección. Incluso la verificación es compleja cuando se incrementa el número de procesos.
- Es una técnica ineficiente si se la utiliza en multiprogramación. Un procesador ejecutando un proceso ***spinning*** puede ser usado de manera más productiva por otro proceso.

⇒***Necesidad de herramientas para diseñar protocolos de sincronización.***

Semáforos

Descriptos en 1968 por Dijkstra
(www.cs.utexas.edu/users/EWD/welcome.html)

Semáforo ⇒ instancia de un tipo de datos abstracto (o un objeto) con sólo 2 operaciones (métodos) *atómicas*: ***P*** y ***V*.**

Internamente el valor de un semáforo es un entero *no negativo:*

- ***V*** → Señala la ocurrencia de un evento (incrementa).
- ***P*** → Se usa para demorar un proceso hasta que ocurra un evento (decrementa).

- Analogía con la sincronización del tránsito para evitar colisiones.
- Permiten proteger *Secciones Críticas* y pueden usarse para implementar *Sincronización por Condición.*

Operaciones Básicas

- **Declaraciones**

  sem s; → NO. Si o si se deben inicializar en la declaración

  sem mutex = 1;

  sem fork[5] = ([5] 1);

- **Semáforo general (o *counting semaphore*)**

  ***P(s)*:** ⟨ await (s > 0) s = s-1; ⟩

  ***V(s)*:** ⟨ s = s+1; ⟩

- **Semáforo binario**

  ***P(b)*:** ⟨ await (b > 0) b = b-1; ⟩

  ***V(b)*:** ⟨ await (b < 1) b = b+1; ⟩

Si la implementación de la demora por operaciones ***P*** se produce sobre una ***cola***, las operaciones son ***fair***

(EN LA MATERIA NO SE PUEDE SUPONER ESTE TIPO DE IMPLEMENTACIÓN)

Problemas básicos y técnicas

Sección Crítica: *Exclusión Mutua*

```
bool lock=false;

process SC[i=1 to n]
{  while (true)
      {  ⟨await (not lock) lock = true;⟩
         sección crítica;
         lock = false;
         sección no crítica;
   }
}
```

Cambio de variable

```
bool free = true;

process SC[i=1 to n]
{  while (true)
      {  ⟨await (free) free = false;⟩
         sección crítica;
         free = true;
         sección no crítica;
   }
}
```

Podemos representar ***free*** con un entero, usar 1 para ***true*** y 0 para ***false*** ⇒ se puede asociar a las operaciones soportadas por los semáforos.

```
int free = 1;

process SC[i=1 to n]
{  while (true)
      {  ⟨await (free==1) free = 0;⟩
         sección crítica;
         free = 1;
         sección no crítica;
   }
}
```

Problemas básicos y técnicas

Sección Crítica: *Exclusión Mutua*

```
int free = 1;

process SC[i=1 to n]
{  while (true)
      {  <await (free==1) free = 0;>
         sección crítica;
         free = 1;
         sección no crítica;
      }
}
```

```
int free = 1;

process SC[i=1 to n]
{  while (true)
      {  <await (free > 0) free = free - 1;>
         sección crítica;
         <free = free + 1>;
         sección no crítica;
      }
}
```

```
sem free= 1;
process SC[i=1 to n]
{  while (true)
      {  P(free);
         sección crítica;
         V(free);
         sección no crítica;
      }
}
```

Es más simple que las soluciones ***busy waiting***.

¿Y si inicializo free= 0?

Problemas básicos y técnicas

Barreras: señalización de eventos

- **Idea:** un semáforo para cada *flag* de sincronización. Un proceso setea el *flag* ejecutando ***V***, y espera a que un *flag* sea seteado y luego lo limpia ejecutando ***P***.
- **Barrera para dos procesos:** necesitamos saber cada vez que un proceso llega o parte de la barrera ⇒ *relacionar los estados de los dos procesos.*

Semáforo de señalización ⇒ generalmente inicializado en 0. Un proceso señala el evento con V(s); otros procesos esperan la ocurrencia del evento ejecutando P(s).

```
sem llega1=0, llega2=0;

process Worker1
{ ..........
   V(llega1); P(llega2);
   ..........
}

process Worker2
{ ..........
   V(llega2); P(llega1);
   ..........
}
```

Puede usarse la barrera para dos procesos para implementar una ***butterfly barrier*** para ***n***, o sincronización con un coordinador central.

¿Qué sucede si los procesos primero hacen P y luego V?

Problemas básicos y técnicas

Productores y Consumidores: *semáforos binarios divididos*

Semáforo Binario Dividido (Split Binary Semaphore). Los semáforos binarios b_1, ..., b_n forman un SBS en un programa si el siguiente es un invariante global:

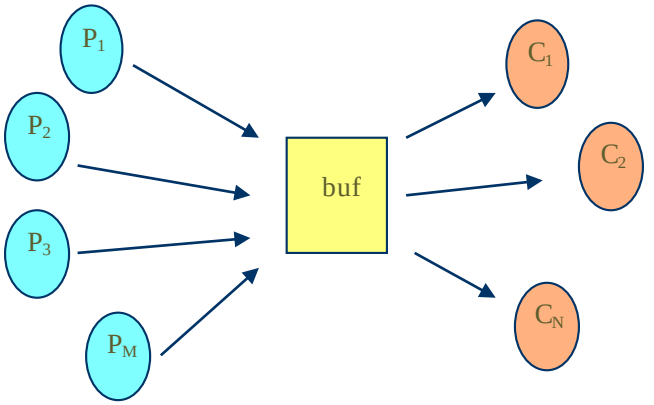
$$SPLIT: 0 \leq b_1, + ... + b_n \leq 1$$

- Los b_i pueden verse como un único semáforo binario ***b*** que fue dividido en ***n*** semáforos binarios.
- Importantes por la forma en que pueden usarse para implementar EM (en general la ejecución de los procesos inicia con un ***P*** sobre un semáforo y termina con un ***V*** sobre otro de ellos).
- Las sentencias entre el ***P*** y el ***V*** ejecutan con exclusión mutua.

Problemas básicos y técnicas

Productores y Consumidores: *semáforos binarios divididos*

Ejemplo: buffer unitario compartido con múltiples productores y consumidores. Dos operaciones: ***depositar*** y ***retirar*** que deben alternarse.




```
typeT buf;   sem vacio = 1, lleno = 0;

process Productor [i = 1 to M]
{ while(true)
     {  …
        producir mensaje datos
        P(vacio); buf = datos; V(lleno); #depositar
     }
}

process Consumidor[j  = 1 to N]
{ while(true)
     {  P(lleno); resultado = buf;  V(vacio); #retirar
        consumir  mensaje resultado
        …
     }
}
```

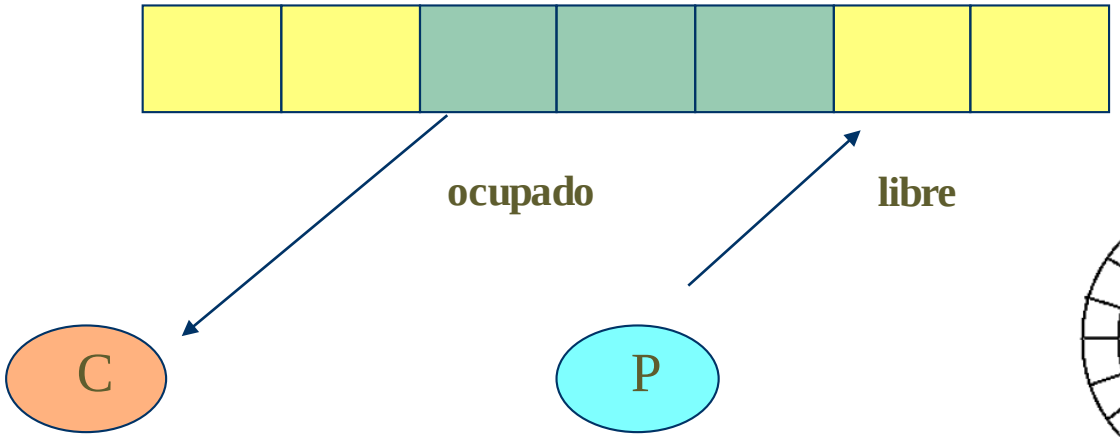
vacio y ***lleno*** (juntos) forman un ***"semáforo binario dividido"***.

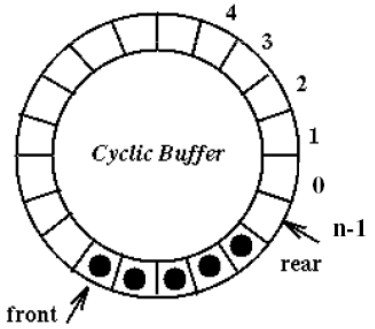
Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

Contadores de Recursos: cada semáforo cuenta el número de unidades libres de un recurso determinado. Esta forma de utilización es adecuada cuando los procesos compiten por recursos de ***múltiples unidades.***

Ejemplo: un buffer es una cola de mensajes depositados y aún no buscados. Existe UN productor y UN consumidor que ***depositan*** y ***retiran*** elementos del buffer.





Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

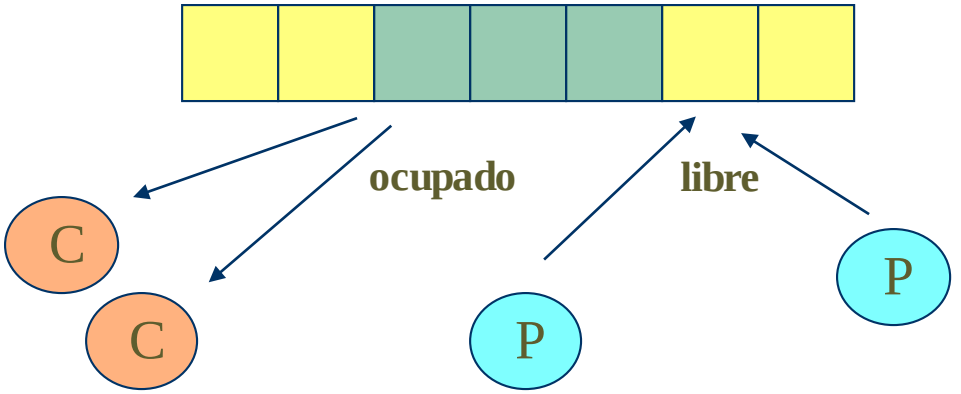
```
typeT buf[n];   int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;

process Productor
{ while(true)
    {  …
       producir mensaje datos
       P(vacio); buf[libre] = datos; libre = (libre+1) mod n;  V(lleno); #depositar
    }
}

process Consumidor
{ while(true)
    {  P(lleno); resultado = buf[ocupado];  ocupado = (ocupado+1) mod n; V(vacio); #retirar
       consumir  mensaje resultado
       …
    }
}
```

- ***vacio*** cuenta los lugares libres, y ***lleno*** los ocupados.
- ***depositar*** y ***retirar*** se pudieron asumir atómicas pues sólo hay un productor y un consumidor.
- ¿Qué ocurre si hay más de un productor y/o consumidor?

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

Si hay más de un productor y/o más de un consumidor, las operaciones de depositar y retirar en sí mismas son SC y deben ejecutar con Exclusión Mutua ¿Cuáles serían las consecuencias de no protegerlas?

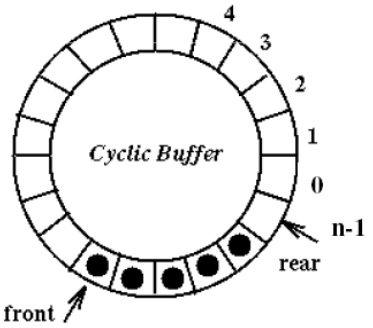


Si no se protege cada slot, podría retirarse dos veces el mismo dato o perderse datos al sobrescribirlo.

Problemas básicos y técnicas

Buffers Limitados: *Contadores de Recursos*

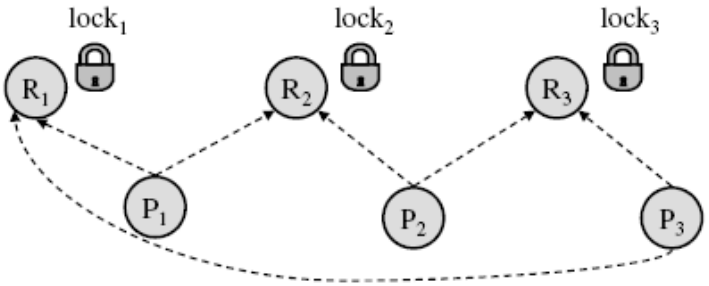
```
typeT buf[n];  int ocupado = 0, libre = 0;
sem  vacio = n, lleno = 0;
sem mutexD = 1, mutexR = 1;

process Productor [i = 1..M]
{ while(true)
     {  producir mensaje datos
        P(vacio);
        P(mutexD); buf[libre] = datos; libre = (libre+1) mod n; V(mutexD);
        V(lleno);
     }
}

process Consumidor  [i = 1..N]
{ while(true)
     {  P(lleno);
        P(mutexR); resultado = buf[ocupado];  ocupado = (ocupado+1) mod n; V(mutexR);
        V(vacio);
        consumir mensaje resultado
      }
}
```

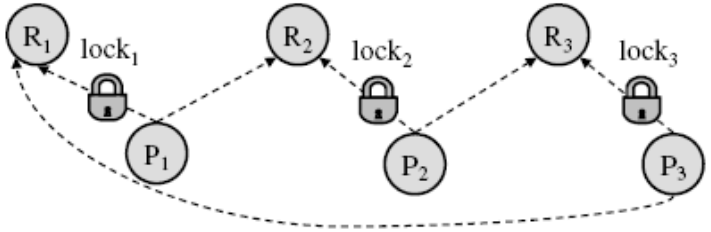


Problemas básicos y técnicas

Varios procesos compitiendo por varios recursos compartidos

- Problema de varios procesos (***P***) y varios recursos (***R***) cada uno protegido por un ***lock***.
- Un proceso debe adquirir los ***locks*** de todos los recursos que necesita.
- Puede caerse en *deadlock* cuando varios procesos compiten por conjuntos superpuestos de recursos.
- Por ejemplo: cada ***P[i]*** necesita ***R[i]*** y ***R[(i+1) mod n]*** $\Rightarrow$ ¿Cuándo se da el ***Deadlock***?



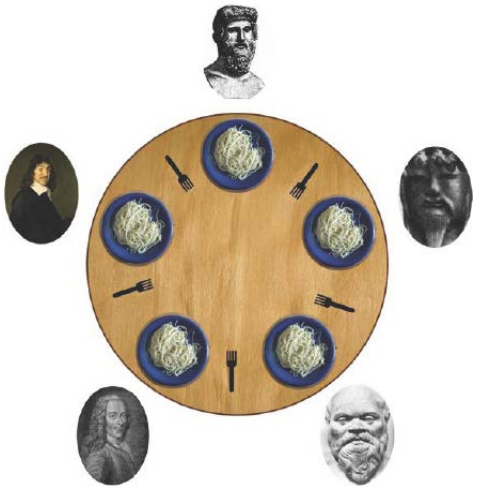


Problemas básicos y técnicas

Problema de los filósofos: *exclusión mutua selectiva*

- Problema de exclusión mutua entre procesos que compiten por el acceso a conjuntos superpuestos de variables compartidas.
- ***Problema de los filósofos:***

```
process Filosofo [i = 0 to 4]
{  while (true)
      {  adquiere tenedores;
         come;
         libera tenedores;
         piensa;
      }
}
```



- ***Cada tenedor es una SC***: puede ser tomado por un único filósofo a la vez ⇒ pueden representarse los tenedores por un arreglo de semáforos.
- Levantar un tenedor ⇒ ***P*** Bajar un tenedor ⇒ ***V***
- Cada filósofo necesita el tenedor izquierdo y el derecho.
- ¿Qué efecto puede darse si todos los filósofos hacen ***exactamente*** lo mismo?.

Problemas básicos y técnicas

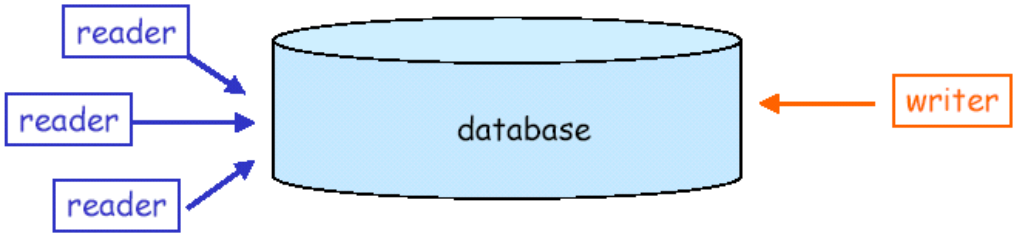
Problema de los filósofos: *exclusión mutua selectiva*

```
sem  tenedores [5] = {1,1,1,1,1};

process Filososfos[i = 0..3]
{ while(true)
     {  P(tenedor[i]);  P(tenedor[i+1]);
        comer;
        V(tenedor[i]);  V(tenedor[i+1]);
     }
}

process Filososfos[4]
{ while(true)
     {  P(tenedor[0]);  P(tenedor[4]);
        comer;
        V(tenedor[0]);  V(tenedor[4]);
     }
}
```

Problemas básicos y técnicas

Lectores y escritores

- ***Problema:*** dos clases de procesos (*lectores* y *escritores*) comparten una Base de Datos. El acceso de los *escritores* debe ser exclusivo para evitar interferencia entre transacciones. Los *lectores* pueden ejecutar concurrentemente entre ellos si no hay escritores actualizando.



- Procesos asimétricos y, según el scheduler, con diferente prioridad.
- Es también un problema de ***exclusión mutua selectiva***: clases de procesos compiten por el acceso a la BD.
- Diferentes soluciones:
  - Como problema de exclusión mutua.
  - Como problema de sincronización por condición.

Problemas básicos y técnicas

Lectores y escritores: *como problema de exclusión mutua*

- Los escritores necesitan acceso mutuamente exclusivo.
- Los lectores (como grupo) necesitan acceso exclusivo con respecto a cualquier escritor.

```
sem rw = 1;

process Lector [i = 1 to M]
{  while(true)
      { …
        P(rw);
        lee la BD;
        V(rw);
      }
}

process Escritor [j = 1 to N]
{  while(true)
      { …
        P(rw);
        escribe la BD;
        V(rw);
       }
}
```

No hay concurrencia entre lectores

- Los lectores (como grupo) necesitan bloquear a los escritores, pero sólo el primero necesita tomar el *lock* ejecutando *P(rw)*.
- Análogamente, sólo el último lector debe hacer *V(rw)*.

Problemas básicos y técnicas

Lectores y escritores: *como problema de exclusión mutua*

```
int nr = 0;          # número de lectores activos
sem rw = 1;          # bloquea el acceso a la BD
```

```
process Escritor [j = 1 to N]
{  while(true)
        { …
          P(rw);
          escribe la BD;
          V(rw);
         }
}
```

```
process Lector [i = 1 to M]
{ while(true)
      { …
        <  nr = nr + 1;  if (nr == 1) P(rw); >
        lee la BD;
        <  nr = nr - 1; if (nr == 0) V(rw);  >
      }
}
```

Problemas básicos y técnicas

Lectores y escritores: *como problema de exclusión mutua*

```
int nr = 0;          # número de lectores activos
sem rw = 1;          # bloquea el acceso a la BD
sem mutexR= 1;       # bloquea el acceso de los lectores a nr
```

```
process Escritor [j = 1 to N]
{  while(true)
        { …
          P(rw);
          escribe la BD;
          V(rw);
        }
}
```

```
process Lector [i = 1 to M]
{ while(true)
      { …
        P(mutexR);
        nr = nr + 1;
        if (nr == 1) P(rw);
        V(mutexR);
        lee la BD;
        P(mutexR);
        nr = nr - 1;
        if (nr == 0) V(rw);
        V(mutexR);
      }
}
```

Problemas básicos y técnicas

Lectores y escritores: *sincronización por condición*

- Solución anterior ⇒ preferencia a los lectores ⇒ no es ***fair***.
- Otro enfoque ⇒ introduce la técnica ***passing the baton***: emplea SBS para brindar exclusión y despertar procesos demorados.
- Puede usarse para implementar ***await*** arbitrarios, controlando de forma precisa el orden en que los procesos son despertados
- En este caso, pueden contarse (por medio de *nr* y *nw*) los procesos de cada clase intentando acceder a la BD, y luego restringir el valor de los contadores. ¿Cuáles son los estados buenos y malos de ***nr*** y ***nw***?

```
int nr = 0, nw = 0;

process Lector [i = 1 to M]
{ while(true)
    { …
      ⟨ await (nw == 0) nr = nr + 1; ⟩
      lee la BD;
      ⟨ nr = nr – 1; ⟩
    }
}
```

```
process Escritor [j = 1 to N]
{ while(true)
    { …
      ⟨ await (nr==0 and nw==0) nw=nw+1; ⟩
      escribe la BD;
      ⟨ nw = nw – 1; ⟩
    }
}
```

Problemas básicos y técnicas

Técnica Passing de Baton

- En algunos casos, await puede ser implementada directamente usando semáforos u otras operaciones primitivas. *Pero no siempre…*
- En el caso de las guardas de los await en la solución anterior, se superponen en que el protocolo de entrada para escritores necesita que tanto **nw** como **nr** sean 0, mientras para lectores sólo que **nw** sea 0.
- Ningún semáforo podría discriminar entre estas condiciones → ***Passing the baton.***

Passing the baton: técnica general para implementar sentencias ***await.***

Cuando un proceso está dentro de una SC mantiene el *baton* (*testimonio*, *token*) que significa permiso para ejecutar.

Cuando el proceso llega a un ***SIGNAL*** (sale de la SC), pasa el *baton* (control) a otro proceso. Si ningún proceso está esperando por el *baton* (es decir esperando entrar a la SC) el *baton* se libera para que lo tome el próximo proceso que trata de entrar.

Problemas básicos y técnicas

Técnica Passing de Baton

La sincronización se expresa con sentencias atómicas de la forma:

$$\mathbf{F_1 : \langle\ S_i\ \rangle} \quad \mathbf{o} \quad \mathbf{F_2 : \langle\ await\ (B_j)\ S_j\ \rangle}$$

Puede hacerse con semáforos binarios divididos (SBS).

e semáforo binario inicialmente ***1*** (controla la entrada a sentencias atómicas).

Utilizamos un semáforo b_j y un contador d_j cada uno con guarda diferente B_j; todos inicialmente *0*.

b_j se usa para demorar procesos esperando que B_j sea *true*.
d_j es un contador del número de procesos demorados sobre b_j.

e y los b_j se usan para formar un SBS: a lo sumo uno a la vez es ***1***, y cada camino de ejecución empieza con un ***P*** y termina con un único ***V***.

Problemas básicos y técnicas

Técnica Passing de Baton

⟨ S_i ⟩

F_1 : **P(e);**
S_i;
SIGNAL;

⟨ **await** **(B_j) S_j** ⟩

F_2 : **P(e);**
if (not B_j) {$d_j = d_j + 1$; V(e); P(b_j); }
S_j;
SIGNAL

SIGNAL: **if (B_1 and $d_1 > 0$) {$d_1 = d_1 - 1$; V(b_1)}**
□ ...
□ (B_n and $d_n > 0$) {$d_n = d_n - 1$; V(b_n)}
□ else V(e);
fi

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing de Baton*

```
int nr = 0, nw = 0;

process Lector [i = 1 to M]
{ while(true)
    { …
      ⟨ await (nw == 0) nr = nr + 1; ⟩
      lee la BD;
      ⟨ nr = nr – 1; ⟩
    }
}


process Escritor [j = 1 to N]
{ while(true)
    { …
      ⟨ await (nr==0 and nw==0) nw=nw+1; ⟩
      escribe la BD;
      ⟨ nw = nw – 1; ⟩
     }
}
```

```
int nr = 0, nw = 0, dr = 0, dw = 0;
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M]
{ while(true) {
        P(e);
        if (nw > 0){dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        SIGNAL1 ;
        lee la BD;
        P(e); nr = nr – 1; SIGNAL2 ;
    }
}

process Escritor [j = 1 to N]
{ while(true) {
        P(e);
        if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
        nw = nw + 1;
        SIGNAL3 ;
        escribe la BD;
        P(e); nw = nw – 1; SIGNAL4 ;
    }
}
```

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing de Baton*

```
int nr = 0, nw = 0, dr = 0, dw = 0;
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M]
{ while(true)
    {  P(e);
       if (nw > 0){dr = dr+1; V(e); P(r); }
       nr = nr + 1;
       SIGNAL1 ;
       lee la BD;
       P(e); nr = nr – 1; SIGNAL2 ;
    }
}

process Escritor [j = 1 to N]
{ while(true)
    {  P(e);
       if (nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
       nw = nw + 1;
       SIGNAL3 ;
       escribe la BD;
       P(e); nw = nw – 1; SIGNAL4 ;
    }
}
```

El rol de los **SIGNAL$_i$** es el de señalizar ***exactamente*** a uno de los semáforos $\Rightarrow$ los procesos se van pasando el ***baton***.

SIGNAL$_i$ es una abreviación de:

```
if (nw == 0 and dr > 0)
      {dr = dr - 1; V(r);}
elsif (nr == 0 and nw == 0 and dw > 0)
      {dw = dw - 1; V(w);}
else  V(e);
```

Algunos de los SIGNAL se pueden simplificar.

Problemas básicos y técnicas

Lectores y escritores: *Técnica Passing de Baton*

```
int nr = 0, nw = 0, dr = 0, dw = 0;          sem e = 1, r = 0, w = 0:
```

```
process Lector [i = 1 to M]
{  while(true)
      {  P(e);
        if (nw > 0) {dr = dr+1; V(e); P(r); }
        nr = nr + 1;
        if (dr > 0) {dr = dr – 1; V(r); }
        else V(e);
        lee la BD;
        P(e);
        nr = nr – 1;
        if (nr == 0 and dw > 0)
              {dw = dw – 1; V(w); }
        else V(e);
       }
}
```

```
process Escritor [j = 1 to N]
{ while(true)
   {  P(e);
      if (nr > 0 or  nw > 0)
              {dw=dw+1; V(e); P(w);}
      nw = nw + 1;
      V(e);
      escribe la BD;
      P(e);
      nw = nw – 1;
      if (dr > 0) {dr = dr - 1; V(r); }
      elseif (dw > 0) {dw = dw - 1; V(w); }
      else V(e);
   }
}
```

Da preferencia a los lectores ⇒ ¿Cómo puede modificarse?

Alocación de Recursos y Scheduling

Problema: decidir cuándo se le puede dar a un proceso determinado acceso a un recurso.

Recurso: cualquier objeto, elemento, componente, dato, SC, por la que un proceso puede ser demorado esperando adquirirlo.

Definición del problema: procesos que compiten por el uso de unidades de un recurso compartido (cada unidad está ***libre*** o ***en uso***).

request (parámetros): ⟨await (request puede ser satisfecho) tomar unidades;⟩

release (parámetros): ⟨retornar unidades;⟩

- Puede usarse Passing the Baton:

request (parámetros): P(e);
if (request no puede ser satisfecho) DELAY;
tomar las unidades;
SIGNAL;

release (parámetros): P(e);
retornar unidades;
SIGNAL;

Alocación de Recursos y Scheduling

Alocación Shortest-Job-Next (SJN)

- Varios procesos que compiten por el uso de un recurso compartido *de **una sola unidad***.
- **request** (tiempo,id). Si el recurso está libre, es alocado inmediatamente al proceso ***id***; sino, el proceso ***id*** se demora.
- **release** ( ). Cuando el recurso es liberado, es alocado al proceso demorado (si lo hay) con el mínimo valor de ***tiempo***. Si dos o más procesos tienen el mismo valor de ***tiempo***, el recurso es alocado al que esperó más.
- SJN minimiza el tiempo promedio de ejecución, aunque *es **unfair*** (¿por qué?). Puede mejorarse con la técnica de ***aging*** (dando preferencia a un proceso que esperó mucho tiempo).
- Para el caso general de alocación de recursos (NO SJN):

```
bool libre = true;
request (tiempo,id): ⟨await (libre) libre = false;⟩
release (): ⟨libre = true;⟩
```

Alocación de Recursos y Scheduling

Alocación Shortest-Job-Next (SJN)

- En SJN, un proceso que invoca a *request* debe demorarse hasta que el recurso esté libre y su pedido sea el próximo en ser atendido de acuerdo a la política. El parámetro tiempo entra en juego sólo si un pedido debe ser demorado.

```
request (tiempo, id):
        P(e);
        if (not libre) DELAY;
        libre = false;
        SIGNAL;

release ( ):
        P(e);
        libre = true;
        SIGNAL;
```

- En ***DELAY*** un proceso:
  - Inserta sus parámetros en un conjunto, cola o lista de espera (*pares*).
  - Libera la SC ejecutando V(e).
  - Se demora en un semáforo hasta que *request* puede ser satisfecho.
- En ***SIGNAL*** un proceso:
  - Cuando el recurso es liberado, si pares no está vacío, el recurso es asignado a un proceso de acuerdo a SJN.

- Cada proceso tiene una condición de demora distinta, dependiendo de su posición en ***pares***. El proceso ***id*** se demora sobre el semáforo ***b[id]***.

Alocación de Recursos y Scheduling

Alocación Shortest-Job-Next (SJN)

```
bool libre = true;  Pares = set of (int, int) = ∅;  sem e = 1, b[n] = ([n] 0);

request(tiempo,id):   P(e);
                      if (! libre){ insertar (tiempo, id) en Pares; V(e); P(b[id]); }
                      libre = false;
                      V(e);

release( ):           P(e);
                      libre = true;
                      if (Pares ≠ ∅ ) { remover el primer par (tiempo,id) de Pares; V(b[id]); }
                      else V(e);
```

s es un ***semáforo privado*** si exactamente un proceso ejecuta operaciones ***P*** sobre ***s***. Resultan útiles para señalar procesos individuales. Los semáforos ***b[id]*** son de este tipo.

Alocación de Recursos y Scheduling

Alocación Shortest-Job-Next (SJN)

```
bool libre = true;  Pares = set of (int, int) = ∅;  sem e = 1, b[n] = ([n] 0);

Process Cliente [id: 1..n]
{  int sig;
   //Trabaja
   tiempo = //determina el tiempo de uso del recurso//
   P(e);
   if (! libre) { insertar (tiempo, id) en Pares;
                  V(e);
                  P(b[id]);
                }
   libre = false;
   V(e);
   //USA EL RECURSO
   P(e);
   libre = true;
   if (Pares ≠ ∅ ) { remover el primer par (tiempo, sig) de Pares;
                     V(b[sig]);
                   }
   else V(e);
}
```

¿Que modificaciones deberían realizarse para respetar el orden de llegada?

¿Que modificaciones deberían realizarse para generalizar la solución a recursos de múltiple unidad?

Programación Concurrente

Clase 5

Facultad de Informática

UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Funcionamiento de los monitores:
https://drive.google.com/uc?id=1KL17Ve0K6uW4Y5m3BREz0ZcRaBnkXF97&export=download

- Ejemplos y técnicas de programación con monitores:
https://drive.google.com/uc?id=13NfxOZgijYu8b7h-RdxBnoI4YcHx-NWo&export=download

Monitores

Conceptos básicos

Semáforos ⇒

- Variables compartidas globales a los procesos.
- Sentencias de control de acceso a la ***sección crítica*** dispersas en el código.
- Al agregar procesos, se debe verificar acceso correcto a las ***variables compartidas***.
- Aunque ***exclusión mutua*** y ***sincronización por condición*** son conceptos distintos, se programan de forma similar.

Monitores: módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.

Mecanismo de abstracción de datos:

- Encapsulan las representaciones de recursos.
- Brindan un conjunto de operaciones que son los únicos medios para manipular esos recursos.

Contiene variables que almacenan el estado del recurso y procedimientos que implementan las operaciones sobre él.

Conceptos básicos

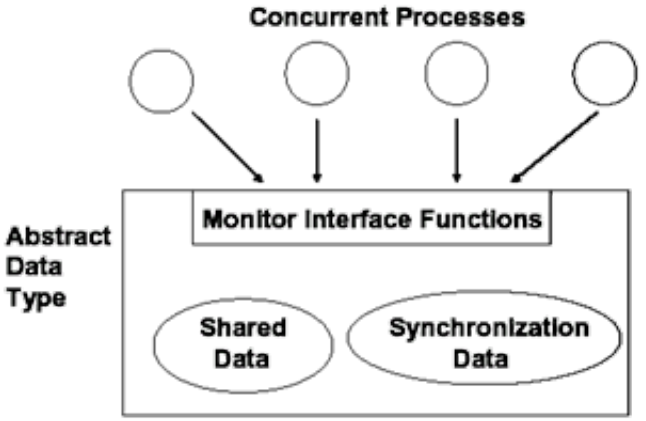
Exclusión Mutua ⇒ implícita asegurando que los ***procedures*** en el mismo monitor no ejecutan concurrentemente.

Sincronización por Condición ⇒ explícita con variables condición.

Programa Concurrente ⇒ procesos activos y monitores pasivos. Dos procesos interactúan invocando ***procedures*** de un monitor.

Ventajas:

- Un proceso que invoca un ***procedure*** puede ignorar cómo está implementado.
- El programador del monitor puede ignorar cómo o dónde se usan los ***procedures***.



Notación

- Un monitor agrupa la representación y la implementación de un recurso compartido, se distingue a un monitor de un TAD en procesos secuenciales en que es compartido por procesos que ejecutan concurrentemente. Tiene ***interfaz*** y ***cuerpo***:
  - La ***interfaz*** especifica operaciones que brinda el recurso.
  - El ***cuerpo*** tiene variables que representan el estado del recurso y *procedures* que implementan las operaciones de la *interfaz*.
- Sólo los nombres de los ***procedures*** son visibles desde afuera. Sintácticamente, los llamados al monitor tienen la forma:

  $$NombreMonitor.op_i\ (argumentos)$$

- Los ***procedures*** pueden acceder sólo a variables permanentes, sus variables locales, y parámetros que le sean pasados en la invocación.
- El programador de un monitor no puede conocer a priori el orden de llamado.

Notación

monitor ***NombreMonitor*** {
  declaraciones de variables permanentes;
  código de inicialización

  procedure op_1 (par. $formales_1$)
    { cuerpo de op_1
     }
  .....
  procedure op_n (par. $formales_n$)
    { cuerpo de op_n
     }
}

Ejemplo de uso de monitores

Tenemos 5 procesos empleados que continuamente hacen algún producto. Hay un proceso coordinador que cada cierto tiempo debe ver la cantidad total de productos hechos.

```
process empleado[id: 0..4] {
    while (true)
     { ……
       TOTAL.incrementar();
       ……
     }
}
```

```
process coordinador{
    int c;
    while (true)
     { ……
       TOTAL.verificar(c);
       ……
     }
}
```

```
monitor TOTAL {
   int cant = 0;

   procedure incrementar ()
     {   cant = cant+1;
      }

   procedure verificar (R: out int)
     {   R = cant;
      }
}
```

Ejemplo de uso de monitores

Tenemos dos procesos A y B, donde A le debe comunicar un valor a B (múltiples veces).

```
process A {
    bool ok;
    int aux;
    while (true)  { --Genera valor a enviar en aux
        ok = false;
        while (not ok) → Buffer.Enviar(aux, ok);
        ……
    }
}
```

BUSY WAITING

```
process B {
    bool ok;
    int aux;
    while (true)  { ……
        ok = false;
        while (not ok) → Buffer.Recibir(aux, ok);
        --Trabaja con en vlor aux recibido
    }
}
```

```
monitor Buffer{
   int dato;
   bool hayDato = false;

   procedure Enviar (D: in int; Ok: out bool)
     {   Ok = not hayDato;
         if (Ok) {  dato = D;
                    hayDato  = true;
                 }
     }

   procedure Recibir (R: out int; Ok: out bool)
     {   Ok = hayDato;
         if (Ok) {  R = dato;
                    hayDato  = false;
                 }
     }
}
```

Sincronización

La ***sincronización por condición*** es programada explícitamente con ***variables condición*** → cond cv;

El valor asociado a ***cv*** es una cola de procesos demorados, ***no visible directamente*** al programador. Operaciones sobre las *variables condición*:

- **wait(cv)** → el proceso se demora al final de la cola de ***cv*** y deja el acceso exclusivo al monitor.
- **signal(cv)** → despierta al proceso que está al frente de la cola (si hay alguno) y lo saca de ella. ***El proceso despertado recién podrá ejecutar cuando readquiera el acceso exclusivo al monitor.***
- **signal_all(cv)** → despierta todos los procesos demorados en ***cv***, quedando vacía la cola asociada a ***cv***.

➢ Disciplinas de señalización:

- Signal and continued ⇒ ***es el utilizado en la materia.***
- Signal and wait.

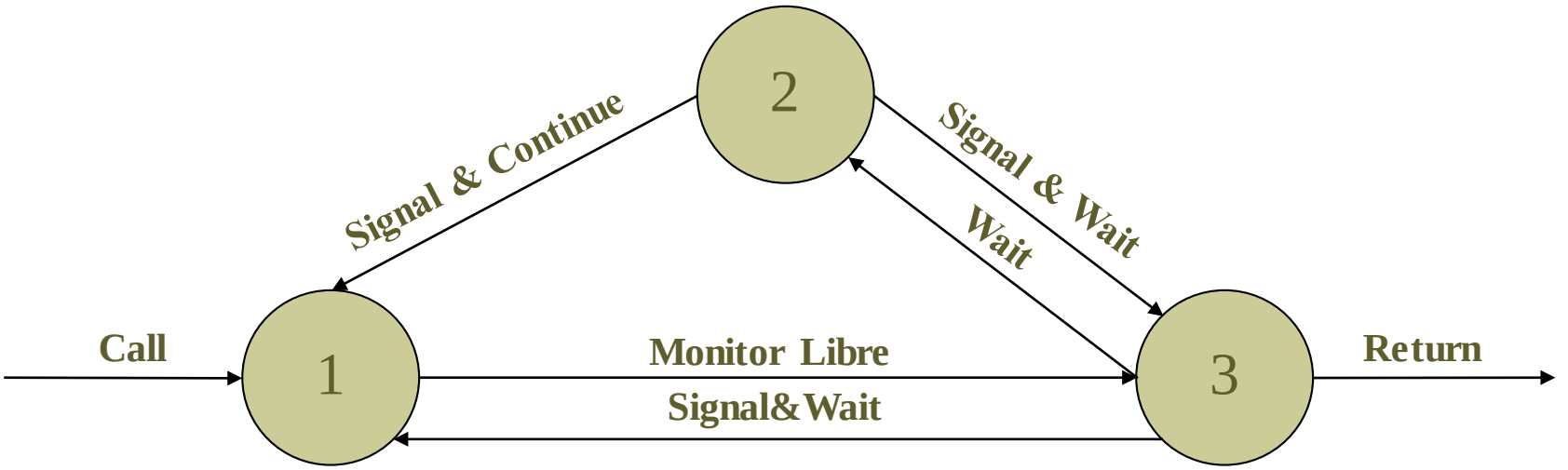
Operaciones adicionales

Operaciones adicionales que NO SON USADAS EN LA PRÁCTICA sobre las *variables condición*:

- **empty(cv)** → retorna *true* si la cola controlada por ***cv*** está vacía.
- **wait(cv, rank)** → el proceso se demora en la cola de ***cv*** en orden ascendente de acuerdo al parámetro *rank* y deja el acceso exclusivo al monitor.
- **minrank(cv)** → función que retorna el mínimo ranking de demora.

Sincronización

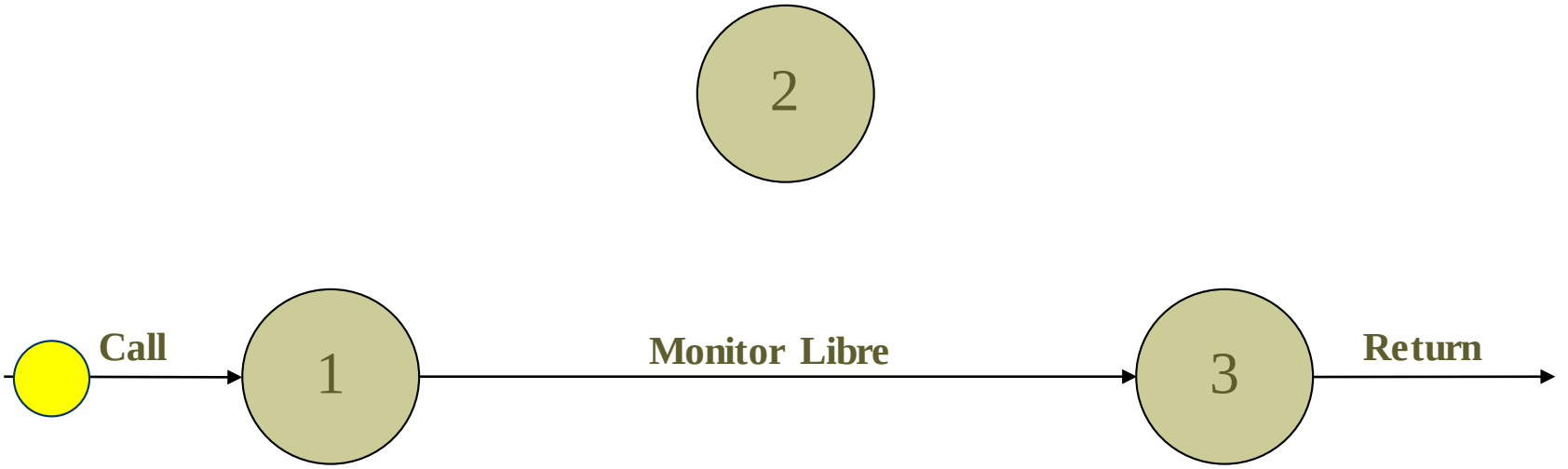
Signal and continue* vs. *Signal and Wait



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

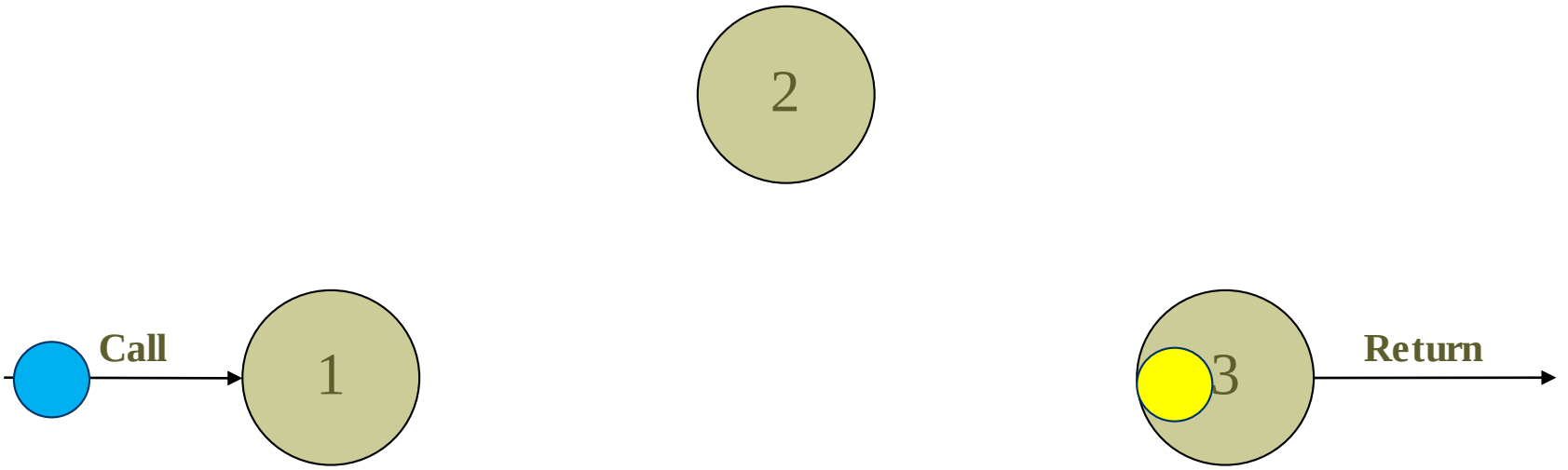
Llamado – Monitor Libre



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

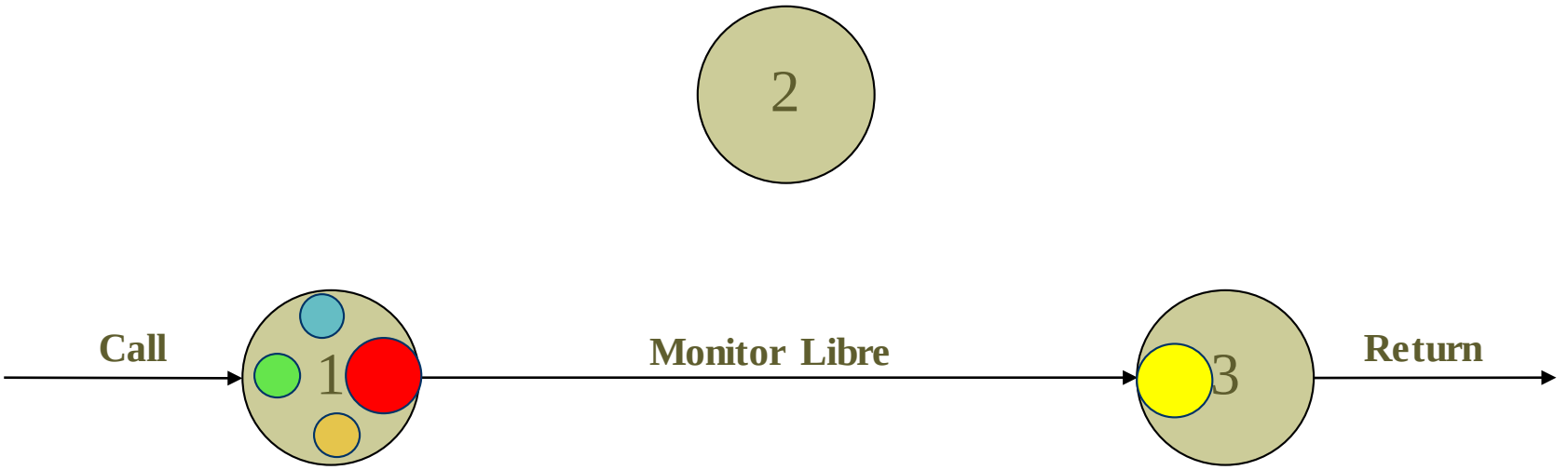
Llamado – Monitor Ocupado



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

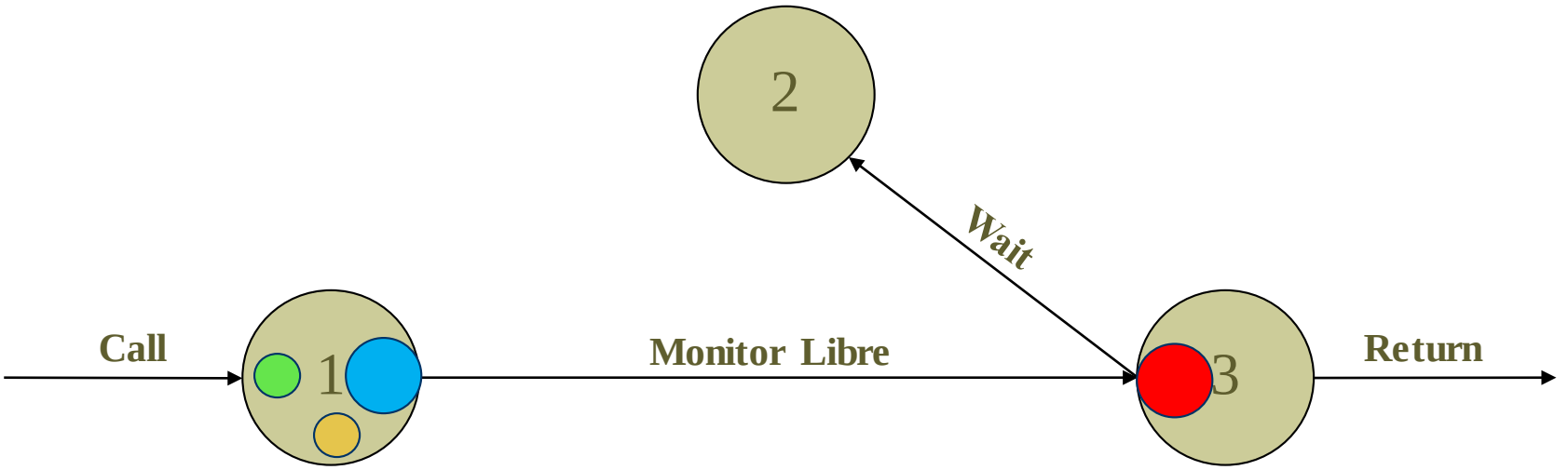
Liberación del Monitor



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

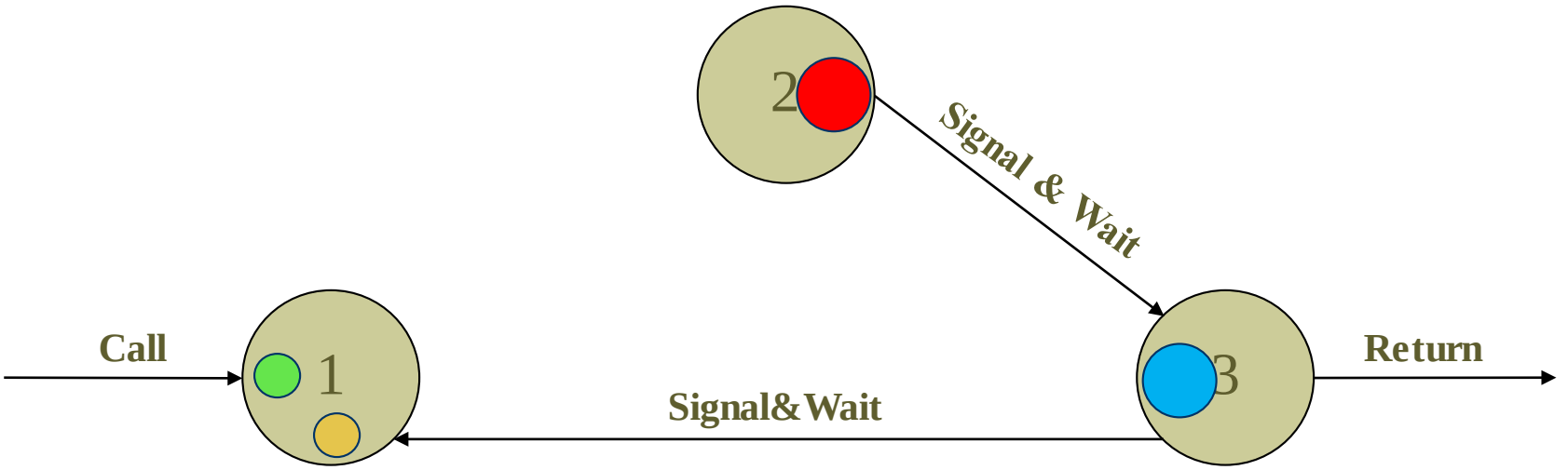
Wait



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

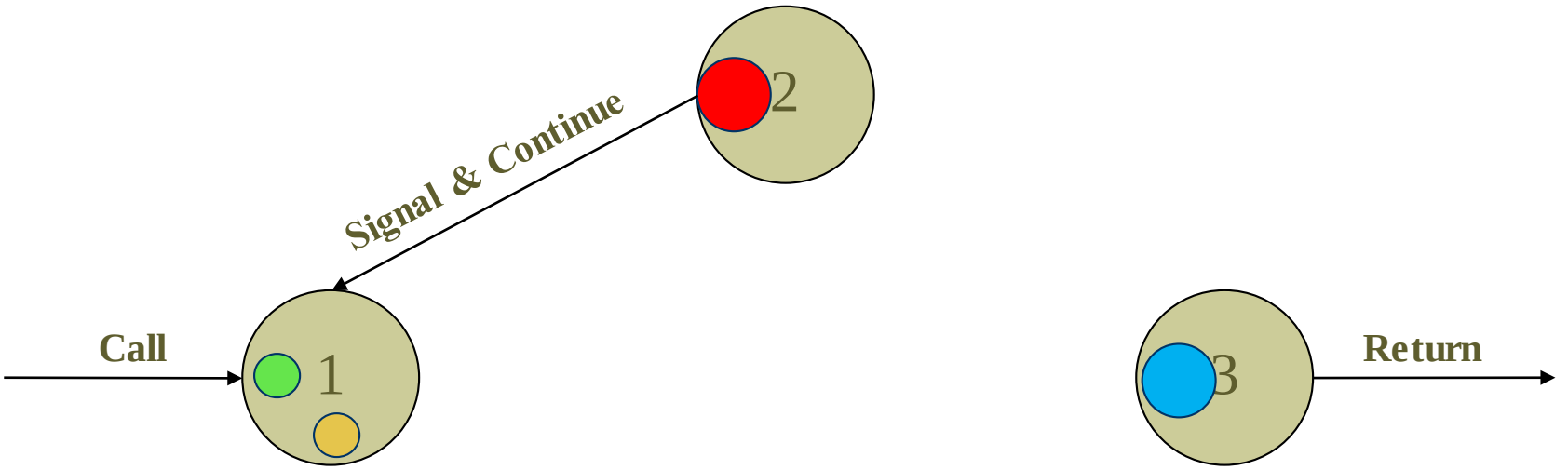
Signal - Disciplina Signal and Wait



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

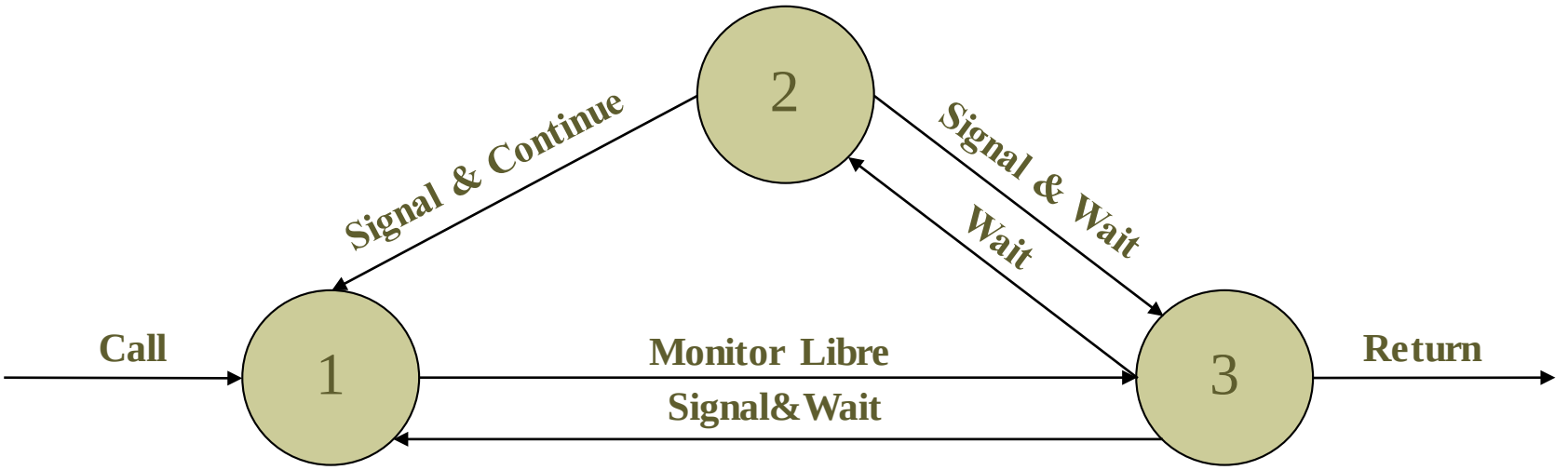
Signal - Disciplina Signal and Continue



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

Signal and continue* vs. *Signal and Wait



1- Espera acceso al monitor.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

Resumen: *diferencia entre las disciplinas de señalización*

- ***Signal and Continued*:** el proceso que hace el *signal* continua usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al *wait*).

- ***Signal and Wait*:** el proceso que hace el *signal* pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al *wait*.

Sincronización

Resumen: *diferencia entre wait/signal con P/V*

WAIT	P
El proceso siempre se duerme	El proceso sólo se duerme si el semáforo es 0.

SIGNAL	V
Si hay procesos dormidos despierta al primero de ellos. En caso contrario no tiene efecto posterior.	Incrementa el semáforo para que un proceso dormido o que hará un P continue. No sigue ningún orden al despertarlos.

Ejemplo de uso de monitores

Tenemos dos procesos A y B, donde A le debe comunicar un valor a B (múltiples veces).

```
process A {
   int aux;
   while (true)
      { --Genera valor a enviar en aux
        Buffer.Enviar (aux);
        ……
      }
}
```

```
process B {
   int aux;
   while (true)
       { …….
         Buffer.Recibir (aux);
         --Trabaja con en vlor aux recibido
        }
}
```

```
monitor Buffer{
   int dato;
   bool hayDato = false;
   cond P, C;

   procedure Enviar (D: in int)
     {   if (hayDato) → wait (P);
         dato = D;
         hayDato = true;
         signal (C);
     }

   procedure Recibir (R: out int)
     {  if (not hayDato) → wait (C);
        R = dato;
        hayDato = false;
        signal (P);
     }
}
```

Ejemplos y técnicas

Ejemplo

Simulación de semáforos: *condición básica*

```
monitor Semaforo
{  int s = 1;   cond pos;

   procedure  P ()
      { if (s == 0) wait(pos);
        s = s-1;
      };

   procedure V ()
      { s = s+1;
        signal(pos);
      };
};
```

Puede quedar el semáforo con un valor menor a 0 (no cumple las propiedades de los semáforos).

```
monitor Semaforo
{  int s = 1;   cond pos;

   procedure  P ()
      { while (s == 0) wait(pos);
        s = s-1;
      };

   procedure V ()
      { s = s+1;
        signal(pos);
      };
};
```

¿Qué diferencia hay con los semáforos?

¿Que pasa si se quiere que los procesos pasen el P en el orden en que llegan?

Técnicas de Sincronización

Simulación de semáforos: *Passing the Conditions*

Simulación de Semáforos

```
monitor Semaforo
{  int s = 1;   cond pos;

   procedure  P ()
      { if (s == 0) wait(pos)
        else s = s-1;
      };

   procedure V ()
     { if (empty(pos) ) s = s+1
       else signal(pos);
     };
};
```

Como resolver este problema al no contar con la sentencia *empty*.

```
monitor Semaforo
{  int s = 1, espera = 0;  cond pos;

   procedure  P ()
      { if (s == 0) { espera ++; wait(pos);}
        else  s = s-1;
      };

   procedure V ()
     { if (espera == 0 ) s = s+1
       else { espera --; signal(pos);}
     };
};
```

Técnicas de Sincronización

Alocación SJN: *Wait con Prioridad*

Alocación SJN

```
monitor Shortest_Job_Next
{ bool libre = true;
  cond turno;

  procedure request (int tiempo)
     { if (libre) libre = false;
       else wait (turno, tiempo);
     };

  procedure release ()
     { if (empty(turno)) libre = true
       else signal(turno);
     };
}
```

- Se usa ***wait*** con prioridad para ordenar los procesos demorados por la cantidad de tiempo que usarán el recurso.
- Se usa ***empty*** para determinar si hay procesos demorados.
- Cuando el recurso es liberado, si hay procesos demorados se despierta al que tiene mínimo ***rank***.
- ***Wait*** no se pone en un loop pues la decisión de cuándo puede continuar un proceso la hace el proceso que libera el recurso.

¿Como resolverlo sin wait con prioridad?

Técnicas de Sincronización

Alocación SJN: *Variables Condición Privadas*

- Se realiza Passing the Condition, manejando el orden explícitamente por medio de una cola ordenada y variables condición privadas.

```
monitor  Shortest_Job_Next
{ bool libre = true;
   cond turno[N];
   cola  espera;

   procedure request (int id, int tiempo)
       { if (libre) libre = false
          else { insertar_ordenado(espera, id, tiempo);
                   wait (turno[id]);
                 };
       };

   procedure release ()
      { if (empty(espera)) libre = true
        else   { sacar(espera, id);
                   signal(turno[id]);
                 };
      };
}
```

Técnicas de Sincronización

Buffer Limitado: *Sincronización por Condición Básica*

Buffer Limitado

```
monitor Buffer_Limitado
{  typeT buf[n];
   int ocupado = 0, libre = 0;  cantidad = 0;
   cond not_lleno, not_vacio;

   procedure depositar(typeT datos)
      { while (cantidad == n) wait (not_lleno);
        buf[libre] = datos;
        libre = (libre+1) mod n;
        cantidad++;
        signal(not_vacio);
      }

   procedure retirar(typeT &resultado)
      { while (cantidad == 0) wait(not_vacio);
        resultado=buf[ocupado];
        ocupado=(ocupado+1) mod n;
        cantidad--;
        signal(not_lleno);
      }
}
```

Técnicas de Sincronización

Lectores y escritores: *Broadcast Signal*

Lectores y escritores

```
monitor Controlador_RW
{  int nr = 0, nw = 0;
   cond  ok_leer, ok_escribir
   procedure pedido_leer( )
      { while (nw > 0) wait (ok_leer);
        nr = nr + 1;
       }
   procedure libera_leer( )
      { nr = nr - 1;
        if (nr == 0) signal (ok_escribir);
       }
   procedure pedido_escribir( )
       { while (nr>0 OR nw>0) wait (ok_escribir);
        nw = nw + 1;
       }
   procedure libera_escribir( )
      { nw = nw - 1;
        signal (ok_escribir);
        signal_all (ok_leer);
       }
}
```

- El monitor arbitra el ***acceso a la BD***.
- Los procesos dicen cuándo quieren acceder y cuándo terminaron ⇒ requieren un monitor con 4 procedures:
  - pedido_leer
  - libera_leer
  - pedido_escribir
  - libera_escribir

Técnicas de Sincronización

Lectores y escritores: *Passing the Condition*

Otra solución al problema de lectores y escritores

```
monitor Controlador_RW
{  int nr = 0, nw = 0, dr = 0, dw = 0;
   cond  ok_leer, ok_escribir

   procedure pedido_leer( )
    { if (nw > 0)
                { dr = dr +1;
                  wait (ok_leer);
                }
      else  nr = nr + 1;
    }

   procedure libera_leer( )
    { nr = nr - 1;
      if (nr == 0 and dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                  nw = nw +1;
                }
    }

   procedure pedido_escribir( )
      { if (nr>0 OR nw>0)
                { dw = dw +1;
                  wait (ok_escribir);
                }
        else  nw = nw + 1;
      }

    procedure libera_escribir( )
     { if (dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                }
       else { nw = nw -1;
              if (dr > 0)
                   { nr = dr;
                     dr = 0;
                     signal_all (ok_leer);
                   }
            }
      }
}
```

Técnicas de Sincronización

Diseño de un reloj lógico: *Covering conditions*

```
monitor Timer
{  int hora_actual = 0;
   cond chequear;

   procedure demorar(int intervalo)
    {  int hora_de_despertar;
       hora_de_despertar=hora_actual+intervalo;
       while (hora_de_despertar>hora_actual)
                wait(chequear);
    }

   procedure tick( )
     { hora_actual = hora_actual + 1;
       signal_all(chequear);
      }
}
```

Diseño de un reloj lógico

- ***Timer*** que permite a los procesos dormirse una cantidad de unidades de tiempo.
- Ejemplo de controlador de recurso (reloj lógico) con dos operaciones:
  - ***demorar(intervalo):*** demora al llamador durante intervalo ticks de reloj.
  - ***tick:*** incrementa el valor del reloj lógico. Es llamada por un proceso que es despertado periódicamente por un timer de hardware y tiene alta prioridad de ejecución.

Ineficiente → mejor usar *wait con prioridad* o *variables condition privadas*

Técnicas de Sincronización

Diseño de un reloj lógico: *Wait con prioridad*

El mismo ejemplo anterior del reloj lógico utilizando *wait con prioridad*:

```
monitor Timer
{  int hora_actual = 0;
   cond espera;

   procedure demorar(int intervalo)
    {  int hora_de_despertar;
       hora_de_despertar = hora_actual + intervalo;
       wait(espera, hora_a_despertar);
    }

   procedure tick( )
     { hora_actual = hora_actual + 1;
       while (minrank(espera) <= hora_actual)
              signal (espera);
      }
}
```

Técnicas de Sincronización

Diseño de un reloj lógico: *Variables conditions privadas*

El mismo ejemplo anterior del reloj lógico utilizando *variables conditions privadas*:

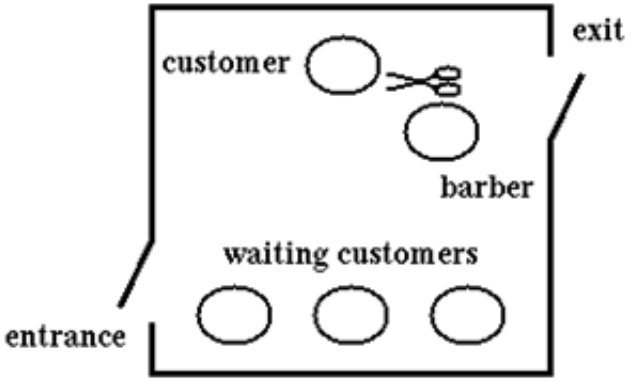
```
monitor Timer
{  int hora_actual = 0;
   cond espera[N];
   colaOrdenada dormidos;

   procedure demorar(int intervalo, int id)
    {  int hora_de_despertar;
       hora_de_despertar = hora_actual + intervalo;
       Insertar(dormidos, id, hora_de_despertar);
       wait(espera[id]);
    }

   procedure tick( )
     { int aux, idAux;
       hora_actual = hora_actual + 1;
       aux = verPrimero (dormidos);
       while (aux <= hora_actual)
            { sacar (dormidos, idAux)
              signal (espera[idAux]);
              aux = verPrimero (dormidos);
             }
      }
}
```

Técnicas de Sincronización

Peluquero dormilón: *Rendezvous*

Problema del peluquero dormilón (sleeping barber).

Una ciudad tiene una peluquería con 2 puertas y unas pocas sillas. Los clientes entran por una puerta y salen por la otra. Como el negocio es chico, a lo sumo un cliente o el peluquero se pueden mover en él a la vez. El peluquero pasa su tiempo atendiendo clientes, uno por vez. Cuando no hay ninguno, el peluquero duerme en su silla. Cuando llega un cliente y encuentra que el peluquero está durmiendo, el cliente lo despierta, se sienta en la silla del peluquero, y duerme mientras el peluquero le corta el pelo. Si el peluquero está ocupado cuando llega un cliente, éste se va a dormir en una de las otras sillas. Después de un corte de pelo, el peluquero abre la puerta de salida para el cliente y la cierra cuando el cliente se va. Si hay clientes esperando, el peluquero despierta a uno y espera que se siente. Sino, se vuelve a dormir hasta que llegue un cliente.



Técnicas de Sincronización
Peluquero dormilón: *Rendezvous*

- ***Procesos*** ⇒ ***clientes*** y ***peluquero***.
- ***Monitor*** ⇒ ***administrador de la peluquería***. Tres procedures:
  - ***corte_de_pelo***: llamado por los clientes, que retornan luego de recibir un corte de pelo.
  - ***proximo_cliente***: llamado por el peluquero para esperar que un cliente se siente en su silla, y luego le corta el pelo.
  - ***corte_terminado***: llamado por el peluquero para que el cliente deje la peluquería.
- El peluquero y un cliente necesitan una serie de etapas de sincronización ***(rendezvous)***:
  - El peluquero tiene que esperar que llegue un cliente, y este tiene que esperar que el peluquero esté disponible.
  - El cliente necesita esperar que el peluquero termine de cortarle el pelo, indicado cuando le abre la puerta de salida.
  - Antes de cerrar la puerta de salida, el peluquero necesita esperar hasta que el cliente haya dejado el negocio.

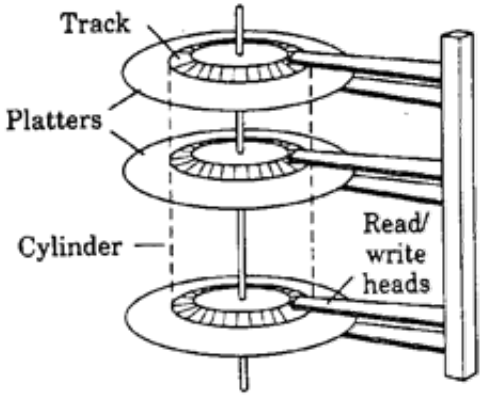
→ el peluquero y el cliente atraviesan una serie de etapas de sincronización, comenzando con un *rendezvous* similar a una barrera entre dos procesos, pues ambas partes deben arribar antes de que cualquiera pueda seguir.

Técnicas de Sincronización

Peluquero dormilón: *Rendezvous*

```
monitor  Peluqueria {
    int peluquero = 0, silla = 0, abierto = 0;
    cond peluquero_disponible, silla_ocupada, puerta_abierta, salio_cliente;

    procedure corte_de_pelo() {
        while (peluquero == 0)   wait (peluquero_disponible);
        peluquero = peluquero - 1;
        signal (silla_ocupada);
        wait (puerta_abierta);
        signal (salio_cliente);
     }

     procedure proximo_cliente(){
          peluquero = peluquero + 1;
          signal(peluquero_disponible);
          wait(silla_ocupada);
     }

     procedure corte_terminado() {
          signal(puerta_abierta);
          wait(salio_cliente);
     }
}
```

Ejemplo: *Scheduling de disco*



- El disco contiene "platos" conectados a un eje central y que rotan a velocidad constante. Las pistas forman círculos concéntricos ⇒ concepto de cilindro de información.
- Los datos se acceden posicionando una cabeza lectora/escritora sobre la pista apropiada, y luego esperando que el plato rote hasta que el dato pase por la cabeza.

  dirección física → cilindro, número de pista, y desplazamiento
- Para acceder al disco, un programa ejecuta una instrucción de E/S específica. Los parámetros para esa instrucción son:
  - dirección física del disco
  - el número de bytes a transferir
  - el tipo de transferencia a realizar (read o write)
  - la dirección de un buffer.

Ejemplo: *Scheduling de disco*

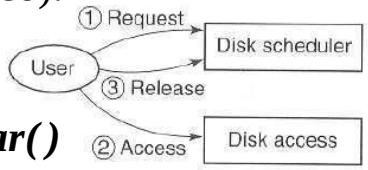
- El tiempo de acceso al disco depende de tres cantidades:
  a. Seek time para mover una cabeza al cilindro apropiado.
  b. Rotational delay.
  c. Transmission time (depende solo del número de bytes).
- a) y b) Dependen del estado del disco (seek time >> rotational delay) ⇒ para reducir el tiempo de acceso promedio conviene minimizar el movimiento de la cabeza (reducir el tiempo de seek).
- El scheduling de disco puede tener distintas políticas:
  - ***Shortest-Seek-Time (SST):*** **s**elecciona siempre el pedido pendiente que quiere el cilindro más cercano al actual. Es unfair.
  - ***SCAN, LOOK, o algoritmo del ascensor:*** se sirven pedidos en una dirección y luego se invierte. Es fair. *Problema***:** un pedido pendiente justo detrás de la posición actual de la cabeza no será servido hasta que la cabeza llegue al final y vuelva (gran varianza del tiempo de espera).
  - ***CSCAN o CLOOK:*** se atienden pedidos en una sola dirección. Es fair y reduce la varianza del tiempo de espera.

Ejemplo: *Scheduling de disco*
Monitor separado

El ***scheduler*** es implementado por un monitor para que los datos sean accedidos solo por un proceso usuario a la vez.

El monitor provee dos operaciones: ***pedir*** y ***liberar***.

- Un proceso usuario que quiere acceder al cilindro ***cil*** llama a ***pedir(cil)***, y retoma el control cuando el scheduler seleccionó su pedido. Luego, el proceso usuario accede al disco (llamando a un procedure o comunicándose con un proceso manejador del disco).



- Luego de acceder al disco, el usuario llama a liberar:

  Scheduler_Disco.pedir(cil) - Accede al disco - ***Scheduler_Disco.liberar()***

- Suponemos cilindros numerados de 0 a MAXCIL y scheduling CSCAN.
- A lo sumo un proceso a la vez puede tener permiso para usar el disco, y los pedidos pendientes son servidos en orden CSCAN.
- ***posicion*** es la variable que indica posición corriente de la cabeza (cilindro que está siendo accedido por el proceso que está usando el disco).
- Para implementar CSCAN, hay que distinguir entre los pedidos pendientes a ser servidos en el scan corriente y los que serán servidos en el próximo scan.

Ejemplo: *Scheduling de disco*

Monitor separado

```
monitor Scheduler_Disco
{ int posicion = -1, v_actual = 0, v_proxima = 1;
  cond scan[2];

  procedure pedir(int cil)
    { if (posicion == -1)  posicion = cil;
      elseif (cil > posicion)   wait(scan[v_actual],cil);
      else  wait(scan[v_proxima],cil);
    }

  procedure liberar()
     { if (!empty(scan[v_actual]))  posicion = minrank(scan[v_actual]);
       elseif (!empty(scan[v_proxima]))
            { v_actual :=: v_proxima;
              posicion = minrank(scan[v_actual]);
            }
       else  posicion = -1;
       signal(scan[v_actual]);
     }
}
```

Ejemplo: *Scheduling de disco*
Monitor intermedio

- Problemas de la solución anterior:
  - La presencia del *scheduler* es visible al proceso que usa el disco. Si se borra el *scheduler*, los procesos usuario cambian.
  - Todos los procesos *usuario* deben seguir el protocolo de acceso. Si alguno no lo hace, el scheduling falla.
  - Luego de obtener el acceso, el proceso debe comunicarse con el *driver de acceso* al disco a través de 2 instancias de *buffer limitado*.

MEJOR: usar un monitor como intermediario entre los procesos usuario y el disk driver. El monitor envía los pedidos al disk driver en el orden de preferencia deseado.

- Mejoras:
  - La interfaz al disco usa un único monitor, y los usuarios hacen un solo llamado al monitor por acceso al disco.
  - La existencia o no de scheduling es transparente.
  - No hay un protocolo multipaso que deba seguir el usuario y en el cual pueda fallar.

Ejemplo: *Scheduling de disco*

Monitor intermedio

```
monitor Interfaz_al_Disco
{  variables permanentes para estado, scheduling y transferencia de datos.

   procedure usar_disco(int cil, parámetros de transferencia y resultados)
      {  esperar turno para usar el manejador
         almacenar parámetros de transferencia en variables permanentes
         esperar que se complete la transferencia
         recuperar resultados desde las variables permanentes
      }

   procedure buscar_proximo_pedido(algunType &resultados)
      {  seleccionar próximo pedido
         esperar a que se almacenen los parámetros de transferencia
         setear resultados a los parámetros de transferencia
      }

   procedure transferencia_terminada(algunType resultados)
      {  almacenar los resultados en variables permanentes
         esperar a que resultados sean recuperados por el cliente
      }
}
```

Ejemplo: *Scheduling de disco*

Monitor intermedio

```
monitor Interfaz_al_disco
{  int posicion = -2, v_actual = 0, v_proxima = 1, args = 0, resultados = 0;
   cond scan[2];
   cond args_almacenados, resultados_almacenados, resultados_recuperados;
   argType area_arg; resultadoType area_resultado;

   procedure usar_disco (int cil; argType params_transferencia;resultType &params_resultado)
      {  if (posicion == -1)    posicion = cil;
         elseif (cil > posicion)   wait(scan[v_actual],cil);
         else wait(scan[v_proxima],cil);
         area_arg = parametros_transferencia;
         args = args+1; signal(args_almacenados);
         wait(resultados_almacenados);
         parametros_resultado = area_resultado;
         resultados = resultados-1;
         signal(resultados_recuperados);
      }
```

Ejemplo: *Scheduling de disco*

Monitor intermedio

```
procedure buscar_proximo_pedido  (argType &parametros_transferencia)
      { int temp;
        if (!empty(scan[v_actual]))   posicion = minrank(scan[v_actual]);
        elseif (!empty(scan[v_proxima]))
              { v_actual :=: v_proxima;
                posicion = minrank(scan[v_actual]);
              }
        else  posicion = -1;
        signal(scan[v_actual]);
        if (args == 0) wait(args_almacenados);
        parametros_transferencia  = area_arg; args = args-1;
       }

   procedure transferencia_terminada  (resultType valores_resultado)
      { area_resultado := valores_resultado;
        resultados = resultados+1;
        signal(resultados_almacenados);
        wait(resultados_recuperados);
      }
 }
```

Programación Concurrente

Clase 6

Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Librería Pthreads: https://drive.google.com/uc?id=1T2TBPMgrc1ax0z_vrDWMiSaSXJOTJ9GR&export=download

- Semáforos y monitores en Pthreads: https://drive.google.com/uc?id=1gLhJodkLt8ukBGLPNgauoHRX2HGX0sN9&export=download

Pthreads

Pthreads

Thread: proceso "liviano" que tiene su propio contador de programa y su pila de ejecución, pero no controla el "contexto pesado" (por ejemplo, las tablas de página).

- Algunos sistemas operativos y lenguajes proveen mecanismos para permitir la programación de aplicaciones "multithreading".
- En principio estos mecanismos fueron heterogéneos y poco portables $\Rightarrow$ a mediados de los 90 la organización POSIX auspició el desarrolló de una biblioteca en C para multithreading **(*Pthreads*).**
- Pthreads es una bilbioteca para programación paralela en ***memoria compartida,*** se pueden crear threads, asignarles atributos, darlos por terminados, identificarlos, etc.

POSIX – API de Threads

- Numerosas APIs para el manejo de Threads.
- Normalmente llamada Pthreads, POSIX a emergido como un API estandard para manejo de Threads, provista por la mayoría de los vendedores.
- Los conceptos que se discutirán son independientes de la API y pueden ser igualmente válidos para utilizar JAVA Threads, NT Threads, Solaris Threads, etc.
- Funciones reentrantes.

Pthreads - Creación y terminación

- Pthreads provee funciones básicas para especificar concurrencia:

```
#include <pthread.h>

int pthread_create (pthread_t  *thread_handle, const pthread_attr_t  *attribute,
                    void * (*thread_function)(void *), void *arg);

int pthread_exit (void *res);

int pthread_join (pthread_t   thread, void **ptr);

int pthread_cancel (pthread_t thread);
```

- El “main” debe esperar a que todos los threads terminen.

Pthreads – Primitivas de Sincronización
Exclusión mutua

- Las secciones críticas se implementan en Pthreads utilizando ***mutex locks*** (bloqueo por exclusión mutua) por medio de variables ***mutex***.
- Una variable ***mutex*** tienen dos estados: locked (bloqueado) and unlocked (desbloqueado). En cualquier instante, sólo UN thread puede bloquear un *mutex*. *Lock* es una operación atómica.
- Para entrar en la sección crítica un Thread debe lograr tener control del *mutex* (bloquearlo).
- Cuando un Thread sale de la SC debe desbloquear el *mutex*.
- Todos los *mutex* deben inicializarse como desbloqueados.

Pthreads – Primitivas de Sincronización
Exclusión mutua

- La API Pthreads provee las siguientes funciones para manejar los *mutex*:

```
int pthread_mutex_lock ( pthread_mutex_t *mutex);

int pthread_mutex_unlock (pthread_mutex_t *mutex);

int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *lock_attr);
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

El escenario de productores-consumidores impone las siguientes restricciones:

- Un thread productor no debe sobrescribir el buffer compartido cuando el elemento anterior no ha sido tomado por un thread consumidor.
- Un thread consumidor no puede tomar nada de la estructura compartida hasta no estar seguro de que se ha producido algo anteriormente.
- Los consumidores deben excluirse entre sí.
- Los productores deben excluirse entre sí.
- En este ejemplo el buffer es de tamaño 1.

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Main de la solución al problema de productores-consumidores.

```
pthread_mutex_t     mutex;
int hayElemento;
tipo_elemento  Buffer;
...
main()
   { hayElemento=  0;
     pthread_init ();
     pthread_mutex_init(&mutex,  NULL);

       /* Create y join de threads productores y consumidores*/
    }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Código para los *productores*.

```
void *productor (void *datos)
  {  tipo_elemento elem;
     int ok;
     ....
     while (true)
           { ok = 0;
             generar_elemento(&elem);
             while (ok = = 0)
               {  pthread_mutex_lock(&mutex);
                  if (hayElemento==  0)
                    {  Buffer  = elem;
                       hayElemento  = 1;
                       ok = 1;
                     }
                  pthread_mutex_unlock(&mutex);
                }
           }
   }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Exclusión Mutua

Código para los *consumidores*.

```
void *consumidor(void *datos)
  {  int ok;
     tipo_elemento elem;
     ....
     while (true)
        { ok = 0;
          while (ok == 0)
              {  pthread_mutex_lock(&mutex);
                 if (hayElemento  == 1)
                    { elem = Buffer;
                      hayElemento  = 0;
                      ok = 1;
                     }
                 pthread_mutex_unlock(&mutex);
              }
           procesar_elemento(elem);
        }
   }
```

Pthreads - Primitivas de Sincronización
Tipos de Exclusión Mutua (*Mutex*)

- Pthreads soporta tres tipos de Mutexs (Locks): Normal, Recursive y Error Check
  - ✓ Un Mutex con el atributo Normal NO permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él (deadlock).
  - ✓ Un Mutex con el atributo Recursive SI permite que un thread que lo tienen bloqueado vuelva a hacer un lock sobre él. Simplemente incrementa una cuenta de control.
  - ✓ Un Mutex con el atributo ErrorCheck responde con un reporte de error al intento de un segundo bloqueo por el mismo thread.
- El tipo de Mutex puede setearse entre los atributos antes de su inicialización.

Pthreads - Primitivas de Sincronización
Overhead de Bloqueos por Exclusión Mutua

- Los locks representan puntos de serialización → si dentro de las secciones críticas ponemos segmentos largos de programa tendremos una degradación importante de perfomance.

- A menudo se puede reducir el overhead por espera ociosa, utilizando la función pthread_mutex_trylock. Retorna el control informando si pudo hacer o no el lock.

  int pthread_mutex_trylock (pthread_mutex_t *mutex_lock).

  - ✓ Evita tiempos ociosos.
  - ✓ Menos costoso por no tener que manejar las colas de espera.

Pthreads - Primitivas de Sincronización
Variables Condición

- Podemos utilizar variables de condición para que un thread se autobloquee hasta que se alcance un estado determinado del programa.
- Cada variable de condición estará asociada con un predicado. Cuando el predicado se convierte en verdadero (TRUE) la variable de condición da una señal para el/los threads que están esperando por el cambio de estado de la condición.
- Una única variable de condición puede asociarse a varios predicados (difícil el debug).
- Una variable de condición siempre tiene un mutex asociada a ella. Cada thread bloquea este mutex y testea el predicado definido sobre la variable compartida.
- Si el predicado es falso, el thread espera en la variable condición utilizando la función pthread_cond_wait (NO USA CPU).

Pthreads - Primitivas de Sincronización
Variables Condición

La API Pthreads provee las siguientes funciones para manejar las variables condición:

```
int pthread_cond_wait ( pthread_cond_t *cond,
                        pthread_mutex_t *mutex)

int pthread_cond_timedwait ( pthread_cond_t *cond,
                             pthread_mutex_t *mutex
                             const struct timespec *abstime)

int pthread_cond_signal (pthread_cond_t *cond)

int pthread_cond_broadcast (pthread_cond_t *cond)

int pthread_cond_init ( pthread_cond_t *cond,
                        const pthread_condattr_t *attr)

int pthread_cond_destroy (pthread_cond_t *cond)
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Main de la solución al problema de productores-consumidores.

```
pthread_cond_t vacio, lleno;
pthread_mutex_t mutex;
int hayElemento;
tipo_elemento Buffer;
...
main()
   { ...
     hayElemento= 0;
     pthread_init();
     pthread_cond_init(&vacio, NULL);
     pthread_cond_init(&lleno, NULL);
     pthread_mutex_init(&mutex, NULL);
     ...
   }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Código para los *productores*.

```
void *productor(void *datos)
   { tipo_element elem;

     while (true)
       { generar_elemento(elem);
         pthread_mutex_lock  (&mutex);
         while (hayElemento  = = 1)
                pthread_cond_wait (&vacio, &mutex);
          Buffer  = elem;
          hayElemento  = 1;
          pthread_cond_signal  (&lleno);
          pthread_mutex_unlock  (&mutex);
       }
  }
```

Pthreads - Primitivas de Sincronización Productores/Consumidores con Variables Condición

Código para los *consumidores*.

```
void *consumidor(void *datos)
   {  tipo_element elem;

      while (true))
       { pthread_mutex_lock (&mutex);
         while (hayElemento = = 0)
                  pthread_cond_wait (&lleno, &mutex);
         elem= Buffer;
         hayElemento = 0;
         pthread_cond_signal (&vacio);
         pthread_mutex_unlock (&mutex);
         procesar_elemento(elem);
       }
   }
```

Pthreads – Atributos y sincronización

- La API Pthreads permite que se pueda cambiar los atributos por defecto de las entidades, utilizando attributes objects.
- Un attribute object es una estructura de datos que describe las propiedades de la entidad en cuestión (thread, mutex, variable de condición).
- Una vez que estas propiedades están establecidas, el attribute object es pasado al método que inicializa la entidad.
- Ventajas
  - ✓ Esta posibilidad mejora la modularidad.
  - ✓ Facilidad de modificación del código.

Pthreads – Atributos para Threads

- La API Pthreads provee las siguientes funciones para manejar los atributos para Threads:

```
int pthread_attr_init (pthread_attr_t  *attr);

int pthread_attr_destroy (pthread_attr_t  *attr);
```

- Las propiedades asociadas con el *attribute object* pueden ser cambiadas con las siguientes funciones:

```
pthread_attr_setdetachstate

pthread_attr_setguardsize_np

pthread_attr_setstacksize

pthread_attr_setinheritsched

pthread_attr_setschedpolicy

pthread_attr_setschedparam
```

Pthreads – Atributos para *Mutex*

- La API Pthreads provee las siguientes funciones para manejar los atributos para Mutex:

```
int pthread_mutexattr_init (pthread_mutexattr_t  *attr);

int pthread_mutexattr_settype_np ( pthread_mutexattr_t *attr,   int type);
```

- Aquí *type* especifica el tipo de *mutex* y puede tomar los valores:
  - PTHREAD_MUTEX_NORMAL_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

Semáforos en Pthreads

Semáforos con Pthreads

- Los threads pueden sincronizar por semáforos (*librería semaphore.h*).
- Declaración y operaciones con semáforos en Pthreads:
  - ✓ sem_t semaforo→ se declaran globales a los threads.
  - ✓ sem_init (&semaforo, alcance, inicial) → en esta operación se inicializa el semáforo semaforo. Inicial es el valor con que se inicializa el semáforo. Alcance indica si es compartido por los hilos de un único proceso (0) o por los de todos los procesos (≠ 0).
  - ✓ sem_wait(&semaforo) → equivale al P.
  - ✓ sem_post(&semaforo) → equivale al V.
  - ✓ Existen funciones extras para: wait condicional, obtener el valor de un semáforo y destruir un semáforo (ESTE TIPO DE FUNCIONES EXTRAS NO SE PUEDEN USAR EN LA PRÁCTICA DE LA MATERIA).

Semáforos con Pthreads

Productor / consumidor

- Las funciones de ***Productor*** y ***Consumidor*** serán ejecutadas por threads independientes.
- Acceden a un buffer compartido (***datos***).
- El productor deposita una secuencia de enteros de ***1*** a ***numItems*** en el buffer.
- El consumidor busca estos valores y los suma.
- Los semáforos ***vacio*** y ***lleno*** garantizan el acceso alternativo de productor y consumidor sobre el buffer.

```
#include <pthread.h>
#include <semaphore.h>
#define SHARED 1

void *Productor(void *);
void *Consumidor(void *);

sem_t  vacio, lleno;
int dato, numItems;
```

```
int main(int argc, char * argv[ ])
{    ……
     sem_init (&vacio, SHARED, 1);
     sem_init (&lleno, SHARED, 0);
     ……
     pthread_create (&pid, &attr,  Productor, NULL);
     pthread_create (&cid, &attr, Consumidor, NULL);
     pthread_join (pid, NULL);
     pthread_join (cid, NULL);
}
```

Semáforos con Pthreads

Productor / consumidor

```
void *Productor (void *arg)
{ int item;
  for (item = 1; item <= numItems;  item++)
     {   sem_wait(&vacio);
         dato = item;
         sem_post(&lleno);
      }
   pthreads_exit();
}

void *Consumidor  (void *arg)
{   int total = 0, item, aux;
    for (item = 1; item <= numItems;  item++)
       { sem_wait(&lleno);
         aux = dato;
         sem_post(&vacio);
         total = total + aux;
       }
    printf("TOTAL:   %d\n", total);
    pthreads_exit();
}
```

Monitores en Pthreads

Monitores con Pthreads

- Pthreads no permite manejar la Exclusión Mutua por medio de las variables *mutex*.
- Pthreads nos permite manejar la Sincronización por Condición utilizando *variables condición* para que un *thread* se auto bloquee hasta que se alcance un estado determinado del programa. Una variable de condición siempre tiene un *mutex* asociada a ella.
- Pthreads no posee "*Monitores*", pero con las dos herramientas que mencionamos se puede similar el uso de monitores: con mutex se hace la exclusión mutua que nos brindaba implicitamente el monitor, y con las variables condición la sincronización.
  - El acceso exclusive al monitor se simula usando una variable *mutex* la cual se bloquea antes del llamada al *procedure* y se desbloquea al terminar el mismo (una variable *mutex* diferente para cada monitor).
  - Cada llamado de un proceso a un procedure de un monitor debe ser reemplazado por el código de ese procedure.

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Esta es la solución que vimos en la teoría de monitores para el problema de lectores/escritores. Ahora veremos como simularla en ***Pthreads***.

```
monitor Controlador
{  int nr = 0, nw = 0, dr = 0, dw = 0;
   cond  ok_leer, ok_escribir

   procedure pedido_leer()
    { if (nw > 0)
                  { dr = dr +1;
                    wait (ok_leer);
                  }
      else  nr = nr + 1;
    }

   procedure libera_leer()
    { nr = nr - 1;
      if (nr == 0 and dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                  nw = nw +1;
                }
    }

   procedure pedido_escribir()
     { if (nr>0 OR nw>0)
                  { dw = dw +1;
                    wait (ok_escribir);
                  }
       else  nw = nw + 1;
     }

   procedure libera_escribir()
    { if (dw > 0)
                { dw = dw -1;
                  signal (ok_escribir);
                }
      else { nw = nw -1;
             if (dr > 0)
                  { nr = dr;
                    dr = 0;
                    signal_all (ok_leer);
                  }
           }
     }
}
```

```
Process lector[id: 0..L-1]
{  while (true)
     { Controlador.pedido_leer();
       //Leer sobre la BD
       Controlador.libera_leer();
     };
}

Process escritor[id: 0..E-1]
{  while (true)
     { Controlador.pedido_escribir();
       //Leer sobre la BD
       Controlador.libera_escribir();
     };
}
```

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{  int nr = 0, nw = 0, dr = 0, dw = 0, i;
   pthread_cond_t   ok_leer, ok_escribir;
   pthread_t  lectores[L], escritores[E];
   ……
   pthread_init();
   pthread_cond_init(&ok_leer,  NULL);
   pthread_cond_init(&ok_escribir,  NULL);
   ……
   for (i=0; i<E;i++) pthread_create (&escritores[i], &attr,  Escritor, NULL);
   for (i=0; i<L;i++) pthread_create (&lectores[i], &attr,  Lector, NULL);
}
```

Por cada monitor se require un *mutex* para similar la EM implicita de los mismos.

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Como hay solo un monitor se pone solo una variable *mutex* que además de declarer se inicializa en el *main* del programa.

```
#include <pthread.h>

void *Escritor(void *);
void *Lector(void *);

int main(int argc, char * argv[ ])
{  int nr = 0, nw = 0, dr = 0, dw = 0, i;
   pthread_cond_t   ok_leer, ok_escribir;
   pthread_t  lectores[L], escritores[E];
   pthreads_mutex_t   mutex;
   ……
   pthread_init();
   pthread_cond_init(&ok_leer,  NULL);
   pthread_cond_init(&ok_escribir,  NULL);
   pthread_mutex_init(&mutex,  NULL);
   ……
   for (i=0; i<E;i++) pthread_create (&escritores[i], &attr,  Escritor, NULL);
   for (i=0; i<L;i++) pthread_create (&lectores[i], &attr,  Lector, NULL);
}
```

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

- Se agrega en los procesos el bloque y desbloqueo de *mutex* en los llamados a los procedure del monitor.

```
void *lector (void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       Controlador.pedido_leer();
       pthread_mutex_unlock (&mutex);
       //Leer sobre la BD
       pthread_mutex_lock (&mutex);
       Controlador.libera_leer();
       pthread_mutex_unlock (&mutex);
     }
}
```

```
void *escritor (void*)
{  while (true)
     { pthread_mutex_lock (&mutex);
       Controlador.pedido_escribir();
       pthread_mutex_unlock (&mutex);
       //Leer sobre la BD
       pthread_mutex_lock (&mutex);
       Controlador.libera_escribir ();
       pthread_mutex_unlock (&mutex);
     }
}
```

El próximo paso es reemplazar los llamados de los procedimientos por el código de los mismos

Monitores con Pthreads

Ejemplo: *Lectores y escritores*

```
void *escritor (void*)
{   while (true)
     { pthread_mutex_lock (&mutex);
       if (nr>0 OR nw>0)
         { dw = dw +1;
           pthread_cond_wait (& ok_escribir, &mutex);
         }
       else  nw = nw + 1;
       pthread_mutex_unlock (&mutex);
       //Escribe sobre la BD
       pthread_mutex_lock (&mutex);
       if (dw > 0)
         { dw = dw -1;
           pthread_cond_signal(&ok_escribir);
         }
       else
         { nw = nw -1;
           if (dr > 0)
             { nr = dr;
               dr = 0;
               pthread_cond_broadcast(&ok_leer);
             };
           };
       pthread_mutex_unlock (&mutex);
     };
   pthreads_exit();
};
```

```
void *lector(void*)
{   while (true)
     { pthread_mutex_lock (&mutex);
       if (nw>0)
         { dr = dr +1;
           pthread_cond_wait (& ok_leer, &mutex);
         }
       else  nr = nr + 1;
       pthread_mutex_unlock (&mutex);
       //Leer sobre la BD
       pthread_mutex_lock (&mutex);
       nr = nr – 1;
       if (nr == 0 and dw > 0)
         { dw = dw -1;
           pthread_cond_signal(&ok_escribir);
           nw = nw + 1;
         };
       pthread_mutex_unlock (&mutex);
     };
   pthreads_exit();
};
```

Programación Concurrente

Clase 7

Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Programación concurrente en memoria distribuida: https://drive.google.com/uc?id=1Xdh_8do7D0XmRQBNDHmHf6MA3usfZaUS&export=download

- Pasaje de Mensajes Asincrónicos (PMA): https://drive.google.com/uc?id=1h3mD73WydEjYSlGXMxQCiJLwsyM3QEbt&export=download

Programación concurrente en memoria distribuida

Conceptos generales

- Arquitecturas de memoria distribuida ⇒ *procesadores* + *memo local* + *red de comunicaciones* + ***mecanismo de comunicación / sincronización*** ⇒ ***intercambio de mensajes.***
- ***Programa distribuido:*** programa concurrente comunicado por mensajes. Supone la ejecución sobre una arquitectura de memoria distribuida, aunque puedan ejecutarse sobre una de memoria compartida (o híbrida).
- ***Primitivas de pasaje de mensajes:*** interfaz con el sistema de comunicaciones ⇒ semáforos + datos + sincronización.
- Los procesos *SOLO* ***comparten canales*** (físicos o lógicos). Variantes para los canales:
  - Mailbox, input port, link.
  - Uni o bidireccionales.
  - Sincrónicos o asincrónicos.

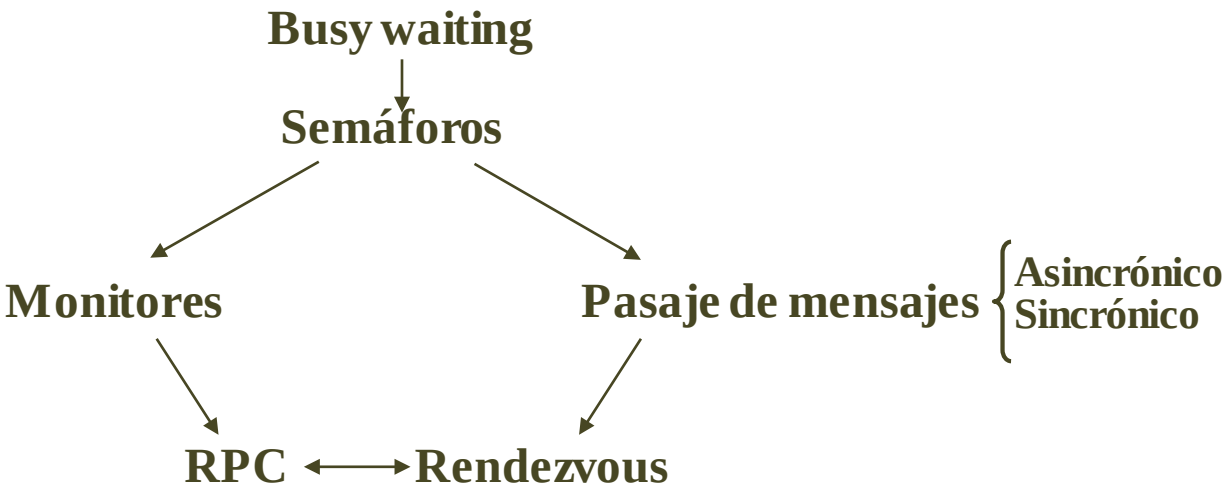
Características

- Los canales son lo único que comparten los procesos
  - Variables locales a un proceso (“cuidador”).
  - La exclusión mutua no requiere mecanismo especial.
  - Los procesos interactúan comunicándose.
  - Accedidos por primitivas de envío y recepción.
- Mecanismos para el Procesamiento Distribuido:
  - Pasaje de Mensajes Asincrónicos (PMA)
  - Pasaje de Mensajes Sincrónico (PMS)
  - Llamado a Procedimientos Remotos (RPC)
  - Rendezvous
- La sincronización de la comunicación interproceso depende del patrón de interacción:
  - Productores y consumidores (Filtros o pipes)
  - Clientes y servidores
  - Pares que interactúan

Cada mecanismo es más adecuado para determinados patrones

Relación entre mecanismos de sincronización

- ***Semáforos*** ⇒ mejora respecto de ***busy waiting***.
- ***Monitores*** ⇒ combinan Exclusión Mutua implícita y señalización explícita.
- ***PM*** ⇒ extiende semáforos con datos.
- ***RPC*** y ***rendezvous*** ⇒ combina la interface procedural de monitores con PM implícito.



Pasaje de Mensajes Asincrónicos (PMA)

Uso de canales en PMA

- ***PMA*** ⇒ canales = colas de mensajes enviados y aún no recibidos.
- ***Declaración de canales*** → **chan** ***ch*** **(** id_1 **:** $tipo_1$**, ... ,** id_n **:** $tipo_n$ **)**
  - **chan** entrada (char);
  - **chan** acceso_disco (INT cilindro, INT bloque, INT cant, CHAR* buffer)**;**
  - **chan** resultado[n] (INT);
- ***Operación Send*** → un proceso agrega un mensaje al final de la cola ("ilimitada") de un canal ejecutando un *send*, que no bloquea al emisor:

  send ch(expr1, ... , exprn);
- ***Operación Receive*** → un proceso recibe un mensaje desde un canal con *receive*, que demora ("bloquea") al receptor hasta que en el canal haya al menos un mensaje; luego toma el primero y lo almacena en variables locales:

  receive ch(var_1***, ... ,*** var_n***);***

  Las variables del receive deben tener los mismos tipos que la declaración del canal.

  Receive es una primitiva ***bloqueante***, ya que produce un delay. **Semántica**: el proceso NO hace nada hasta recibir un mensaje en la cola correspondiente al canal. ***NO*** es necesario hacer polling.

Características de los canales

- Acceso a los contenidos de cada canal: atómico y respeta orden FIFO.
- En principio los canales son ilimitados, aunque las implementaciones reales tendrán un tamaño de buffer asignado.
- Se supone que los mensajes NO se pierden ni modifican y que todo mensaje enviado en algún momento puede ser "leído".
- ***empty(ch)*** → determina si la cola de un canal está vacía. Útil cuando el proceso puede hacer trabajo productivo mientras espera un mensaje, **pero debe usarse con cuidado**.
  - O podría ser ***false***, y no haber más mensajes cuando sigue ejecutando (si no en el único en recibir por ese canal).
  - La evaluación de ***empty*** podría ser ***true***, y sin embargo existir un mensaje al momento de que el proceso reanuda la ejecución.

Características de los canales

Los canales son declarados globales a los procesos, ya que pueden ser compartidos. Según la forma en que se usan podría ser:

- Cualquier proceso puede enviar o recibir por alguno de los canales declarados. En este caso suelen denominarse ***mailboxes***.
- En algunos casos un canal tiene un solo receptor y muchos emisores (***input port***).
- Si el canal tiene un único emisor y un único receptor se lo denomina ***link***: provee un "camino" entre el emisor y sus receptores.

Ejemplo

```
chan entrada(char), salida(char [CantMax]);

Process Carac_a_Linea
{ char linea [CantMax], int i = 0;
  WHILE (true)
     { receive entrada (linea[i]);
       WHILE (linea[i] ≠ CR and i < CantMax)
          {  i := i + 1;
               receive entrada (linea[i]);
          }
       linea [i] := EOL;
       send salida(linea);
       i := 0;
     }
}
```

```
Process Proceso_1
{  char a;
   WHILE (true)
      {  leer_carácter_por_teclado(a);
         send entrada(a);
      }
}

Process Proceso_2
{  char res[CantMax];
   WHILE (true)
      { receive salida(res);
        imprimir_en_pantalla(res);
      }
}
```

Productores y consumidores (*filtro)*

Red de Ordenación

- ***Filtro:*** proceso que recibe mensajes de uno o más canales de entrada y envía mensajes a uno o más canales de salida. La salida de un filtro es función de su estado inicial y de los valores recibidos.
- Esta función del filtro puede especificarse por un predicado que relacione los valores de los mensajes de salida con los de entrada.
- ***Problema:*** ordenar una lista de *N* números de modo ascendente. Podemos pensar en un filtro *Sort* con un canal de entrada (*N* números desordenados) y un canal de salida (*N* números ordenados).

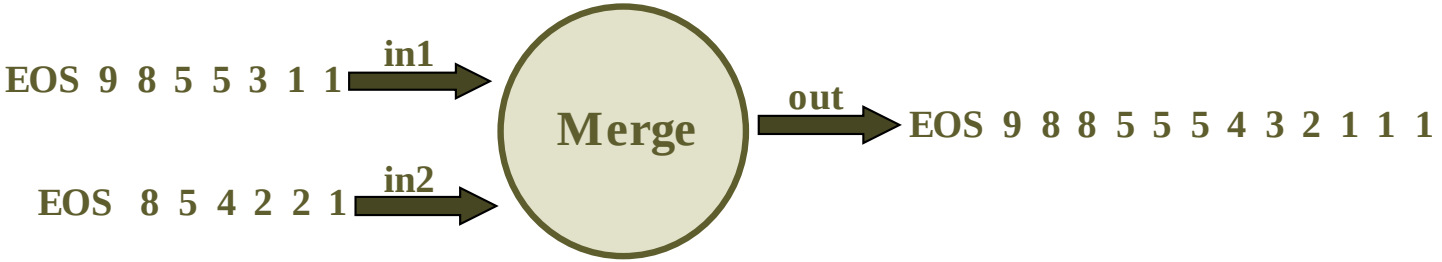
```
Process Sort
        {  receive todos los números del canal entrada;
           ordenar los números;
           send de los números ordenados por el canal OUTPUT;
         }
```

- ¿Cómo determina *Sort* que recibió todos los números?
  - conoce *N*.
  - envía *N* como el primer elemento a recibir por el canal *entrada*.
  - cierra la lista de *N* números con un valor especial o “centinela”.

Productores y consumidores (*filtro*)

Red de Ordenación

- Solución más eficiente que la "secuencial" ⇒ red de pequeños procesos que ejecutan en paralelo e interactúan para "armar" la salida ordenada ***(merge network).***
- ***Idea:*** mezclar repetidamente y en paralelo dos listas ordenadas de *N1* elementos cada una en una lista ordenada de *2*N1* elementos.
- Con ***PMA***, pensamos en *2* canales de entrada por cada canal de salida, y un carácter especial *EOS* cerrará cada lista parcial ordenada.
- La red es construida con filtros ***Merge***:
  - Cada *Merge* recibe valores de dos *streams* de entrada ordenados, ***in1*** e ***in2***, y produce un *stream* de salida ordenado, ***out***.
  - Los *streams* terminan en ***EOS***, y *Merge* agrega ***EOS*** al final.
  - ¿Cómo implemento *Merge*?. Comparar repetidamente los próximos dos valores recibidos desde ***in1*** e ***in2*** y enviar el menor a ***out***.



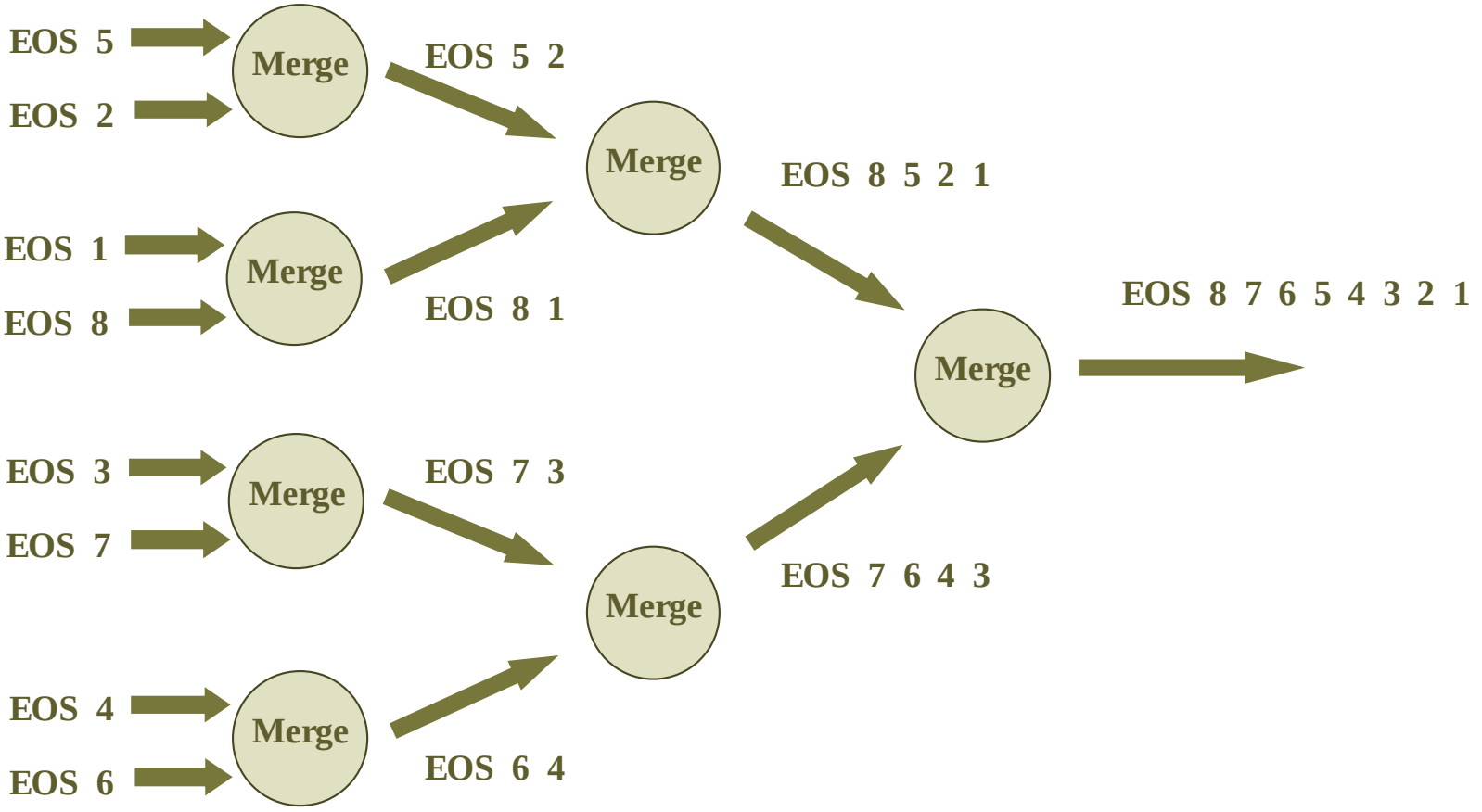
Productores y consumidores (*filtro)*

Red de Ordenación

```
chan  in1(int), in2(int), out(int);

Process Merge
{  int v1, v2;

   receive in1(v1);
   receive in2(v2);
   while (v1 ≠ EOS) and (v2 ≠ EOS)
      { if  (v1 ≤ v2) { send out(v1); receive in1(v1); }
        else  { send out(v2); receive in2(v2); }
      }

   if  (v1 == EOS) while (v2 ≠ EOS) {send out(v2); receive in2(v2);}
   else  while (v1 ≠ EOS) {send out(v1); receive in1(v1);}

   send out (EOS);
}
```

Productores y consumidores (*filtro)*

Red de Ordenación



Productores y consumidores (*filtro)*

Red de Ordenación

- *n*-1 procesos; el ancho de la red es $\log_2 n$.
- Canales de entrada y salida compartidos
- Puede programarse usando:
  - ***Static naming*** (arreglo global de canales, y cada instancia de *Merge* recibe desde 2 elementos del arreglo y envía a otro ⇒ embeber el árbol en un arreglo).
  - ***Dynamic naming*** (canales globales, parametrizar los procesos, y darle a cada proceso 3 canales al crearlo; todos los *Merge* son idénticos, pero se necesita un coordinador).
- Los filtros podemos conectarlos de distintas maneras. Solo se necesita que la salida de uno cumpla las suposiciones de entrada del otro ⇒ pueden reemplazarse si se mantienen los comportamientos de entrada y salida.

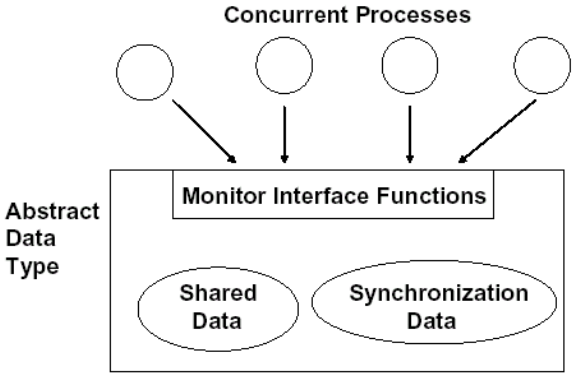
Clientes y Servidores.
Monitores Activos

- ***Servidor:*** proceso que maneja pedidos (“*requests*”) de otros procesos ***clientes***. ¿Cómo implementamos *C/S* con *PMA*?

- Dualidad entre monitores y PM: cada uno de ellos puede simular al otro.

Monitor ⇒ *manejador de recurso.* Encapsula variables permanentes que registran el estado, y provee un conjunto de procedures. Los simulamos, usando procesos servidores y PM, como procesos activos en lugar de como conjuntos pasivos de procedures.

```
monitor Mname
 {   declaración de  variables permanentes;
     código de inicialización;
     procedure op(formales) { cuerpo de op;}
 }
```



Clientes y Servidores.

Monitores Activos

- Un ***Servidor*** es un proceso que maneja pedidos (requerimientos) de otros procesos clientes. Veremos cómo implementar Cliente/Servidor con PMA.

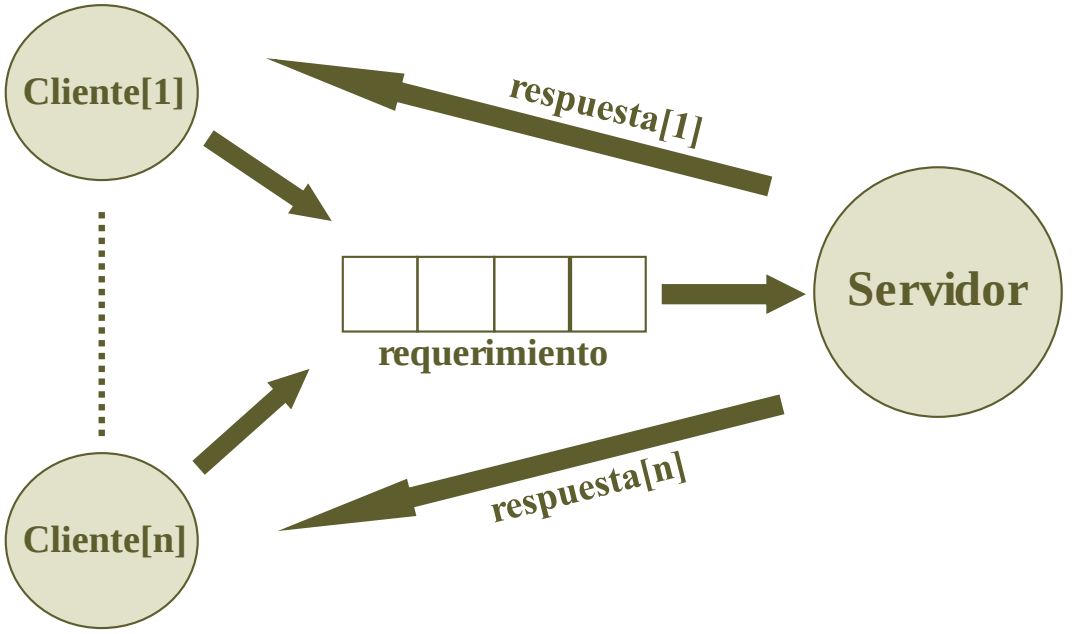
- Un proceso ***Cliente*** que envía un mensaje a un canal de requerimientos general, luego recibe el resultado desde un canal de respuesta propio (¿por qué?).

- En un sistema distribuido, lo natural es que el proceso ***Servidor*** resida en un procesador físico y M procesos ***Cliente*** residan en otros N procesadores (N<= M).

Clientes y Servidores.

Monitores Activos – 1 operación

- Para simular ***Mname***, usamos un proceso server ***Servidor***.
- Las variables permanentes serán variables locales de ***Servidor***.
- Llamado: un proceso ***cliente*** envía un mensaje a un canal de ***requerimiento***.
- Luego recibe el resultado por un canal de ***respuesta*** propio



Clientes y Servidores.

Monitores Activos – 1 operación

```
chan requerimiento (int idCliente, tipos de los valores de entrada );
chan respuesta[n] (tipos de los resultados );

Process Servidor
      {  int idCliente;
         declaración de variables permanentes;
         código de inicialización;
         while (true)
            { receive requerimiento (IdCliente, valores de entrada);
              cuerpo de la operación op;
              send respuesta[IdCliente] (resultados);
            }
      }

Process Cliente [i = 1 to n]
      {  send requerimiento (i, argumentos);
         receive respuesta[i] (resultados);
      }
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones

- Podemos generalizar esta solución de *C/S* con una única operación para considerar múltiples operaciones.
- El ***IF*** del *Servidor* será un ***CASE*** con las distintas clases de operaciones.
- El cuerpo de cada operación toma datos de un canal de entrada en ***args*** y los devuelve *al cliente adecuado* en resultados.

```
type clase_op = enum(op1, ..., opn);
type tipo_arg = union(arg1 : tipoAr1, ..., argn : tipoArn );
type tipo_result = union(res1 : tipoRe1, ..., resn : tipoRen );

chan request(int idCliente, clase_op, tipo_arg);
chan respuesta[n](tipo_result);

Process Servidor .....
Process Cliente [i = 1 to n]   .....
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones

```
Process Servidor
{  int IdCliente; clase_op oper; tipo_arg args;
   tipo_result resultados;
   código de inicialización;
   while ( true)
      {  receive request(IdCliente, oper, args);
         if ( oper == op1 ) { cuerpo de op1;}
            .....
         elsif ( oper == opn ) { cuerpo de opn;}
         send respuesta[IdCliente](resultados);
      }
}

Process Cliente [i = 1 to n]
{  tipo_arg mis_args;
   tipo_result  mis_resultados;
   send request(i, opk, mis_args);
   receive respuesta[i] (mis_resultados);
}
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones y variables condición

- Hasta ahora el monitor no requería variables condición ya que el *Servidor* no requería demorar la atención de un pedido de servicio. ***Caso general:*** monitor con múltiples operaciones y con sincronización por condición. Para los clientes, la situación es transparente ⇒ ***cambia el servidor.***

- Consideramos un caso específico de manejo de múltiples unidades de un recurso (ejemplos: *bloques de memoria, impresoras*).
  - Los clientes "adquieren" y devuelven unidades del recurso.
  - Las unidades libres se insertan en un "conjunto" sobre el que se harán las operaciones de INSERTAR y REMOVER.
  - El número de unidades disponibles es lo que "controla" nuestra variable de sincronización por condición.

```
Monitor Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades  = valores iniciales;
    cond libre;

    procedure adquirir( int  *Id )
            {  if (disponible == 0)  wait(libre)
               else  disponible --;
               remove(unidades, id);
            }

    procedure liberar(int id )
            {   insert(unidades, id);
                if (empty(libre)) disponible ++
                else signal(libre);
            }
 }
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones y variables condición

- Caso en que el servidor tiene dos operaciones:
  - Si no hay unidades disponibles, el servidor no puede esperar hasta responder al pedido ⇒ debe salvarlo y diferir la respuesta.
  - Cuando una unidad es liberada, atiende un pedido salvado (si hay) enviando la unidad.

```
type clase_op = enum(adquirir, liberar);
chan request(int idCliente, claseOp oper, int idUnidad );
chan respuesta[n] (int id_unidad);

Process Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    queue pendientes;
    while (true)
     {  receive request (IdCliente, oper, id_unidad);
        if (oper == adquirir)
           {   if (disponible > 0)
                  {  disponible = disponible - 1;
                     remove (unidades, id_unidad);
                     send respuesta[IdCliente] (id_unidad);
                  }
               else  push (pendientes, IdCliente);
           }
        else
           { if empty (pendientes)
                {  disponible= disponible + 1;
                   insert(unidades, id_unidad);
                }
             else
                {  pop (pendientes, IdCliente);
                   send respuesta[IdCliente](id_unidad);
                }
           }
     } //while
}    //process  Administrador_Recurso

Process Cliente[i = 1 to n]
{   int id_unidad;
    send request(i, adquirir, 0);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send request(i, liberar, id_unidad);
}
```

Clientes y Servidores.

Monitores Activos – Múltiples operaciones y variables condición

- El monitor y el Servidor muestran la dualidad entre monitores y PM: hay una correspondencia directa entre los mecanismos de ambos.
- La eficiencia de monitores o PM depende de la arquitectura física de soporte:
  - Con MC conviene la invocación a procedimientos y la operación sobre variables condición.
  - Con arquitecturas físicamente distribuidas tienden a ser más eficientes los mecanismos de PM.
- Dualidad entre Monitores y Pasaje de Mensajes

Programas con Monitores		***Programas basados en PM***
• Variables permanentes	↔	Variables locales del servidor
• Identificadores de procedures	↔	Canal *request* y tipos de operación
• Llamado a procedure	↔	*send request*( ); *receive respuesta*
• Entry del monitor	↔	*receive request*( )
• Retorno del procedure	↔	*send respuesta*( )
• Sentencia *wait*	↔	Salvar pedido pendiente
• Sentencia *signal*	↔	Recuperar/ procesar pedido pendiente
• Cuerpos de los procedure	↔	Sentencias del “case” de acuerdo a la clase de operación.

Clientes y Servidores.

Sentencia de Alternativa Múltiple

- Resolución del mismo problema con sentencias de alternativa múltiple. Ventajas y desventajas

```
chan pedido (int idCliente);
chan liberar (int idUnidad);
chan respuesta[n] (int idUnidad);

Process Administrador_Recurso
{   int disponible = MAXUNIDADES;
    set unidades = valor inicial disponible;
    int id_unidad, idCliente;
    while (true)
     {  if  ( (not empty(pedido) and (disponible >0) ) →
              receive pedido (idCliente);
              disponible = disponible - 1;
              remove (unidades, id_unidad);
              send respuesta[idCliente] (id_unidad);
        □ (not empty(liberar)) →
              receive liberar (id_unidad);
               disponible= disponible + 1;
               insert(unidades, id_unidad);
        } //if
      }  //while
  }    //process  Administrador_Recurso
```

```
Process Cliente[i = 1 to n]
 {  int id_unidad;

    send pedido (i);
    receive respuesta[i](id_unidad);
     //Usa la unidad
    send liberar (id_unidad);
 }
```

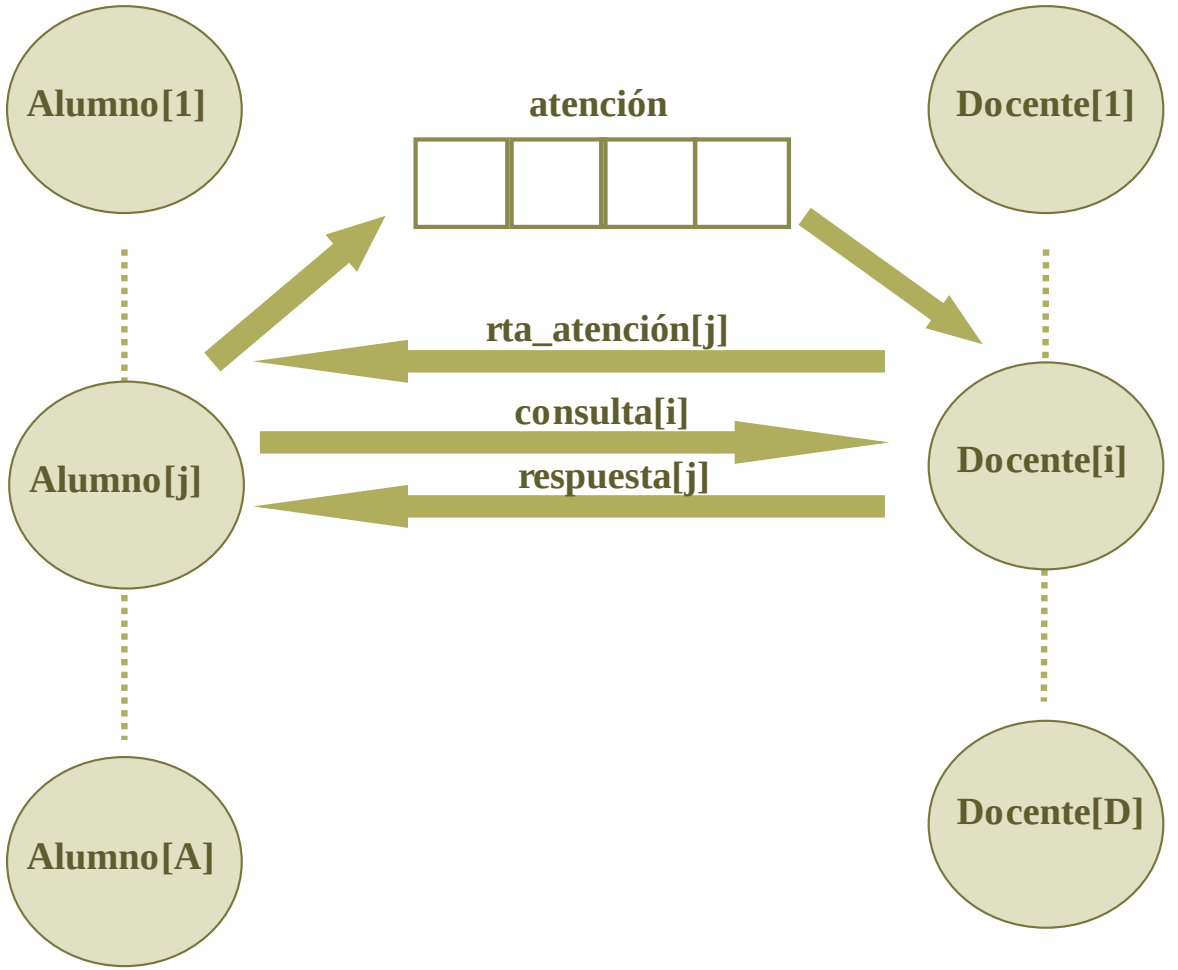
Clientes y Servidores.
Continuidad Conversacional

- Existen ***A*** alumnos que hacen consultas a ***D*** docentes.
- El alumno espera hasta que un docente lo atiende, y a partir de ahí le comienza a realizar las diferentes consultas hasta que no le queden dudas.
- Los alumnos son los procesos "*clientes*", y los docentes los procesos "*Servidores*". Los procesos servidores son idénticos, y cualquiera de ellos que esté libre puede atender un requerimiento de un alumno.

*Todos los alumnos pueden pedir atención por un **canal global** y recibirán respuesta de un docente dado por un **canal propio**. ¿Por qué?*

Clientes y Servidores.

Continuidad Conversacional



Clientes y Servidores.
Continuidad Conversacional

```
chan atención (int);
chan consulta[D] (string);
chan rta_atención[A](int);
chan respuesta[A] (string);

Process Alumno [a = 1 to A]
    {  int idDocente;
       string preg, res;
       send atención (a);
       receive rta_atención[a] (idDocente);
       while (tenga consultas para hacer)
          { send consulta[idDocente](preg);
            receive respuesta[a](res);
          }
       send consulta [idDocente] ('FIN');
     }
```

```
Process Docente [d = 1 to D]
{    string preg, res;
     int idAlumno;
     bool seguir = false;

     while (true)
       {  receive atención (idAlumno);
          send rta_atención[idAlumno](d);
          seguir = true;
          while (seguir)
             { receive consulta[d](preg);
               if (preg == 'FIN')  seguir = false
               else
                    { res = resolver la pregunta (preg)
                      send respuesta [idAlumno](res);
                    }
             }
       }
}
```

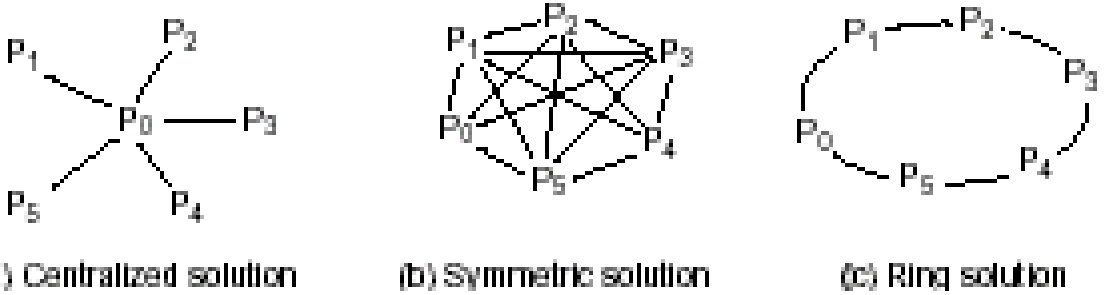
- Este ejemplo de interacción entre clientes y servidores se denomina ***continuidad conversacional*** (desde la solicitud de atención hasta la última consulta).
- ***atención*** es un canal compartido por el que cualquier *Docente* puede recibir. Si cada canal puede tener un solo receptor, se necesita otro proceso intermedio. ¿Para qué?.

Pares (peers) interactuantes.

Intercambio de Valores

Problema: cada proceso tiene un dato local *V* y los *N* procesos deben saber cuál es el menor y cuál el mayor de los valores.

Ejemplo donde los procesadores están conectados por tres modelos de arquitectura: ***centralizado***, ***simétrico*** y en ***anillo circular***.



Pares (peers) interactuantes.

Intercambio de Valores – Solución centralizada

- Cada proceso tiene un valor ***V*** local. Al final todos los procesos deben conocer el mínimo y máximo valor de todo el sistema.
- La arquitectura centralizada es apta para una solución en que todos envían su dato local ***V*** al procesador central, éste ordena los ***N*** datos y reenvía la información del mayor y menor a todos los procesos $\Rightarrow$ ***2(N-1)*** mensajes.

```
chan valores(int),  resultados[n-1] (int minimo, int maximo);
```

```
Process P[0]
{  int v; int nuevo, minimo = v, máximo = v;
   for [i=1 to n-1]
      {    receive valores (nuevo);
           if (nuevo < minimo)  minimo = nuevo;
           if (nuevo > maximo)  maximo = nuevo;
       }
   for [i=1 to n-1]
       send resultados [i-1] (minimo, maximo);
}
```

```
Process P[i=1 to n-1]
{  int v;  int minimo, máximo;
   send valores (v);
   receive resultados[i-1](minimo, maximo);
}
```

- ¿Se puede usar un único canal de resultados?

Pares (peers) interactuantes.

Intercambio de Valores – Solución simétrica

- En la arquitectura simétrica o "full conected" hay un canal entre cada par de procesos. Todos los procesos ejecutan el mismo algoritmo.
- Cada proceso trasmite su dato local ***V*** a los ***N-1*** restantes procesos. Luego recibe y procesa los ***N-1*** datos que le faltan, de modo que en paralelo toda la arquitectura está calculando el mínimo y el máximo y toda la arquitectura tiene acceso a los ***N*** datos.
- Ejemplo de solución *SPMD*: cada proceso ejecuta el mismo programa pero trabaja sobre datos distintos $\Rightarrow$***N(N-1)*** mensajes.
- Si disponemos de una primitiva de ***broadcast***, serán nuevamente ***N*** mensajes.

```
chan valores[n] (int);
Process P[i=0 to n-1]
{ int v=…, nuevo, minimo = v, máximo=v;
  for [k=0 to n-1  st  k <> i ]
      send valores[k] (v);
  for [k=0 to n-1  st  k <> i ]
    { receive valores[i] (nuevo);
      if (nuevo < minimo) minimo = nuevo;
      if (nuevo > maximo) maximo = nuevo;
    }
}
```

- ¿Se puede usar un único canal?

Pares (peers) interactuantes.

Intercambio de Valores – Solución en anillo circular

- Un tercer modo de organizar la solución es tener un anillo donde ***P[i]*** recibe mensajes de ***P[i-1]*** y envía mensajes a ***P[i+1]***. ***P[n-1]*** tiene como sucesor a ***P[0]***.
- Esquema de 2 etapas. En la primera cada proceso recibe dos valores y los compara con su valor local, trasmitiendo un máximo local y un mínimo local a su sucesor. En la segunda etapa todos deben recibir la circulación del máximo y el mínimo global.
  - ***P[0]*** deberá ser algo diferente para "arrancar" el procesamiento.
  - Se requerirán (***2N)-1*** mensajes.
  - Notar que si bien el número de mensajes es lineal (igual que en la centralizada) los tiempos pueden ser muy diferentes. ¿Por qué?.

```
chan valores[n] (int minimo, int maximo);
Process P[0]
{ int v=…, minimo = v, máximo=v;
  send valores[1] (minimo, maximo);
  receive valores[0] (minimo, maximo);
  send valores[1] (minimo, maximo);
}

Process P[i=1 to n-1]
{ int v=…, minimo, máximo;
  receive valores[i] (minimo, maximo);
  if (v<minimo) minimo = v;
  if (v> maximo) maximo = v;
  send valores[(i+1) mod n] (minimo, maximo);
  receive valores[i] (minimo, maximo);
  if (i < n-1) send valores[i+1] (minimo, maximo);
 };
```

Pares (peers) interactuantes.
Comentarios sobre las soluciones

- ***Simétrica*** es la más corta y sencilla de programar, pero usa el mayor número de mensajes (si no hay broadcast). Pueden transmitirse en paralelo si la red soporta transmisiones concurrentes, pero el overhead de comunicación acota el speedup.
- ***Centralizada y anillo*** usan n° lineal de mensajes, pero tienen distintos patrones de comunicación que llevan a distinta performance:
  - En ***centralizada***, los mensajes al coordinador se envían casi al mismo tiempo ⇒ sólo el primer *receive* del coordinador demora mucho.
  - En ***anillo***, todos los procesos son *productores y consumidores*. El último tiene que esperar a que todos los otros (uno por vez) reciban un mensaje, hacer poco cómputo, y enviar su resultado.

    Los mensajes circulan 2 veces completas por el anillo ⇒ Solución inherentemente lineal y lenta para este problema.

Programación Concurrente

Clase 8

Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Pasaje de Mensajes Sincrónicos (PMS): https://drive.google.com/uc?id=1xJS1-HKs6FMWRfH6tprV2vZ0Nu-2Hj0R&export=download

- CSP – Lenguaje para PMS: https://drive.google.com/u/1/uc?id=1TgzNTMds7QPzHpLaYY4LjzM9lOVeB9Xx&export=download

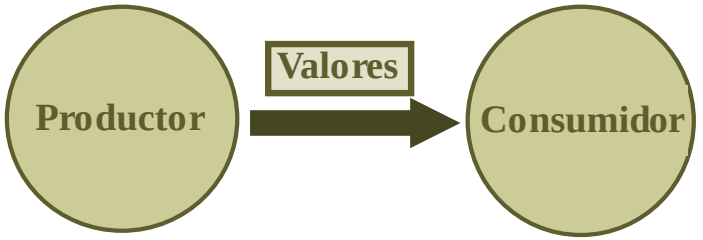
Pasaje de Mensajes Sincrónicos (PMS)

Diferencia con PMA

- Los canales son de tipo ***link*** o punto a punto (1 emisor y 1 receptor).
- La diferencia entre *PMA* y *PMS* es la primitiva de trasmisión ***Send***. En *PMS* es bloqueante y la llamaremos (por ahora) ***sync_send***.
  - El trasmisor queda esperando que el mensaje sea recibido por el receptor.
  - La cola de mensajes asociada con un *send* sobre un canal se reduce a 1 mensaje $\Rightarrow$ **menos** memoria.
  - Naturalmente el grado de concurrencia se reduce respecto de la sincronización por PMA *(los emisores se bloquean).*
- Si bien ***send*** y ***sync_send*** son similares (en algunos casos intercambiables) la semántica es diferente y las posibilidades de ***deadlock*** son mayores en comunicación sincrónica.

Ejemplo: *productor / consumidor*

```
chan valores(int);

Process Productor
{  int datos[n];
   for [i=0 to n-1]
      {  #Hacer cálculos productor
         sync_send valores (datos[i]);
      }
}

Process Consumidor
{  int resultados[n];
   for [i=0 to n-1]
      {  receive valores (resultados[i]);
         #Hacer cálculos consumidor
      }
}
```



Comentarios de PMS

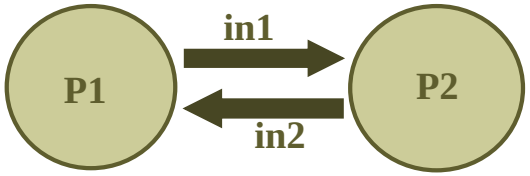
- Si los cálculos del productor se realizan mucho más rápido que los del consumidor en las primeras ***n/2*** operaciones, y luego se realizan mucho más lento durante otras ***n/2*** interacciones:
  - Con PMS los pares send/receive se completarán asumiendo la demora del proceso que más tiempo consuma. Si la relación de tiempo fuera 10 a 1 significaría multiplicar por 10 los tiempos totales.
  - Con PMA, al principio el productor es más rápido y sus mensajes se encolan. Luego el consumidor es más rápido y "descuenta" tiempo consumiendo la cola de mensajes.
- ***Mayor concurrencia en PMA.*** Para lograr el mismo efecto en PMS se debe interponer un proceso ***"buffer"***.
- ¿Que pasa si existe más de un productor/consumidor?.

Comentarios de PMS

- La concurrencia también se reduce en algunas interacciones Cliente/Servidor.
  - Cuando un cliente está liberando un recurso, no habría motivos para demorarlo hasta que el servidor reciba el mensaje, pero con PMS se tiene que demorar.
  - Otro ejemplo se da cuando un cliente quiere escribir en un display gráfico, un archivo u otro dispositivo manejado por un proceso servidor. Normalmente el cliente quiere seguir inmediatamente después de un pedido de write.
- Otra desventaja del PMS es la mayor probabilidad de ***deadlock***. El programador debe ser cuidadoso de que todas las sentencias ***send*** (sync_send) y ***receive*** hagan matching.

Deadlock en PMS

- Dos procesos que intercambian valores.




```
chan in1(int), in2(int);

Process P1
{    int valorA = 1, valorB;
     sync_send in2(valorA);
     receive in1(valorB);
}

Process P2
{    int valorA, valorB = 2;
     sync_send in1(valorB);
     receive in2 (valorA);
}
```

```
chan in1(int), in2(int);

Process P1
{    int valorA = 1, valorB;
     sync_send in2(valorA);
     receive in1(valorB);
}

Process P2
{    int valorA, valorB = 2;
     receive in2 (valorA);
     sync_send in1(valorB);
}
```

CSP– Lenguaje para PMS

El lenguaje CSP (Hoare, 1978)

- CSP (Communicating Sequential Processes, Hoare 1978) fue uno de los desarrollos fundamentales en Programación Concurrente. Muchos lenguajes reales (OCCAM, ADA, SR) se basan en CSP.
- Las ideas básicas introducidas por Hoare fueron PMS y ***comunicación guardada***: PM con waiting selectivo.
- ***Canal***: link directo entre dos procesos en lugar de mailbox global. Son *half-duplex* y nominados.
- Las sentencias de Entrada (**?** o **query**) y Salida (**!** o **shriek** o **bang**) son el único medio por el cual los procesos se comunican.

  process A { . . . B ! e; . . . } **process B { . . . A ? x; . . . }**

- Para que se produzca la comunicación, deben ***matchear***, y luego ***se ejecutan simultáneamente***.
- Efecto: sentencia de ***asignación distribuida***.

El lenguaje CSP (Hoare, 1978)

- Formas generales de las sentencias de comunicación:

Destino ! port(e_1, ..., e_n);

Fuente ? port(x_1, ..., x_n);

  - *Destino* y *Fuente* nombran un proceso simple, o un elemento de un arreglo de procesos. *Fuente* puede nombrar *cualquier* elemento de un arreglo (*Fuente*[*]).
  - ***port*** son etiquetas que se usan para distinguir entre distintas clases de mensajes que un proceso podría recibir (puede omitirse si es sólo uno).
  - Dos procesos se comunican cuando ejecutan sentencias de comunicación que hacen ***matching***.

***A* ! canaluno(dato);** ** *B* ? canaluno(resultado);**

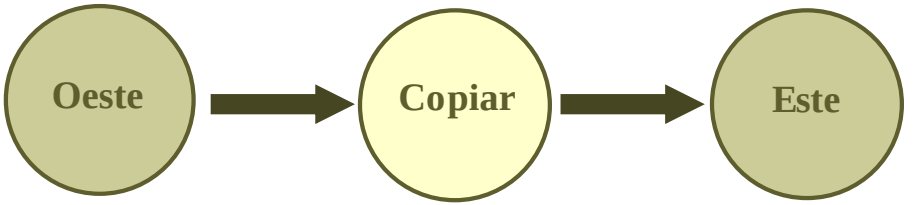
Ejemplos básicos

- *Proceso* filtro que copia caracteres recibidos del proceso *Oeste* al proceso *Este*:

```
Process Oeste
{   char c;
    do true →
            Generar(c);
            Copiar ! (c);
    od
}
```

```
Process Copiar
{   char c;
    do true →
            Oeste ? (c) ;
            Este ! (c);
    od
}
```

```
Process Este
{   char c;
    do true →
            Copiar ? (c) ;
            Usar(c);
    od
}
```



Ejemplos básicos

- Server que calcula el MCD de dos enteros con el algoritmo de Euclides. ***MCD*** espera recibir entrada en su port ***args*** desde un cliente. Envía la respuesta al port ***resultado*** del cliente.

```
Process MCD
{   int id, x, y;
    do true →
            Cliente[*] ? args(id, x, y);
            do  x > y → x := x - y;
            □   x < y → y := y - x;
            od
            Cliente[id] ! resultado(x);
    od
}
```

- *Cliente[i]* se comunica con *MCD* ejecutando:

```
...
MCD ! args(i, v1, v2);
MCD ? resultado(r);
...
```

Comunicación Guardada

- Limitaciones de **?** y **!** ya que son bloqueantes. Hay problema si un proceso quiere comunicarse con otros (quizás por ≠ ports) sin conocer el orden en que los otros quieren hacerlo con él.
- Por ejemplo, el proceso Copiar podría extenderse para hacer buffering de k caracteres: si hay más de 1 pero menos de k caracteres en el buffer, Copiar podría recibir otro carácter o sacar 1.

- Las operaciones de comunicación (**?** y **!**) pueden ser ***guardadas***, es decir hacer un AWAIT hasta que una condición sea verdadera.

- El **do** e **if** de CSP usan los *comandos guardados* de Dijkstra (B → S).

Comunicación Guardada

Las sentencias de comunicación guardada soportan comunicación no determinística:

$$B; C \rightarrow S;$$

- ***B*** puede omitirse y se asume *true*.
- ***B*** y ***C*** forman la guarda.
- La guarda tiene éxito si ***B*** es *true* y ejecutar ***C*** no causa demora.
- La guarda falla si ***B*** es *falsa*.
- La guarda se bloquea si ***B*** es *true* pero ***C*** no puede ejecutarse inmediatamente.

Comunicación Guardada

Las sentencias de comunicación guardadas aparecen en ***if*** y ***do***.

if B_1***; comunicación***$_1 \rightarrow S_1$***;***
□ B_2***; comunicación***$_2 \rightarrow S_2$***;***
fi

Ejecución:

- *Primero*, se evalúan las guardas.
  - Si todas las guardas fallan, el if termina sin efecto.
  - Si al menos una guarda tiene éxito, se elige una de ellas (no determinísticamente).
  - Si algunas guardas se bloquean, se espera hasta que alguna de ellas tenga éxito.
- *Segundo*, luego de elegir una guarda exitosa, se ejecuta la sentencia de comunicación de la guarda elegida.
- *Tercero*, se ejecuta la sentencia S_i.

La ejecución del do es similar (se repite hasta que todas las guardas fallen).

Comunicación Guardada

Podemos re-programar Copiar para usar comunicación guardada:

```
Process Copiar
    {   char c;
        do Oeste ? (c) → Este!(c);
        od
    }
```

Extendemos *Copiar* para manejar un buffer de tamaño 2. Luego de ingresar un carácter, el proceso que copia puede estar recibiendo un segundo carácter de Oeste o enviando uno a Este.

```
Process Copiar2
    {   char c1, c2;
        Oeste ? (c1);
        do Oeste ? (c2) → Este ! (c1) ;
                          c1=c2;
        □  Este ! (c1) → Oeste? (c1);
        od
    }
```

Ejemplo

Copiar con un buffer limitado

```
Process Copiar
{  char buffer[80];
   int front = 0, rear = 0, cantidad = 0;
   do cantidad <80; Oeste?(buffer[rear]) → cantidad = cantidad + 1;
                                           rear = (rear + 1) MOD 80;
    □ cantidad >0; Este!(buffer[front]) → cantidad := cantidad - 1;
                                          front := (front + 1) MOD 80;
   od
}
```

- Con ***PMA***, procesos como *Oeste* e *Este* ejecutan a su propia velocidad pues hay buffering implícito.
- Con ***PMS***, es necesario programar un proceso adicional para implementar buffering si es necesario.

Ejemplo

Asignación de Recursos

```
Process Alocador
{   int disponible = MaxUnidades;
    set unidades = valores iniciales;
    int indice, idUnidad;

    do disponible >0;  cliente[*] ? acquire(indice) → disponible = disponible -1;
                                                      remove (unidades, idUnidad);
                                                      cliente[indice] ! reply(idUnidad);
    □  cliente[*] ? release(indice, idUnidad) →  disponible = disponible +1;
                                                 insert (unidades, idUnidad);
    od
}
```

La solución es concisa. Usa múltiples ***ports*** y un brazo del ***do*** para atender cada una. Se demora en un mensaje ***acquire*** hasta que haya unidades, y no es necesario salvar los pedidos pendientes.

Ejemplo

Intercambio de Valores

```
Process P1
{    int valor1 = 1, valor2;
     if  P2 ! (valor1) → P2 ? (valor2);
     □ P2 ? (valor2) → P2 ! (valor1);
     fi
}

Process P2
{    int valor1 , valor2 = 2;
     if  P1 ! (valor2) → P1 ? (valor1);
     □ P1 ? (valor1) → P1 ! (valor2);
     fi
}
```

Esta solución simétrica **NO** tiene *deadlock* porque el no determinismo en ambos procesos hace que se acoplen las comunicaciones correctamente. Si bien es simétrica, es más compleja que la de PMA...

Ejemplo

*La Criba de Eratóstenes para la g*eneración de números primos

- **Problema:** generar todos los primos entre 2 y n $\rightarrow$ *2* 3 4 5 6 7 8 . . . N
- Comenzando con el primer número (2), recorremos la lista y borramos los múltiplos de ese número. Si n es impar: 2 3 5 7 . . . n
- Pasamos al próximo número (3) y borramos sus múltiplos.
- Siguiendo hasta que todo número fue considerado, los que quedan son todos los primos entre 2 y n.
- La criba *captura* primos y *deja caer* múltiplos de los primos.
- ***¿Cómo paralelizar?*** Pipe de procesos filtro: cada uno recibe un stream de números de su predecesor y envía un stream a su sucesor. El primer número que recibe es el próximo primo, y pasa los *no múltiplos*.

Ejemplo

*La Criba de Eratóstenes para la g*eneración de números primos

```
Process Criba[1]
{    int p = 2;

     for [i = 3 to n by 2]  Criba[2] ! (i);
}

Process Criba[i = 2 to L]
{    int p, proximo;

     Criba[i-1] ? (p);
     do Criba[i-1] ? (proximo) →
                  if  ((proximo MOD p) <> 0 ) and (i < L) →  Criba[i+1] ! (proximo);
     od
}
```

- El número total de procesos *Criba* (*L*) debe ser lo suficientemente grande para garantizar que se generan todos los primos hasta *n*.
- Excepto *Criba*[1], los procesos terminan bloqueados esperando un mensaje de su predecesor. Cuando el programa para, los valores de *p* en los procesos son los primos. Puede modificarse con centinelas.

Ejemplo

Ordenación de un Arreglo

- **Problema:** ordenar un arreglo de n valores en paralelo (n par, orden no decreciente).
- Dos procesos *P1* y *P2*, cada uno inicialmente con ***n/2*** valores (arreglos ***a1*** y ***a2*** respectivamente).
- Los ***n/2*** valores de cada proceso se encuentran ordenados inicialmente.
- **Idea:** realizar una serie de intercambios. En cada uno *P1* y *P2* intercambian ***a1[mayor]*** y ***a2[menor]***, hasta que *a1[mayor] < a2[menor]*.

Ejemplo

Ordenación de un Arreglo

```
Process P1
{  int nuevo, a1[1:n/2]; const mayor = n/2;
   ordenar a1 en orden no decreciente
   P2 ! (a1[mayor]);
   P2 ? (nuevo)
   do  a1[mayor] > nuevo →
       poner nuevo en el lugar correcto en a1, descartando el viejo a1[mayor]
       P2 ! (a1[mayor]);
       P2 ? (nuevo);
    od
}

Process P2
{  int nuevo, a2[1:n/2]; const menor = 1;
   ordenar a2 en orden no decreciente
   P1 ? (nuevo);
   P1 ! (a2[menor]);
   do  a2[menor] < nuevo →
       poner nuevo en el lugar correcto en a2, descartando el viejo a2[menor]
       P1 ? (nuevo);
       P1 ? (a2[menor]);
    od
}
```

Ejemplo

Ordenación de un Arreglo

- Notar que en la implementación del intercambio, las sentencias de entrada y salida son bloqueantes → usamos una solución asimétrica.
- Para evitar *deadlock*, *P1* primero ejecuta una salida y luego una entrada, y *P2* ejecuta primero una entrada y luego una salida.
- ***Comunicación guardada para programar una solución simétrica:*** esta solución es más costosa de implementar.

```
P1: … if P2 ? (nuevo) → P2 ! (a1[mayor])
      □ P2 ! (a1[mayor]) → P2 ? (nuevo)
      fi…

P2: … if P1 ? (nuevo) → P1 ! (a2[menor])
      □ P1 ! (a2[menor]) → P1 ? (nuevo)
      fi …
```

- Mejor caso → los procesos intercambian solo un par de valores.
- Peor caso → intercambian n/2 + 1 valores: n/2 para tener cada valor en el proceso correcto y uno para detectar terminación.

Ejemplo

Ordenación de un Arreglo

- Solución con ***b*** procesos ***P[1:b]***, inicialmente con ***n/b*** valores cada uno.
- Cada uno primero ordena sus ***n/b*** valores. Luego ordenamos los ***n*** elementos usando aplicaciones paralelas repetidas del algoritmo compare-and-exchange
- Cada proceso ejecuta una serie de rondas:
  - En las impares, cada proceso con número impar juega el rol de ***P1***, y cada proceso con número par el de ***P2***.
  - En las rondas pares, cada proceso numerado par juega el rol de ***P1***, y cada proceso impar el rol de ***P2***.
- **Ejemplo:** *b=4* - Algoritmo odd/even exchange sort

ronda	**P[1]**	**P[2]**	**P[3]**	**P[4]**
0	8	7	6	5
1	7	8	5	6
2	7	5	8	6
3	5	7	6	8
4	5	6	7	8

Ejemplo

Ordenación de un Arreglo

- Cada intercambio progresa hacia una lista totalmente ordenada.
- ¿Cómo pueden detectar los procesos si toda la lista está ordenada?. Un proceso individual no puede detectar que la lista entera está ordenada después de una ronda pues conoce solo dos porciones:
  - Se puede usar un coordinador separado. Después de cada ronda, los procesos le dicen a éste si hicieron algún cambio a su porción.
    - ✓ ***2b*** mensajes de overhead en cada ronda.
  - Que cada proceso ejecute suficientes rondas para garantizar que la lista estará ordenada (en general, al menos ***b*** rondas).
    - ✓ Cada proceso intercambia hasta ***n/b+1*** mensajes por ronda.
    - ✓ El algoritmo requiere hasta $b^2*(n/b + 1)$ intercambio de mensajes.

Programación Concurrente

Clase 9

Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Paradigmas de Interacción entre Procesos: https://drive.google.com/uc?id=1a0QUfXjpdM26pTIGzSEjm7zGRtCxAs9d&export=download

- Librería para Pasaje de Mensajes (MPI): https://drive.google.com/uc?id=1tlOM5BaPD2KSlRIUVYWnwO-8SDqLPlE8&export=download

Paradigmas de Interacción entre Procesos

Paradigmas para la interacción entre procesos

- 3 esquemas básicos de interacción entre procesos: *productor/consumidor*, *cliente/servidor* e *interacción entre pares*.
- Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros **paradigmas** o modelos de interacción entre procesos.

Paradigma 1: *master / worker*
Implementación distribuida del modelo *Bag of Task*.

Paradigma 2: *algoritmos heartbeat*
Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

Paradigma 3: *algoritmos pipeline*
La información recorre una serie de procesos utilizando alguna forma de receive/send.

Paradigmas para la interacción entre procesos

Paradigma 4: *probes (send) y echoes(receive)*
La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información.

Paradigma 5: *algoritmos broadcast*
Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

Paradigma 6: *token passing*
En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

Paradigma 7: *servidores replicados*
Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

Paradigmas para la interacción entre procesos

Manager/Worker

- El concepto de ***bag of tasks*** usando variables compartidas supone que un conjunto de workers comparten una "bolsa" con tareas independientes. Los workers sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa (ejemplo en LINDA manejando un espacio compartido de tuplas).

- La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers.

- Analizaremos la implementación de este paradigma con mensajes en lugar de MC. Para esto un proceso *manager* implementará la "bolsa" manejando las tasks, comunicándose con los workers y detectando fin de tareas. **Se trata de un esquema C/S**.

- Ejemplo: multiplicación de matrices ralas.

Paradigmas para la interacción entre procesos

Heartbeat

- Paradigma *heartbeat* ⇒ útil para soluciones iterativas que se quieren paralelizar.
- Usando un esquema "*divide & conquer*" se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte.
- Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos.
- Cada "paso" debiera significar un progreso hacia la solución.
- Formato general de los worker:

```
process worker [i =1 to numWorkers]
  {  declaraciones e inicializaciones locales;
     while (no terminado)
       {  send valores a los workers vecinos;
          receive valores de los workers vecinos;
          Actualizar valores locales;
       }
  }
```

- Ejemplo: grid computations (imágenes), autómatas celulares (simulación de fenómenos como incendios o crecimiento biológico).

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Los procesadores están conectados por canales bidireccionales. Cada uno se comunica sólo con sus vecinos y conoce esos links.

¿Cómo puede cada procesador determinar la topología completa de la red?

- Modelización:
  - Procesador ⇒ proceso
  - Links de comunicación ⇒ canales compartidos.
- Soluciones: los vecinos interactúan para intercambiar información local.

Algoritmo Heartbeat: se expande enviando información; luego se contrae incorporando nueva información.

- Procesos ***Nodo[p:1..n]***.
- Vecinos de ***p***: ***vecinos[1:n]*** → vecinos[q] es true si q es vecino de p.
- ***Problema:*** computar ***top*** (matriz de adyacencia), donde ***top[p,q]*** es true si p y q son vecinos.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Cada nodo debe ejecutar un nº de rondas para conocer la topología completa. Si el diámetro ***D*** de la red es conocido se resuelve con el siguiente algoritmo.

```
chan topologia[1:n] ([1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n],  bool nuevatop[1:n,1:n],  top[1:n,1:n] = ([n*n] false);
   top[p,1..n] = vecinos;

   for (r =0 ; r < D; r++)
      { for [q = 1 to n st vecinos[q] ] send topologia[q](top);
        for [q = 1 to n st vecinos[q] ]
          {  receive topologia[p](nuevatop);
             top = top or nuevatop;
          }
      }
 }
```

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

- Rara vez se conoce el valor de ***D***.
- Excesivo intercambio de mensajes $\Rightarrow$ los procesos cercanos al "centro" conocen la topología más pronto y no aprenden nada nuevo en los intercambios.
- El tema de la terminación $\Rightarrow$ ¿local o distribuida?
- ***¿Cómo se pueden solucionar estos problemas?***
  - Después de ***r*** rondas, ***p*** conoce la topología a distancia ***r*** de él. Para cada nodo ***q*** dentro de la distancia ***r*** de ***p***, los vecinos de ***q*** estarán almacenados en la fila ***q*** de ***top*** $\Rightarrow$ ***p*** ejecutó las rondas suficientes tan pronto como cada fila de ***top*** tiene algún valor ***true***.
  - Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos.
- No siempre la terminación se puede determinar localmente.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)

Process Nodo[p:1..n]
{  bool vecinos[1:n], activo[1:n] = vecinos, top[1:n,1:n] = ([n*n]false), nuevatop[1:n,1:n];
   bool qlisto, listo = false;
   int emisor;
   top[p,1..n] = vecinos;
   while (not listo)
     {   for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);
         for [q = 1 to n st activo[q] ]
            {   receive topologia[p](emisor,qlisto,nuevatop);
                top = top or nuevatop;
                if (qlisto) activo[emisor] = false;
             }
          if (todas las filas de top tiene 1 entry true) listo=true;
     }
    for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
    for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop);
}
```

Paradigmas para la interacción entre procesos

Pipeline

- Un pipeline es un arreglo lineal de procesos "filtro" que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida.
- Estos procesos ("workers") pueden estar en procesadores que operan en paralelo, en un primer esquema *a lazo abierto* (W_1 en el INPUT, W_n en el OUTPUT).
- Un segundo esquema es el pipeline *circular*, donde W_n se conecta con W_1. Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe.
- En un tercer esquema posible (*cerrado*), existe un proceso coordinador que maneja la "realimentación" entre W_n y W_1.
- Ejemplo: multiplicación de matrices en bloques.

Paradigmas para la interacción entre procesos

Probe-Echo

- Arboles y grafos son utilizados en muchas aplicaciones distribuidas como búsquedas en la WEB, BD, sistemas expertos y juegos.
- Las arquitecturas distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan.
- DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos en un árbol o grafo. Este paradigma es el análogo concurrente de DFS.
- ***Prueba-eco*** se basa en el envío de un mensajes (“probe”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“echo”).
- Los **probes** se envían en paralelo a todos los sucesores.
- Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un número fijo de nodos activos (ejemplo: redes móviles).

Paradigmas para la interacción entre procesos

Broadcast

- En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva ***broadcast***:

broadcast ch(m);

- Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden.

- Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. Ejemplo: semáforos distribuidos, la base es un ***ordenamiento total de eventos de comunicación*** mediante el uso de ***relojes lógicos***.

Paradigmas para la interacción entre procesos

Token Passing

- Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“token”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida. Un ejemplo del primer tipo de algoritmos es el caso de tener que controlar ***exclusión mutua distribuida***.
- Ejemplos de recolección de información de estado son los algoritmos de detección de terminación en computación distribuida.
- Aunque el problema de la SC se da principalmente en programas de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez. Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BD.
- Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o *token ring* (descentralizado y fair).

Paradigmas para la interacción entre procesos

Servidores Replicados

- Un server puede ser replicado cuando hay múltiples instancias de un recurso: cada server maneja una instancia.
- La replicación también puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.
- Ejemplo: problema de los filósofos
  - Modelo ***centralizado*****:** los Filósofo se comunican con ***UN*** proceso Mozo que decide el acceso o no a los recursos.
  - Modelo ***distribuido*****:** supone **5 procesos Mozo,** cada uno manejando un tenedor. Un Filósofo puede comunicarse con **2** Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos **NO se comunican entre ellos**.
  - Modelo ***descentralizada*****:** cada Filósofo ve **un único** Mozo. Los Mozos se comunican entre ellos (cada uno con sus **2** vecinos) para decidir el manejo del recurso asociado a "su" Filósofo.

Librerías para manejo de PM

Operaciones Send y Receive

- Los prototipos de las operaciones son:

  Send (void *sendbuf, int nelems, int dest)

  Receive (void *recvbuf, int nelems, int source)

- Ejemplo:

P0

```
a = 100;
send(&a, 1, 1);
a = 0;
```

P1

```
receive(&a, 1, 0)
printf("%d\n", a);
```

- La semántica del SEND requiere que en P1 quede el valor 100 (no 0).

- Diferentes protocolos para Send y Receive.

Send y Receive bloqueante

- Para asegurar la semántica del SEND → no devolver el control del Send hasta que el dato a trasmitir esté seguro (Send bloqueante).
- Ociosidad del proceso.
- Hay dos posibilidades:
  - Send/Receive bloqueantes sin buffering.
  - Send/Receive bloqueantes con buffering.

Send y Receive no bloqueante

- Para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente.
- Requiere un posterior chequeo para asegurarse la finalización de la comunicación.
- Deja en manos del programador asegurar la semántica del SEND.
- Hay dos posibilidades:
  - Send/Receive no bloqueantes sin buffering.
  - Send/Receive no bloqueantes con buffering.

MPI

Message Passing Interface

Librería MPI (Interfaz de Pasaje de Mensajes)

- Existen numerosas librerías para pasaje de mensaje (no compatibles).
- MPI define una librería estándar que puede ser empleada desde C o Fortran (y potencialmente desde otros lenguajes).
- El estándar MPI define la sintaxis y la semántica de más de 125 rutinas.
- Hay implementaciones de MPI de la mayoría de los proveedores de hardware.
- Modelo SPMD.
- Todas las rutinas, tipos de datos y constantes en MPI tienen el prefijo "MPI_". El código de retorno para operaciones terminadas exitosamente es MPI_SUCCESS.
- Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send y MPI_Recv.

Librería MPI - Inicio y finalización de MPI

- ***MPI_Init***: se invoca en todos los procesos antes que cualquier otro llamado a rutinas MPI. Sirve para inicializar el entorno MPI.

```
MPI_Init (int *argc, char **argv)
```

Algunas implementaciones de MPI requieren argc y argv para inicializar el entorno

- ***MPI_Finalize***: se invoca en todos los procesos como último llamado a rutinas MPI. Sirve para cerrar el entorno MPI.

```
MPI_Finalize ()
```

Librería MPI - Comunicadores

- Un comunicador define el dominio de comunicación.
- Cada proceso puede pertenecer a muchos comunicadores.
- Existe un comunicador que incluye a todos los procesos de la aplicación MPI_COMM_WORLD.
- Son variables del tipo MPI_Comm → almacena información sobre que procesos pertenecen a él.
- En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

Librería MPI - Adquisición de Información

- ***MPI_Comm_size***: indica la cantidad de procesos en el comunicador.

  MPI_Comm_size (MPI_Comm comunicador, int *cantidad).

- ***MPI_Comm_rank***: indica el "rank" (identificador) del proceso dentro de ese comunicador.

  MPI_Comm_rank (MPI_Comm comunicador, int *rank)

  - rank es un valor entre [0..cantidad]
  - Cada proceso puede tener un rank diferente en cada comunicador.

EJEMPLO:

```
#include <mpi.h>

main(int argc, char *argv[])
 {  int cantidad, identificador;

    MPI_Init(&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &cantidad);
    MPI_Comm_rank(MPI_COMM_WORLD, &identificador);
    printf("Soy %d de %d \n", identificador, cantidad);
    MPI_Finalize();
 }
```

Librería MPI - Tipos de Datos para las comunicaciones

Tipo de Datos MPI	Tipo de Datos C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Librería MPI - Comunicación punto a punto

- Diferentes protocolos para Send.
  - Send bloqueantes con buffering (Bsend).
  - Send bloqueantes sin buffering (Ssend).
  - Send no bloqueantes (Isend).
- Diferentes protocolos para Recv.
  - Recv bloqueantes (Recv).
  - Recv no bloqueantes (Irecv).

Librería MPI - Comunicación bloqueante punto a punto

- MPI_Send, MPI_Ssend, MPI_Bsend: rutina básica para enviar datos a otro proceso.

```
MPI_Send (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador)
```

  - Valor de Tag entre [0..MPI_TAG_UB].

- MPI_Recv: rutina básica para recibir datos a otro proceso.

```
MPI_Recv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador,
          MPI_Status *estado)
```

  - Comodines MPI_ANY_SOURCE y MPI_ANY_TAG._
  - Estructura MPI_Status

```
typedef struct MPI_Status { int MPI_SOURCE;
                            int MPI_TAG;
                            int MPI_ERROR; }
```

  - MPI_Get_count para obtener la cantidad de elementos recibidos

```
MPI_Get_count(MPI_Status *estado, MPI_Datatype tipoDato, int *cantidad)
```

Ejemplo

Dos procesos intercambian valores (14 y 25). Solución empleando MPI:

```
# include  <mpi.h>
main (INT argc, CHAR *argv [ ] )  {
      INT id, idAux;
      INT longitud=1;
      INT valor, otroValor;
      MPI_status  estado;

      MPI_Init (&argc, &argv);

      MPI_Comm_Rank (MPI_COMM_WORLD, &id);
      IF (id == 0)  { idAux = 1; valor = 14;}
      ELSE { idAux = 0; valor = 25; }

      MPI_send (&valor, longitud,  MPI_INT, idAux,  1, MPI_COMM_WORLD);
      MPI_recv (&otroValor, 1, MPI_INT, idAux, 1, MPI_COMM_WORLD, &estado);
      printf (“process %d received a %d\n”, id, otroValor);
      MPI_Finalize  ( );
 }
```

Ejemplo

En este caso resolvemos el mismo ejercicio pero para que no haya Deadlock si el Send actúa como Ssend.

```
# include  <mpi.h>
main (INT argc, CHAR *argv [ ] )  {
      INT id;
      INT valor, otroValor;
      MPI_status  estado;

      MPI_Init (&argc, &argv);
      MPI_Comm_Rank (MPI_COMM_WORLD, &id);
      IF (id == 0)  { valor = 14;
                        MPI_send (&valor, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
                        MPI_recv (&otroValor, 1, MPI_INT, 1, 1, MPI_OMM_WORLD, &estado);
                      }
      ELSE { valor = 25;
               MPI_recv (&otroValor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &estado);
               MPI_send (&valor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
             }
      printf (“process %d received a %d\n”, id, otroValor);
      MPI_Finalize  ( );
 }
```

Librería MPI - Comunicación no bloqueante punto a punto

- Comienzan la operación de comunicación e inmediatamente devuelven el control (no se asegura que la comunicación finalice correctamente).

```
MPI_Isend (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador,
           MPI_Request *solicitud)
```

```
MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador,
           MPI_Request *solicitud)
```

- MPI_Test: testea si la operación de comunicación finalizó.

```
MPI_Test (MPI_Request *solicitud, int *flag, MPI_Status *estado)
```

- MPI_Wait: bloquea al proceso hasta que finaliza la operación.

```
MPI_Wait (MPI_Request *solicitud, MPI_Status *estado)
```

- Este tipo de comunicación permite solapar computo con comunicación. Evita overhead de manejo de buffer. Deja en manos del programador asegurar que se realice la comunicación correctamente.

Librería MPI - Comunicación no bloqueante punto a punto

Código usando comunicación bloqueante

```
EJEMPLO: main (int argc, char *argv[])
          {  int cant, id, *dato, i;
             MPI_Status estado;

             dato = (int *) malloc (100 * sizeof(int));
             MPI_Init(&argc,&argv);
             MPI_Comm_rank (MPI_COMM_WORLD, &id);

             if (id == 0)
               { cant = atoi(argv[1])%100;
                 MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
                 for (i=0; i< 100; i++) dato[i]=0;
               }
             else
               { MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
                 MPI_Get_count(&estado, MPI_INT, &cant);
                 //PROCESA LOS DATOS;
               };
             MPI_Finalize;
          }
```

Para usar comunicación NO bloqueante (¿alcanza con cambiar el Send por Isend?)

Librería MPI - Comunicación no bloqueante punto a punto

Código anterior usando comunicación no bloqueante

```
EJEMPLO: main (int argc, char *argv[])
          {  int cant, id, *dato, i;
             MPI_Status estado;
             MPI_Request req;

             dato = (int *) malloc (100 * sizeof(int));
             MPI_Init(&argc,&argv);
             MPI_Comm_rank (MPI_COMM_WORLD, &id);

             if (id == 0)
               { cant = atoi(argv[1]);
                 //INICIALIZA dato
                 MPI_Isend(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD, &req);
                 //TRABAJA
                 MPI_Wait(&req, &estado);
                 for (i=0; i< 100; i++) dato[i]=0;
               }
             else
               { MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
                 MPI_Get_count(&estado, MPI_INT, &cant);
                 //PROCESA LOS DATOS;
               };
             MPI_Finalize;
          }
```

Librería MPI - Comunicación no bloqueante punto a punto

```
EJEMPLO: main (int argc, char *argv[])
          {  int id, *dato, i, flag;
             MPI_Status estado;
             MPI_Request req;

             dato = (int *) malloc (100 * sizeof(int));
             MPI_Init(&argc,&argv);
             MPI_Comm_rank (MPI_COMM_WORLD, &id);

             if (id == 0)
               { //INICIALIZA dato
                 MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
               }
             else
               { MPI_Irecv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD ,&req);
                 MPI_Test(&req, &flag,&estado);
                 while (!flag)
                   { //Trabaja mientras espera
                     MPI_Test(&req, &flag,&estado);
                   };
                 //PROCESA LOS DATOS;
               };
             MPI_Finalize;
          }
```

Librería MPI – Consulta de mensajes pendientes

- Información de un mensaje antes de hacer el Recv (Origen, Cantidad de elementos, Tag).
- MPI_Probe: bloquea el proceso hasta que llegue un mensaje que cumpla con el origen y el tag.

```
MPI_Probe (int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)
```

- MPI_Iprobe: cheqea por el arribo de un mensaje que cumpla con el origen y tag.

```
MPI_Iprobe (int origen, int tag, MPI_Comm comunicador, int *flag, MPI_Status *estado)
```

- Comodines en Origen y Tag.

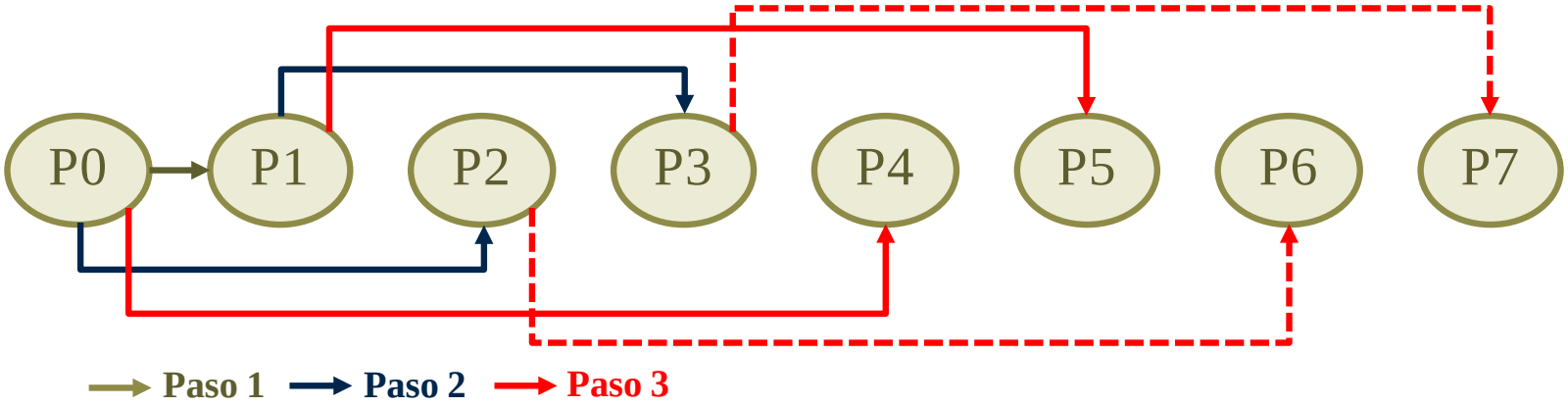
¿Cuando y porque usar cada uno?

Librería MPI - Comunicaciones Colectivas

MPI provee un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado con un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva:

- MPI_Barrier
- MPI_Bcast
- MPI_Scatter - MPI_Scatterv
- MPI_Gather - MPI_Gatherv
- MPI_Reduce
- Otras…

Ventajas del uso de comunicaciones colectivas.



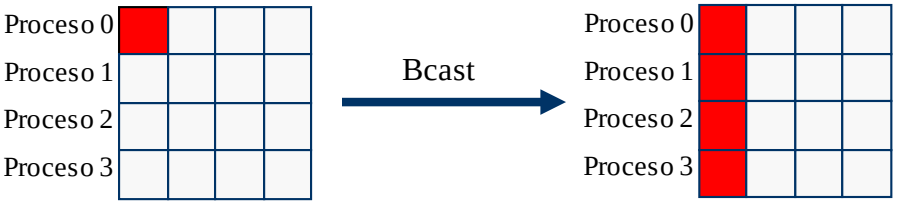
Librería MPI - Comunicaciones Colectivas

- Sincronización en una barrera.

MPI_Barrier(MPI_Comm comunicador)

- Broadcast: un proceso envía el mismo mensaje a todos los otros procesos (incluso a él) del comunicador.

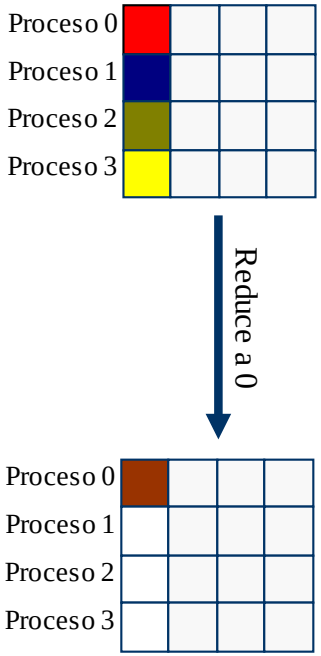
MPI_Bcast (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, MPI_Comm comunicador)



Librería MPI - Comunicaciones Colectivas (cont.)

- Reducción de todos a uno: combina los elementos enviados por cada uno de los procesos (inclusive el destino) aplicando una cierta operación.

MPI_Reduce (void *sendbuf, void *recvbuf, int cantidad, MPI_Datatype tipoDato, MPI_Op operación, int destino , MPI_Comm comunicador)



Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

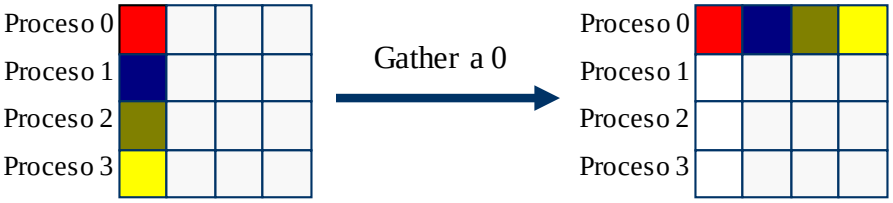
Librería MPI - Comunicaciones Colectivas (cont.)

- Gather: recolecta el vector de datos de todos los procesos (inclusive el destino) y los concatena en orden para dejar el resultado en un único proceso.
  - Todos los vectores tienen igual tamaño.

MPI_Gather (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)

  - Los vectores pueden tener diferente tamaño.

MPI_Gatherv (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int *cantsRec, int *desplazamientos, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)



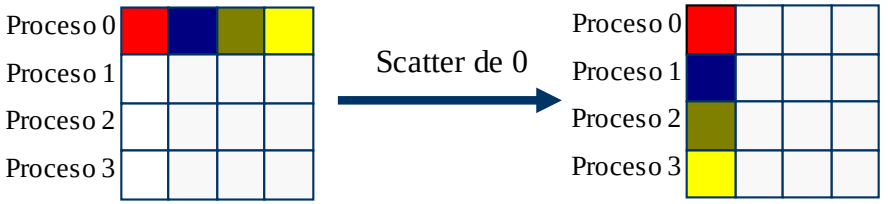
Librería MPI - Comunicaciones Colectivas (cont.)

- Scatter: reparte un vector de datos entre todos los procesos (inclusive el mismo dueño del vector).
  - Reparte en forma equitativa (a todos la misma cantidad).

```
MPI_Scatter (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec,
             MPI_Datatype tipoDatoRec, int origen, MPI_Comm comunicador)
```

  - Puede darle a cada proceso diferente cantidad de elementos.

```
MPI_Scatterv (void *sendbuf, int *cantsEnvio, int *desplazamientos, MPI_Datatype tipoDatoEnvio,
              void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int origen,
              MPI_Comm comunicador)
```



Minimizando los overheads de comunicación.

- Maximizar la localidad de datos.
- Minimizar el volumen de intercambio de datos.
- Minimizar la cantidad de comunicaciones.
- Considerar el costo de cada bloque de datos intercambiado.
- Replicar datos cuando sea conveniente.
- Lograr el overlapping de cómputo (procesamiento) y comunicaciones.
- En lo posible usar comunicaciones asincrónicas.
- Usar comunicaciones colectivas en lugar de punto a punto

Programación Concurrente

Clase 10

Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Introducción a RPC y Rendezvous: https://drive.google.com/uc?id=1U8hzHSogTzRDMnNbWe8nu1crRfqH2lh0&export=download
- RPC: https://drive.google.com/uc?id=1ZV78mvFRUWWbIjesgJD9_0kwEzN9MWr-&export=download
- Rendezvous: https://drive.google.com/uc?id=1tDtjlD60cDooWsUxqXa1YK74x9DLOsFM&export=download
- ADA: https://drive.google.com/uc?id=1WWGcgv2R71tcKBSr2clih4VO0TCFF4Eg&export=download

RPC y Rendezvous

Conceptos Básicos

- El Pasaje de Mensajes se ajusta bien a problemas de filtros y pares que interactúan, ya que se plantea la ***comunicación unidireccional***.
- Para resolver C/S la comunicación bidireccional obliga a especificar 2 tipos de canales (requerimientos y respuestas).
- Además, cada cliente necesita un canal de reply distinto...
- RPC (Remote Procedure Call) y Rendezvous ⇒ técnicas de comunicación y sincronización entre procesos que suponen ***un canal bidireccional*** ⇒ ideales para programar aplicaciones C/S
- RPC y Rendezvous combinan una interfaz "tipo monitor" con operaciones exportadas a través de llamadas externas (CALL) con mensajes sincrónicos (demoran al llamador hasta que la operación llamada se termine de ejecutar y se devuelvan los resultados).

Diferencias entre RPC y Rendezvous

- Difieren en la manera de servir la invocación de operaciones.
  - Un enfoque es declarar un *procedure* para cada operación y crear un nuevo proceso (al menos conceptualmente) para manejar cada llamado (RPC porque el llamador y el cuerpo del *procedure* pueden estar en distintas máquinas). Para el cliente, durante la ejecución del servicio, es como si tuviera en su sitio el proceso remoto que lo sirve *(Ej: JAVA)*.
  - El segundo enfoque es hacer *rendezvous* con un proceso existente. Un *rendezvous* es servido por una *sentencia de Entrada* (o accept) que espera una invocación, la procesa y devuelve los resultados *(Ej:Ada)*.

RPC (Remote Procedure Call)

Remote Procedure Call (RPC)

- Los programas se descomponen en ***módulos*** (con procesos y procedures), que pueden residir en espacios de direcciones distintos.
- Los procesos de un módulo pueden compartir variables y llamar a procedures de ese módulo.
- Un proceso en un módulo puede comunicarse con procesos de otro módulo sólo invocando procedimientos exportados por éste.
- Los módulos tienen especificación e implementación de procedures

module ***Mname***
  headers de procedures exportados (visibles)
body
  declaraciones de variables
  código de inicialización
  cuerpos de procedures exportados
  procedures y procesos locales
end

Remote Procedure Call (RPC)

- Los procesos locales son llamados ***background*** para distinguirlos de las operaciones exportadas.
- Header de un procedure visible:

  op ***opname*** (formales) [**returns** result]

- El cuerpo de un procedure visible es contenido en una declaración proc:

  proc ***opname***(identif. formales) **returns** identificador resultado
  declaración de variables locales
  sentencias
  end

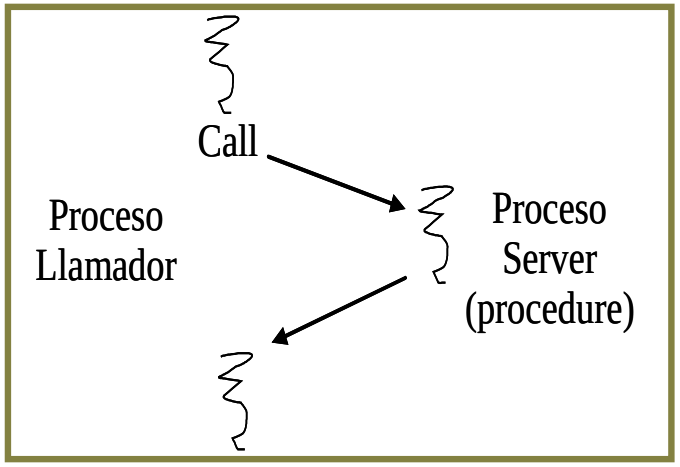
- Un proceso (o procedure) en un módulo llama a un procedure en otro ejecutando:

  call ***Mname.opname*** **(argumentos)**

- Para un llamado local, el nombre del módulo se puede omitir.

Remote Procedure Call (RPC)

- La implementación de un llamado intermódulo es distinta que para uno local, ya que los dos módulos pueden estar en distintos espacios: un **nuevo proceso** sirve el llamado, y los argumentos son pasados como mensajes entre el llamador y el proceso server.
- El llamador se demora mientras el proceso servidor ejecuta el cuerpo del procedure que implementa *opname*.
- Cuando el server vuelve de *opname* envía los resultados al llamador y termina. Después de recibir los resultados, el llamador sigue.
- Si el proceso llamador y el procedure están en el mismo espacio de direcciones, es posible evitar crear un nuevo proceso.
- En general, un llamado será remoto ⇒ se debe crear un proceso server o alocarlo de un pool preexistente.



Sincronización en módulos

Por sí mismo, RPC es solo un mecanismo de comunicación.

- Aunque un proceso llamador y su server sincronizan, el único rol del server es actuar en nombre del llamador (como si éste estuviera ejecutando el llamado ⇒ la sincronización entre ambos es implícita).
- Necesitamos que los procesos en un módulo sincronicen (procesos server ejecutando llamados remotos y procesos del módulo). Esto comprende Exclusión Mutua y Sincronización por Condición.
- Existen dos enfoques para proveer sincronización, dependiendo de si los procesos en un módulo ejecutan:
  - ***Con exclusión mutua (un solo proceso por vez).***
  - ***Concurrentemente***.

Sincronización en módulos

- *Si ejecutan con Exclusión Mutua* las variables compartidas son protegidas automáticamente contra acceso concurrente, pero es necesario programar sincronización por condición.

- *Si pueden ejecutar concurrentemente* necesitamos mecanismos para programar exclusión mutua y sincronización por condición ***(cada módulo es un programa concurrente)*** ⇒ podemos usar cualquier método ya descripto (semáforos, monitores, o incluso rendezvous).

- Es más general asumir que los procesos pueden ejecutar concurrentemente (más eficiente en un multiprocesador de memoria compartida). Asumimos que procesos en un módulo ejecutan concurrentemente, usando por ejemplo *time slicing*.

Ejemplo Cliente/Servidor

Time Server

- Módulo que brinda servicios de *timing* a procesos cliente en otros módulos.
- Dos operaciones visibles: ***get_time*** y ***delay(interval)***
- Un proceso interno que continuamente inicia un ***timer*** por hardware, luego incrementa el tiempo al ocurrir la interrupción de ***timer***.

```
module TimeServer
    op get_time( ) returns INT;
    op delay(INT interval, INT myid);
  body
    INT tod = 0;
    SEM m= 1;
    SEM d[n] = ([n] 0);
    QUEUE of (INT waketime, INT id's) napQ;

   proc get_time ( ) returns time
     {   time := tod; }

   proc delay(interval, myid)
     {   INT waketime = tod + interval;
         P(m);
         insert ((waketime, myid) napQ);
         V(m);
         P(d[myid]);
     }
```

```
Process Clock
  { Inicia timer por hardware;
    WHILE (true)
      { Esperar interrupción,
        luego rearrancar timer;
        tod := tod + 1;
        P(m);
        WHILE tod ≥ min(waketime, napQ)
            { remove ((waketime, id), napQ);
              V(d[id]);
            }
        V(m);
      }
  }


end TimeServer;
```

Ejemplo Cliente/Servidor

Time Server

- Múltiples clientes pueden llamar a *get_time* y a *delay* a la vez $\Rightarrow$ múltiples procesos “servidores” estarían atendiendo los llamados concurrentemente.
- Los pedidos de *get_time* se pueden atender concurrentemente porque sólo significan leer la variable ***tod***.
- Pero, *delay* y *clock* necesitan ejecutarse con Exclusión Mutua porque manipulan ***napQ***, la cola de procesos cliente "durmiendo”.
- El valor de ***myid*** en *delay* se supone un entero único entre 0 y n-1. Se usa para indicar el semáforo privado sobre el cual está esperando un cliente.

Ejemplo Cliente/Servidor

Manejo de caches en un sistema de archivos distribuido

- Versión simplificada de un problema que se da en sistemas de archivos y BD distribuidos.
- Suponemos procesos de aplicación que ejecutan en una WS, y archivos de datos almacenados en un FS. Los programas de aplicación que quieren acceder a datos del FS, llaman procedimientos ***read*** y ***write*** del módulo local ***FileCache***. Leen o escriben arreglos de caracteres.
- Los archivos se almacenan en el FS en bloques de 1024 bytes, fijos. El módulo ***FileServer*** maneja el acceso a bloques del disco; provee dos procedimientos (***ReadBlk*** y ***WriteBlk***).
- El módulo ***FileCache*** mantiene en cache los bloques recientemente leídos. Al recibir pedido de ***read***, ***FileCache*** primero chequea si los bytes solicitados están en su cache. Sino, llama al procedimiento ***readblock*** del ***FileServer***. Algo similar ocurre con los ***write***.

Ejemplo Cliente/Servidor

Manejo de caches en un sistema de archivos distribuido

```
Module FileCache    # ubicado en cada workstation
    op read (INT count ;  result CHAR buffer[ *] );
    op write (INT count; CHAR buffer[*] );
body
   cache de N bloques; descripción de los registros de cada file; semáforos para sincronizar acceso al cache;

    proc read (count, buffer)
      {   IF (los datos pedidos no están en el cache)
            {    seleccionar los bloques del cache a usar;
                 IF (se necesita vaciar parte del cache)  FileServer.writeblk(....);
                 FileServer.readblk(....);
             }
          buffer= número de bytes requeridos del cache;
      }

    proc write(count, buffer)
      {   IF (los datos apropiados no están en el cache)
            {   seleccionar los bloques del cache a usar;
                IF (se necesita vaciar parte del cache)  FileServer.writeblk(....);
            }
          bloquedeCache= número de bytes desde buffer;
      }

end FileCache;
```

Ejemplo Cliente/Servidor

Manejo de caches en un sistema de archivos distribuido

- Los llamados de los programas de aplicación de las WS son locales a su ***FileCache***, pero desde estos módulos se invocan los procesos remotos de ***FileServer***.
- ***FileCache*** es un server para procesos de aplicación; ***FileServer*** es un server para múltiples clientes ***FileCache***, uno por WS.
- Si existe un ***FileCache*** por programa de aplicación, no se requiere sincronización interna entre los ***read*** y ***write***, porque sólo uno puede estar activo. Si múltiples programas de aplicación usaran el mismo ***FileCache***, tendríamos que usar semáforos para implementar la EM en el acceso a ***FileCache***.
- En cambio en ***FileServer*** se requiere sincronización interna, ya que atiende múltiples ***FileCache*** y contiene un proceso ***DiskDriver*** (la sincronización no se muestra en el código).

Ejemplo Cliente/Servidor

Manejo de caches en un sistema de archivos distribuido

```
Module FileServer     # ubicado en el servidor
    op readblk (INT fileid, offset;  result CHAR blk[1024] );
    op writeblk (INT fileid, offset; CHAR blk[1024] );
body
    cache de bloques;  cola de pedidos  pendientes;  semáforos para acceso al cache y a la cola;
    proc readblk (fileid, offset, blk)
        {  IF (los datos pedidos no están en el cache) {encola el pedido;  esperar que la lectura sea procesada;}
           blk= bloques pedidos del disco;
        }
    proc writeblk (fileid, offset, blk)
        {  Ubicar el bloque en cache;
           IF (es necesario grabar físicamente en disco) {encola el pedido; esperar que la escritura sea procesada;}
           bloque cache = blk;
        }
    process DiskDriver
        {  WHILE (true)
             {  esperar por un pedido de acceso físico al disco; arrancar una operación física; esperar interrupción;
               despertar el proceso que está esperando completar el request;
             }
        }
end FileServer;
```

Ejemplo Pares Interactuantes

Intercambio de valores

- Si dos procesos de diferentes módulos deben intercambiar valores, cada módulo debe exportar un procedimiento que el otro módulo llamará.

```
module  Intercambio  [i = 1 to 2]
    op depositar(int);
body
    int otrovalor;
    sem listo = 0;

 proc  depositar(otro)
    { otrovalor = otro;
      V(listo);
    }

 process Worker
   {  int mivalor;
      call Intercambio[3-i].depositar(mivalor);
      P(listo); .....
   }

end Intercambio
```

RPC en JAVA
Remote Method Invocation (RMI)

- Java soporta el uso de RPC en programas distribuidos mediante la invocación de métodos remotos (RMI).
- Una aplicación que usa RMI tiene 3 componentes:
  - Una interfase que declara los headers para métodos remotos.
  - Una clase server que implementa la interfase.
  - Uno o más clientes que llaman a los métodos remotos.
- El server y los clientes pueden residir en máquinas diferentes.

Rendezvous

Rendezvous

- **RPC** por si mismo sólo brinda un mecanismo de comunicación intermódulo. Dentro de un módulo es necesario programar la sincronización. Además, a veces son necesarios procesos extra sólo para manipular los datos comunicados por medio de RPC (ej: Merge).

- **Rendezvous** combina ***comunicación*** **y** ***sincronización***:
  - Como con RPC, un proceso cliente ***invoca*** una operación por medio de un ***call***, pero esta operación es servida por un proceso existente en lugar de por uno nuevo.
  - Un proceso servidor usa una ***sentencia de entrada*** para esperar por un ***call*** y actuar.
  - Las operaciones se atienden una por vez más que concurrentemente.

Rendezvous

- La especificación de un módulo contiene declaraciones de los ***headers*** de las operaciones exportadas, pero el cuerpo consta de un único proceso que sirve operaciones.

- Si un módulo exporta ***opname***, el proceso server en el módulo realiza ***rendezvous*** con un llamador de ***opname*** ejecutando una ***sentencia de entrada***:

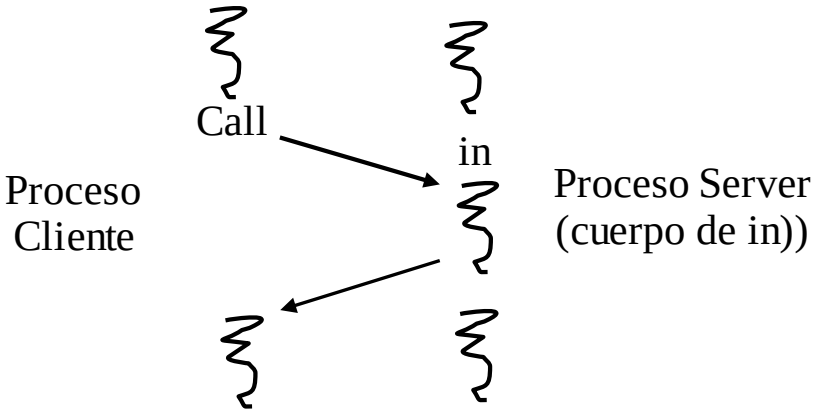
  in *opname* (parámetros formales) → **S; ni**

- Las partes entre ***in*** y ***ni*** se llaman ***operación guardada***.

- Una sentencia de entrada demora al proceso server hasta que haya al menos un llamado pendiente de ***opname***; luego elige el llamado pendiente más viejo, copia los argumentos en los parámetros formales, ejecuta **S** y finalmente retorna los parámetros de resultado al llamador. Luego, ambos procesos pueden continuar.

Rendezvous

- A diferencia de RPC el server es un proceso activo.



- Combinando comunicación guardada con rendezvous:

in op$_1$ (formales$_1$) **and B$_1$ by e$_1$ → S$_1$;**
□ **...**
□ **op$_n$** (formales$_n$) **and B$_n$ by e$_n$ → S$_n$;**
ni

  - Los B_i son ***expresiones de sincronización*** opcionales.
  - Los e_i son ***expresiones de scheduling*** opcionales.

} Pueden referenciar a los parámetros formales.

Ejemplo
Buffer limitado

```
module BufferLimitado
   op depositar (typeT), retirar (OUT typeT);

body
   process Buffer
     {  queue buf;
        int cantidad = 0;

        while (true)
          {  in depositar (item) and cantidad < n →  push (buf, item);
                                                     cantidad = cantidad + 1;
             □ retirar (OUT item) and cantidad > 0 →  pop (buf, item);
                                                     cantidad = cantidad - 1;
             ni
          }
     }
end BufferLimitado
```

Ejemplo

Filósofos Centralizado

```
module  Mesa
   op tomar(int),  dejar(int);
body
   process Mozo
      { bool comiendo[5] =([5] false);
        while (true)
              in tomar(i) and not (comiendo[izq(i)] or comiendo[der(i)])  → comiendo[i] = true;
              □ dejar(i) → comiendo[i] = false;
              ni
      }
end Mesa

module  Persona [i = 0 to 4]
Body
   process Filosofo
      { while (true)
          {   call Mesa.tomar(i);
              come;
              call Mesa.dejar(i);
              piensa;
          }
   }
```

Ejemplo
Time Server

- A diferencia del ejemplo visto para RPC, *waketime* hace referencia a la hora que debe despertarse.

```
module TimeServer
   op get_time (OUT int);
   op delay (int);
   op tick ();

body TimeServer
   process Timer
      { int tod = 0;
        while (true)
              in get_time (OUT time) → time = tod;
              □ delay (waketime) and waketime <= tod by waketime → skip;
              □ tick () → tod = tod + 1; reiniciar timer;
              ni
      }
end TimeServer
```

Ejemplo
Alocador SJN

```
module  Alocador_SJN
   op pedir(int), liberar();

body
   process SJN
            {  bool libre = true;
               while (true)
                  in pedir (tiempo) and libre by tiempo → libre = false;
                  □ liberar ( ) → libre = true;
                  ni
             }
end SJN_Allocator
```

ADA– Lenguaje con Rendezvous

El lenguaje ADA

- Desarrollado por el Departamento de Defensa de USA para que sea el estandard en programación de aplicaciones de defensa (desde sistemas de Tiempo Real a grandes sistemas de información).
- Desde el punto de vista de la concurrencia, un programa Ada tiene ***tasks*** (tareas) que pueden ejecutar independientemente y contienen primitivas de sincronización.
- Los puntos de invocación (entrada) a una tarea se denominan ***entrys*** y están especificados en la parte visible (header de la tarea).
- Una tarea puede decidir si acepta la comunicación con otro proceso, mediante la primitiva ***accept***.
- Se puede declarar un ***type task***, y luego crear instancias de procesos (tareas) identificado con dicho tipo (arreglo, puntero, instancia simple).

Tasks

- La forma más común de especificación de task es:

  TASK *nombre* **IS**
  declaraciones de ENTRYs
  end;

- La forma más común de cuerpo de task es:

  TASK BODY *nombre* **IS**
  declaraciones locales
  BEGIN
  sentencias
  END *nombre***;**

- Una especificación de TASK define una única tarea.
- Una instancia del correspondiente ***task body*** se crea en el bloque en el cual se declara el TASK.

Sincronización
Call: *Entry Call*

- El ***rendezvous*** es el principal mecanismo de sincronización en Ada y también es el mecanismo de comunicación primario.

- ***Entry:***
  - Declaración de ***entry simples y familia de entry*** (parámetros IN, OUT y IN OUT)**.**
  - ***Entry call.*** La ejecución demora al llamador hasta que la operación E terminó (o abortó o alcanzó una excepción). → **Tarea.entry (parámetros)**
  - ***Entry call condicional***:

```
select entry call;
        sentencias adicionales;
else
        sentencias;
end select;
```

  - ***Entry call temporal:***

```
select entry call;
        sentencias adicionales;
or delay tiempo
        sentencias;
end select;
```

Sincronización

Sentencia de Entrada: *Accept*

- La tarea que declara un entry sirve llamados al entry con *accept*:

  accept *nombre* **(parámetros formales) do** sentencias **end *nombre*;**

- Demora la tarea hasta que haya una invocación, copia los parámetros reales en los parámetros formales, y ejecuta las sentencias. Cuando termina, los parámetros formales de salida son copiados a los parámetros reales. Luego ambos procesos continúan.

- La ***sentencia wait selectiva*** soporta comunicación guardada.

  select *when* $B_1 \Rightarrow$ ***accept*** E_1; sentencias$_1$
  or **...**
  or ***when*** $B_n \Rightarrow$ ***accept*** E_n; sentencias$_n$
  end select;

  - Cada línea se llama ***alternativa***. Las cláusulas ***when*** son opcionales.
  - Puede contener una alternativa ***else, or delay, or terminate.***
  - Uso de atributos del entry: ***count, calleable.***

Ejemplo

Mailbox para 1 mensajes

```
TASK TYPE Mailbox IS
      ENTRY Depositar (msg: IN mensaje);
      ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

A, B, C : Mailbox;

TASK BODY Mailbox IS
    dato: mensaje;
BEGIN
      LOOP
        ACCEPT Depositar (msg: IN mensaje) DO dato := msg;  END Depositar;
        ACCEPT Retirar (msg: OUT mensaje) DO msg := dato;  END Retirar;
      END LOOP;
END Mailbox;
```

Podemos utilizar estos mailbox para manejar mensajes: *A.Depositar(x1); B.Depositar(x2); C.Retirar(x3);*

Ejemplo
Lectores-Escritores

```
Procedure Lectores-Escritores is
   Task Sched IS
          Entry  InicioLeer;
          Entry  FinLeer;
          Entry  InicioEscribir;
          Entry  FinEscribir;
   End Sched;
   Task type Lector;
   Task body Lector is
      Begin
         Loop
            Sched.InicioLeer; … Sched.FinLeer;
          End loop;
   End Lector;
   Task type Escritor;
   Task body Escritor is
      Begin
         Loop
            Sched.InicioEscribir; … Sched.FinEscribir;
          End loop;
   End Lector;
   VecLectores: array (1..cantL) of Lector;
   VecEscritores: array (1..cantE) of Escritor;
```

```
Task body Sched is
        numLect: integer :=0;
    Begin
        Loop
            Select
                When InicioEscribir'Count = 0 =>
                    accept  InicioLeer;
                    numLect := numLect+1;
            or   accept FinLeer;
                 numLect := numLect-1;
            or   When numLect = 0 =>
                    accept InicioEscribir;
                    accept FinEscribir;
                    For i in 1..InicioLeer‘count loop
                        accept InicioLeer;
                        numLect:= numLect +1;
                    End loop;
            End select;
        End loop;
End Sched;

Begin
    Null;
End Lectores-Escritores
```

Ejemplo

Mailbox para N mensajes (Buffer Limitado)

- Solución vista para Rendezvous general

```
module BufferLimitado
   op depositar (typeT), retirar (OUT typeT);

body
   process Buffer
     {  queue buf;
        int cantidad = 0;

        while (true)
          {  in depositar (item) and cantidad < n →  push (buf, item);
                                                     cantidad = cantidad + 1;
             □ retirar (OUT item) and cantidad > 0 →  pop (buf, item);
                                                     cantidad = cantidad - 1;
             ni
          }
     }
end BufferLimitado
```

Ejemplo

Mailbox para N mensajes (Buffer Limitado) – con una cola

- Solución en ADA

```
TASK Mailbox IS
      ENTRY Depositar (msg: IN mensaje);
      ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

TASK BODY Mailbox IS
       buf: queue;
       cantidad integer := 0;
BEGIN
      LOOP
            SELECT
                  WHEN cantidad < N => ACCEPT Depositar (msg: IN mensaje) DO
                                             push (buf, msg);
                                             cantidad := cantidad +1;
                                       END Depositar;
            OR
                  WHEN cantidad > 0 => ACCEPT Retirar (msg: OUT mensaje) DO
                                             pop (buf, msg);
                                             cantidad = cantidad -1;
                                       END Retirar;
            END SELECT;
      END LOOP;
END Mailbox;
```

Ejemplo

Mailbox para N mensajes (Buffer Limitado) – con un arreglo

```
TASK Mailbox IS
      ENTRY Depositar (msg: IN mensaje);
      ENTRY Retirar (msg: OUT mensaje);
END Mailbox;

TASK BODY Mailbox IS
       datos: array (0..N-1) of mensaje;
       cant, pri, ult integer := 0;
BEGIN
      LOOP
            SELECT
                  WHEN cant < N => ACCEPT Depositar (msg: IN mensaje) DO
                                          ult := (ult+1) MOD N; datos[ult] := msg; cant := cant +1;
                              END Depositar;
            OR
                  WHEN cant > 0 => ACCEPT Retirar (msg: OUT mensaje) DO
                                    msg := datos[pri]; pri := (pri+1) MOD N; cant := cant -1;
                               END Retirar;
            END SELECT;
      END LOOP;
END Mailbox;
```

Ejemplo
Filósofos Centralizado

- Solución vista para Rendezvous general

```
module  Mesa
   op tomar(int),  dejar(int);
body
   process Mozo
      { bool comiendo[5] =([5] false);
        while (true)
              in tomar(i) and not (comiendo[izq(i)] or comiendo[der(i)])  → comiendo[i] = true;
              □ dejar(i) → comiendo[i] = false;
              ni
      }
end Mesa

module  Persona [i = 0 to 4]
Body
  process Filosofo
     { while (true)
          {   call Mesa.tomar(i);
              come;
              call Mesa.dejar(i);
              piensa;
          }
   }
```

Ejemplo
Filósofos Centralizado

- Solución en ADA – Múltiples entry

```
TASK Mesa IS
      ENTRY Tomar0; ENTRY Tomar1; ENTRY Tomar2; ENTRY Tomar3; ENTRY Tomar4;
      ENTRY Dejar (id: IN integer);
END Mesa;

TASK BODY Mesa IS
      Comiendo:  array (0..4) of bool := (0..4=> false);
BEGIN
    For i in 0..4 loop
        Filosofos(i).identificacion(i);
    end loop;
    LOOP
     SELECT
          when (not (comiendo(4) or comiendo(1)) => ACCEPT Tomar0; comiendo(0) := true;
     OR  when (not (comiendo(0) or comiendo(2)) => ACCEPT Tomar1; comiendo(1) := true;
     OR  when (not (comiendo(1) or comiendo(3)) => ACCEPT Tomar2; comiendo(2) := true;
     OR  when (not (comiendo(2) or comiendo(4)) => ACCEPT Tomar3; comiendo(3) := true;
     OR  when (not (comiendo(3) or comiendo(0)) => ACCEPT Tomar4; comiendo(4) := true;
     OR  ACCEPT Dejar(id: IN integer) do
                comiendo(id) := false;
          end Dejar;
      END SELECT;
    END LOOP;
END Mesa;
```

Ejemplo
Filósofos Centralizado

- Solución en ADA – Múltiples entry

```
TASK TYPE Filosofo IS
    ENTRY Identificacion (ident: IN integer);
END Filosofo;

Filosofos: array (0..4) of Filosofo;

TASK BODY FIlosofo IS
    id: integer;
BEGIN
    ACCEPT Identificacion (ident : IN integer) do
            id := ident;
    End Identificacion;
    LOOP
        if ( id = 0) then Mesa.Tomar0;
        else if  ( id = 1) then Mesa.Tomar1;
             else if  ( id = 2) then Mesa.Tomar2;
                  else if  ( id = 3) then Mesa.Tomar3;
                       else if  ( id = 4) then Mesa.Tomar4;
        //Come
        Mesa.Dejar(id);
        //Piensa
    END LOOP;
END Mesa;
```

Ejemplo
Filósofos Centralizado

- Solución en ADA – Encolar pedidos

```
TASK TYPE Filosofo IS
      ENTRY Identificacion (ident: IN integer);
      ENTRY Comer;
END Filosofo;
TASK Mesa IS
      ENTRY Tomar (id: IN integer);
      ENTRY Dejar (id: IN integer);
END Mesa;

Filosofos: array (0..4) of Filosofo;

TASK BODY FIlosofo IS
      id: integer;
BEGIN
      ACCEPT Identificacion (ident : IN integer) do id := ident;  End Identificacion;
      LOOP
            Mesa.Tomar(id);
            Accept Comer;
            //Come
            Mesa.Dejar(id);
            //Piensa
      END LOOP;
END Mesa;
```

Ejemplo

Filósofos Centralizado

- Solución en ADA – Encolar pedidos

```
TASK BODY Mesa IS
      Comiendo: array (0..4) of bool := (0..4=> false);
      QuiereC: array (0..4) of bool := (0..4=> false);
      aux: integer;
BEGIN
    For i in 0..4 loop   Filosofos(i).identificacion(i);     end loop;
    LOOP
      SELECT
            ACCEPT Tomar(id: IN integer)  do    aux := id; END Tomar;
            if (not (comiendo((aux+1) mod 5) or comiendo((aux-1) mod 5)) )  then
                 comiendo(aux) = true;
                 Filosofos(aux).Comer;
            else  QuiereC (aux) := true;      end if;
      OR  ACCEPT Dejar(id:  IN integer)  do   aux := id; end Dejar;
            comiendo(aux) := false;
            for i in 0..4 loop
               if (QuiereC(i)  and not (comiendo((i+1)  mod 5) or comiendo((i-1)  mod 5)) )  then
                        comiendo(i) = true;
                        QuiereC(i)  := false;
                        Filosofos(i).Comer;
               end if;
            end loop;
       END SELECT;
    END LOOP;
END Mesa;
```

Ejemplo
Time Server

- Solución vista para Rendezvous general

```
module TimeServer
   op get_time (OUT int);
   op delay (int);
   op tick ();

body TimeServer
   process Timer
      { int tod = 0;
        while (true)
              in get_time (OUT time) → time = tod;
              □ delay (waketime) and waketime <= tod by waketime → skip;
              □ tick () → tod = tod + 1; reiniciar timer;
              ni
      }
end TimeServer
```

Ejemplo
Time Server

- Solución en ADA

```
PROCEDURE DESPERTADORES IS
Task TimeServer is
        entry get_time  (hora: OUT int);      entry delay (hd, id: IN int);    entry tick;
End TimeServer;
Task Reloj;
Task Type Cliente Is
     entry Identificar (identificacion: IN integer);     entry seguir;
End Cliente;
ArrClientes: array (1..C) of Cliente;
Task Body Cliente Is
     id: integer;    hora: integer;
BEGIN
     ACCEPT Identificar (identificacion : IN integer) do id := identificación;  End Identificar;
     TimeServer.get_time(hora);
     TimeServer.delay(hora+….., id);
     ACCEPT seguir;
End Cliente;
Task Body Reloj is
BEGIN
   loop      delay(1);  TimeServer.tick;    end loop;
End  Reloj;
```

Ejemplo
Time Server

- Solución en ADA

```
Task Body TimeServer is
      actual: integer := 0;
      dormidos: colaOrdenada;
      auxId, auxHora: integer;
BEGIN
 LOOP
   SELECT
        when (tick'count =0) =>ACCEPT get_time (hora: OUT integer) do  hora := actual; END get_time;
  OR when (tick'count =0) =>ACCEPT delay(hd, id: IN integer) do agregar(dormidos, (id,hd)); END delay;
  OR ACCEPT tick;
        actual := actual +1;
        while (not empty (dormidos)) and then (VerHoraPrimero(dormidos) <= actual)) loop
                sacar(dormidos, (auxId, auxHora);
                ArrClientes(auxId).seguir;
        end loop;
   END SELECT;
 END LOOP;
End TimeServer;

BEGIN
     for i in 1..C loop
          ArrClientes(i).identificacion(i);
     end loop;
END DESPERTADORES;
```

Ejemplo
Alocador SJN

- Solución vista para Rendezvous general

```
module  Alocador_SJN
   op pedir(int), liberar();

body
   process SJN
            {  bool libre = true;
               while (true)
                  in pedir (tiempo) and libre by tiempo → libre = false;
                  □ liberar ( ) → libre = true;
                  ni
             }
end SJN_Allocator
```

Ejemplo
Alocador SJN

- Solución en ADA

```
PROCEDURE SchedulerSJN IS
Task Alocador_SJN is
        entry pedir (tiempo, id: IN integer);
        entry liberar;
End Alocador_SJN ;

Task Type Cliente Is
    entry Identificar (identificacion: IN integer);
    entry usar;
End Cliente;
ArrClientes: array (1..C) of Cliente;
Task Body Cliente Is
    id: integer;   tiempo: integer;
BEGIN
    ACCEPT Identificar (identificacion : IN integer) do id := identificación;  End Identificar;
    loop
        //trabaja y determna el valor de tiempo
        Alocador_SJN.pedir(id, tiempo);
        Accept usar;
        //Usa el recurso
        Alocador_SJN.liberar;
    end loop;
End Cliente;
```

Ejemplo
Alocador SJN

- Solución en ADA

```
Task Body Alocador_SJN is
      libre: boolean := true;
      espera: colaOrdenada;
      tiempo, aux: integer;
Begin
   loop
      aux := -1;
      select
          accept Pedir (tiempo, id: IN integer) do
                if (libre) then libre:= false; aux := id;
                else agregar(espera, (id, tiempo)); end if;
          end Pedir;
      or accept liberar;
          if (empty (espera)) then libre := true;
          else sacar(espera, (aux, tiempo); end if;
      end select;
      if (aux <> -1) then ArrClientes(aux).usar;   end if;
   end loop;
End Alocador_SJN ;

BEGIN
     for i in 1..C loop
          ArrClientes(i).identificacion(i);
     end loop;
END SchedulerSJN;
```

Programación Concurrente

Clase 11

Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- Arquitecturas Paralelas:
https://drive.google.com/u/0/uc?id=1bt3lCyqcbLOjKw30gNe3qH9WV0s4Hn7U&export=download

- Diseño de Algoritmos Paralelos:
https://drive.google.com/u/0/uc?id=1CsHAANKAdZg654UjkkT65c6tM3nGqwO_&export=download

- Métricas de Rendimiento:
https://drive.google.com/uc?id=1b7DRKwCaOx0mH9sEzmWeGroW-VluJcv7&export=download

- Paradigmas de Programación Paralela:
https://drive.google.com/uc?id=15zyZYIL5VWxU-JxxOwjMdC7LhmkchNKv&export=download

Arquitecturas Paralelas

Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

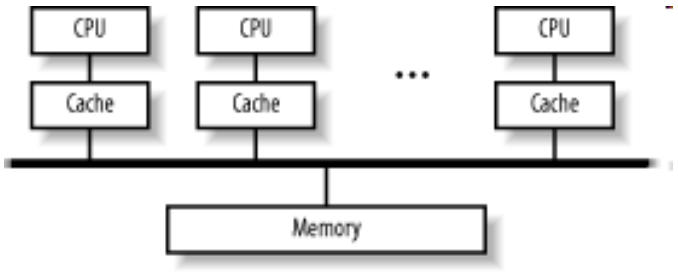
- Por la organización del espacio de direcciones.
- Por la granularidad.
- Por el mecanismo de control.
- Por la red de interconexión.

Clasificación por el Espacio de Direcciones

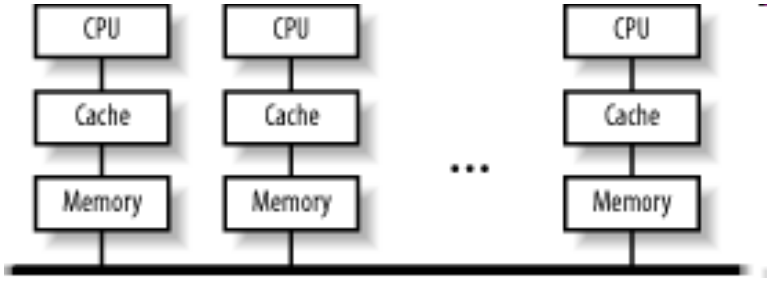
- Las arquitecturas paralelas se clasifican según su espacio de direcciones en:
  - Memoria Compartida.
  - Memoria Distribuida.
- Esta clasificación se relaciona con el modelo de comunicación a utilizar:
  - Accesos a Memoria Compartida (memoria compartida).
  - Intercambio de mensajes (principalmente memoria distribuida).
- En algunos casos también tenemos en la misma plataforma ambos mecanismos.

Clasificación por el Espacio de Direcciones

- Multiprocesadores de memoria compartida.
  - Interacción modificando datos en la memoria compartida.
  - Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problemas de sincronización y consistencia.
  - Esquemas NUMA para mayor número de procesadores distribuidos.
  - Problema de consistencia.



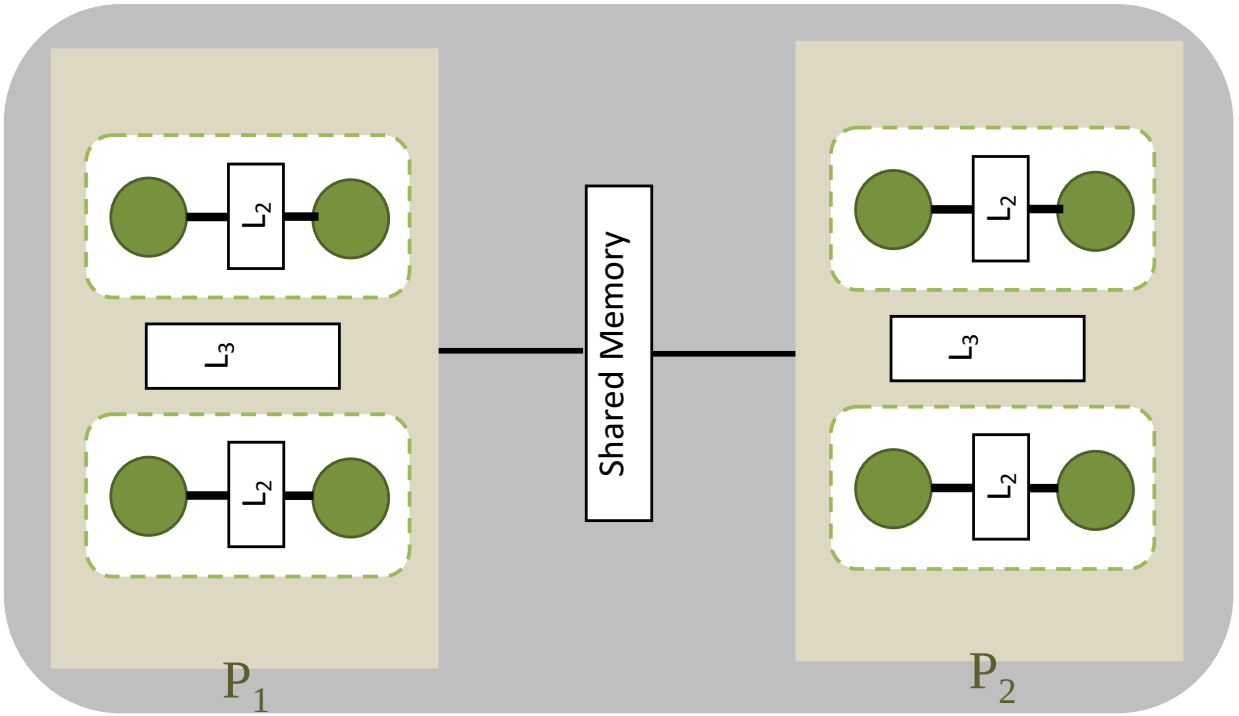
Esquema UMA



Esquema NUMA

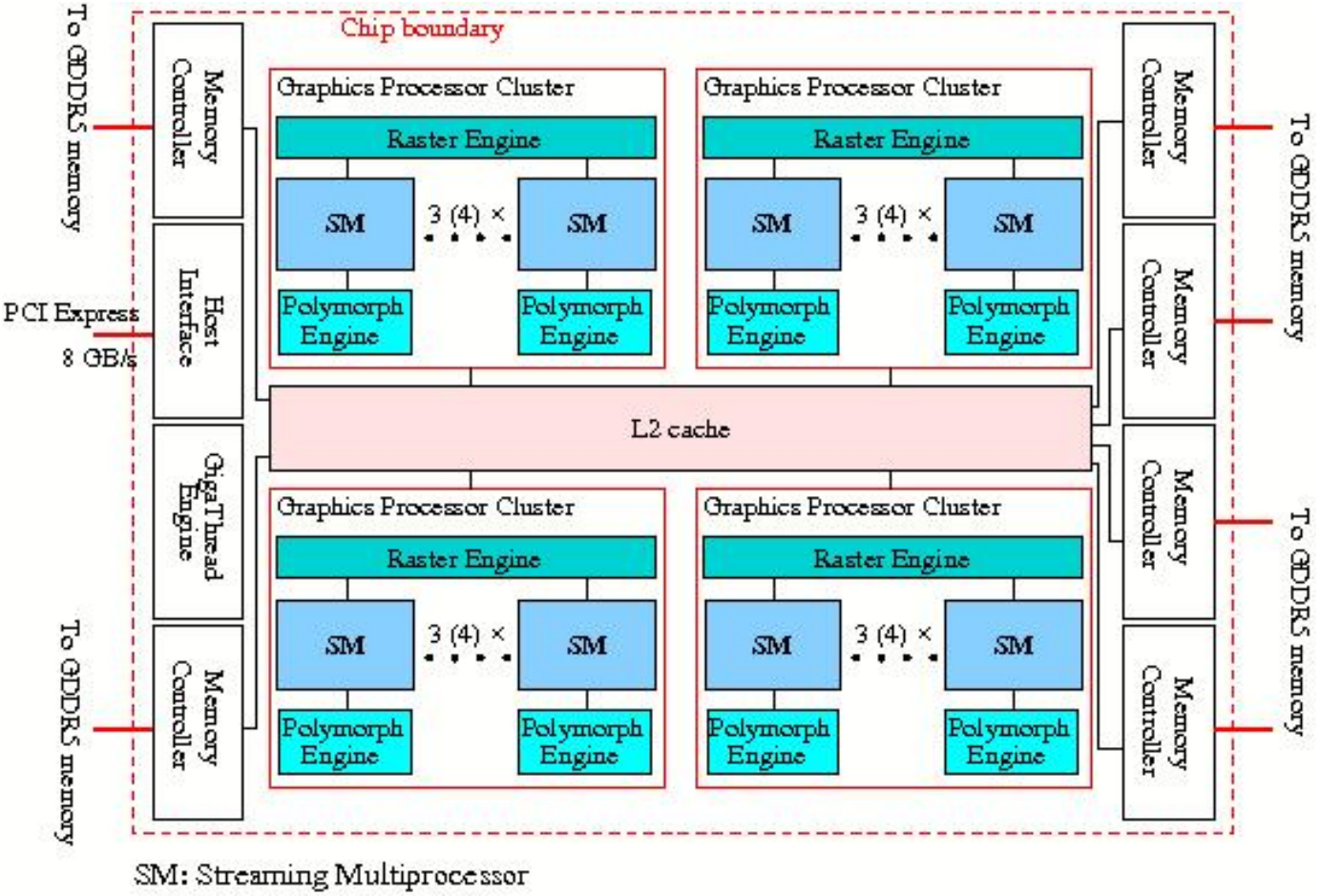
Clasificación por el Espacio de Direcciones

- Ejemplo de multiprocesador de memoria compartida: multicore de 8 núcleos.



Clasificación por el Espacio de Direcciones

- Ejemplo de multiprocesador de memoria compartida: GPU.



Clasificación por el Espacio de Direcciones

- Multiprocesadores con memoria distribuida.
  - Procesadores conectados por una red.
  - Memoria local (no hay problemas de consistencia).
  - Interacción es sólo por pasaje de mensajes.
  - Grado de acoplamiento de los procesadores:
    - Multicomputadores (máquinas fuertemente acopladas). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
    - Memoria compartida distribuida.
    - Clusters.
    - Redes (multiprocesador débilmente acoplado).

Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

- Por la organización del espacio de direcciones.
- Por la granularidad.
- Por el mecanismo de control.
- Por la red de interconexión.

Clasificación por el Mecanismo de Control

Propuesta por Flynn ("Some computer organizations and their effectiveness", 1972).

Se basa en la manera en que las ***instrucciones*** son ejecutadas sobre los ***datos***.

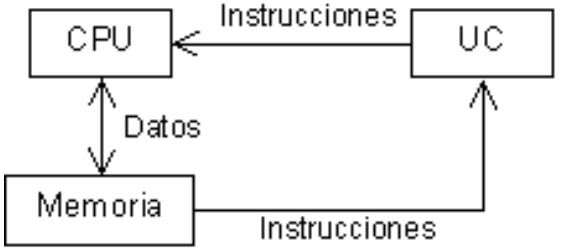
Clasifica las arquitecturas en 4 clases:

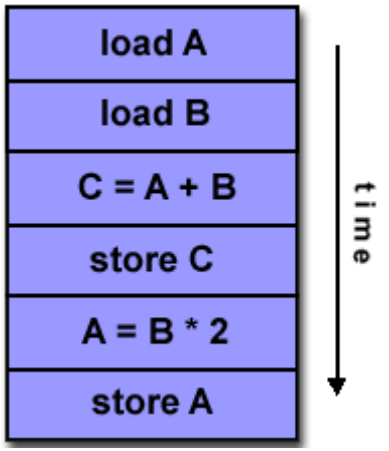
- SISD (Single Instruction Single Data).
- SIMD (Single Instruction Multiple Data).
- MISD (Multiple Instruction Single Data).
- MIMD (Multiple Instruction Multiple Data).

Clasificación por el Mecanismo de Control

SISD: Single Instruction Single Data

- Instrucciones ejecutadas en secuencia, una por ciclo de instrucción.
- La memoria afectada es usada sólo por ésta instrucción.
- Usada por la mayoría de los uní procesadores.
- La CPU ejecuta instrucciones (decodificadas por la UC) sobre los datos. La memoria recibe y almacena datos en las escrituras, y brinda datos en las lecturas.
- Ejecución determinística.

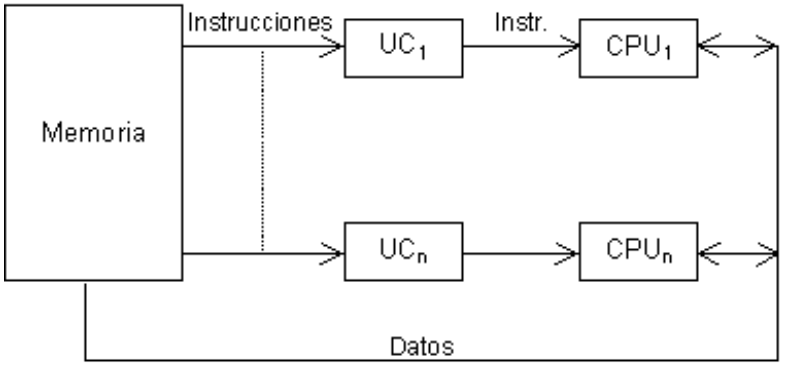




Clasificación por el Mecanismo de Control

MISD: Multiple Instruction Single Data

- Los procesadores ejecutan un flujo de instrucciones distinto pero comparten datos comunes.
- Operación sincrónica (en lockstep).
- No son máquinas de propósito general (“hipotéticas”, Duncan).
- Ejemplos posibles:
  - Múltiples filtros de frecuencia operando sobre una única señal.
  - Múltiples algoritmos de criptografía intentando crackear un único mensaje codificado.



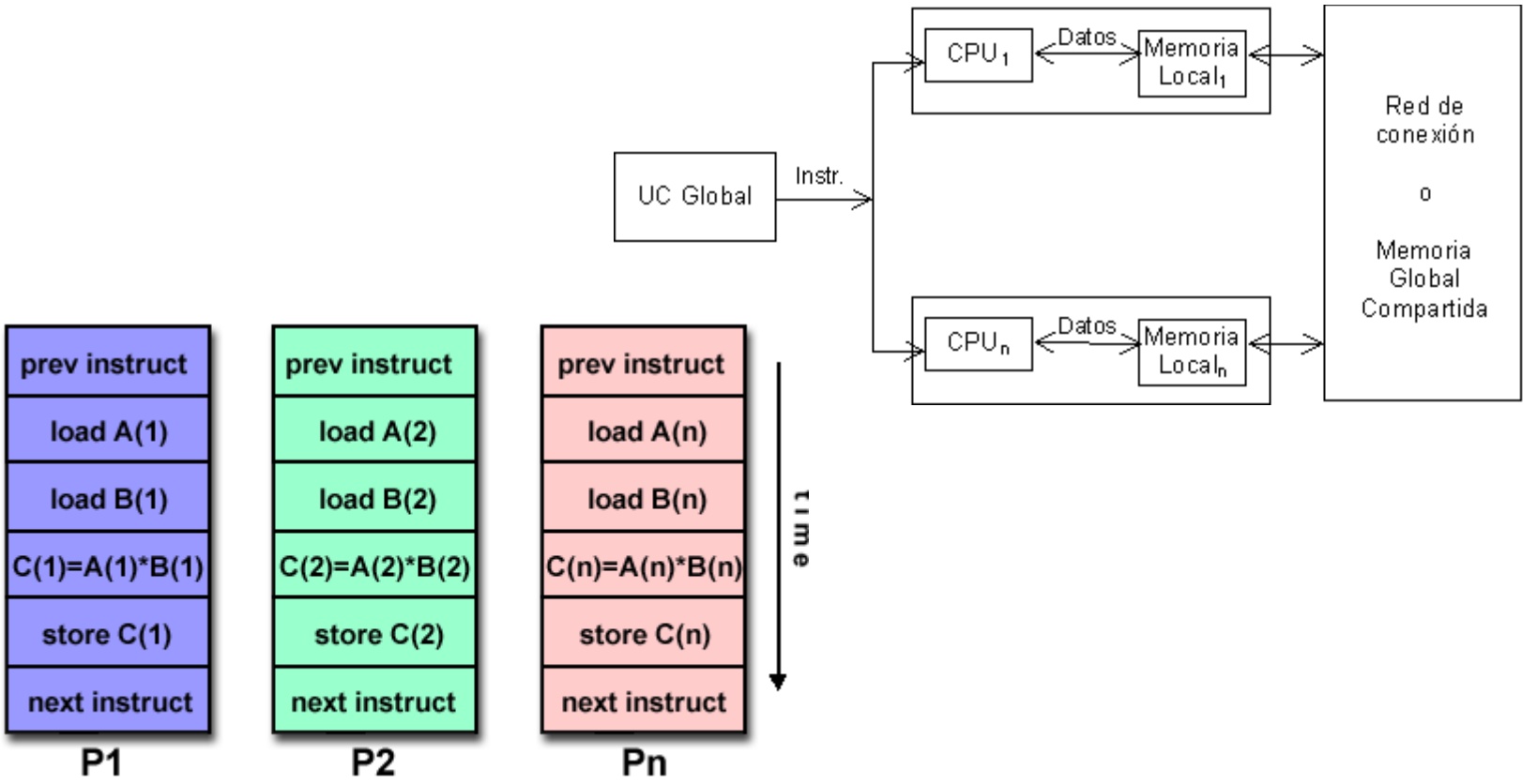
Clasificación por el Mecanismo de Control

SIMD: Single Instruction Multiple Data

- Conjunto de procesadores idénticos, con sus memorias, que ejecutan la misma instrucción sobre distintos datos.
- Los procesadores en general son muy simples.
- El host hace broadcast de la instrucción. Ejecución sincrónica y determinística.
- Pueden deshabilitarse y habilitarse selectivamente procesadores para que ejecuten o no instrucciones.
- Adecuados para aplicaciones con alto grado de regularidad, (por ejemplo procesamiento de imágenes).

Clasificación por el Mecanismo de Control

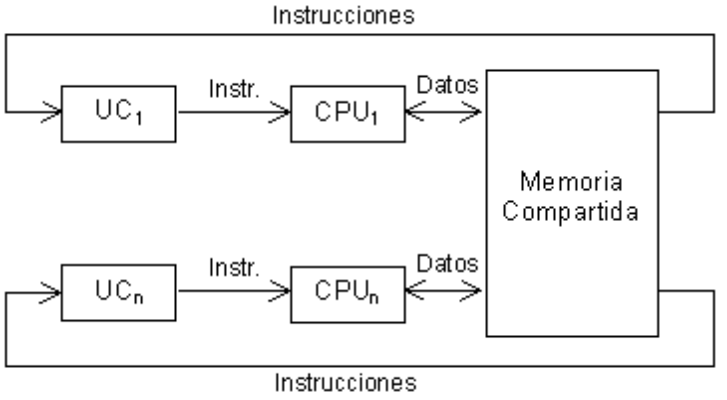
- **Ejemplos de máquina SIMD:** Array Processors. CM-2, Maspar MP-1 y 2, Illiac IV.

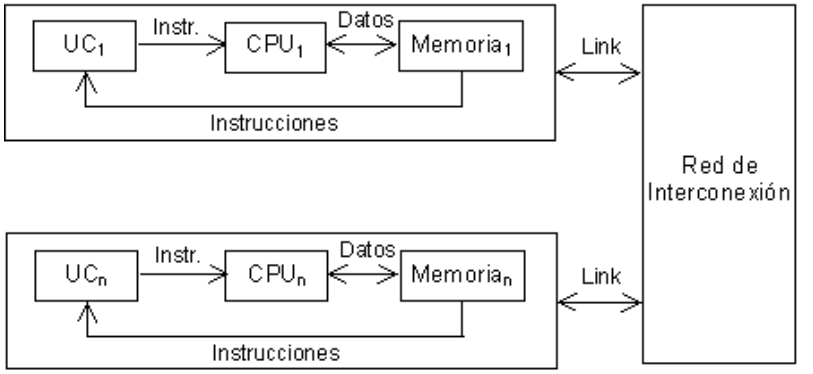


Clasificación por el Mecanismo de Control

MIMD: Multiple Instruction Multiple Data

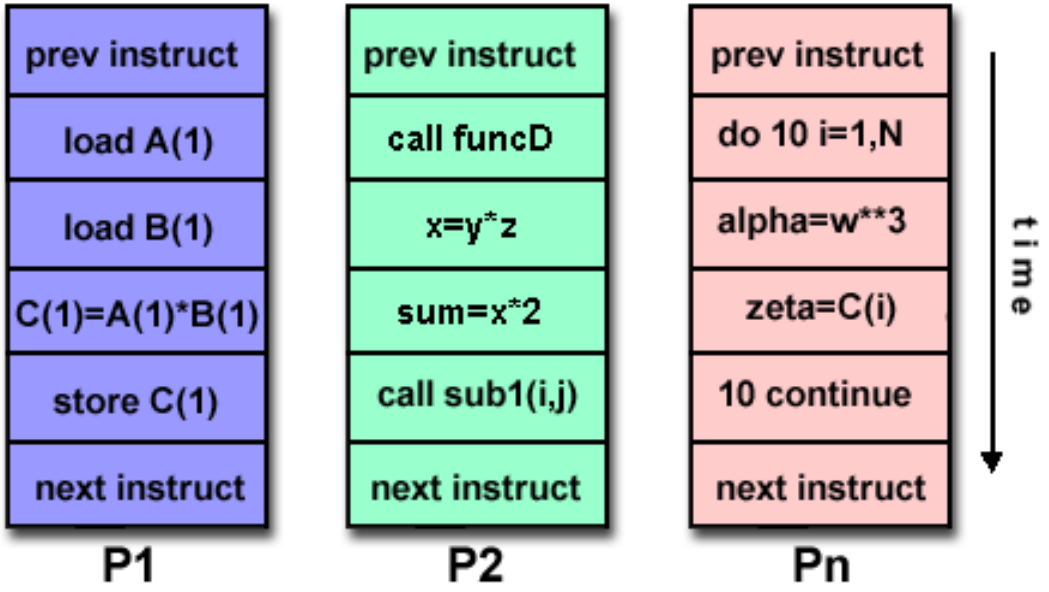
- Cada procesador tiene su propio flujo de instrucciones y de datos ⇒ cada uno ejecuta su propio "programa" a su ritmo.
- Pueden ser con memoria compartida o distribuida.
- Sub-clasificación de MIMD:
  - MPMD (multiple program multiple data): cada procesador ejecuta su propio programa (ejemplo con PVM).
  - SPMD (single program multiple data): hay un único programa fuente y cada procesador ejecuta su copia independientemente (ejemplo con MPI).





Clasificación por el Mecanismo de Control

- **Ejemplos de máquina MIMD:** nCube 2, iPSC, CM-5, Paragon XP/S, máquinas DataFlow, red de transputers, multicores, cluster.



Clasificación de arquitecturas paralelas

Hay diferentes enfoques para clasificar las arquitecturas paralelas:

- Por la organización del espacio de direcciones.
- Por la granularidad.
- Por el mecanismo de control.
- Por la red de interconexión.

Clasificación por la Red de Interconexión

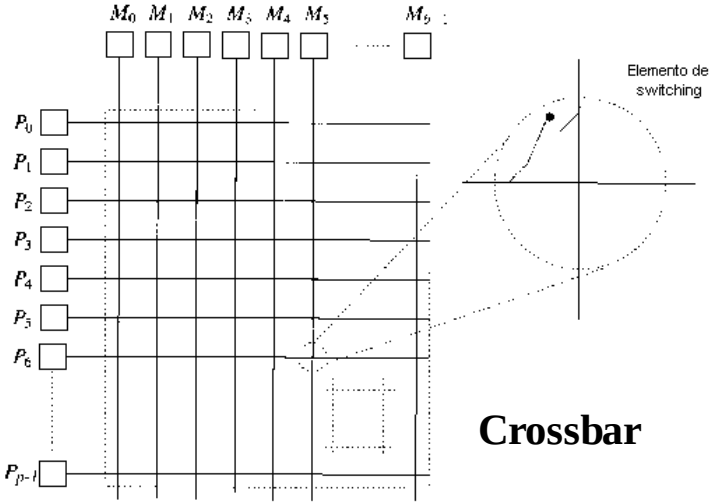
Tanto en memoria compartida como en pasaje de mensajes las máquinas pueden construirse conectando procesadores y memorias usando diversas redes de interconexión:

- Las ***redes estáticas*** constan de links punto a punto. Típicamente se usan para máquinas de pasaje de mensajes.
- Las ***redes dinámicas*** están construidas usando switches y enlaces de comunicación. Normalmente para máquinas de memoria compartida.

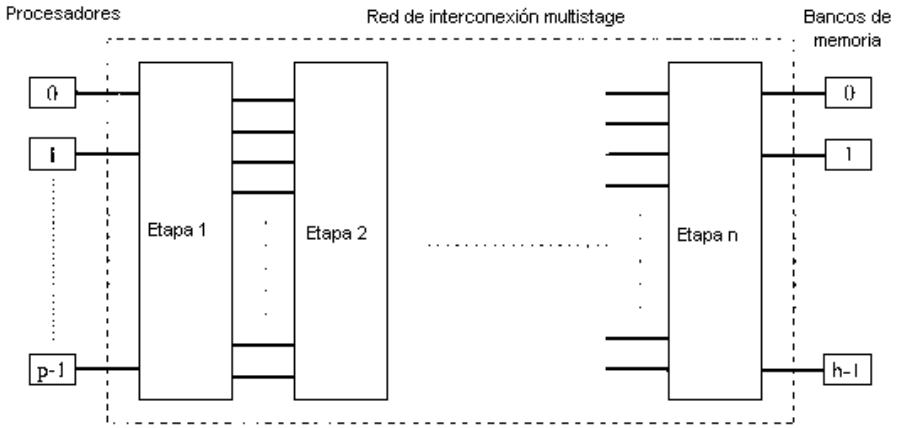
El diseño de la red de interconexión depende de una serie de factores (ancho de banda, tiempo de startup, paths estáticos o dinámicos, operación sincrónica o asincrónica, topología, costo, etc.).

Clasificación por la Red de Interconexión

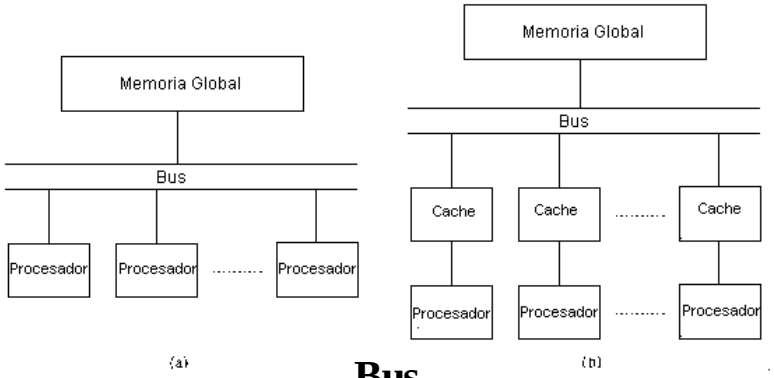
Redes de interconexión dinámicas



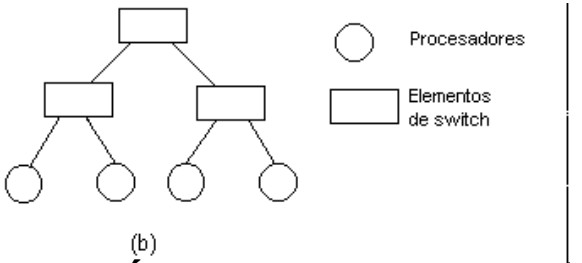
Crossbar



Multistage



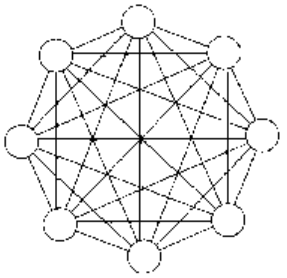
Bus



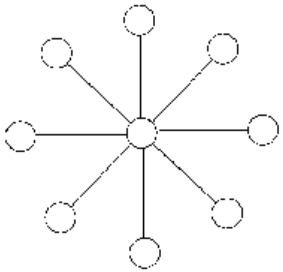
Árbol dinámico

Clasificación por la Red de Interconexión

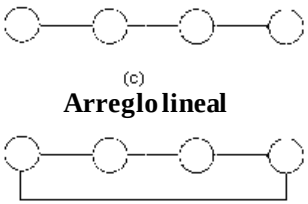
Redes de interconexión estáticas



(a)
Completamente conectada

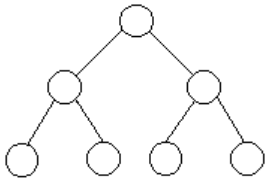


(b)
Estrella

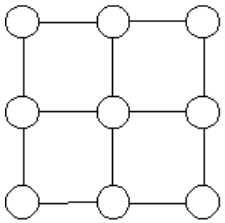


(c)
Arreglo lineal

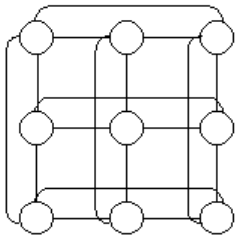
(d)
Anillo



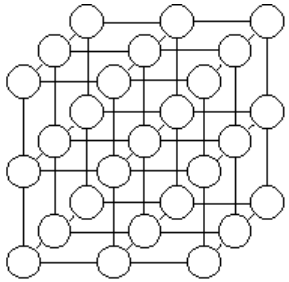
(a)
Árbol estático



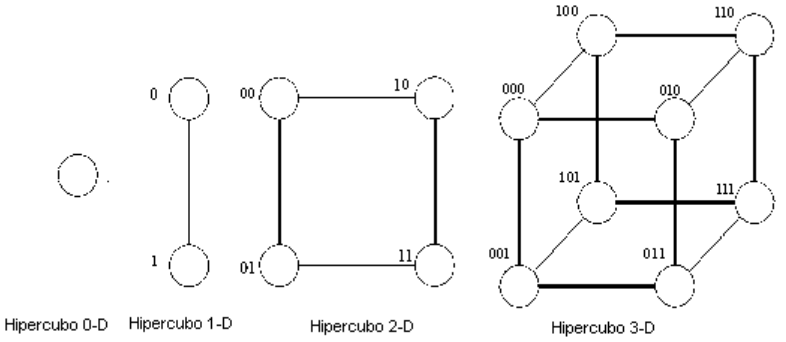
(a)
Mesh



(b)
Toro



(c)
Mesh 3D



Un hipercubo d-dimensional tiene $p=2^d$ procesadores

Diseño de Algoritmos Paralelos

Diseño de Algoritmos Paralelos

- La mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes.
- Puede darse un enfoque metódico para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas, y reducir el costo de *backtracking* por malas elecciones.
- Aspectos independientes de la máquina tales como la concurrencia son considerados tempranamente, y los aspectos específicos de la máquina se demoran.

Diseño de Algoritmos Paralelos

¿Por qué es compleja la programación paralela?

- Decidir cuál es la granularidad óptima de las tareas.
- Mapear tareas y datos a los nodos físicos de procesamiento (¿en forma estática o dinámica?)
- Manejar comunicación y sincronización.
- Asegurar corrección. Evitar deadlocks. Evitar desbalances.
- Obtener un cierto grado de Tolerancia a Fallos.
- Manejar la heterogeneidad.
- Lograr escalabilidad en todos los casos (potencia, tamaño de la arquitectura y del problema).
- Consumo energético.

Diseño de Algoritmos Paralelos

- Para diseñar un algoritmo paralelo se deben realizar alguno de los siguientes pasos:
  - Identificar porciones de trabajo (tareas) concurrentes.
  - Mapear tareas a procesos en distintos procesadores.
  - Distribuir datos de entrada, intermedios y de salida.
  - Manejo de acceso a datos compartidos.
  - Sincronizar procesos.

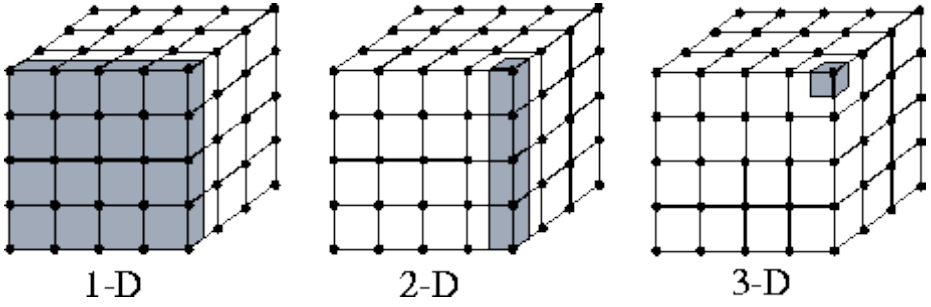
- Pasos Fundamentales: ***Descomposición en Tareas*** y ***Mapeo de Procesos a Procesadores***.

Descomposición en tareas

- Para desarrollar un algoritmo paralelo el primer punto es descomponer el problema en sus componentes funcionales concurrentes (procesos/tareas).
- Se trata de definir un gran número de pequeñas tareas para obtener una descomposición de grano fino, para brindar la mayor flexibilidad a los algoritmos paralelos potenciales.
- En etapas posteriores, la evaluación de los requerimientos de comunicación, arquitectura de destino, o temas de IS pueden llevar a descartar algunas posibilidades detectadas en esta etapa, revisando la partición original y aglomerando tareas para incrementar su tamaño o granularidad.
- Esta descomposición puede realizarse de muchos modos. Un primer concepto es pensar en tareas de igual código (normalmente ***paralelismo de datos o dominio***) pero también podemos tener diferente código (***paralelismo funcional***).

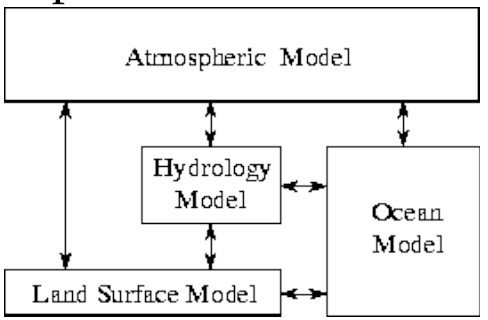
Descomposición en Tareas

- ***Descomposición de datos***: determinar una división de los datos (en muchos casos, de igual tamaño) y luego asociarle el cómputo (típicamente, cada operación con los datos con que opera).
- Esto da un número de tareas, donde cada uno comprende algunos datos y un conjunto de operaciones sobre ellos. Una operación puede requerir datos de varias tareas, y esto llevará a la ***comunicación***.
- Son posibles distintas particiones, basadas en diferentes estructuras de datos. Por ejemplo, diferentes formas de descomponer una estructura 3D de datos. Inicialmente la de grano más fino.



Descomposición en Tareas

- ***Descomposición funcional***: primero descompone el cómputo en tareas disjuntas y luego trata los datos.
- Los requerimientos de datos pueden ser disjuntos (partición completa) o superponerse significativamente (necesidad de comunicación para evitar replicación de datos). En el segundo caso, probablemente convenga descomponer el dominio.
- Inicialmente se busca no replicar cómputo y datos. Esto puede revisarse luego para reducir costos.
- La descomposición funcional tiene un rol importante como técnica de estructuración del programa, para reducir la complejidad del diseño general. Modelos computacionales de sistemas complejos pueden estructurarse como conjuntos de modelos más simples conectados por interfaces.



Aglomeración

- El algoritmo resultante de las etapas anteriores es abstracto en el sentido de que no es especializado para ejecución eficiente en una máquina particular.
- Esta etapa revisa las decisiones tomadas con la visión de obtener un algoritmo que ejecute en forma eficiente en una clase de máquina real.
- En particular, se considera si es útil combinar o aglomerar las tareas para obtener otras de mayor tamaño. También se define si vale la pena replicar datos y/o computación.

Aglomeración

- 3 objetivos, a veces conflictivos, que guían las decisiones de aglomeración y replicación:
  - ***Incremento de la granularidad***: intenta reducir la cantidad de comunicaciones combinando varias tareas relacionadas en una sola.
  - ***Preservación de la flexibilidad***: al juntar tareas puede limitarse la escalabilidad del algoritmo. La capacidad para crear un número variante de tareas es crítica si se busca un programa portable y escalable.
  - ***Reducción de costos de IS***: se intenta evitar cambios extensivos, por ejemplo, reutilizando rutinas existentes.

Características de las Tareas

Una vez que tenemos el problema separado en tareas conceptualmente independientes, tenemos una serie de características de las mismas que impactarán en la perfomance alcanzable por el algoritmo paralelo:

- Generación de las tareas.
- El tamaño de las tareas.
- Conocimiento del tamaño de las tareas.
- El volumen de datos asociado con cada tarea.

Mapeo de tareas a procesadores

- Se especifica dónde ejecuta cada tarea.
- Este problema no existe en uniprocesadores o máquinas de memoria compartida con scheduling de tareas automático.
- **Objetivo:** minimizar tiempo de ejecución. Dos estrategias, que a veces conflictúan: ubicar tareas que pueden ejecutar concurrentemente en ≠ procesadores para mejorar la concurrencia o poner tareas que se comunican con frecuencia en = procesador para incrementar la localidad.
- El problema es NP-completo: no existe un algoritmo de tiempo polinomial tratable computacionalmente para evaluar tradeoffs entre estrategias en el caso general. Existen heurísticas para clases de problema.

Mapeo de tareas a procesadores

- Normalmente tendremos más tareas que procesadores físicos.
- Los algoritmos paralelos (o el scheduler de ejecución) deben proveer un mecanismo de “mapping” entre tareas y procesadores físicos.
- Nuestro lenguaje de especificación de algoritmos paralelos debe poder indicar claramente las tareas que pueden ejecutarse concurrentemente y su precedencia/prioridad para el caso que no haya suficientes procesadores para atenderlas.
- La dependencia de tareas condicionará el balance de carga entre procesadores.
- La interacción entre tareas debe tender a minimizar la comunicación de datos entre procesadores físicos.

Criterio para el mapeo de tareas a procesadores

- Un buen mapping es crítico para el rendimiento de los algoritmos paralelos.

  1. Tratar de mapear tareas independientes a diferentes procesadores.
  2. Asignar prioritariamente los procesadores disponibles a las tareas que estén en el camino crítico.
  3. Asignar tareas con alto nivel de interacción al mismo procesador, de modo de disminuir el tiempo de comunicación físico.

  ⇒ Notar que estos criterios pueden oponerse entre sí … por ejemplo el criterio 3 puede llevarnos a NO paralelizar.

- Debe encontrarse un equilibrio que optimice el rendimiento paralelo ⇒ MAPPING DETERMINA LA EFICIENCIA DEL ALGORITMO.

Métricas de Rendimiento

Métricas del paralelismo

- En el mundo serial la performance con frecuencia es medida teniendo en cuenta los requerimientos de tiempo y memoria de un programa.
- En un algoritmo paralelo para resolver un problema interesa saber cuál es la ganancia en performance.
- Hay otras medidas que deben tenerse en cuenta siempre que favorezcan a sistemas con mejor tiempo de ejecución.
- A falta de un modelo unificador de cómputo paralelo, el tiempo de ejecución depende del tamaño de la entrada y de la arquitectura y número de procesadores (*sistema paralelo = algoritmo + arquitectura sobre la que se implementa*).

Métricas del paralelismo

- La diversidad torna complejo el análisis de performance...
  - ¿Qué interesa medir?
  - ¿Qué indica que un sistema paralelo es mejor que otro?
  - ¿Qué sucede si agrego procesadores?
- En la medición de performance es usual elegir un problema y testear el tiempo variando el número de procesadores. Aquí subyacen las nociones de speedup y eficiencia, y la *ley de Amdahl*.
- Otro tema de interés es la *escalabilidad*, que da una medida de usar eficientemente un número creciente de procesadores.

Métricas del paralelismo
Speedup (S)

- S es el cociente entre el tiempo de ejecución del algoritmo serial conocido más rápido (T_s) y el tiempo de ejecución paralelo del algoritmo elegido (T_p):

$$S = \frac{T_S}{T_P}$$

- Speedup óptimo depende de la arquitectura (en homogénea P).

$$S_{óptimo} = \sum_{i=0}^{P} \frac{PotenciaCalculo(i)}{PotenciaCalculo(mejor)}$$

- Rango de valores: en general entre *0* y $S_{óptimo}$
- Speedup lineal o perfecto, sublineal y superlineal.

Métricas del paralelismo
Eficiencia (E)

- Cociente entre Speedup y Speedup Óptimo.

$$E = \frac{S}{S_{óptimo}}$$

- Mide la fracción de tiempo en que los procesadores son *útiles* para el cómputo.

- El valor está entre 0 y 1, dependiendo de la efectividad de uso de los procesadores. Cuando es 1 corresponde al speedup perfecto.

Escalabilidad de los Sistemas Paralelos

- Es muy difícil extrapolar la perfomance de un sistema paralelo, a partir de configuraciones con pocos procesadores y conjuntos de datos reducidos.

  - No sirven los estudios con 2, 4, 8 procesadores que proyectan el *Sp* alcanzable con 128 o 256 procesadores o el tiempo de procesamiento cuando tengamos 100 o 1000 veces más datos… ¿Por qué?

  - Básicamente porque los resultados con pequeños conjuntos de datos están afectados por la localidad en el manejo de la memoria, y los resultados con pocos procesadores porque las comunicaciones no computan los costos relacionados con la distancia entre procesadores y la disminución del ancho de banda efectivo.

Métricas del paralelismo
Factores que limitan el Speedup

- Alto porcentaje de código secuencial (*Ley de Amdahl*).
- Alto porcentaje de entrada/salida respecto de la computación.
- Algoritmo no adecuado (necesidad de rediseñar).
- Excesiva contención de memoria (rediseñar código para localidad de datos).
- Tamaño del problema (puede ser chico, o fijo y no crecer con *p*).
- Desbalance de carga (produciendo esperas ociosas en algunos procesadores).
- Overhead paralelo: ciclos adicionales de CPU para crear procesos, sincronizar, etc.

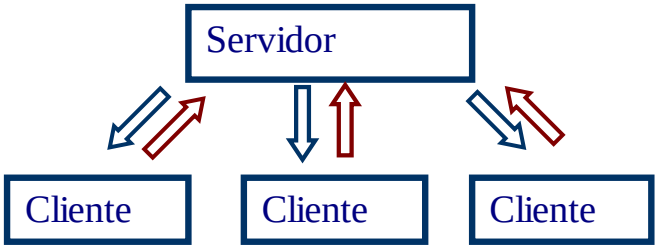
Paradigmas de Programación Paralela

Paradigmas de Programación Paralela

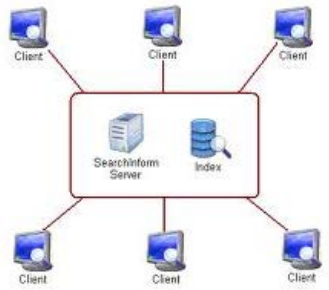
- ***Paradigma de programación***: clase de algoritmos que resuelve distintos problemas, pero tienen la misma estructura de control.
- Para cada paradigma puede escribirse un esqueleto algorítmico que define la estructura de control común.
- Dentro de la programación paralela pueden encontrarse paradigmas que permiten encuadrar los problemas en alguno de ellos.
- En cada paradigma, los patrones de comunicación son muy similares en todos los casos.

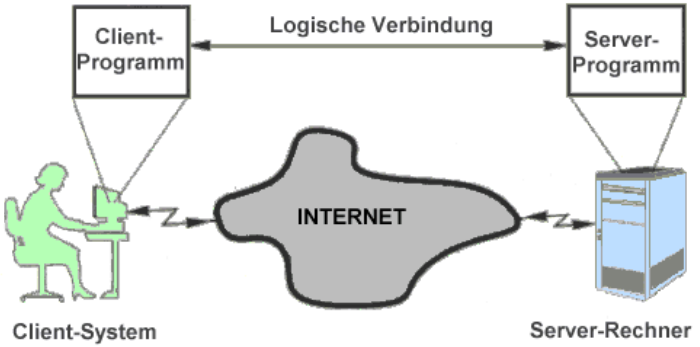
Cliente / Servidor

- Cliente-servidor es el esquema dominante en las aplicaciones de procesamiento distribuido.
- Los servidores son procesos que esperan pedidos de servicios de múltiples clientes. Naturalmente unos y otros pueden ejecutarse en procesadores diferentes. Comunicación bidireccional. Atención de a un cliente a la vez, o a varios con multithreading.
- Mecanismos de invocación variados (rendezvous, RPC, monitores).
- El soporte distribuido puede ser simple (LAN) o extendido a la WEB.



Cliente/Servidor





Master/slave o master/worker

- Basado en organizaciones del mundo real.
- El master envía iterativamente datos a los workers y recibe resultados de éstos.
- Posible "cuello de botella" (por ejemplo, por tareas muy chicas o *slaves* muy rápidos) → elección del grano adecuado.
- Dos casos de acuerdo a las dependencias de las iteraciones:
  - ✓ Iteraciones dependientes: el master necesita los resultados de todos los workers para generar un nuevo conjunto de datos.
  - ✓ Entradas de datos independientes: los datos llegan al maestro, que no necesita resultados anteriores para generar un nuevo conjunto de datos

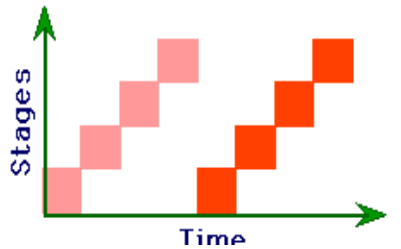
Master/slave o master/worker

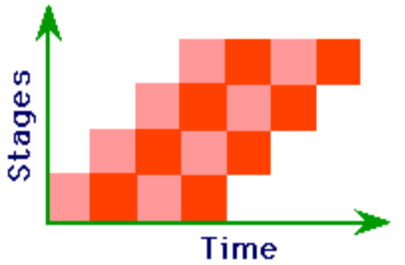
- Dos opciones para la distribución de los datos:
  - ✓ Distribuir todos los disponibles, de acuerdo a alguna política (estático).
  - ✓ Bajo petición o demanda (dinámico).
- Existen variantes, pero básicamente un procesador es responsable de la coordinación y los otros de resolver los problemas asignados.
- Es una variación de SPMD donde hay dos programas en lugar de sólo uno.
- Casos:
  - ✓ Procesadores heterogéneos y con distintas velocidades → problemas con el balance de carga.
  - ✓ Trabajo que debe realizarse en "fases" → sincronización.
  - ✓ Generalización a modelo multi-nivel o jerárquico.

Pipeline y Algoritmos Sistólicos

- El problema se particiona en una secuencia de pasos. El stream de datos pasa entre los procesos, y cada uno realiza una tarea sobre él.
- Ejemplo: filtrado, etiquetado y análisis de escena en imágenes.
- Mapeo natural a un arreglo lineal de procesadores.

Pipe





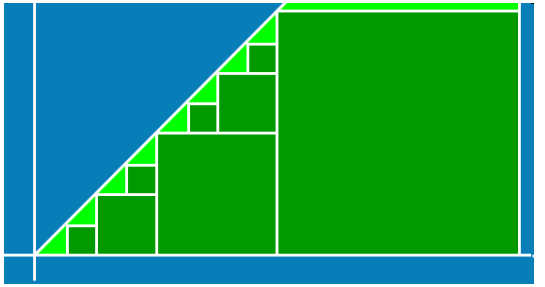
- Extensiones:
  - ✓ Procesadores especializados no iguales.
  - ✓ Más de un procesador para una tarea determinada.
  - ✓ El flujo puede no ser una línea simple (ejemplo: ensamble de autos con varias líneas que son combinadas) → procesamiento sistólico.

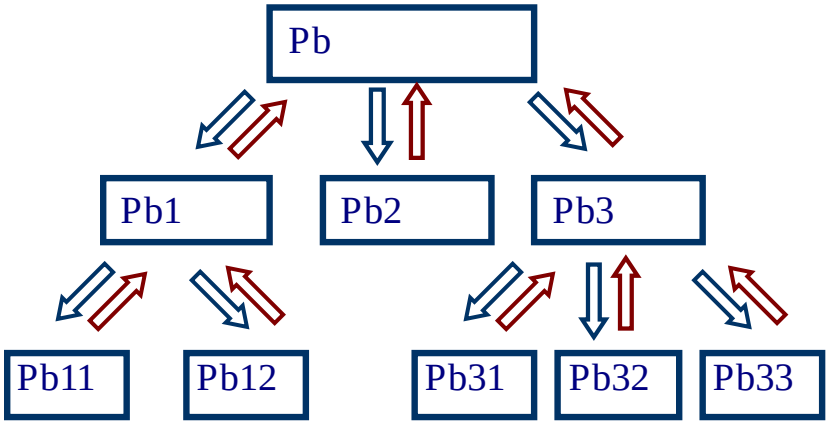
Dividir y Conquistar

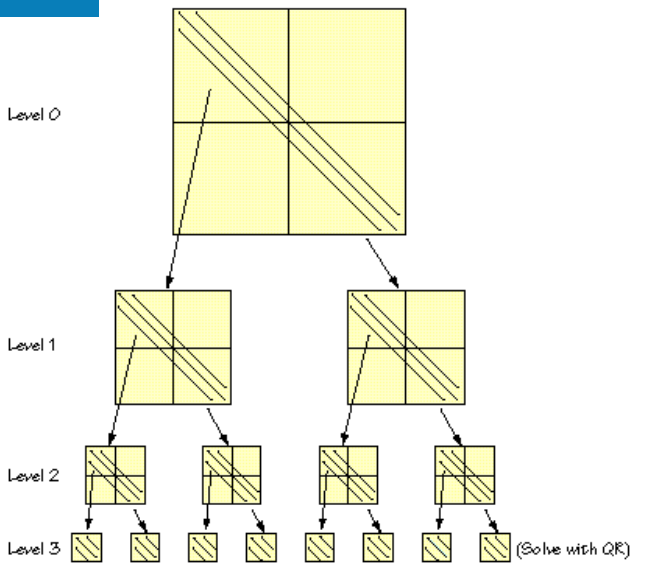
- En general implica ***paralelismo recursivo*** donde el problema general (programa) puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos (*dividir y conquistar*).
- División repetida de problemas y datos en subproblemas más chicos (fase de dividir); resolución independiente de éstos (conquistar), con frecuencia de manera recursiva. Las soluciones son combinadas en la solución global (fase de combinar).
- La subdivisión puede corresponderse con la descomposición entre procesadores. Cada subproblema puede mapearse a un procesador. Cada proceso recibe una fracción de datos: si puede los procesa; sino, crea un n° de “hijos” y les distribuye los datos.

Dividir y Conquistar

- Ejemplos clásicos son el “sorting by merging”, el cálculo de raíces en funciones continuas, problema del viajante.

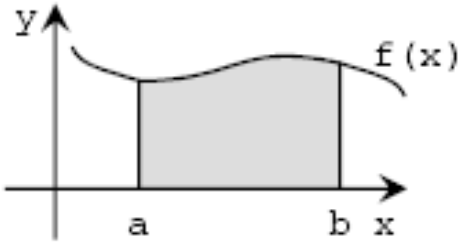


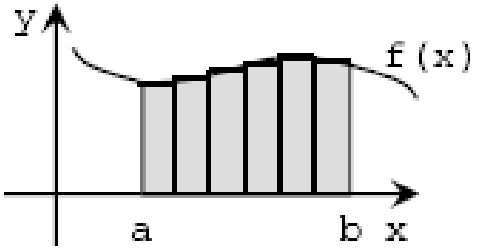




Dividir y Conquistar

Ejemplo el "Problema de la cuadratura": calcular una aproximación de la integral de una función continua *f*(x) en el intervalo de *a* a *b*



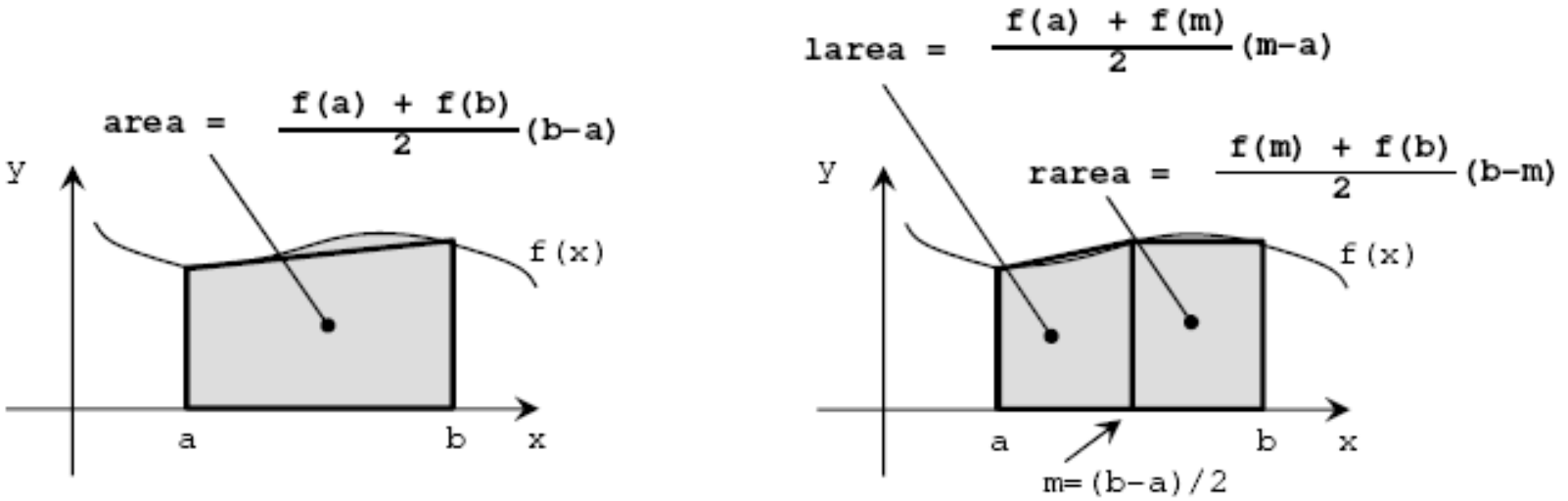


Solución secuencial iterativa (usando el método trapezoidal):

```
double fl = f(a), fr, area = 0.0;
double dx = (b-a)/ni;
for [x = (a+dx) to b by dx]
  {  fr = f(x);
     area = area + (fl+fr) * dx / 2;
     fl = fr;
  }
```

Dividir y Conquistar

Procedimiento recursivo adaptivo



(a) First approximation (area)

(b) Second approximation (larea + rarea)

Si ***abs((larea + rarea) - area) > e***, repetir el cómputo para cada intervalo ***[a,m]*** y ***[m,b]*** de manera similar hasta que la diferencia entre aproximaciones consecutivas esté dentro de un dado ***e.***

Dividir y Conquistar

Procedimiento secuencial

```
double quad(double l,r,fl,fr,area) {
  double m = (l+r)/2;
  double fm = f(m);
  double larea = (fl+fm)*(m-l)/2;
  double rarea = (fm+fr)*(r-m)/2;
  if (abs((larea+rarea)-area) > e) {
     larea = quad(l,m,fl,fm,larea);
     rarea = quad(m,r,fm,fr,rarea);

  }
  return (larea+rarea);
}
```

Procedimiento paralelo

```
double quad(double l,r,fl,fr,area) {
   double m = (l+r)/2;
   double fm = f(m);
   double larea = (fl+fm)*(m-l)/2;
   double rarea = (fm+fr)*(r-m)/2;
   if (abs((larea+rarea)-area) > e) {
      co larea = quad(l,m,fl,fm,larea);
      || rarea = quad(m,r,fm,fr,rarea);
      oc
   }
   return (larea+rarea);
}
```

- Dos llamados recursivos son independientes y pueden ejecutarse en paralelo.
- Uso: area = quad (a, b ,f(a), f(b), (f(a) + f(b)) * (b-a) /2)

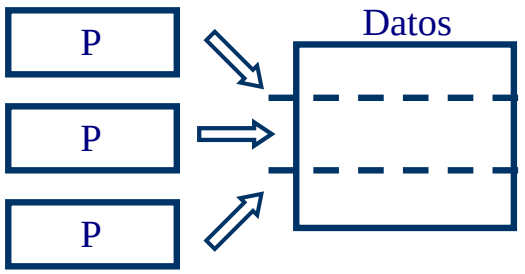
SPMD

- El programador genera un programa único que ejecuta cada nodo sobre una porción del dominio de datos. La diferente evaluación de un predicado en sentencias condicionales permite que cada nodo tome distintos caminos del programa

- Dos fases: 1) elección de la distribución de datos y 2) generación del programa paralelo

1) Determina el lugar que ocuparán los datos en los nodos. La carga es proporcional al número de datos asignado a cada nodo. Dificultades en computaciones irregulares y máquinas heterogéneas.

2) Convierte al programa secuencial en SPMD. En la mayoría de los lenguajes, depende de la distribución de datos.



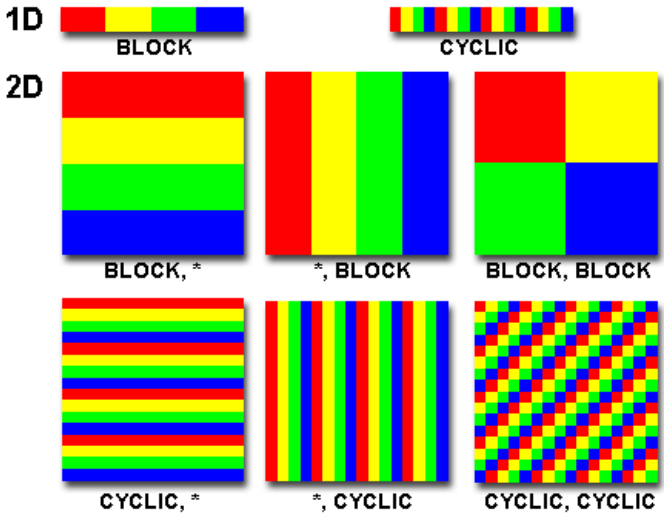
SPMD

SPMD

- Suele implicar ***paralelismo iterativo*** donde un programa consta de un conjunto de procesos los cuales tiene 1 o más *loops*. Cada proceso es un programa iterativo.

- Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones.

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}$$

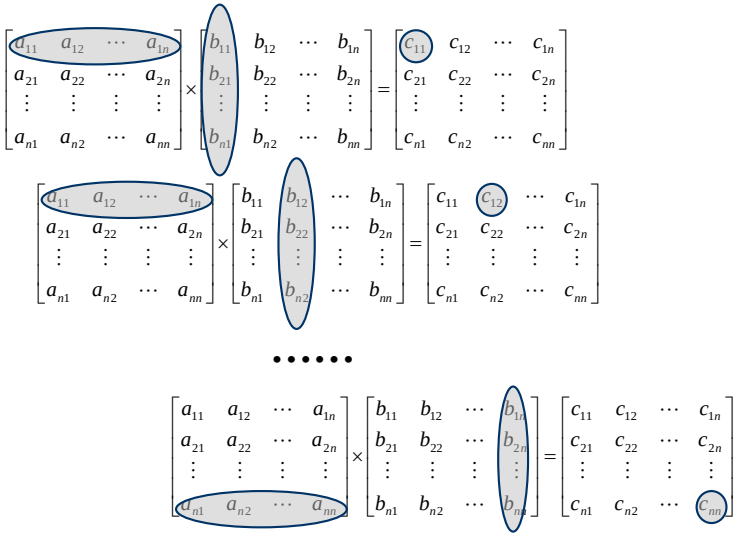


SPMD

Ejemplo de SPMD: multiplicación de matrices.

Solución secuencial:

```
double a[n,n], b[n,n], c[n,n];
for [i = 1 to n]
  { for [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```

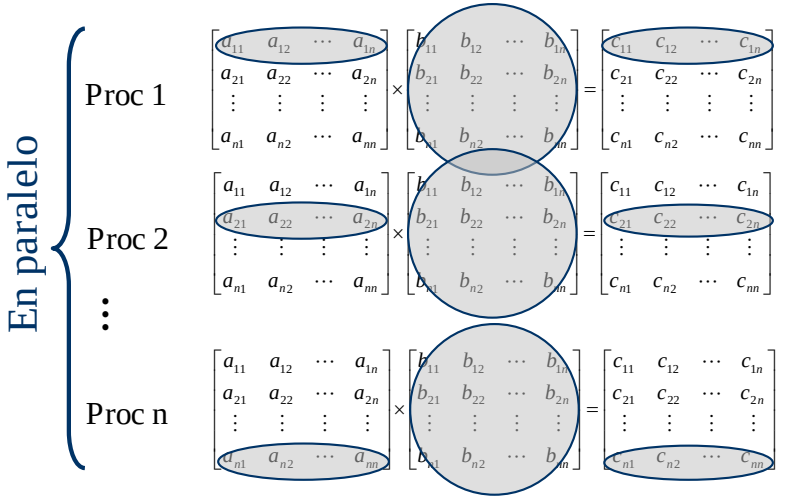


- El loop interno calcula el producto interno de la fila *i* de la matriz *a* por la columna *j* de la matriz *b* y obtiene *c*[*i*,*j*].
- El cómputo de cada producto interno es independiente. Aplicación *embarrasingly parallel* (muchas operaciones en paralelas).
- Diferentes acciones paralelas posibles.

SPMD

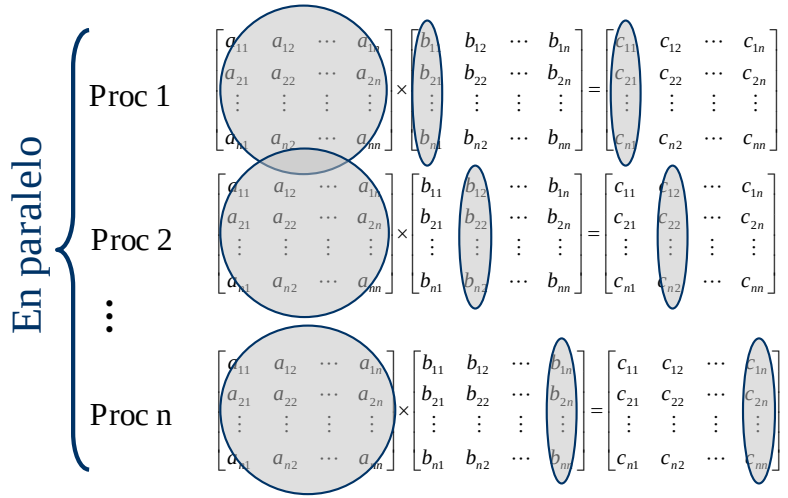
Solución paralela por fila:

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { for [j = 1 to n]
      { c[i,j] = 0;
        for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
  }
```



Solución paralela por columna:

```
double a[n,n], b[n,n], c[n,n];
co [j = 1 to n]
  { for [i = 1 to n]
      { c[i,j] = 0;
        for [k = 1 to n]
          c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
  }
```



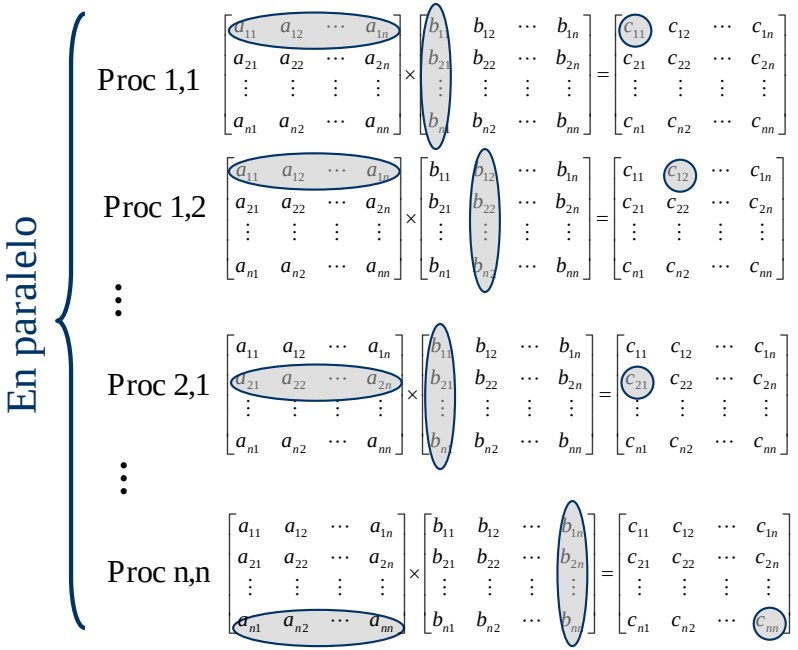
SPMD

Solución paralela por celda (opción 1):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n , j= 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
```

Solución paralela por celda (opción 2):

```
double a[n,n], b[n,n], c[n,n];
co [i = 1 to n]
  { co [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
      }
  }
```



SPMD

Solución paralela por fila con process:

```
process fila [i = 1 to n]
  { for [j = 1 to n]
     { c[i,j] = 0;
       for [k = 1 to n]
         c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
     }
  }
```

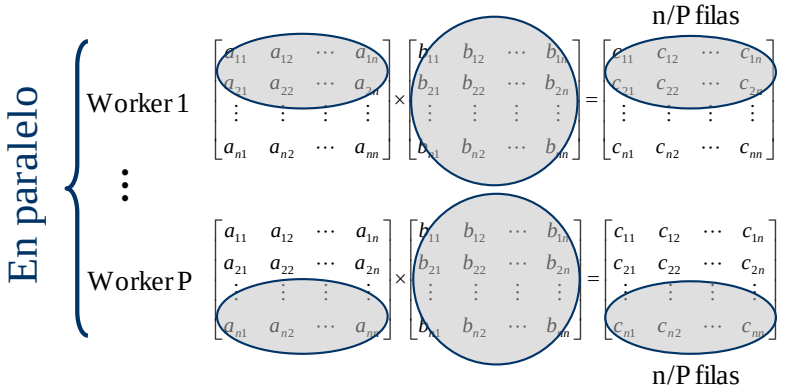
¿Qué sucede si hay menos de *n* procesadores?

- Se puede dividir la matriz resultado en *strips* (subconjuntos de filas o columnas) y usar un proceso ***worker*** por strip.
- El tamaño del strip óptimo es un problema interesante para balancear costo de procesamiento con costo de comunicaciones.

SPMD

Solución paralela por strips:
(*P* procesadores con *P*<*n*)

```
process worker [ w = 1 to P]
  { int primera = (w-1)*(n/P) + 1;
    int ultima = primera + (n/P) - 1;
    for [i = primera to ultima]
      { for [j = 1 to n]
          { c[i,j] = 0;
            for [k = 1 to n]
              c[i,j] = c[i,j] + (a[i,k]*b[k,j]);
          }
      }
  }
```



Ejercicio: a) Si P=8 y n=120. ¿Cuántas asignaciones, sumas y productos hace cada procesador?.
b) Si P1=...=P7 y los tiempos de asignación son 1, de suma 2 y de producto 3; y si P8 es 2 veces más lento. ¿Cuánto tarda el proceso total?. ¿Cuál es el speedup?. ¿Qué puede hacerse para mejorar el speedup?.