I
Programación Concurrente
Clases 1 y 2: Conceptos básicos. Comunicación y sincronización. Interferencia.

Ramiro Martínez D'Elía

2021

Índice general

Capítulo 1

Programa concurrente

1.1. Definición

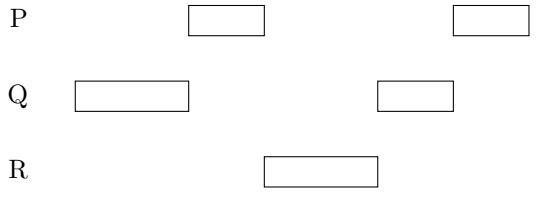
La ***concurrencia*** es la capacidad de ejecutar múltiples actividades de forma simultánea. Un ***programa concurrente***, en consecuencia, es la especificación de uno o más programas secuenciales que pueden ejecutarse de forma concurrente como procesos o tareas.

Básicamente es un concepto de software que, dependiendo la arquitectura dará lugar a definiciones como: programación paralela y programacion distribuida.

Por último, vale aclarar que, la concurrencia no implica paralelismo.

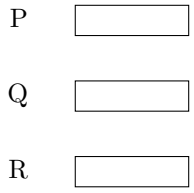
Concurrencia interleaved

- Procesamiento simultáneo lógicamente.
- Ejecución intercalada en un único CPU.
- Pseudo paralelismo.



Concurrencia simultánea

- Procesamiento simultáneo físicamente.
- Ejecución en múltiples CPU.
- Paralelismo full.



1.2. Patrones de resolución

1.2.1. Introducción

Al igual que con los programas secuenciales, puede existir un gran número de soluciones concurrentes. No obstante, todas siguen los mismos patrones de diseño.

Antes de abordar los patrones de resolución, o diseño, clasificaremos a los procesos de un programa concurrente de la siguiente forma.

- **Peer (par)**: Proceso, posiblemente, identico al resto.
- **Filter (filtro)**: Proceso donde su entrada de datos está conectada a la salida de otro.
- **Client (cliente)**: Proceso que realiza solicitudes, o peticiones, y aguarda por su respectiva respuesta.
- **Server (servidor)**: Proceso que espera peticiones, y envia respuestas en función de las solicitudes recibidas.

1.2.2. Paralelismo iterativo

En el ***paralelismo iterativo*** un programa consta de un conjunto de procesos, posiblemente idénticos, que cooperarán para resolver un único problema. Donde cada proceso es un algoritmo iterativo que, trabaja sobre un subconjunto de datos del problema.

Tipos de procesos, que encontramos, en este patrón: ***peers***.

Multiplicación de matrices (por filas)

$$C_{1,1} = A_{1,n}xB_{n,1} = A_{1,1}xB_{1,1} + A_{1,2}xB_{2,1} + A_{1,3}xB_{3,1} \tag{1.1}$$

$$C_{1,2} = A_{1,n}xB_{n,2} = A_{1,1}xB_{1,2} + A_{1,2}xB_{2,2} + A_{1,3}xB_{3,2} \tag{1.2}$$

$$A \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} xB \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix} = C \begin{bmatrix} c_{1,1} & c_{1,2} & c_{1,3} \\ c_{2,1} & c_{2,2} & c_{2,3} \\ c_{3,1} & c_{3,2} & c_{3,3} \end{bmatrix} \tag{1.3}$$

```
1  # c[n,n] =  matriz resultante.
2  double  a[n,n], b[n,n], c[n,n];
3
4  # Lanzamos 1 proceso por cada fila de A.
5  co  [a_row = 1 to n]
6
7    # Iteramos sobre las columnas de B
8    # En funcion del proceso id:a_row
9    for [b_col = 1 to n]
10
11     # Inicializamos la celda acumuladora de C.
12     c[a_row, b_col] = 0;
13
14     # Iteramos entre las columnas (A) y filas (B) de interes.
15     for [k = 1 to n]
16       c[a_row, b_col] += a[a_row, k] * b[k, b_row];
17   end;
18 oc
```

> 💎 ¿Qué pasaría si, hay menos de n procesadores? (Ver apunte de Gemas)

1.2.3. Paralelismo recursivo

En este patrón, el programa puede descomponerse en procesos recursivos que trabajan sobre partes del conjunto total de datos. Es un claro ejemplo de la metodología ***divide y conqueror***. Por ejemplo, problema del viajante.

Tipos de procesos, que encontramos, en este patrón: ***peers***.

1.2.4. Productores y consumidores

En este patrón, se observan procesos que se comunican. Unos produciendo información y, los otros, consumiendo dicha información. Es común, organizar los procesos en forma de tubería (pipe); a través de la cual fluye la información. Por ejemplo, aplicar filtros sobre imágenes.

Tipos de procesos, que encontramos, en este patrón: ***filters***.

> 💎 ¿Podemos ordenar un vector utilizando este patrón? (Ver apunte de Gemas)

1.2.5. Clientes y servidores

En este patrón, existen procesos que; realizan pedidos y esperan por una respuesta (clientes) y, otros procesos que, esperan pedidos para procesar (servidores.) Este tipo de soluciones, predominan en aplicaciones distribuidas.

Tipos de procesos, que encontramos, en este patrón: ***clientes y servidores***.

1.2.6. Pares que interactúan

En este esquema los procesos, posiblemente idénticos, resuelven parte del problema y se comunican para avanzar en la tarea y completar el objetivo. Similar al iterativo pero, requiere de sincronización por condición (barriers) entre pares.

Tipos de procesos, que encontramos, en este patrón: ***peers***.

1.3. Estados, acciones e historias

El ***estado***, de un programa concurrente, consiste en los valores de las variables del programa en un instante de tiempo dado. Pudiendo ser, variables explícitas (las definidas por el programador) como variables implícitas (program counter, stack pointer, etc ...).

La ejecución, del programa concurrente, puede ser vista como el intercalado de secuencias de ***acciones atómicas*** ejecutadas por cada proceso. Así, cada ejecución particular del programa puede ser vista como una ***historia (trace)***: $s_1 \rightarrow s_2 \rightarrow ...s_n$.

Cada ejecución de un programa concurrente, produce una historia. Inclusive, hasta en los programas más triviales, el número de historias posibles puede ser enorme. Por ejemplo; para el siguiente programa existen 2 (dos) historias posibles:

```
1  int x = 5;       # (a)
2
3  co x = 0;        # (b)
4  /  x = x + 1;    # (c)
5  oc;
```

Trace	Resultado
$A \rightarrow B \rightarrow C$	$X = 1$
$A \rightarrow C \rightarrow B$	$X = 0$

1.4. Acciones atómicas

Una ***acción atómica***, es una acción que realiza un cambio de estado indivisible. Esto significa que, cualquier estado intermedio que pueda existir en la implementación de la acción no debe ser visible a los otros procesos.

Las acciones atómicas, pueden ser ***condicionales*** o ***incondicionales***. Según tengan, o no, una condición de guarda B que demore su ejecución hasta que la misma sea verdadera.

Una ***acción atómica de grano fino***, es aquella implementada por el hardware (CPU) en el cual el programa concurrente es ejecutado.

En un programa secuencial, las sentencias de asignación parecen ser atómicas. Esto es porque ningún estado intermedio es visible al programa. En los programas concurrentes, no se puede afirmar lo mismo. Una sentencia de asignación, puede estar implementada por una secuencia de instrucciones máquina. Por ejemplo, para el siguiente programa.

```
1  int x = 2,
2    y = 2;
3
4  co  z = x + y;      # (1)
5  /   x = 3; y = 4;   # (2)
6  oc
```

```
1  # Proceso (1)
2  Load    PosMemX, Temp;
3  Add     PosMemY, Temp;
4  Store   Temp, PosMemZ
5
6  # Proceso (2)
7  Store   3, PosMemX;
8  Store   4, PosMemY;
```

Variable	Valor final
x	3
y	4
z	4, 5, 6, ó 7

La cantidad de valores posibles, para z, se debe a que; en algunas historias el proceso (2) invalidó las suposiciones realizadas por (1). Por ejemplo, si:

$$(1)_2 \rightarrow (2)_7 \rightarrow (1)_3 \rightarrow (1)_4 \rightarrow (2)_8 \Rightarrow z = 4$$

La suposición, en este caso, fue que el proceso (1) realizó la suma con $x = 2$. La cual quedó invalidada, cuando el proceso (2) realizó $x = 3$.

Ese evento, negativo, es conocido como ***interferencia***. El cual se produce, cuando un proceso toma una acción que invalida las suposiciones hechas por otro proceso. Técnicas para evitar la interferencia, pueden ser:

- Variables disjuntas (disjoint variables)
- Afirmaciones debilitadas (weakened assertions)
- Invariantes globales (global invariants)
- Sincronización (synchronization)

1.5. Propiedad de ASV

Una forma de detectar que una sentencia no posee interferencias y, por consiguiente, su ejecución parecerá atómica es verificar que la misma posea, a lo sumo, una referencia crítica.

Una ***referencia crítica***, en una expresión, es la referencia a una variable que es modificada por otro proceso. Toda expresión que contenga a lo sumo una referencia crítica, cumple con la ***propiedad de a lo sumo una vez*** y su ejecución parecerá atómica.

Por ejemplo, la sentencia de asignación $\boldsymbol{x = expr}$ satisface la propiedad si se cumple alguna de las siguientes condiciones:

1. Si ***expr*** contiene a lo sumo una referencia crítica, en cuyo caso $\boldsymbol{x}$ no podrá ser leída por otro proceso.
2. Si ***expr*** no contiene referencias críticas, en cuyo caso $\boldsymbol{x}$ podrá ser leída por otro proceso.

Program	Cumple
Uno	Si
Dos	Si
Tres	No

```
1 Program Uno
2 int x = 0,
3   y = 0;
4 co  x = x + 1;
5 /   y = y + 1;
6 oc
7 End.
```

```
1 Program Dos
2 int x = 0,
3   y = 0;
4 co  x = y + 1;
5 /   y = y + 1;
6 oc
7 End.
```

```
1 Program Tres
2 int x = 0,
3   y = 0;
4 co  x = y + 1;
5 /   y = x + 1;
6 oc
7 End.
```

1.6. Sentencia Await

Si una expresión o sentencia de asignación, no cumplen con la propiedad ASV será necesario que la ejecutemos de forma atómica. Aunque, en general, necesitaremos ejecutar secuencias de instrucciones como una única acción atómica.

En ambos casos, necesitaremos utilizar algún mecanismo de sincronización que nos permita construir ***acciones atómicas de grano grueso***. Las cuales, no son más que una secuencia de acciones atómicas de grano fino que parecen ser indivisibles.

Podemos especificar acciones atómicas, a través de parentesis angulares $\langle$ y $\rangle$. Como también, especificar sincronización a través de la sentencia ***await***:

```
1 <await(B) S;>
```

- B es una condición booleana que, especifica una condición de demora.
- S es un conjunto de setencias que eventualmente terminarán, y que su estado interno no será visible por otros procesos.

Por ejemplo, en el siguiente programa; B demora hasta que x contenga un valor positivo entonces, decrementa x.

```
<await(x > 0) a = x-1;>
```

Con la sentencia ***await*** podemos construir acciones atómicas de grano grueso arbitrarias. Por ejemplo, el programa anterior es un ejemplo de la operación P de un semáforo a. No obstante, es complejo implementar esta sentencia en su forma general.

La forma general del ***await*** especifica ambas formas de sincronización (***exlusión mutua*** y ***por condicón***). No obstante podemos utilizar su abreviación para, especificarlas por separado.

Para especificar sincronización por ***exlusión mutua*** podemos emplear $\langle$ ***s*** $\rangle$. El siguiente ejemplo, incrementa a x e y de forma atómica:

```
<x = x + 1; y = y + 1;>
```

La acción anterior, es una ***acción atómica incondicional***. Ya que; no posee una guarda.

Para especificar sincronización por ***por condicón***, podemos emplear $\langle$ ***B*** $\rangle$. El siguiente ejemplo, demora la ejecución del proceso hasta que $count > 0$:

```
<await(count > 0)>
```

La acción anterior, es una ***acción atómica condicional***. Ya que; posee una guarda.

Si la condición de guarda satisface ASV. Podremos implementar el ***await*** mediante ***busy waiting*** o ***spin loops***.

```
BusyWaiting
  do (not B)
    skip;
  od
End.
```

```
SpinLoop
  while (not B);
End.
```

En otros casos, para lograr implementaciones atómicas del await, necesitaremos instrucciones del tipo **Fetch & Add** o **Test & Set**. Con las cuales, poder construir protocolos de E/S para secciones críticas.

1.6.1. Ejemplo Productores/Consumidores

Sea el problema de procesos que producen y consumen datos, en un búffer con capacidad para n elementos. Donde, existen las siguientes restricciones:

- El productor podrá colocar datos, siempre y cuando la cola no este llena (condición).
- El consumidor podrá leer datos, siempre y cuando la cola no este vacía (condición).
- El acceso, al búffer, debe ser atómico para evitar inconsitencias (mutex).

```
int size = 0;
Queue queue;

Process Productor
  while(true)
    <await(size < N) queue.push(dato)>
  end;
End.

Process Consumidor
  while(true)
    <await(size > 0) queue.pop(dato); size--;>
  end;
End.
```

💎 ¿Qué pasaría si, en lugar de una cola fuese un arreglo? (Ver búffers limitados en Apunte II)

1.7. Propiedades

Una propiedad, en un programa concurrente, es un atributo que debe ser cierto en cada historia posible del programa. Existen 2 (dos) grandes categorías, de propiedades.

La primera, es la de ***seguridad (safety)***. La cual, asegura que nada malo ocurra durante la ejecución; para garantizar que el correcto estado final del programa. Dos propiedades fundamentales, dentro de esta categoría, son:

1. ***Exclusión mutua***: El evento negativo, en este caso, sería que 2 (dos) o más programas; accedan a la misma sección crítica en el mismo instante de tiempo.
2. ***Ausencia de deadlock***: El evento negativo, en este caso, sería que un proceso quede demorado, esperando por una condición que nunca será verdadera.

La segunda, es la de ***vida (liveness)***. La cual, asegura que eventualmente algo bueno ocurrirá; para garantizar la correcta terminación del programa. Por ejemplo:

1. Que todo mensaje, eventualmente, sea recibido por su receptor.
2. Que todo proceso, eventualmente, acceda a su sección crítica.

1.8. Scheduling y Fairness

La mayoría de las propiedades de vida, de un programa concurrente, dependen del ***fairness***. El cual, es un concepto, que se ocupa de garantizar que todos los procesos tienen chances de proceder sin importar lo que hagan los demás.

Para que los procesos avancen, acciones atómicas candidatas deben ser elegidas para su ejecución. Dicha tarea de selección, es llevada a cabo por una política de planificación (***scheduling policy***). Según la bibliografía, existen 3 (tres) grados de fairness que toda política de planificación debería proveer.

💎 Ver el ejercicio en el apunte de Gemas.

1.8.1. Incondicionalmente Fair

Una política de planificación cumple con este grado si cada acción atómica, incondicional elegible, eventualmente es ejecutada. Por ejemplo, para el siguiente programa:

```
1 var continue = true;
2 co  while (continue); (P)
3 /   continue = false; (Q)
4 oc
```

Una política donde se conceda CPU, a los procesos, hasta que estos terminan o son demorados; no sería incondicionalmente fair. Ya que si (P) ingresa primero entonces, (Q) no tendría oportunidad de ser ejecutada.

Por el contrario, Round Robin si resulta una política incondicionalmente fair. Ya que, (Q) eventualmente podrá acceder a CPU.

1.8.2. Débilmente Fair

Una política de planificación cumple este grado si, es incondicionalmente fair y cada acción atómica condicional eventualmente es ejecutada. Asumiendo que su condición, se volverá verdadera y permanecerá así hasta ser vista por el proceso ejecutando la acción.

1.8.3. Fuertemente Fair

Una política de planificación cumple este grado si, es incondicionalmente fair y cada acción atómica condicional eventualmente es ejecutada. Asumiendo que su condición, se volverá verdadera con infinita frecuencia hasta ser vista por el proceso ejecutando la acción.

Nota: No es posible idear un planificador que sea práctico y fuertemente fair.

II
Programación Concurrente
Clases 3 a 6: Memoria compartida

Ramiro Martínez D'Elía

2021

Índice general

Capítulo 1

Locks y Barriers

1.1. Introducción

Los programas concurrentes emplean dos tipos básicos de sincronización: exclusión mutua y sincronización por condición. Este capítulo examina 2 (dos) problemas importantes (secciones críticas y barreras).

El problema de la sección crítica se preocupa en implementar acciones atómicas por software. Este problema surge en la mayoría de los programas concurrentes, donde, la exclusión mutua es implementada mediante locks que protegen las secciones críticas.

Una barrera (barrier), es un punto de sincronización al que todos los procesos deben llegar, antes de que cualquier proceso se le permita proceder. Es un problema muy común en los programas paralelos.

1.2. El problema de la sección crítica

En este problema, ***n*** procesos repetidamente ejecutan secciones críticas y no críticas de código. La sección crítica está precedida por un protocolo de entrada y seguida por un protocolo de salida. Los procesos que contengan secciones críticas, deberían ser de la siguiente forma.

```
1  Process SeccionCritica[i = 1 to n]
2    while (true)
3       # Entry protocol
4       # critical section
5       # Exit protocol
6       # Noncritical section
7    end;
8   end;
```

Cada sección crítica, es un conjunto de sentencias que acceden a algún recurso compartido. Mientras que, cada sección no crítica es otra secuencia de instrucciones. Para resolver este problema, es necesario implementar protocolos de entrada y salida que cumplan las siguiente 4 (cuatro) propiedades:

1. **Exclusión mutua:** A lo sumo un proceso podrá estar ejecutando su sección crítica. Esta es una ***propiedad de seguridad***; donde lo malo que puede ocurrir es que 2 (dos), o más, procesos accedan a su sección crítica en el mismo momento.

2. **Ausencia de deadlock:** Si 2 (dos) o más procesos intentan entrar a sus secciones críticas, al menos uno tendrá éxito. Esta es una propiedad ***propiedad de seguridad***; donde lo malo que puede ocurrir es que todos los procesos estén esperando ingresar pero, ninguno sea capaz de lograrlo.

3. **Ausencia de demoras inecesarias:** Si un proceso intenta ingresar a su sección crítica y los demás procesos se encuentran ejecutando sus secciones no críticas o finalizaron, el primer proceso no debe estar impedido de ingresar a su sección crítica. Esta es una ***propiedad de seguridad***, donde lo malo que puede ocurrir es que; un proceso no pueda ingresar a su sección crítica aunque no haya procesos en sus secciones críticas.

4. **Eventual entrada:** Todo procesos que intente ingresar a su sección crítica, eventualmente lo logrará. Esta es una ***propiedad de vida*** y es afectada directamente por la política de scheduling.

Cualquier solución al problema de la sección crítica, también puede ser utilizada para implementar sentencias ***await arbitrarias***.

1.3. Señalizando con barreras

Varios problemas pueden ser resueltos utilizando algoritmos iterativos que sucesivamente computen aproximaciones a la respuesta. Terminando cuando la respuesta final haya sido procesada, o bien, haya convergido.

La idea es utilizar múltiples procesos, para procesar partes disjuntas de una solución en paralelo. La clave principal en la mayoría de los algoritmos paralelos, es que cada iteración depende del resultado de una iteración previa. Así, podemos llegar a la siguiente forma general para todo algoritmo que implemente barreras.

```
1 Process Worker[i = 1 .. n]
2   while (true)
3      # Realiza la tarea i
4      # Espera por los demas procesos
5   end;
6  end;
```

Esto es llamado sincronización por barrera, porque la demora al final de cada iteración representa una barrera a la cual todos los procesos deben llegar, antes de que a cualquier otro se le permita continuar.

1.3.1. Barreras simétricas

Si todos los procesos ejecutan el mismo algoritmo y cada proceso está ejecutando en un procesador distinto. Entonces, todos los procesos deberían llegar a la barrera casi al mismo tiempo.

Esta es la opción más adecuada para programas que ejecuten en máquinas con memoria compartida. Mas adelante, se abordarán algoritmos para este tipo de barreras; el ***butterfly*** y ***dissemination barrier*** más precisamente.

1.3.2. Principio de sincronización por banderas

Algunas soluciones, como ***flags and coordinators***, implementan este principio de sincronización. El cual, se basa en las siguientes premisas:

1. El proceso que espera por un flag de condición, es el único que puede limpiar dicho flag.
2. Un flag no puede ser activado, nuevamente, hasta no ser "limpiado".

Capítulo 2

Soluciones con variables compartidas

Introducción

La sincronización, en esta sección, será implementada mediante la técnica de ***busy waiting***. Donde un proceso evalúa, repetidas veces, una condición hasta que esta se vuelva verdadera.

Ventajas

1. Puede ser implementada utilizando instrucciones, de máquina, disponibles es cualquier procesador moderno.
2. Adecuada si cada proceso se ejecuta en su propio procesador.

Desventajas

1. Ineficiente en arquitecturas monoprocesador.

2.1. Para secciones críticas

2.1.1. Spin locks

Solución de grano fino que utiliza instrucciones atómicas especiales, existentes en la mayoría de los procesadores. Por ejemplo, ***Test and Set (TS)***. Se dice que los procesos dan "vueltas" (spinning) hasta que se libere lock.

```
1  bool lock = false;
2
3  Process CS[i = 1 to n]
4    while (true)
5      while (TS(lock)) skip;    # Entry
6      # Critical Section
7      lock = false;             # Exit
8      # Noncritical Section
9    end;
10 end;
```

Ventajas

1. Cumple con 3 (tres) de los requisitos, para secciones críticas: *garantiza exclusión mutua*, *ausencia de deadlock* y *ausencia de demoras inecesarias*.

Desventajas

1. La *eventual entrada* se garantiza solo con *schedulers fuertemente fairs*. Ya que lock se vuelve verdadera, con infinita frecuencia.
2. No atiende prioridades, es decir; no controla el orden en que los procesos, demorados, entran a su sección crítica.

2.1.2. Implementaciones Fair

Spin locks, no termina siendo del todo adecuada. Sería deseable, contar con algoritmos que:

1. Cumplan las 4 (cuatro) propiedades, de una sección crítica.
2. Solo dependan de *schedulers débilmente fair*.
3. Sean más justos. Es decir, manejen prioridades.

Los algoritmos ***Tie breaker***, ***Ticket*** y ***Bakery*** parecen más adecuados ya que, cumplen con todos los requisitos mencionados anteriormente.

2.1.3. Tie breaker

Este algoritmo asegura la exclusión mutua mediante dos variables, una por proceso, `in1` e `in2`. En caso de que ambas valgan verdadero (empate) emplea una variable adicional, `last`, para determinar cuál fue el último en ingresar a su sección crítica.

```
1  bool in1, in2 = false;
2  int last = 1;

3  Process CS1
4    while (true)
5      last = 1; in1 = true; # Entry
6      while (in2 and last == 1) skip;
7      # Critical Section
8      in1 = false; #Exit
9      # Noncritical Section
10   end;
11 end;
12
```

```
13 Process CS2
14   while (true)
15     last = 2; in2 = true; # Entry
16     while (in1 and last == 2) skip;
17     # Critical Section
18     in2 = false; # Exit
19     # Noncritical section
20   end;
21 end;
```

Ventajas

1. No requiere instrucciones especiales.
2. Prioriza al primer proceso, en iniciar el protocolo de entrada.

Desventajas

1. Dificil generalizarlo a n procesos.

2.1.4. Ticket

El algoritmo ticket, es una solución al problema de la sección crítica generalizada para n procesos, fácil de entender e implementar. El algoritmo se basa en la entrega de tickets (números) a procesos y posteriormente atenderlos en orden de llegada.

Para esto, obligatoriamente, se requiere de alguna instrucción especial que entregue e incremente los números a cada proceso de forma atómica, para evitar duplicados. Esta instrucción puede ser ***Fetch and Add***.

De no existir una instrucción máquina, de estilo ***Fetch and Add***, podemos reemplazarla con otra sección crítica.

```
1  int number = 1;
2  int next = 1;
3  int[] turn[n] = ([n] 0);
```

```
# Con instruccion FA

Process Worker[i = 1..n]
  # Entry protocol
  turn[i] = FA(number, 1);
  while (turns[i] != next) skip;
  # Critical Section
  next = next + 1;        # Exit
  # Noncritical Section
end;
```

```
# Sin instruccion FA

Process Worker[i = 1..n]
  turn[i] = number;
  <number = number + 1>
  while (turns[i] != next) skip;
  # Critical Section
  next = next + 1;
  # Noncritical Section
end;
```

Ventajas

1. Sencillo de implemetar.
2. General para n procesos.

Desventajas

1. La implementación sin instrucciones especiales, puede entregar números repetidos. Esto, decrementa el grado de justicia del algoritmo.

2.1.5. Bakery

Al igual que en el algoritmo ticket, los procesos obtienen un número y esperan a ser atendidos. La diferencia radica en que en el algoritmo bakery, cada proceso debe revisar el número de los demás para obtener uno mayor a todos los que se encuentran demorados.

```
int[] turn[n] = ([n] 0);

Process SC [i = 1 .. n]
  while (true)
    turn[i] = 1;
    turn[i] = max(turn) + 1;
    for (j = 1 to n st j != i)                          # Entry
      while (turn[j] != 0 and turn[i] > turn[j]) skip;
    # Critical Section
    turn[i] = 0;                                        # Exit
    # Noncritical Section
  end;
end;
```

Ventajas

1. No requiere instrucciones especiales.
2. General para n procesos.

Desventajas

1. Resulta complejo, y costoso, calcular el máximo entre n valores.

2.2. Para barreras

2.2.1. Shared counter

La manera más sencilla de especificar una barrera, es la de utilizar un contador compartido iniciado en 0 (cero).

Asumiendo que existan n procesos que necesitan reunirse en un barrera. Cuando un proceso llega a la barrera, incrementa el contador; cuando el contador valga n, todos los procesos podrán continuar.

```
int count = 0;

Process Worker[i = 1 .. n]
  while (true)
    # Realizar tarea ...
    FA(count, 1);
    while (count != n) skip;
  end;
end;
```

Ventajas

1. Solución adecuada para un n pequeño.

Desventajas

1. La variable *count* debe ser reiniciada cuando todos crucen la barrera y, por sobre todo, antes de que cualquier proceso intente incrementarla.
2. Requiere instrucciones especiales.
3. Requiere una política de administración eficiente. La variable *count* es referenciada varias veces, esto puede provocar ***Memory Contention***.

2.2.2. Flags and coordinators

El algoritmo utiliza **dos arreglos como flags**; el primero para marcar la llegada de cada proceso a la barrera. Y el segundo, para indicar a cada proceso que puede continuar. Este último arreglo es actualizado por un proceso coordinador.

```
int[] arrive[n] = ([n] 0);
int[] continue[n] = ([n] 0);

Process Worker[i = 1..n]
  while (true)
    # Realizar tarea
    arrive[i] = 1;
    while (continue[i] == 0) skip;
    continue[i] = 0;
  end;
end;
```

```
Process Coordinator
  while (true)
    for (i = 1 to n)
      while (arrive[i] == 0) skip;
      arrive[i] = 0;
    end;
    for (i = 1 to n)
      continue[i] = 1
  end;
end;
```

Ventajas

1. Resetea correctamente los contadores.
2. Evita ***memory contention***. Ya que, cada elemento del arreglo utiliza una línea de caché distinta.

Desventajas

1. El tiempo de ejecución del coordinador, es proporcional a n. Por su uso, no es recomendable para n muy grandes.

2.2.3. Combined tree barrier

Este algoritmo combina el rol de los workers y el del coordinador, de forma tal que cada worker es, también, un coordinador.

Los procesos son organizados en forma de árbol y se procede con la siguiente lógica: cada nodo worker primero espera a que sus hijos le den la señal de llegada, luego avisa a su padre que él también llegó.

Cuando el nodo raíz, recibe la señal de llegada de sus hijos se sobreentiende que todos los demás workers también lo hicieron. Así, la raíz envía la señal de continuar a sus hijos y así sucesivamente.

```
1  int[n] arrive = 0;
2  int[n] continue = 0;

3  Process Leaf[1..L]
4    # Hacer algo ...
5    arrive[L] = 1;
6    < await (continue[L] == 1) >
7    continue[L] = 0;
8  End.
9
10 Process Root
11   < await(arrive[left] == 1) >
12   arrive[left] = 0;
13   < await(arrive[right] == 1) >
14   arrive[right] = 0;
15   # Hacer algo ...
16   arrive[R] = 1;
17   < continue[left] = 1;
18   continue[right] = 1; >
19 End.
20
21 Process Internal[1..I]
22   < await(arrive[left] == 1) >
23   arrive[left] = 0;
24   < await(arrive[right] == 1) >
25   arrive[right] = 0;
26   # Hacer algo ...
27   arrive[I] = 1;
28   < await(continue[I] == 1) >
29   continue[I] = 0;
30   < continue[left] = 1;
31   continue[right] = 1; >
32 End.
```

Ventajas

1. Útil para n muy grandes ya que, el tiempo de ejecución es proporcional al alto del árbol: $\log_2 n$.
2. Adecuado para máquinas con memoria distribuida.

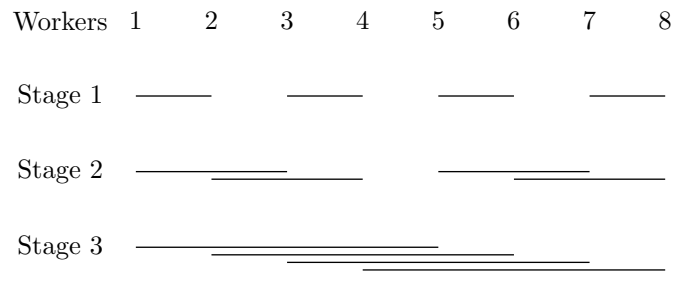
2.2.4. Butterfly

La idea es conectar barreras de pares de procesos, para construir una barrera de n procesos. Asumiendo que *Worker[1:n]* es un arreglo de procesos y que n es potencia de 2, podríamos combinarlos de la siguiente manera.

Por la forma de conexión, es conocida como butterfly barrier. Como se aprecia en la figura, cada proceso se conecta con otro distinto en cada una de sus $\log_2 n$ pasadas. Más precisamente, en cada pasada, cada proceso se sincroniza con otro a una distancia 2^{S-1}.

Cuando un proceso finalizó todas sus pasadas, todos los procesos arribaron a la barrera y pueden proceder. Esto es, porque los procesos están directa o indirectamente sincronizados los unos con los otros.

```
1  int[n] arrive = 0;
2  int stages = log2(n);
3
4  Process Worker[i=1..n]
5    for (s = 1 to stages)
6      arrive[i] = arrive[i] + 1;
7      j = neighbord_for(i, s);
8      while (arrive[j] < arrive[i])
9        skip;
10   end;
11 End.
```



Si n no fuese potencia de 2, podría utilizarse el siguiente n potencia de 2. Generando workers substitutos para cada iteración. Este workaround, decrementa la eficiencia del algoritmo.

2.3. Defectos

La mayoría de los protocolos implementados por ***busy waiting*** son ***complejos*** y la ***separación entre variables***, utilizadas para la sicronización y para cómputo general, es ***poco clara***.

Otro defecto es la ***ineficiencia*** de los protocolos de busy waiting ***en la mayoria de los programas multihilos***. Excepto para el caso de los programas paralelos donde el número de procesos concuerde con el número de procesadores.

No obstante, usualmente, existen más procesos que procesadores y ***resulta menos productivo otorgar CPU a procesos para que hagan spinning en lugar de cómputo***.

La eficiencia es fundamental, en los programas concurrentes, por esto es deseable tener herramientas más eficientes, para el desarrollo de aplicaciones concurrentes, como: ***semáforos*** y ***monitores***.

Con este tipo de herramientas, los procesos demorados no ocupan tiempo CPU hasta que no tengan la posibilidad de ejecutarse. En cuyo caso, serán colocados en una cola de listos.

Capítulo 3

Soluciones con semáforos

Al igual que los semáforos viales sirven para proveer un mecanismo de señalización para prevenir accidentes. En los programas concurrentes, los semáforos sirven para proveer un mecanismo de señalización entre procesos e implementar exclusión mutua y sincronización por condición.

3.1. Sintáxis y semántica

Un semáforo es una variable compartida, que puede pensarse en términos de una instancia de la clase semáforo. Dicha clase, posee solo dos métodos y una variable interna contador.

El método ***v*** es utilizado para señalar la ocurrencia de un evento, y en consecuencia incrementa de forma atómica el contador interno.

El método ***p***, demora al proceso hasta que un evento haya ocurrido y decrementa de forma atómica el contador interno.

Por último, el ***contador*** es una variable que sólo toma valores enteros positivos.

Tipo de semáforo	Valores del contador
Binario	Entre 0 y 1
General	Entre 0 y ∞

```
1  sem s;
2
3  P(s): < await(s > 0) s = s - 1; >
4
5  V(s): < s = s + 1; >
```

3.1.1. Para secciones críticas

El problema de la sección crítica se puede resolver empleando una variable *lock*. La cual, valdrá 1 (*true*) si no hay procesos en su sección crítica ó 0 (*false*) en caso contrario.

Cuando un proceso desea entrar en su sección crítica; primero deberá esperar a que lock valga 1 (*true*) y luego colocar lock en 0 (*false*). Cuando un proceso sale, deberá colocar a lock nuevamente en (*true*).

```
1  sem mutex = 1;
2  Process SC[i = 1 .. n]
3    while (true)
4      P(mutex);     # Entry
5      # Seccion Critica
6      V(mutex);     # Exit
7      # Seccion no critica
8    end;
9  end;
```

3.2. Split Binary Semaphores

La técnica split binary semaphores consiste en combinar 2 (dos), o más, semáforos binarios como si fuesen un solo. Todo conjunto de semáforos, formaran un SBS si cumplen la siguiente regla: $0 \leq s_1 + s_2 + ... + s_n \leq 1$.

3.2.1. Productores y consumidores

Dado un programa donde los procesos se comunican, entre sí, mediante un ***búffer con capacidad para 1 (un) mensaje***. En dicho programa, existiran 2 (dos) clases de procesos: los productores y los consumidores. Los ***productores***; crean mensajes, esperan a que el búffer esté vacío, depositarán su mensaje y marcarán el búffer como lleno. Mientras que, los ***consumidores***; esperan a que el búffer esté lleno, retiran el mensaje y marcan el búffer como vacío.

La forma más sencilla de sincronizar a los procesos es; utilizar semáforos en terminos de los estados posibles del búffer; *empty* y *full*. Juntos, *empty* y *full*, conforman un split binary semaphore que; proveerá exclusión mutua sobre el acceso del búffer.

```
1  sem empty = 1, full = 0;
2  any buffer;
```

```
3  Process Consumer[1..n]
4    while (true)
5      P(full);
6      message = buffer;
7      V(empty);
8    end;
9  End.
10
```

```
11 Process Producer[1..n]
12   while (true)
13     P(empty);
14     buffer = message;
15     V(full);
16   end;
17 End.
```

3.3. Contadores de recursos

Usualmente los procesos compiten por el acceso a recursos limitados. En esos casos, ***semáforos generales***; pueden ser utilizados como contadores de recursos disponibles.

3.3.1. Buffers limitados

Dado un programa donde los procesos se comunican, entre sí, mediante un ***búffer con capacidad para n mensaje***. En dicho programa, existiran 2 (dos) clases de procesos: los productores y los consumidores. Los ***productores***; crean mensajes, esperan a que el búffer esté vacío, depositarán su mensaje y marcarán el búffer como lleno. Mientras que, los ***consumidores***; esperan a que el búffer esté lleno, retiran el mensaje y marcan el búffer como vacío.

En este caso, el recurso son los espacios libres del buffer. Como adición, se debe aplicar exclusión mutua para que distintos consumidores no recuperen el mismo mensaje (mantener consitente *front*) y para, que los productores no sobreescriban mensajes (mantener consistente *near*).

```
1  int front, rear = 0;
2  sem empty = n, full = 0;
3  sem mutexFetch = 1, mutexDeposit = 1;
4  array[n] buffer;
```

```
Process Producer[1..n]
  while (true)
    P(empty);
    P(mutexDeposit);
    buffer[rear] = data;
    rear = (rear + 1) % n;
    V(mutexDeposit);
    V(full);
  end;
End.
```

```
Process Consumer[1..n]
  while (true)
    P(full);
    P(mutexFetch)
    data = buffer[front];
    front = (front + 1) % n;
    V(mutexFetch)
  end;
End.
```

3.4. Exclusión mutua selectiva

La exlusión mutua selectiva, se presenta cuando cada proceso compite contra un subconjunto de procesos (por un recurso). En lugar de, competir contra todos.

3.4.1. Dining Philosophers

Por ejemplo: cinco filósofos se sientan a comer en una mesa redonda donde hay solo cinco tenedores. Cada filósofo, para comer, requiere de dos tenedores. Esto implica que; dos filósofos vecinos no pueden comer al mismo tiempo y que, a lo sumo, solo dos filósofos podrán comer al mismo tiempo.

El problema de exclusión mutua selectiva se da entre cada par de filósofos y un tenedor. Tener en cuenta, de que si fuesen 3 filósofos, no sería un problema de exclusión mutua.

```
sem forks[5] = {0,0,0,0,0}

Process Philosopher[i = 0 to 3]
  while(true)
    p(fork[i]);
    p(fork[i+1]);
    # come ...
    v(fork[i]);
    v(fork[i+1]);
  end;
end;
```

```

Process Philosopher[4]
  while(true)
    p(fork[0]);
    p(fork[4]);
    # come ...
    v(fork[0]);
    v(fork[4]);
  end;
end;
```

💎 Si en lugar de 5 filósofos fueran 3, ¿el problema seguiría siendo de exclusión mutua selectiva? ¿Por qué?
Si hay 3 filósofos, por consiguiente hay 2 tenedores. Lo que significa que todos los procesos son adyacentes, por lo cual la competencia se da entre todos y dejaría de ser un problema de exclusión mutua selectiva.

💎 El problema de los filósofos resuelto de forma centralizada y sin posiciones fijas ¿es de exclusión mutua selectiva? ¿Por qué?
Si los filósofos pueden utilizar cualquier cubierto (sin posiciones fijas) y, en caso de haber disponibilidad, son servidos por un mozo (coordinador). No estamos hablando de exclusión mutua selectiva ya que, los filosofos compiten contra todos no solo contra sus adyacentes.

3.4.2. Lectores y escritores

Otro ejemplo, de exclusión mutua selectiva, donde clases de procesos, compiten por el acceso a un recurso es el siguiente. Procesos escritores, que requieren acceso exclusivo para evitar interferencias. Y procesos, lectores, los cuales pueden acceder de forma concurrente entre sí (siempre y cuando no haya escritores haciendo uso de la base de datos).

El siguiente algoritmo, resuelve el problema. No obstante, esta solución no es fair. Ya que, prioriza lectores por sobre escritores.

```
1  int nr = 0; # lectores activos
2  sem rw = 1; # acceso a bbdd
3  sem mutexR = 1; # acceso a nr

4  Process Reader
5    while (true)
6      P(mutexR)
7      nr = nr + 1;
8      if (nr == 1) P(rw);
9      V(mutexR);
10     # lee en bbdd
11     P(mutexR)
12     nr = nr - 1;
13     if (nr == 0) V(rw);
14     V(mutexR);
15   end;
16 end;
```

```
17 Process Writer
18   while (true)
19     P(rw);
20     # escribe en bbdd
21     V(rw);
22   end;
23 end;
```

💎 Si solo se acepta sólo 1 escritor o 1 lector en la BD, ¿tenemos un problema de exclusión mutua selectiva? ¿Por qué?
En este caso, sería solo un problema de exclusión mutua. Debido a que, por definición, la competencia es entre todos los procesos (lectores vs escritores, lectores vs lectores, escritores vs escritores).

3.5. Passing the Baton

Técnica que utiliza ***split binary semaphores*** para proveer exclusión mutua y despertar procesos demorados (incluso respetando su orden). Empleando esta técnica, podremos ***especificar sentencias await arbitrarias***. Su implementación, respeta la siguiente forma:

1. Un semáforo $\boldsymbol{e}$. Inicialmente en 1 para, controlar los accesos a la sección crítica.
2. Un semáforo b_j para demorar procesos hasta que, su guarda, B_j sea verdadera.
3. Un contador d_j para contar los procesos demorados por b_j.

Cuando un proceso se encuentra en su sección crítica, retiene el permiso de ejecución (***baton***). Al finalizar, le pasa el permiso a otro proceso (si lo hubiera) o bien, lo libera.

3.5.1. Lectores y escritores

Resolveremos el mismo problema, de lectores y escritores, de la sección anterior introduciendo la técnica ***passing the baton***. Donde:

- $\boldsymbol{e} \rightarrow e$
- $b_j \rightarrow r$ y w
- $d_j \rightarrow dr$ y dw

Si bien este algoritmo, sigue priorizando a los lectores; podremos modificar a ***SIGNAL*** para darle la política que quisieramos.

```
int nr = 0,  # Lectores activos
int nw = 0;  # Escritores activos

int dr = 0, # Lectores demorados
int dw = 0; # Escritores demorados

# SBS: 0<=(e+r+w)<=1
sem e = 1,  # Baton
sem r = 0,  # Demora lectores
sem w = 0;  # Demora escritores
```

```
SIGNAL:
  if (nw == 0 & nr > 0)
    dr--;
    V(r);
  elseif (nr == 0 & nw == 0 & dw > 0)
    dw--
    V(w);
  else
    V(e);
```

```
Process Writer[1..n]
  while (true)
    P(e);
    if (nr > 0 or nw > 0)
      dw++;
      V(e);
      P(w);
    end;
    nw++;
    SIGNAL
    # Escribir en la bbbd ...
    P(e);
    nw--;
    SIGNAL
  end;
End.

```

```
Process Reader[1..n]
  while (true)
    P(e);
    if (nw > 0)
      dr++;
      V(e);
      P(r);
    end;
    nr++;
    SIGNAL
    # Leer en la bbbd ...
    P(e);
    nr--;
    SIGNAL
  end;
End.
```

💎 ¿Qué relación encuentra con la técnica de passing the condition?
Ambas técnicas poseen la misma finalidad. Solo que, Passing the condition es utilizada en Monitores.

3.5.2. Alocación de recursos - SJN

En los problemas de alocación de recursos, se deben implementar políticas de acceso generales a un recurso compartido por los procesos.

Dichas políticas deben contemplar: cuándo se le puede dar acceso a un proceso y cuál proceso toma el recurso si hubiese más de uno esperando.

Como su nombre lo indica, una política de ***Short Job Next*** le otorga el recurso al proceso en espera con menor valor de tiempo (el más corto). Su semántica, es la siguiente:

- `request(time, id)`: Un proceso solicita el acceso. Si el recurso está libre, se le otorga. Caso contrario, se coloca al proceso en una lista de espera.
- `release`: Un proceso libera el recurso. Si hay procesos en espera se otorga, el recurso, al proceso con valor de tiempo más chico. Caso contrario, el recurso queda libre.

Este problema puede ser implementado, utilizando la técnica Passing the baton de la siguiente manera:

```
request(time, id):
  P(e);   # mutex sobre free
  if (not free) DELAY;
  free = false;
  SIGNAL;

release():
  P(e);   # mutex sobre free
  libre = true;
  SIGNAL;
```

En **DELAY** un proceso:

- Se coloca en una lista de espera.
- Libera SC ejecutando `V(e)`.
- Demora en un semáforo `b[id]`.

En **SIGNAL** un proceso:

- Cuando se libera el recurso, se le asigna al próximo proceso en espera si lo hubiera.

Esta política minimiza el tiempo promedio de ejecución pero, **no es una solución fair**. Procesos con un tiempo mayor, pueden quedar relegados ante la aparición de procesos con tiempo menor.

Dicha desventaja se puede evitar, implementando políticas de **aging**. Es decir controlando, también, el tiempo en el que un proceso estuvo en lista de espera.

3.6. Instrucciones máquina utilizadas

3.6.1. Fetch and Add

De manera atómica incrementa *number* en *inc* veces y retorna su antiguo valor.

```
1 FA (int number, int inc)
2   < int temp = number;
3   number = number + inc;
4   return temp; >
5 end;
```

3.6.2. Test and Set

De manera atómica setea *lock* en true y retorna su antiguo valor.

```
1 bool TS(bool lock)
2   < bool initial = lock;
3   lock = true;
4   return initial; >
5 end;
```

3.7. Desventajas

Los semáforos resultan un mecanismo de sincronización fundamental. Con ellos, podemos resolver cualquier problema de sincronización.

No obstante, por naturaleza, son un mecanismo de bajo nivel. Un algoritmo que emplea semáforos posee las siguientes características que, pueden llevar a que el programador cometa errores.

- Variables compartidas globales a los procesos.
- Protocolos de entrada a secciones críticas, dispersas en el código.
- Al agregar procesos, se debe verificar el acceso correcto a las variables compartidas.
- La implementación de exclusión mutua y sincronización por condicion son similares, aunque los conceptos sean distintos.

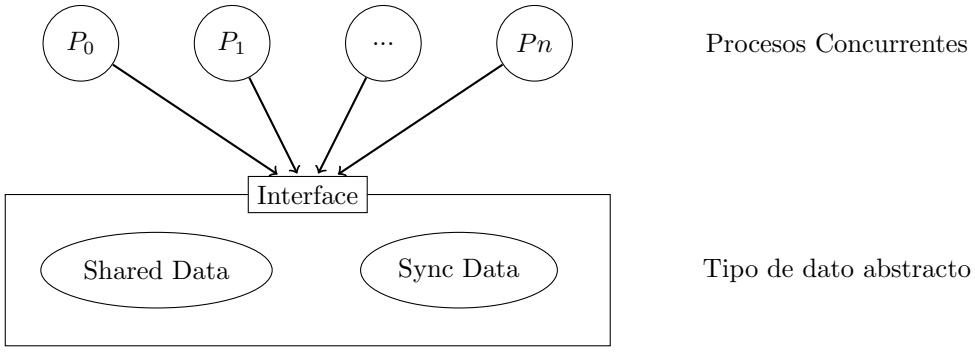
Capítulo 4

Soluciones con Monitores

4.1. Introducción

Los monitores son módulos, del programa, que ofrecen una estructura que permite desarrollar programas concurrentes más limpios con respecto al uso de semáforos.

Principalmente, los monitores, funcionan como una capa de abstracción de datos. Un monitor encapsula la representación de un objeto abstracto (o recurso) y ofrece un conjunto de operaciones para manipularlo.



4.2. Sintáxis y Semántica

Un monitor es utilizado para agrupar la representación e implementación de un recurso compartido (class). La especificación de un monitor consta de las siguientes partes:

- **Interface**: Especifica las operaciones provistas por el recurso.
- **Cuerpo**: Contiene las variables que representan al recurso y procedimientos que implemetan las operaciones de la interfaz.

```
1 Monitor Example
2
3 # Declracion de variables permanentes.
4 # Inicializacion de variables.
5
6 procedure op_test(formals)
7   # Body ...
8 end;
9 End.
```

Las ***variables permanentes*** son compartidas por todos los procedimientos dentro del monitor. Son llamadas permanentes, porque existen y consevan sus valores durante todo el ciclo de vida de un monitor. Ellas son inicializadas antes de cualquier llamado a procedimientos.

Los ***procedimientos***, tienen acceso a las variables permanentes, parámetros formales y sus variables locales. Estas últimas, son copiadas e inicializadas en cada llamado al procedimiento.

La principal ventaja, del uso de monitores, es que pueden ser desarrollados de manera aislada al programa que los vaya a utilizar.

4.3. Sincronización

En monitores, a lo sumo **un procedimiento** puede estar **activo a la vez**. Un procedimiento se considera activo, cuando un proceso externo se encuentra ejecutando alguna de sus sentencias.

De esta forma, la ***exclusión mutua*** se garantiza de forma **implícita**. Esto hace los algoritmos más fáciles de leer ya que, no hay protocolos de entrada y salida explícitos.

En contraste, la ***sincronización por condición*** debe ser especificada de forma **explícita**. Esto se debe a que, las condiciones son diferentes en cada programa.

4.3.1. Variables condición

Una variable condición es utilizada para demorar un proceso que no puede continuar ejecutandose, de forma segura, hasta que algún estado del minitor satisfaga una condición.

También son usadas para despertar procesos demorados, cuando la condición se vuelva verdadera.

Este tipo de variables, solo pueden ser utilizadas dentro de un monitor. Y su declaración, sigue la siguiente forma:

```
1 cond vc;
```

Internamente el valor de `vc` es el de una cola, inicialmente vacía. El programador no manipula directamente la cola, sino que lo hace mediante operaciones especiales.

- `empty`: Retorna `true` si la cola está vacía. Retornará `false`, en caso contrario.
- `wait`: Causa la demora del proceso en ejecución y lo agrega a la cola.
- `signal`: Despierta procesos demorados y los retira de la cola.

Si bien, es posible definir una política de prioridades a la cola de procesos demorados; el comportamiento por defecto es el de una política FIFO. De esta forma: `wait` colocará procesos al final de la cola y `signal` desperatá al proceso que este en el frente.

4.4. Disciplinas de señalización

Cuando un proceso ejecuta `signal`, se encuentra dentro de un monitor y por consiguiente tiene el control del lock implícito asociado al monitor.

Dicha situación plantea un dilema. Si `signal` despierta a otro proceso, ahora tendremos 2 (dos) procesos ejecutandose dentro del monitor y por definición, solo un proceso debe tener acceso exclusivo al monitor.

Ante la situación planteada, podemos adoptar alguna de las siguientes políticas:

1. ***Signal and Continue (SC)***: El proceso *señalizador* continua su ejecución. El proceso *señalizado*, es colocado en la cola de listos y espera a que se libere el monitor para continuar.
2. ***Signal and Wait (SW)***: El proceso *señalizado* continua su ejecución. El proceso *señalizador*, espera a que se libere el monitor para continuar.

4.5. Sincronización básica por condición

4.5.1. Buffers limitados

- `not_full` → señaliza `count < n`
- `empty` → señaliza `count > 0`

```
1  monitor BoundedBuffer
2
3  typeT buffer[n];
4  int front = 0, rear = 0, count = 0;
5  cond not_full, not_empty;

6  procedure deposit(typeT data)
7    while(count == n) wait(not_full);
8    buffer[rear] = data;
9    rear = (rear + 1) % 10;
10   count++;
11   signal(not_empty)
12 end;
```

```
13 procedure fetch(typeT &result)
14   while(count == 0) wait(not_empty);
15   result = buffer[front];
16   front = (front + 1) % n;
17   count--;
18   signal(not_full);
19 end;
```

4.6. Broadcast

4.6.1. Lectores y Escritores

- `oktoread` → señaliza `nw == 0`
- `oktowrite` → señaliza `nw == 0 && nr == 0`
- `signal_all` → despierta a todos (broadcast)

```
1  monitor RW_Controller
2
3  int nr = 0, nw = 0;
4  cond oktoread, oktowrite;

5  procedure request_read()
6    while (nw > 0) wait(oktoread);
7    nr = nr + 1;
8  end;
9
10 procedure release_read()
11   nr = nr - 1;
12   if (nr == 0) signal(oktowrite);
13 end;
```

```
14 procedure request_write()
15   while (nr > 0 || nw > 0)
16     wait(oktowrite);
17   nw = nw + 1;
18 end;
19
20 procedure release_write()
21   nw = nw - 1;
22   signal(oktowrite);
23   signal_all(oktoread);
24 end;
```

4.7. Priority Wait

4.7.1. Shortest Job Next Allocation (SJN)

- Se maneja la cola con una politica SJN, en lugar de FIFO.

```
1  Monitor SJN
2
3  bool free = true,
4  cond turn;

5  procedure request(int time)
6    if (free)
7      free = false;
8    else
9      wait(turn, time)
10 end;

11 procedure release()}
12   if (empty(turn))
13     free = true;
14   else
15     signal(turn);
16 end;
```

III
Programación Concurrente
Clases 7 a 9: Memoria distribuida

Ramiro Martínez D'Elía

2021

Índice general

Capítulo 1

Programa distribuido

1.1. Definición

Los mecanismos de sincronización vistos hasta ahora, están pensados para programas concurrentes que ejecutan en hardware con procesadores que comparten memoria (arquitecturas de memoria compartida).

Sin embargo, hay arquitecturas donde los CPU solo comparten una red de comunicación. Bajo este contexto, los procesos ya no comparten variables sino que comparten ***canales***.

Los ***canales*** son abstracciones de una red de comunicación física y al ser el único recurso compartido entre procesos, toda variable es local a un solo proceso y solo accedida por ese proceso. Esto descarta la necesidad de exclusión mutua.

La ausencia de variables compartidas, permite que los procesos puedan ejecutarse en procesadores distribuidos. Por esta razón, los programas concurrentes que utilizan pasaje de mensajes son llamados ***programas distribuidos***.

1.2. Canales

Los canales son ***estructuras FIFO*** (First In First Out) que contienen mensajes pendientes. Estas estructuras soportan 2 (dos) **operaciones atómicas *send* y *receive***. Para iniciar una comunicación, un proceso envía un mensaje por el canal; y otro lo adquiere recibiendolo desde el mismo canal.

La operación ***send*** puede ser **asincrónica** (no demora al proceso que la invoca) y **sincrónica** (demora al proceso que la invoca, hasta que su mensaje fuese recibido). La operación ***receive*** **siempre demora** al proceso que la invoca.

La estructura FIFO y la atomicidad de las primitivas send y receive aseguran que; los mensajes eventualmente serán recibidos, los mensajes no serán corrompidos y serán entregados en el orden en que fueron encolados.

```
1  channel ch1(string), ch2(string);
2
3  Program Uno
4    string respuesta;
5    send ch2("Hola");
6    receive ch1(respuesta);
7  End.
8
9  Program Dos
10   string saludo;
11   receive ch2(saludo);
12   send ch1("Aca esta tu respuesta ...);
13 End.
```

Según la forma en que se utilicen, los canales, pueden clasificarse en los siguientes tipos:

- **Mailbox**: Cualquier proceso puede enviar y recibir datos, por alguno de los canales declarados. Relación *n:n*.
- **Input Port**: El canal tiene un único receptor y *(*n) emisores. Relación *1:n*.

- **Link**: El canal tiene un único receptor y un único emisor. Relación *1:1*.

Capítulo 2

Patrones de resolución

Los tipos de procesos que podemos encontrar en los programas distribuidos, son los mismos que en un programa concurrente con memoria compartida (peers, filters, clientes y servidores). Lo mismo ocurre, con los patrones de resolución.

A continuación, abordaremos un patrón en particular: ***interacting peers***, donde se describirá la forma de interconexión entre procesos. Todos los ejemplos, implementan ***pasaje de mensajes asincrónicos (PMA)***.

2.1. Iteracting peers: intercambiando valores

Supongamos el problema donde existen n procesos, cada uno con un valor local v. El objetivo es, lograr que todos los procesos conozcan el máximo y mínimo valor de v de entre todos.

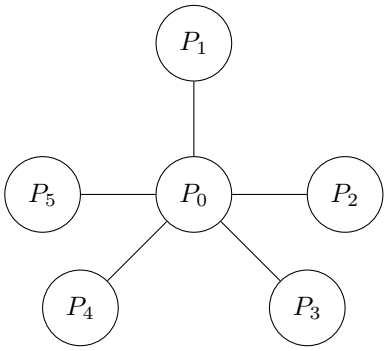
En este caso, los procesos son identicos. Por lo cual, podemos diseñar una solución basada en el esquema de pares que interactúan.

A contuniación se detallan soluciones, empleando 3 (tres) formas distintas de conexión entre procesos: ***centralizada***, ***simétrica*** y ***circular***.

2.1.1. Solución centralizada

En esta solución, utilizaremos un proceso central que; recibirá los valores de los demás. Computará los máximos y mínimos e informará al resto.

```
1  channel values(int),
2      results[n](int, int);
3
4  Process Peer[i = 1 to n-1]
5    int v, min, max;
6    send values(v);
7    receive results[i](min, max);
8  End.
9
10 Process CentralPeer[0]
11   int v, min, max, temp;
12
13   for(i=1 to n-1)
14     receive values(temp);
15     # Compara y actualiza ...
16   end;
17
18   for(i=1 to n-1)
19     send results[i](min, max);
20 End.
```

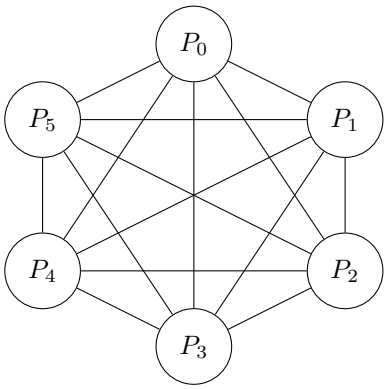


2.1.2. Solución simétrica

En esta solución, todos los procesos realizan el mismo algoritmo. Cada par de procesos, se encuentran conectados mediante un canal.

Cada proceso, envía los datos al resto, y así cada uno puede conocer todos los valores y efecturar el cálculo de máximos y mínimos en paralelo.

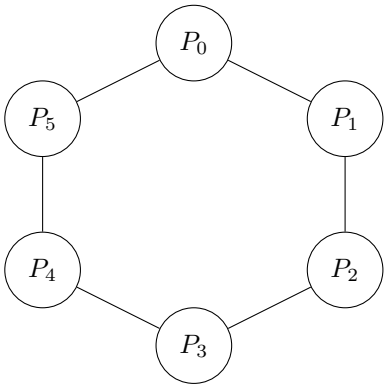
```
1  channel values[n](int, int);
2
3  Process Peer[i = 0 to n]
4    int v, min, max, temp;
5
6    for(j=0 to n st j != i)
7      send values[j](v);
8
9    for(j=0 to n st j != i)
10     receive receive[i](temp)
11     # Compara y actualiza.
12 End.
```



2.1.3. Solución circular

En esta solución, cada proceso recibe mensajes de su antecesor y envía mensajes a su sucesor. Con este enfoque, harán falta 2 (dos) iteraciones al anillo. La primera, para obtener el máximo y el mínimo. Y la segunda, para notificar a todos los procesos.

```
1  channel values[n](int, int);
2
3  Process Initial[0]
4    int v, min = v, max = v;
5
6    # Inicia primera iteracion ...
7    send values[1](min, max);
8    receive values[0](min, max);
9
10   # Inicia segunda iteracion ...
11   send values[1](min, max);
12 End.
13
14 Process Peer[i = 1 to n-1]
15   int v, min, max;
16
17   # Primera iteracion ...
18   receive values[i](min, max);
19   # Compara y actualiza valores ...
20   send values[(i+1) mod n](min, max);
21
22   # Segunda iteracion
23   receive values[i](min, max);
24   send values[(i+1) mod n](min, max);
25 End.
```



2.1.4. Conclusiones

Desde el punto de vista de la cantida de mensajes enviados y la eficiencia, podemos obtener las siguientes conclusiones para las soluciones descriptas.

Solución	**# Mensajes**	**Orden**	**Conclusión**
Centralizada	$2(n-1)$	Lineal $O(n)$	No hay paralelismo, toda la carga del cómputo recae en CentralPeer. Menor overhead en el pasaje de mensajes. Menor eficiencia a mayor n.
Simétrica	$n(n-1)$	Cuadrático $O(n^2)$	Cómputo paralelo pero, mayor overhead en el pasaje de mensjaes.
Circular	$2(n-1)$	Lineal $O(n)$	Cómputo paralelo. Mayor eficiencia para n muy grandes.

Nota: Con la primitiva ***broadcast***, la cantidad de mensajes se reduce a n para todos los casos. Lo que hace, dicha primitiva es envíar un mensaje (de forma concurrente) a todos los procesos que escuchen el canal utilizado.

Capítulo 3

PMS - Pasaje de Mensajes Sincrónico

3.1. CSP y Comunicación guardada

Con CSP (***Communicating Sequential Processes***) se introduce el concepto de ***PMS y comunicación guardada***. Los procesos, se enviarán mensajes. entre ellos, mediante links directos. La forma general de comunicación, entre procesos, en CSP es de la siguiente forma:

```
1 DestinationProc!portName(...);
2 SourceProc?portName(...);
```

Siendo los operadores ***!*** y ***?*** para el envío y recepción de mensajes respectivamente. Dos procesos, entablabran comunicación cuando ejecuten sentencias de comunicación que hagan match entre sí.

El siguiente ejemplo, muestra un proceso (Copy) que copia caracteres entre 2 (dos) procesos East y West.

```
1 Process Copy
2
3 Char c;
4
5 while (true)
6   west?(c)
7   east!(c);
8 end.
9 End.
```

La principal limitación de ***!*** y ***?*** es que, son primitivas bloqueantes. Usualmente, se desea, que un proceso pueda comunicarse con los demás; sin importar el orden en que los otros quieran comunicarse con el.

Por ejemplo, si modificamos el ejemplo de Copy de forma tal que; ahora se deben copiar de a 2 caracteres. El proceso Copy, entonces, debe esperar a recibir los 2 caracteres desde West, antes de retransmitirlos. Esto provoca que, East quede bloqueado.

```
1 Process Copy
2
3 Char c1, c2;
4
5 while (true)
6   west?(c1); west?(c2);
7   east!(c1); east!(c1);
8 end.
9 End.
```

La comunicación guardada, ofrece un tipo de ***comunicación no determinística*** que resuelve la limitación planteada. ***Combinando sentencias condicionales y de comunicación*** de la siguiente forma $B; C \rightarrow S$, donde:

- ***B*** es una sentencia condicional, que de no existir se asume *verdadera*.
- ***C*** es una sentencia de comunicación.
- ***D*** es un conjunto de sentencias a ejecutar.
- Juntas ***B*** y ***C*** forman una ***guarda*** que "protegen"la ejecución de ***S***

Con respecto al funcionamiento de las ***guardas***, se debe tener en cuenta lo siguiente:

- La guarda tiene ***éxito*** si, ***B*** es verdadera y ***C*** no causa demora (tiene mensajes → ejecución inmediata).
- La guarda ***falla*** si, ***B*** es falsa.
- La guarda se ***bloquea*** si, ***B*** es verdadera y ***C*** causa demora.

Podemos aplicar comunicación guardada sobre 2 (dos) tipos de estructuras del control: el ***if*** y el ***do***. Cada una, con la siguiente semántica de ejecución:

- Si ***todas las guardas fallan*** (en simultáneo); la estructura de control finaliza sin efectos.
- Si al menos una guarda tiene éxito; se elige una de manera no deterministica, se ejecuta su ***B*** y luego su ***C***. En el caso del ***if***, la estructura finaliza.
- Si todas las ***guardas están bloqueadas***; se espera hasta que alguna tenga éxito.

La versión del proceso Copy, para 2 caracteres, se puede mejorar utilizando comunicación guardada del siguiente modo:

```
1  Process Copy
2
3  char c1, c2;
4  west?(c1);
5
6  do  true; west?(c2) -> east!(c1); c1=c2;
7  []  true; east!(c1) -> west?(c1);
8  end;
9
10 End.
```

Aplicando comunicación guardada, el proceso *Copy*, una vez recibido el primer caracter podrá (de forma no determinística) esperar al segundo caracter o enviar el primero a *East*. De esta forma, logramos reducir el bloqueo en la comunicación entre *Copy* y *East*.

Capítulo 4

RPC, Rendezvous y Notaciones primitivas

4.1. Introducción

La técnica de pasaje de mensajes, resulta óptima para resolver problemas del tipo productores/consumidores y pares que interactúan. Ya que, estos plantean un tipo de comunicación unidireccional entre procesos.

No obstante, el uso de esta técnica para problemas del tipo cliente/servidor; no resulta del todo óptima. Ese tipo de problemas supone una comunicación bidireccional. Es decir, un cliente realiza una petición y, posteriormente, un servidor le otorga una respuesta. Esto nos obliga a definir un gran número de canales:

- Al menos 1 (un) canal por el cual, el servidor reciba peticiones.
- Un canal de respuesta para cada cliente.

Los mecanismos de ***RPC*** y ***Rendezvous***, resuelven de forma más adecuada los problemas del tipo cliente/servidor. Combinando PMS con aspectos de monitores.

Como en ***monitores***, un módulo (o proceso) expone operaciones públicas. Dichas operaciones, son invocadas por otro módulo (o proceso) utilizando la sentencia ***call***.

Como en ***PMS***, la eejcución de una sentencia ***call*** demora al llamador hasta obtener una respuesta.

Las ***operaciones***, resultan ***canales de comunicación bidireccional***. Desde el llamador hacia el proceso que sirve la llamada, y luego, de vuelta al llamador.

4.2. Remote Procedure Calls (RPC)

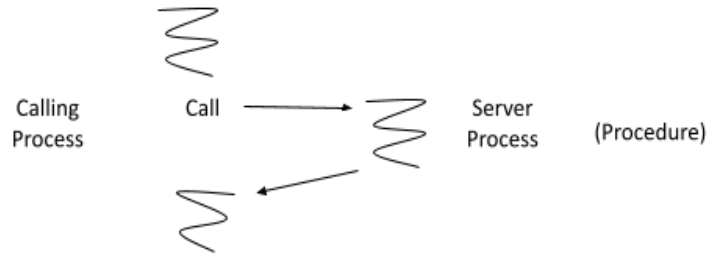
Los programas se descomponen en módulos (con procesos y procedimientos). Los procesos, de un módulo, son llamados ***background*** para diferenciarlos de los procesos exportados (***servers***). El siguiente ejemplo, muestra de forma trivial la especificación de un módulo.

```
Module module_name

op opname(params)[returns type];        # (1) Interfaz public

Body

int x;                                  # (2) Definicion de variables locales
x = 0;                                  # e inicializacion

proc opname(params)[returns type]       # (3) Definicion de procedimientos exportados
begin
  # Cuerpo del procedimiento.
end;

proc local_proc(params)[returns type]   # (4) Definicion de procedimientos locales
begin
  # Cuerpo del procedimiento.
end;

Process Main                            # (5) Definicion de procesos locales
begin
  # Cuerpo del proceso bakground
end;
End module_name;
```

Los procesos background tienen acceso a las variables y pueden invocar procedimientos del mismo módulo. A su vez, como sucede en monitores, pueden invocar procedimientos definidos en otros módulos. Siempre y cuando, estos últimos, sean exportados por dicho módulo.

La ejecución de una llama intermódulo, difiere con respecto a una local.

- Un nuevo proceso atiende la llamada y los argumentos son pasados como mensajes entre el llamador y el proceso servidor.
- El proceso llamador que demorado mientras se ejecuta el cuerpo del procedimiento invocado.
- Cuando el proceso servidor finaliza su ejecución del procedimiento invocado, retorna los resultados y finaliza.
- El proceso llamador, continua su ejecución.



4.2.1. Sincronización

Con RPC, la ***sincronización entre proceso llamador y servidor está implícita***. No obstante, necesitamos mecanismos para que procesos server y background sincronicen dentro de un módulo. Existen 2 (dos) enfoques para proveer sincronización dentro de un módulo.

A lo sumo, un proceso activo
Apropidado para entornos monoprocesador ya que, reduce el context switch.
Mutex → Implícito.
Condición → Explícito (semáforos/monitores).

Ejecución concurrente
Apropidado, de forma natural, para entornos multriprocesador.
Mutex → Explícito (semáforos/monitores).
Condición → Explícito (semáforos/monitores).

4.2.2. Time Server

El siguiente módulo es una adaptación del ejemplo del libro.

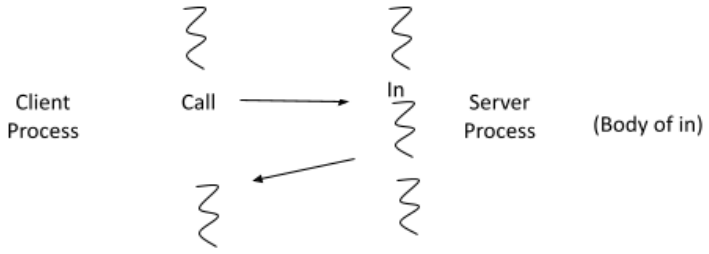
```
1  module TimeServer
2
3  op get_time(): int;
4  op delay(int interval);
5
6  body
7
8  int time = 0;                    # Tiempo inicial del TimeServer (en ms).
9  Queue delayed(waketime, id);     # Cola de procesos dormidos, por la operacion delay.
10 Sem m = 1;                       # Semaforo para garantizar mutex, en el acceso a delayed.
11 Sem d = ([n] 0);                 # Arreglo de semaforos para demorar procesos.
12
13 proc get_time(): int             # Retorna el tiempo actual del servidor en ms.
14   return time;
15 end;
16
17 proc delay(int interval)         # Demora un proceso tantos ms, como interval lo indique.
18   P(m);
19   waketime = time + interval;
20   delayed.push(waketime, id);
21   V(m);
22   P(d[id]);
23 end;
24
25 process Clock                    # Proceso background. Mantiene actualizada la variable time
26   while (true)                 # y despierta procesos demorados por la operacion delay.
27     time++;
28     P(m);
29     while (time >= menor waketime en delayed)
30       delayed.pop(waketime, id);
31       V(d[id]);
32     end;
33     V(m);
34   end;
35 end;
36 End.
```

4.3. Rendezvous

Al igual que RPC, el programa se descompone en módulos, se brindan los mecanismos para la ***comunicación intermódulo*** y se demoran a los procesos llamadores.

La cabecera de un módulo, debe especificar las operaciones exportadas. Mientras que el cuerpo, del módulo, contendrá ***un único proceso***; que atenderá las llamadas a las operaciones en lugar de crear un nuevo proceso servidor.

Dicho proceso, esperara por una call y actuar. Las llamadas, a operaciones, serán atendidas de a una por vez aunque los procesos se ejecuten concurrentemente. Esto ***provee sincronización dentro del módulo***.



4.3.1. Sentencias Input

Suponiendo que, un módulo exporta la siguiente operación: `op opname(formals)`. El proceso servidor realiza rendezvous con un llamador de `opname`, ejecutando una sentencia ***input***. Dicha sentencia, respeta la siguiente forma:

```
in opname(formals) -> S; ni
```

Las partes entre `in` y `ni` son conocidas como ***operaciones guardadas***. La guarda nombra las operaciones y provee sus parámetros. En cuanto a `S`, es un conjunto de sentencias a ejecutar en la invocación de la operación. Los parámetros, de la operación, tienen alcance de a toda la guarda; por consiguente `S` podrá leer y escribir sus valores.

Una sentencia `input`, demora al proceso servidor hasta que haya al menos una llamada pendiente a `opname`. Entonces, elige la llamada pendiente más antigua, copia los argumentos en los parámetros formales, ejecuta `S` y finalmente, retorna los resultados al proceso llamador.

Es posible ofrecer ***comunicación guardada***, en una sentencia `input` de la siguiente forma:

```
In opname1 (formals) and B1 by E1 -> S1
[] opname2 (formals) and B2 by E2 -> S2
Ni
```

Donde ***E*** supone una ***condicion booleana de sincronización***. Mientras que, ***E*** una ***expresión de scheduling***. Ambas expresiones son opcionales.

4.3.2. Buffer limitado

```
Module Buffer

op depositar(int);
op retirar(int);

Body
Process Buffer
  Queue buffer;
  int cant = 0;
  int cantMax = n;

  while(true)
    in depositar(x) and (cant < cantMax) ->
      buffer.push(x);
      cant++;
    [] retirar(x) and (cat > 0) ->
      buffer.pop(x);
      cant--
    ni
  end;
end;
End.
```

4.3.3. ADA

Un programa, escrito en ADA, está compuesto por subprogramas, paquetes y tareas. Donde ***rendezvous*** es el principal mecanismo de comunicación y sincronización.

Las tareas (***tasks***), son procesos independientes que cuentan con una especificación y un cuerpo (***body***). En la especificación, se definen las operaciones visibles por otros procesos (***entry***). El cuerpo, contiene variables locales (junto con su inicialización) y sentencias de atención (***entry calls***).

```
Procedure nombreProceso

Task nombreTarea Is
  # Declaracion de entries
end;

Task Body nombreTarea Is
  # Declaracion de locales e inicializacion
Begin
  # Sentencias ...
end;

End nombreProceso;
```

Un programa en ADA, atiende solicitudes a procedimientos mediante la sentencia `accept`. La cual equivale a `input` pero, su ejecución es determinística y no posee guardas. Dicha sentencia, posee la siguiente forma:

```
Accept E (formals) do
  # Sentencias
end;
```

De ser necesario, comunicación guardada con un comportamiento no determinístico. ADA ofrece la sentencia `select`. No obstante, `select` resulta menos poderosa que la notación formal establecida en rendezvous; por 2 (dos) motivos.

1. No soporta expresiones de scheduling.
2. La expresión de sincronización, no puede referenciar a los parámetros formales de la operación.

4.3.4. Buffer limitado con ADA

```
Procedure nombreProceso

Task Mailbox Is
  Entry Depositar (msg: IN string)
  Entry Retirar (msg: OUT string)
End;

Task Body Mailbox Is
  Array datos[n];
  Int count, first, last = 0;
Begin
  Loop
    Select
      When (count < n) -> Accept Depositar (msg: IN string)
        # Sentencias de la operacion Depositar
      End Depositar;
      Or
      When (count > 0) -> Accept Retirar (msg: OUT string)
        # Sentencias de la operacion Retirar
      End Retirar;
    End;
  End;
End;
End nombreProceso;
```

4.4. Notaciones primitivas

Mecanismo que ***combina RPC, Rendezvous y PMA en un paquete coherente***. Con este mecanismo, será posible implementar una solución óptima al problema de filtros y pares que interactúan mediante

el uso de las técnicas de RPC y Rendezvous.

Los programas, son colecciones de módulos que exportan operaciones visibles para otros módulos o procesos.

Una operación puede ser invocada por call (sincrónico) o por send (asincrónico). Como también una operación puede ser servida por RPC o Rendezvous (según la elección del programador del módulo).

Invocación	**Servicio**	**Efecto**
call	proc	Llamado a procedimiento.
call	in	Rendezvous.
send	proc	Creación dinámica de un proceso.
send	in	PMA.

Capítulo 5

Paradigmas de intercacción entre procesos

5.1. Introducción

Existen 3 (tres) esquemas básicos de interacción entre procesos: productores/consumidores, cliente/servidor y pares que interactúan. Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a nuevos modelos o paradigmas de interacción entre procesos.

5.2. Manager and workers

Es la implementación distribuida del modelo ***bag of tasks***. Diversos procesos workers comparten una bolsa de tareas. Cada worker repetidamente toma una tarea de la bolsa, la ejecuta y posiblemente genere una (o más) tareas nuevas para depositar en la bolsa.

La principal ventaja de este paradigma, en cuanto a cómputo paralelo, es la facilidad para escalar y balancear la carga de trabajo de cada worker.

Por el otro lado un proceso manager representa la bolsa de tareas. Es decir, mantiene la lista de tareas, recolecta los resultados y detecta la finalización de todas las tareas.

En escencia, un manager funciona como un server. Mientras que, un worker como un cliente.

Caso de uso: multiplicación de matrices.

5.3. Heartbeat

Los procesos, periódicamente, deben intercambiar información con mecanismos de tipo send/receive.

En particular, puede ser utilizado cuando los datos están divididos entre los procesos workers. Donde, cada worker es responsable de actualizar una parte de los datos. Dicha actualización, dependerá de los valores que contenga cada worker o sus vecinos inmediatos.

La lógica de este paradigma, tiene relación con su nombre. Los procesos se expanden enviando información a sus vecinos y se contraen recibiendo información actualizada desde sus vecinos.

La interacción entre vecinos, dada por la disposición de las primitivas send/receive, produce una especie de barrera de sincronización entre ellos. Impidiendo que, cada worker inicie su fase de actualización antes de que su vecino no finalice la suya.

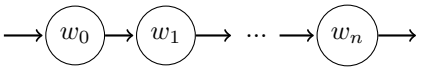
Caso de uso: paralelizar soluciones iterativas → topología de una red, procesamiento de imágenes.

5.4. Pipeline

Los procesos, funcionan como filtros y se encuentran conectados entre sí; formando una colección de filtros. De esta manera, la información fluye por los procesos utilizando alguna forma de receive/send. Existen 3 (tres) formas de conectar los procesos:

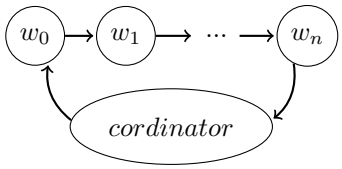
Pipeline Abierto

La fuente de entrada y salida, del pipeline, no está especificada.



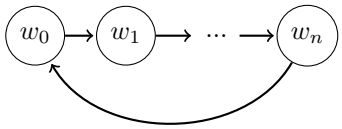
Pipeline Cerrado

Pipeline abierto conectado por un coordinador. El coordinador, produce los datos de entrada para w_0 y consume la salida de w_n.



Pipeline Circular

La salida de w_n está conectada a la entrada de w_0. Útil cuando la solución, no puede obtenerse en una sola pasada.



5.5. Probes and echoes

La interacción entre procesos permite recorrer estructuras dinámicas, diseminando y recolectando información.

La estructura de muchos programas distribuidos, puede verse como árboles o gráfos. En el cual, los procesos son nodos y los canales aristas.

Una primitiva ***probe*** es un mensaje enviado por un nodo, a todos sus sucesores. Mientras que, una primitiva ***echo*** es su respectiva respuesta. Este paradigma, es el análogo al algoritmo Deep Search First (DFS).

Caso de uso: Cuando se desea recorrer una red, de la cual no se conoce la cantidad de nodos → redes móviles.

5.6. Broadcasts

Los procesos suelen enviar mensajes a todos los demás, de forma paralela, mediante una primitiva ***broadcast***.

Dicha primitiva, no se asume atómica, por lo cual los procesos pueden recibir los mensajes en distinto orden al cual fueron enviados. La primitiva ***receive*** funciona de manera tradicional, sobre el canal de cada proceso.

Caso de uso: Mensaje broadcast en redes LAN.

5.7. Token passing

Lo procesos se pasan, entre sí, un token que garantiza el acceso exclusivo a un recurso compartido.

Caso de uso: Exclusión mutua en ambientes distribuidos → Exclusión mutua con toke ring.

5.7.1. Exclusión mutua con toke ring

Existe un proceso *helper* por cada proceso que quiera solicitar acceso a una SC. Los procesos *helper*, esperan por una solicitud de acceso (*entryRequest*).

En caso de tener solicitudes pendientes (y poseer el token), concederán el acceso al proceso correspondiente (*grant*). Luego, esperaran a recibir una solicitud de salida (*exitRequest*).

Una vez finalizada la atención, pasarán el token al proceso *helper* siguiente.

```
# Exclusion mutua con toke ring (PMA).
Chan token[n];
Chan entryRequest[n];
Chan exitRequest[n];
Chan grant[n];

Process Worker[i=1..n]
while(true)
  send entryRequest[i]();
  receive grant[i]();
  # Seccion Critica ...
  send exitRequest[i];
end;
end;

Process Helper[i=1..n]
while(true)
  receive token[i]();
  if (!empty(entryRequest[i]))
    receive entryRequest[i]();
    send grant[i]();
    receive exitRequest[i]();
  end;
  send token[(i mod n) + 1]();
end;
end;
```

5.8. Servidores replicados

Los procesos servidores gestionan el acceso a un recurso. Al existir n instancias del recurso, se replicará un servidor por cada una. También, puede utilizarse para abstraer a los clientes la cantidad real de recursos.

Caso de uso: En el problema de los filósofos, estos le solicitan sus cubiertos (recursos) a un mozo (servidor). Los filósofos, desconocen la existencia de cubiertos ajenos.

IV
Programación Paralela
Clase 10: Introducción a la Programación Paralela

Ramiro Martínez D'Elía

2021

Índice general

Capítulo 1

Conceptos

1.1. Programa paralelo

Tipo de programa, concurrente, escrito para resolver un problema en menos tiempo que su análogo secuencial. El objetivo principal es reducir el tiempo de ejecución, o resolver problemas más grandes o con mayor precisión en el mismo tiempo.

Un programa paralelo puede escribirse usando variables compartidas o pasaje de mensajes. La elección la dicta el tipo de arquitectura.

Para determinar si una solución paralela tiene mejores prestaciones que su análogo secuencial, existen nociones como: ***speedup***, ***eficiencia***, ***escalabilidad***, y ***ley de Amndhal***.

1.2. Speedup

La métrica de speedup (o eficiencia) **evalúa el tiempo total de ejecución** de un programa. El speedup está dado por la siguiente fórmula:

$$S = Speedup = T_1/T_p$$

Donde T_1 es el tiempo de una solución secuencial ejecutando en 1 CPU y T_p es el tiempo de una solución paralela con p CPUs.

Del resultado obtenido, podemos concluir las siguientes afirmaciones:

- Si $S = P \rightarrow$ speedup ***lineal*** (o ***perfecto***).
- Si $S < P \rightarrow$ speedup ***sublineal***
- Si $S > P \rightarrow$ speedup ***superlineal***.

Por ejemplo, si el tiempo de una solución secuencial es de 600 segundos y el de una solución paralela es de 60 segundos utilizando 10 CPU.

$$Speedup = T_1/T_p = 600/60 = 10$$

Como $S = P \rightarrow$ speedup lineal

1.3. Eficiencia

Esta métrica permite conocer **qué tan bien aprovechará procesadores extras** un programa paralelo. La eficiencia, está dada por la siguiente fórmula:

$$E = Eficiencia = Speedup/p$$

Donde p es la cantididad de CPUs. Del resultado obtenido, podemos concluir las siguientes afirmaciones:

- Con speedup perfecto $\rightarrow E = 1$.
- Con speedup sublineal $\rightarrow E < 1$.
- Con speedup superlineal $\rightarrow E > 1$.

Por ejemplo, si el tiempo de una solución secuencial es de 600 segundos y el de una solución paralela es de 60 segundos utilizando 10 CPU.

$$Speedup = T_1/T_p = 600/60 = 10$$

$$Eficiencia = Speedup/P = 10/10 = 1$$

Como el speedup obtenido fue lineal ($S = P$), entonces se cumplió que $E = 1$.

1.4. Escalabildiad

Las métricas de **speedup** y **eficiencia**, son **relativas**. Ellas dependen del número de procesadores, el tamaño de los datos y el algoritmo utilizado. Por esto, de forma general, se dice que **un programa paralelo es escalable si; su eficiencia se mantienen constante para un rango amplio de CPU**.

Por ejemplo: una solución es paralelizada sobre P procesadores de dos maneras. En la primera, el speedup está regido por la función $S = P - 5$. Mientras que, en la otra $S = P/2$. ¿Qué solución se comportará más eficientemente al crecer P?

P	**S = P - 5**	**S = P / 2**
6	$(6-5)/6 = 1,66$	$(6/2)/6 = 0,5$
10	$(10-5)/10 = 0,5$	$(10/2)/10 = 0,5$
20	$(20-5)/20 = 0,75$	$(10/2)/20 = 0,5$

La solución regida por el speedup $S = P/2$ es la más escalable. Ya que, el valor de E se mantiene constante al incrementar el número de procesadores.

1.5. Ley de Amdhal

m Un programa típico consta de 3 (tres) etapas; entrada de datos, cómputo y salida de resultados.

La ley de Amdhal postula que; para todo algoritmo **existe un speedup máximo alcanzable**, independiente del número de procesadores. Ese valor dependerá de la cantiadd de código paralelizable.

$$Limite = 1/1 - Paralelizable$$

Por ejemplo: suponga un programa secuencial donde las etapas (1) y (2) consumen cada una el 10 % del tiempo de ejecución y no pueden ser paralelizadas. La etapa (3) consume el 80 % restante. Si el programa tarda 100 unidades de tiempo (ut), ¿cuál es el limite de mejora alcanzable?

$$Limite = 1/1 - 0,8 = 1/0,2 = 5$$

El mejor speedup alcanzable es 5. Esto quiere decir que; el tiempo de ejecución, del algoritmo paralelizado, será como máximo hsta 5 veces más rápido ($100/5 = 20ut$)

1.6. Granularidad

Cuando el número de procesadores crece, la carga de cómputo en cada unidad tiende a decrecer y las comunicaciones aumentan. Esta relación se conoce como **granularidad**.

La granularidad de una aplicación o máquina paralela está dada por la relación entre el cómputo promedio y la comunicación promedio que realice.

Si la granularidad del algoritmo difiere a la de la arquitectura, normalmente se tendrá pérdida de performance.

	Grano grueso	**Grano fino**
Arquitectura	Pocos CPU, con mayor capacidad de cómputo.	Muchos CPU, con menor capacidad de cómputo.
Aplicación	Pocos procesos, con responsabilidades más amplias. Se espera, menor comunicación entre ellos.	Muchos procesos, con responsabilidades más chicas. Se espera, mayor comunicación entre ellos.

Programación Concurrente

Cuestionario

Ramiro Martínez D'Elía

2021

1. Defina programa concurrente, programa paralelo y programa distribuido.

Apunte I: sección 1.1, Apunte II: sección 1.1 y Apunte IV: 1.1

2. Responda

***a*) ¿A qué se denomina propiedad del programa? ¿Qué son las propiedades de seguridad y vida? Ejemplificar.**

***b*) Defina fairness. Relacionar dicho concepto con las políticas de scheduling.**

***c*) Describa los distintos tipos de fairness**

***d*) ¿Cuáles son las propiedades que debe cumplir un protocolo de E/S a una sección crítica?**

***e*) Cuáles son los defectos que presenta la sincronización por busy waiting? Diferencie esta situación respecto de los semáforos.**

***f*) Explique la semántica de la instrucción de grano grueso AWAIT y su relación con las instrucciones Test & Set o Fetch & Add.**

a) Apunte I: sección 1.7

b) Apunte I: sección 1.8

c) Apunte I: sección 1.8

d) Apunte II: sección 1.2

e) Apunte II: sección 2.3

f) Apunte I: sección 1.6

3. Definir el problema general de alocación de recursos y su resolución mediante una política SJN. ¿Minimiza el tiempo promedio de espera? ¿Es fair? Si no lo es, plantee una alternativa que lo sea.

Apunte II: sección 3.5.2

4. ¿En qué consiste la técnica de passing the baton? Aplicar este concepto a la resolución del problema de lectores y escritores.

Apunte II: secciones 3.5 y 3.5.1

5. Explique el concepto de broadcast y sus dificultades de implementación en un ambiente distribuido, con pasaje de mensajes sincrónico y asincrónico.

Sobre primitiva broadcast: Apunte III sección 2.1.4

En un ambiente con memoria compartida, este tipo de primitivas es más sencilla de implementar.

En ambientes distribuidos, al no haber variables compartidas, se debe utilizar pasaje de mensajes. Esto, supone ciertas dificultades para cada caso.

Con pasaje de mensajes asincrónicos (PMA), podemos plantear los siguientes escenarios de implementación:

Utilizar un único canal

El emisor deberá enviar tantos mensajes, como receptores existan. Este escenario evita demoras innecesarias, en el envío de mensajes, ya que `send` no es bloqueante.

Un canal por proceso

El emisor deberá enviar el mensaje por tantos canales, como receptores existan. Con este escenario, tenemos demoras innecesarias: un proceso listo debe esperar a que el emisor itere sobre su canal.

Con pasaje de mensajes sincrínicos (PMS), los canales son punto a punto. Por este motivo, solo podemos llevar a cabo la implementación "un canal por proceso" planteada en PMA. Aunque, en este caso, resulte aún más ineficiente ya que `send` causa demora.

6. Responder sobre exclusión mutua selectiva

***a*) ¿Por qué el problema de los filósofos es de exclusión mutua selectiva? Si en lugar de 5 filósofos fueran 3, ¿el problema seguiría siendo de exclusión mutua selectiva? ¿Por qué?**

***b*) El problema de los filósofos resuelto de forma centralizada y sin posiciones fijas ¿es de exclusión mutua selectiva? ¿Por qué?**

***c*) El problema de los lectores-escritores es de exclusión mutua selectiva? ¿Porque?**

***d*) Si en el problema de los lectores-escritores se acepta sólo 1 escritor o 1 lector en la BD, ¿tenemos un problema de exclusión mutua selectiva? ¿Por qué?**

a) Apunte II: sección 3.4.1

b) Apunte II: sección 3.4.1

c) Apunte II: sección 3.4.2

d) Apunte II: sección 3.4.2

7. Sea el problema en el cual N procesos poseen inicialmente cada uno un valor V, y el objetivo es que todos conozcan cual es el máximo y cuál es el mínimo de todos los valores.

***a*) Plantee conceptualmente posibles soluciones con las siguientes arquitecturas de red: Centralizada, simétrica y anillo circular. No implementar.**

***b*) Analice las soluciones anteriores desde el punto de vista del número de mensajes y la performance global del sistema.**

a) Apunte III: sección 2.1

b) Apunte III: sección 2.1.4

8. Defina el concepto de granularidad. ¿Qué relación existe entre la granularidad de programas y de procesadores?

Apunte IV: sección 1.6

9. ¿En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad?

Apunte III: sección 3.1

10. ¿Cuál es la utilidad de la técnica de passing the baton? ¿Qué relación encuentra con la técnica de passing the condition?

Apunte II: sección 3.5

11. Explique sintéticamente los 7 paradigmas de interacción entre procesos en programación distribuida. Ejemplifique.

Apunte III: capítulo 5

12. Responder sobre programación paralela

***a*) ¿Cuál es el objetivo de la programación paralela?**

***b*) Defina las métricas de speedup y eficiencia. ¿Cuál es el significado de cada una de ellas (que miden)? ¿Cuál es el rango de valores para cada una?**

***c*) ¿En qué consiste la ley de Amdahl?**

a) Apunte IV: sección 1.1

b) Apunte IV: sección 1.2 y sección 1.3

c) Apunte IV: sección 1.5

13. Suponga que el tiempo de ejecución de un algoritmo secuencial es de 1000 unidades de tiempo, de las cuales el 80 % corresponden a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo?

Apunte IV: sección 1.5 (Ley de Amdahl) está resuelto un ejercicio similar.

14. ¿Qué significa el problema de interferencia en un programa concurrente? ¿Cómo puede evitarse?

Apunte I: sección 1.4

15. Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función S=p-1 y en el otro por la función S=p/2. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

Apunte I: sección 1.4 (Escalabilidad). Hay un ejercicio similar resuelto.

16. Sea el problema de alocacion Shortest Job Next

***a*) ¿Funciona correctamente con disciplina de señalización Signal and continue? Justifique.**

***b*) ¿Funciona correctamente con disciplina de señalización signal and wait? Justifique.**

a) Con esta disciplina, todo proceso señalizado es enviado a una cola de listos; donde competirá contra el resto por el acceso al recurso. En cuyo caso, podría perder contra otro proceso y la meta de SJN no se estaría concretando.

b) Con esta disciplina, todo proceso señalizado retoma su ejecución de inmediato. Por tal motivo, esta disciplina es la más adecudada para la implementación.

Nota$_1$: El enunaciado incluye el mismo algoritmo que el del apunte de monitores.

Nota$_2$: Para el LJN (Lowest Job Next) aplican las mismas respuestas.

17. En los protocolos de acceso a sección crítica vistos en clase, cada proceso ejecuta el mismo algoritmo. Una manera alternativa de resolver el problema es usando un proceso coordinador. En este caso, cuando cada proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le de permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador. Desarrolle protocolos para los procesos SC[i] y el coordinador usando sólo variables compartidas (no tenga en cuenta la propiedad de eventual entrada).

```
1 int request[n] = ([n] 0);
2 int grant[n] = ([n] 0);
```

```
Process Worker[i = 1..n]

# Seccion no critica ...

request[i] = 1;
while(grant[i] == 0) skip;
# Seccion Critica
grant[i] = 0;

# Seccion no critica ...
end;
```

```
Process Coordinator
while (true)
  for (j = 1; j < n; j++)
    if (request[j] == 1)
      request[j] = 0;
      grant[j] = 1;
      while (grant[j] == 1) skip;
    end;
  end;
end;
end;
```

18. Dado el siguiente bloque de código, indique para cada inciso qué valor queda en aux, o si el código queda bloqueado. Justifique su respuesta.

```
aux = -1;
...
if (A == 0); P2?(aux) -> aux = aux + 2; # (1)
[] (A == 1); P3?(aux) -> aux = aux + 5; # (2)
[] (B == 0); P3?(aux) -> aux = aux + 7; # (3)
fi
```

***a*) Si el valor de A = 1 y B = 2 antes del if, y solo P2 envia el valor 6.**

***b*) Si el valor de A = 0 y B = 2 antes del if, y solo P2 envia el valor 8.**

***c*) Si el valor de A = 2 y B = 0 antes del if, y solo P3 envia el valor 6.**

***d*) Si el valor de A = 2 y B = 1 antes del if, y solo P3 envia el valor 9.**

***e*) Si el valor de A = 1 y B = 0 antes del if, y solo P3 envia el valor 14.**

***f*) Si el valor de A = 0 y B = 0 antes del if, P3 envia el valor 9 y P2 el valor 5.**

a) La guarda 1 queda bloqueada; su condición booleana es verdadera pero, su sentencia de comunicación generá bloqueo.

b) Solo la guarda 1 tiene éxito. Su sentencia guardada es ejecutada. El programa finaliza con `aux = 10`.

c) Solo la guarda 3 tiene éxito. Su sentencia guardada es ejecutada. El programa finaliza con `aux = 13`.

d) Todas las guardas fallan porque sus condiciones booleanas se evalúan como falsas. El programa finaliza con `aux = -1`.

e) Las guardas 2 y 3, tienen éxito. Se elegirá una de las dos, de forma no deterministica y se ejecutará su sentencia guardada. El programa finalizará con `aux = 19 o 21`.

f) Las guardas 1 y 3, tienen éxito. Se elegirá una de las dos, de forma no deterministica y se ejecutará su sentencia guardada. El programa finalizará con `aux = 16 o 7`.

19. Dado el siguiente programa, indique si es posible que finalice.

```
bool continue = true;
bool try = false;
co  while (continue) { try = true; try = false; } #(P)
/   <await(try) continue = false> #(Q)
oc
```

Con una política débilmente fair, el programa podría no terminar. Ya que, `try` no se mantiene verdadera hasta ser vista por (Q).

Con una política fuertemente fair, el programa podría terminar. Ya que, `try` se convierte en verdadera con infinita frecuencia. Ante este escenario, eventualmente, (Q) verá a `try` como vedadera.

20. Suponga que n^2 procesos organizados en forma de grilla cuadrada. Cada proceso puede comunicarse solo con los vecinos izquierdo, derecho, de arriba y de abajo (los procesos de las esquinas tienen solo 2 vecinos, y los otros en los bordes de la grilla tienen 3 vecinos). Cada proceso tiene inicialmente un valor local v.

***a*) Escriba un algoritmo heartbeat que calcule el máximo y el mínimo de los n^2 valores. Al terminar el programa, cada proceso debe conocer ambos valores.**

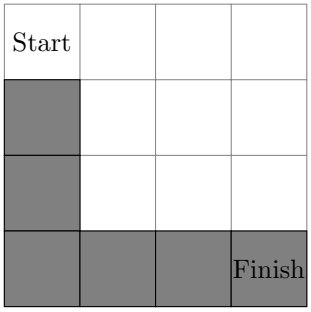
***b*) Analice la solución desde el punto de vista del número de mensajes.**

***c*) ¿Puede realizar alguna mejora para reducir el número de mensajes?**

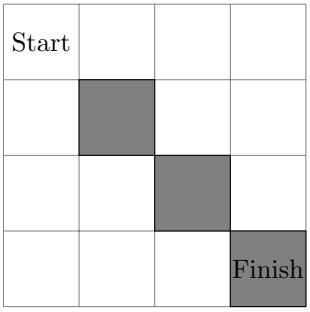
***d*) Indique que mecanismo de pasaje de mensajes utilizaría. Justifique.**

a) La idea del algoritmo es que, cada proceso pueda propagar su valor hasta el proceso más lejano. La mayor distancia entre procesos, sin diagonales, está dada por $2(n-1)$.

Sin diagonal y n=4



Con diagonal y n=4




```
1  chan valores[1:n,1:n](int);
2
3  Process P[i=1..n;j=1..n]
4
5  int v, nuevo;
6  int min = v;
7  int max = v;
8  int rondas = 2(n-1);
9
10 for(k=1; k <= rondas; k++)
11
12   foreach(vecinos => (x,y))
13     send valores[x,y](v);
14
15   foreach(vecinos => (x,y))
16     receive valores[i,j](nuevo);
17     if (nuevo < min) min = nuevo;
18     if (nuevo > max) max = nuevo;
19   end;
20 end;
21 end;
```

b) Desde el punto de vista de los mensajes, podemos obtener las siguientes conclusiones.

Tipo	Cantidad	Vecinos	Mensajes
Esquina	4	2	$cantidad * vecinos * rondas$
Borde	$4(n-2)$	3	$cantidad * vecinos * rondas$
Internos	$(n-2)^2$	4	$cantidad * vecinos * rondas$

c) Podemos hacer que cada proceso, también, se comunique con sus diagonales. De esta forma; los procesos esquina tendrán 3 vecinos, los bordes 5 vecinos y el resto 8 vecinos.

Esto incrementará el número de mensajes enviados por cada proceso pero, también reducirá en la mitad la cantidad de rondas $(n-1)$. Generando una reducción en la cantidad final de mensajes.

Tipo	Cantidad	Vecinos
Esquina	4	3
Borde	$4(n-2)$	5
Internos	$(n-2)^2$	8

n	Sin Diagonales	Con Diagonales
4	288	252
6	1200	1100
8	3136	2940

d) En mi opinión, utilizaría PMA. Particularmente, porque su primitiva `send` no es bloqueante. Esto nos evita demoras innecesarias al momento de propagar el valor de un proceso, en particular con n muy grandes.

21. Suponga los siguientes programas concurrentes. Asuma que EOS es un valor especial que indica el fin de la secuencia de mensajes y que los procesos son iniciados desde el programa principal.

P1

```
chan canal (double)
process Genera {
  int fila, col;  double sum;
  for [fila= 1 to 10000]
    for [col = 1 to 10000]
      send canal (a(fila,col));
  send canal (EOS)  }
```

```
process Acumula {
  double valor, sumT;
  sumT=0;
  receive canal (valor);
  while valor<>EOS {
    sumT = sumT + valor
    receive canal (valor); }
  printf (sumT);
}
```

P2

```
chan canal (double)
process Genera {
  int fila, col;  double sum;
  for [fila= 1 to 10000] {
    sum=0;
    for [col = 1 to 10000]
      sum=sum+a(fila,col);
    send canal (sum);  }
  send canal (EOS)  }
```

```
process Acumula {
  double valor, sumT;
  sumT=0;
  receive canal (valor);
  while valor<>EOS {
    sumT = sumT + valor
    receive canal (valor); }
  printf (sumT);
}
```

***a*) ¿Qué hacen los programas?**

***b*) Analice desde el punto de vista del número de mensajes.**

***c*) Analice desde el punto de vista de la granularidad de procesos.**

***d*) ¿Cuál de los programas le parece el más adecuado para ejecutar sobre una arquitectura tipo cluster PCs? Justifique.**

a) Ambos programas realizan la suma de los elementos de una matriz. Aunque, difieren en el modo en que lo hacen.

En el `Programa 1`: el proceso `Genera` envía todos los valores al proceso `Acumula`. El proceso `Acumula` suma los valores que recibe desde `Genera`.

En el `Programa 2`: el proceso `Genera` realiza la suma de los elementos de una fila y envia el resultado a `Acumula`. El proceso `Acumula`, recibe los resultados parciales y los adiciona en su acumulador.

b) En el `Programa 1`, el proceso `Genera` envía todos los elementos de la matriz. Por lo cual, en esta solución, se están enviando $n * n = n^2$ mensajes.

En el `Programa 2`, el proceso `Genera` envía los valores acumulados de cada fila. Por lo cual, en esta solución, se están enviando n mensajes.

c) El programa `P2` tiene mayor granularidad, con respecto a `P1`. Esto se debe a que, requiere menor comunicación (menos mensajes) para completar la tarea.

d) Las arquitecturas cluster PC pueden ser catalogadas como grano grueso. Esto es, porque se componen de mucho procesadores; otorgando mayor potencia para cómputo. No obstante, tienen una menor performance en cuanto a comunicación.

Es deseable que la granularidad de la aplicación, se ajuste a la de la arquitectura. Para hacer un uso apropiado de los recursos y evitar rendimientos menores.

En este escenario, el programa, `P1` parece el más adecuado para ejecutar en la arquitectura planteada.

Programación Concurrente

Preguntas de final

Ramiro Martínez D'Elía

2021

1. **Defina programa concurrente, programa paralelo y programa distribuido.**

   Apunte I: sección 1.1, Apunte II: sección 1.1 y Apunte IV: 1.1

2. **Responda**

   a) **¿A qué se denomina propiedad del programa? ¿Qué son las propiedades de seguridad y vida? Ejemplificar.**

      Apunte I: sección 1.1, Apunte II: sección 1.1 y Apunte IV: 1.1

   b) **Defina fairness. Relacionar dicho concepto con las políticas de scheduling.**

      Apunte I: sección 1.7

   c) **Describa los distintos tipos de fairness**

      Apunte I: sección 1.8

   d) **¿Cuáles son las propiedades que debe cumplir un protocolo de E/S a una sección crítica?**

      Apunte II: sección 1.2

   e) **Cuáles son los defectos que presenta la sincronización por busy waiting? Diferencie esta situación respecto de los semáforos.**

      Apunte II: sección 2.3

   f) **Explique la semántica de la instrucción de grano grueso AWAIT y su relación con las instrucciones Test & Set o Fetch & Add.**

      Apunte I: sección 1.6

3. **Definir el problema general de alocación de recursos y su resolución mediante una política SJN. ¿Minimiza el tiempo promedio de espera? ¿Es fair? Si no lo es, plantee una alternativa que lo sea.**

   Apunte II: sección 3.5.2

4. **¿En qué consiste la técnica de passing the baton? Aplicar este concepto a la resolución del problema de lectores y escritores.**

   Apunte II: sección 3.5 y 3.5.1

5. **Explique el concepto de broadcast y sus dificultades de implementación en un ambiente distribuido, con pasaje de mensajes sincrónico y asincrónico.**

   Sobre primitiva broadcast → Apunte III: sección 2.1.4

   En un ambiente con memoria compartida, este tipo de primitivas es más sencilla de implementar.

   En ambientes distribuidos, al no haber variables compartidas, se debe utilizar pasaje de mensajes. Esto, supone ciertas dificultades para cada caso.

   Con pasaje de mensajes asincrónicos (PMA), podemos plantear los siguientes escenarios de implementación:

   Utilizar un único canal

El emisor deberá enviar tantos mensajes, como receptores existan. Este escenario evita demoras innecesarias, en el envío de mensajes, ya que `send` no es bloqueante.

Un canal por proceso

El emisor deberá enviar el mensaje por tantos canales, como receptores existan. Con este escenario, tenemos demoras innecesarias: un proceso listo debe esperar a que el emisor itere sobre su canal.

Con pasaje de mensajes sincrónicos (PMS), los canales son punto a punto. Por este motivo, solo podemos llevar a cabo la implementación "un canal por proceso" planteada en PMA. Aunque, en este caso, resulte aún más ineficiente ya que `send` demora.

6. Sobre exclusión mutua selectiva

a) ¿Por qué el problema de los filósofos es de exclusión mutua selectiva? Si en lugar de 5 filósofos fueran 3, ¿el problema seguiría siendo de exclusión mutua selectiva? ¿Por qué?
Apunte II: sección 3.4.1

b) El problema de los filósofos resuelto de forma centralizada y sin posiciones fijas ¿es de exclusión mutua selectiva? ¿Por qué?
Apunte II: sección 3.4.1

c) El problema de los lectores-escritores es de exclusión mutua selectiva? ¿Porque?
Apunte II: sección 3.4.2

d) Si en el problema de los lectores-escritores se acepta sólo 1 escritor o 1 lector en la BD, ¿tenemos un problema de exclusión mutua selectiva? ¿Por qué?
Apunte II: sección 3.4.2

7. Sea el problema en el cual N procesos poseen inicialmente cada uno un valor V, y el objetivo es que todos conozcan cual es el máximo y cuál es el mínimo de todos los valores.

a) Plantee conceptualmente posibles soluciones con las siguientes arquitecturas de red: Centralizada, simétrica y anillo circular. No implementar.
Apunte III: sección 2.1

b) Analice las soluciones anteriores desde el punto de vista del número de mensajes y la performance global del sistema.
Apunte II: sección 2.1.4

8. Defina el concepto de granularidad. ¿Qué relación existe entre la granularidad de programas y de procesadores?

Apunte IV: sección 1.6

9. Dado el siguiente programa concurrente con memoria compartida, y suponiendo que todas las variables están inicializadas en 0 al empezar el programa, y que las instrucciones no son atómicas. Para cada una de las opciones indique verdadero o falso. En caso de ser verdadero indique el camino de ejecución para llegar a ese valor y de ser falso justifique claramente su respuesta.

```
1 P1::
2 If ( x == 0 ) then
3     Y = 4*x+2;
4     X = y+2+x;
```

```
1 P2::
2 If ( x > 0 ) then
3     X = X + 1;
```

```
1 P3::
2 X = x*8+x*2+1;
```

a) El valor de x al terminar el programa es 9.
Instrucciones no atómicas → los bloques IF pueden ser interrumpidos
Verdadero para la siguiente historia:
$P1_2 \rightarrow P2 \rightarrow P3(x = 1) \rightarrow P1_3(y = 2) \rightarrow P1_4(x = 9)$

b) El valor de x al terminar el programa es 6.
Verdadero para la siguiente historia:
$P1_2 \rightarrow P1_3(y = 2) \rightarrow P3(x = 1) \rightarrow P1_4(x = 5) \rightarrow P2(x = 6)$

c) El valor de x al terminar el programa es 11.
Falso: la única forma de que ocurra es que $x = 1$ cuando se ejecute P3. Esto no puede darse, ya que P2 nunca podría incrementar con $x = 0$ y P1 escribe valores mayores sobre x.

d) Y, siempre termina con alguno de los siguientes valores: 10 o 6 o 2 o 0.
- $y = 10$ con $P1_2 \rightarrow P3(x = 1) \rightarrow P2(x = 2) \rightarrow P1_3(y = 10) \rightarrow P1_4(x = 8)$
- $y = 6$ con $P1_2 \rightarrow P2 \rightarrow P3(x = 1) \rightarrow P1_3(y = 6) \rightarrow P1_4(x = 9)$
- $y = 2$ con cualquier historia que inicie con $P1_2 \rightarrow P1_3$. Una vez seteado y, ningún otra instrucción la modifica.
- $y = 0$ con cualquier historia que inicie con $P3$. Ya que $x = 1$ invalidará el if de P1.

10. ¿En qué consiste la comunicación guardada (introducida por CSP) y cuál es su utilidad?
Apunte III: sección 3.1

11. ¿Cuál es la utilidad de la técnica de passing the baton? ¿Qué relación encuentra con la técnica de passing the condition?
Apunte II: sección 3.5

12. Explique sintéticamente los 7 paradigmas de interacción entre procesos en programación distribuida. Ejemplifique.
Apunte III: capitulo 5

13. Responder sobre programación paralela

***a*) ¿Cuál es el objetivo de la programación paralela?**
Apunte IV: sección 1.1

***b*) Defina las métricas de speedup y eficiencia. ¿Cuál es el significado de cada una de ellas (que miden)? ¿Cuál es el rango de valores para cada una?**
Apunte IV: sección 1.2 y sección 1.3

***c*) ¿En qué consiste la ley de Amdahl?**
Apunte IV: sección 1.5

14. Sea el problema de ordenar de menor a mayor un arreglo de A[1..n]

***a*) Escriba un programa donde dos procesos (cada uno con n/2 valores) realicen la operación en paralelo mediante una serie de intercambios.**
En la siguiente implementación, el proceso P1 se queda con los números más chicos en su porción del arreglo. Mientras que, el proceso P2 con los números más grandes.

```
1  Process P1
2  const mayor = n/2;
3  int nuevo,
4      a1[1:n/2];
5
6  # ordenar a1 de forma ascendente
7
8  P2 ! (a1[mayor]); # (1)
9  P2 ? (nuevo);
10
11 do (true) a1[mayor] > nuevo ->
12     # reordenar a1:
13     # decartando a1[mayor]
14     P2 ! (a1[mayor]);
15     P2 ? (nuevo)
16 od
17 end;
```

```
18 Process P2
19 const menor = 1;
20 int nuevo,
21     a1[1:n/2];
22
23 # ordenar a1 de forma ascendente
24
25 P1 ? (nuevo);
26 P1 ! (a1[menor]); # (2)
27
28 do (true) a1[menor] < nuevo ->
29     # reordenar a1:
30     # descartando a1[menor]
31     P1 ? (nuevo);
32     P1 ! (a2[menor]);
33 od
34 end;
```

***b*) ¿Cuántos mensajes intercambian en el mejor de los casos? ¿Y en el peor de los casos?**
Mejor de los casos $\rightarrow$ 2 mensajes totales (uno cada proceso).
La lista ya se encuentra ordenada. Solo se envían los mensajes (1) y (2). Los valores enviados no satifacen la guarda del do, causando su finalización.
Peor de los casos $\rightarrow n$ mensajes totales.
La lista está ordenada de mayor a menor, todos los valores son intercambiados ($n/2$ cada proceso).

c) Implemente una solución general a k procesos, con n/k valores cada uno (*odd-even/exchange sort*)

```
Process worker[i = 1..k]

int largest = n/k, smallest = 1,
    a[1:k], dato;

# Ordeno la porcion del arreglo
# del proceso actual.

for(ronda = 1; ronda <= k; ronda++)

    if (i mod 2 == ronda mod 2)
        # Misma paridad (proceso+ronda par o proceso+ronda impar)
        if (i != k)
            proc[i+1]!(a[largest]);
            proc[i+1]?(dato);

            while (a[largest] > dato)
                # Inserto dato ordenado, pisando a[largest].
                proc[i+1]!(a[largest]);
                proc[i+1]?(dato);
            end;
        end;
    else
        if (i != 1)
            proc[i-1]?(dato);
            proc[i-1]!(a[smallest]);

            while (a[smallest] < dato)
                # inserto dato ordenado, pisando a[smallest].
                proc[i-1]?(dato);
                proc[i-1]!(a[smallest]);
            end;
        end;
    end;
end;
End.
```

d) ¿Cuántos mensajes se intercambian en (c) en el mejor caso? ¿Y en el peor de los casos?

Nota: todas las cuentas, son contando a los procesos inactivos de las rondas pares. Es por eso que, decimos *de forma general*.

Peor caso

En el peor caso, la lista se encuentra ordenada de forma descendiente. Lo que implica que, cada proceso deba intercambiar todos sus valores.

De forma general, cada proceso intercambia $\frac{n}{k+1}$ valores por ronda. El $+1$ puede verse como un proceso fantasma, para que ningun proceso quede inactivo en las rondas pares. Existirán k procesos ejecutandose en k rondas, por lo que la cantidad de mensajes totales está dada por la siguiente forma:

$$k * k * \frac{n}{k+1} = k^2 * \frac{n}{k+1} \tag{1}$$

n	k	Forma general	Valor exacto
8	4	25,6	24
12	4	38,4	36
12	6	61,2	60

Mejor caso

En el mejor caso, la lista se encuentra ordenada de menor a mayor. Siguiendo la lógica anterior, cada proceso enviará a lo sumo 1 (un) mensaje. Dando un total de k^2 mensajes.

e) **¿Cómo modificaría el algoritmo del punto (c) para que termine tan rápido como el arreglo este ordenado? ¿Esto agrega overhead de mensajes? De ser así, ¿Cuánto?**

Se podría implementar un proceso coordinador para lograr la siguiente interacción:. Al final de cada ronda, los procesos le indican al coordinador si efectuaron cambios en su colección de números

- Los procesos, al final de cada ronda, le notifican al coordinador si efectuaron cambios en su colección de números.
- Si el coordinador, para una ronda, no recibe ninguna notificación de cambios entonces; da por concluido el trabajo y notifica a los procesos.

Esto, claramente, agrega un overhead en el pasaje de mensajes. Ya que, luego de cada ronda el coordinador recibirá k mensajes (1 por proceso) y envará otros k mensajes (1 a cada proceso). Dando un overhead, total, de $2k$

f) **¿Considere que es más adecuado para este caso, si pasaje de mensajes sincrónico o asincrónico? Justifique.**

PMS resulta la opción más adecuada ya que los procesos deben sincronizar de a pares. El comportamiento bloqueante del send/receive produce, de forma implícita, una barrera simética.
En caso de emplear PMA, deberíamos implementar de forma explicita la barrera simétrica.

15. Suponga que quiere ordenar N números enteros utilizando pasaje de mensajes con el siguiente algoritmo (odd/even Exchange sort): Hay n procesos P [1: n], con n par. Cada proceso ejecuta una serie de rondas. En las rondas impares, los procesos con número impar P[impar] intercambian valores con P [impar +1] si los números están desordenados. En las rondas pares, los procesos con numero par P[par] intercambian valores con P [par +1] si los números están desordenados (P [1] y P[n] no hacen nada en las rondas pares).

a) **Determine cuantas rondas deben ejecutarse en el peor caso para ordenar los números.**

Peor caso → la lista está ordenada de forma descendente (mayor a menor).
En este caso, el algoritmo requerirá de n rondas. Ya que requiere, enviar el máximo valor n posiciones hacia delante. Con el valor mínimo, ocurre lo mismo; es necesario moverlo n posiciones hacia atrás. Ambos desplazamientos se dan en simultáneo.

b) **¿Considere que es más adecuado para este caso, si pasaje de mensajes sincrónico o asincrónico? Justifique.**

Misma respuesta que en el caso general para k procesos.

16. Dado el siguiente programa concurrente con memoria compartida, tenga en cuenta que las instrucciones no son atómicas:

```
1 x:=4; y:=2; z:=3;
2 co x:=x-z // z:=z*2 // y:=z+4  oc
```

a) **¿Cuáles de las asignaciones dentro de la sentencia co cumplen con la propiedad de a lo sumo una vez? Justifique.**

- `x:=x-z` cumple con ASV. Contiene una referencia crítica (`z`) pero, `x` no es referenciada por ningun otro proceso.
- `z:=z*2` cumple con ASV. No contiene referencias críticas. `z:=z*2` Contiene una referencia crítica (`z`) pero, `y` no es referenciada por ningun otro proceso.

17. Sea el problema de alocacion Shortest Job Next

Nota$_1$: mismo algoritmo que el del apunte de monitores.
Nota$_2$: Para el LJN (Lowest Job Next) aplican las mismas respuestas.

a) **¿Funciona correctamente con disciplina de señalización Signal and continue? Justifique.**

Con esta disciplina, todo proceso señalizado es enviado a una cola de listos; donde competirá contra el resto por el acceso al recurso. En cuyo caso, podría perder contra otro proceso y la meta de SJN no se estaría concretando.

b) **¿Funciona correctamente con disciplina de señalización signal and wait? Justifique.**
Con esta disciplina, todo proceso señalizado retoma su ejecución de inmediato. Por tal motivo, esta disciplina es la más adecudada para la implementación.

18. Suponga que el tiempo de ejecución de un algoritmo secuencial es de 1000 unidades de tiempo, de las cuales el 80 % corresponden a código paralelizable. ¿Cuál es el límite en la mejora que puede obtenerse paralelizando el algoritmo?

Apunte IV: Ley de Amdhal (sección 1.5)

19. Suponga los siguientes métodos de ordenación de menor a mayor para n valores (n par y potencia de dos), utilizando pasaje de mensajes. Asuma que cada proceso tiene almacenamiento local solo para dos valores (el próximo y el mantenido hasta ese momento).
(1) Un pipeline de filtros. El primero hace un input de los valores de a uno por vez, mantiene el mínimo y le pasa los otros al siguiente. Cada filtro hace lo mismo: recibe un stream de valores desde el procesador, mantiene el mínimo y pasa los otros al sucesor.
(2) Una red de procesos filtro como la del dibujo (sort merge network)
(3) Odd/Even Exchange Sort. Odd/even exchange sort. Hay n procesos P[1:n], Cada uno ejecuta una serie de rondas. En las rondas "impares", los procesos con número impar P[impar] intercambian valores con P[impar+1]. En las rondas "pares", los procesos con número par P[par] intercambian valores con P[par+1] (P[1] y P[n] no hacen nada en las rondas "pares"). En cada caso, si los números están desordenados actualizan su valor con el recibido.

a) **¿Cuántos procesos son necesarios en (1) y (2)? Justifique.**

- Pipeline → En este escenario, cada proceso es responsable de retener un valor mínimo. Por consiguiente, se requieren n procesos para obtener el orden correcto.
- Sort Merged Network → En este escenario los procesos se organizan en forma de árbol binario, cuya altura (h) está dada por $log_2 n$. La cantidad de nodos de un árbol equivale a $2^h - 1$.

b) **¿Cuántos mensajes envía cada algoritmo para ordenar los valores? Justifique. NOTA: se pueden obviar en los cálculos los mensajes que se requieren para enviar los EOS.**
Nota: Se asume el peor de los casos para las respuestas.

- Pipeline → Cada proceso envía 1 mensaje menos, del que recibió. De forma general, la cantidad total de mensajes puede verse como la siguiente sumatoria $\sum_{i=1}^{n} n - i$
- Sort Merged Network → Cada nivel del árbol envía n mensajes, sabiendo que la altura del árbol es $log_2 n$ podemos afirmar que se enviarán $n * log_2 n$ mensajes.
- Odd/Even Exchange Sort → El algoritmo requerirá de hasta $k^2 \frac{n}{k+1}$ mensajes.

c) **¿En cada caso, cuáles mensajes pueden ser enviados en paralelo (asumiendo que existe el hardware apropiado) y cuáles son enviados secuencialmente? Justifique.**

- Pipeline → en este caso se podrán enviar tantos mensajes en paralelo, como procesos activos existan en un instante de tiempo dado.
- Sort Merged Network → en este caso la cantidad de mensajes está dada en función de la altura del árbol. Se podrían enviar tantos mensajes, en paralelo, como procesos de una altura determinada se estén ejecutando.
- Odd/Even Exchange Sort → en este caso la cantidad de mensajes, también, está dada en función de la cantidad de procesos activos. En cada ronda, se podrán enviar en paralelo tantos mensajes como pares de procesos activos.

d) **¿Cuál es el tiempo total de ejecución de cada algoritmo? Asuma que cada operación de comparación o de envío de mensaje toma una unidad de tiempo. Justifique. Nota: Se asume el peor de los casos para las respuestas.**

- Pipeline → la lógica en este algoritmo es la de comparar, realizar una asignación si corresponde, y enviar un mensaje. Por lo tanto, cada proceso hara tantas comparaciones como envío de mensajes:

$$\text{unidades de tiempo} \rightarrow 2 * \sum_{i=1}^{n} n - i$$

- Sort Merged Network → La lógica en este algoritmo, es similiar al pipeline (comparar, asignar y enviar). Por tal motivo, cada proceso hara tantas comparaciones como envío de mensajes:

$$\text{unidades de tiempo} \rightarrow 2 * (n * log_2 n)$$

- Odd/Even Exchange Sort → La lógica en este algoritmo, es similiar al pipeline (comparar, asignar y enviar). Por tal motivo, cada proceso hara tantas comparaciones como envío de mensajes:

$$\text{unidades de tiempo} \rightarrow 2 * (k^2 * \frac{n}{k+1})$$

20. Sea la siguiente solución al problema de multiplicación de matrices de n*n con P procesadores trabajando en paralelo.

***a*) Suponga n=128 y que cada procesador es capaz de ejecutar un proceso. ¿Cuántas asignaciones, sumas y productos se hacen secuencialmente (caso en el que P=1)?**

- En este caso $strip \rightarrow n$
- Línea 11: Hace tantas pasadas como columnas tenga B (n) por el tamaño del strip $\rightarrow n^2$
- Línea 15: Hace n pasadas por tantas pasadas como columnas tenga B por el tamaño del strip $\rightarrow n^3$

$Asignaciones = n^3 + n^2 = 128^3 + 128^2 = 2097152 + 16384 = 2113536$
$Sumas = n^3 = 128^3 = 2097152$
$Productos = n^3 = 128^3 = 2097152$

***b*) Manteniendo n=128. Si los procesadores P1 a P7 son iguales, y sus tiempos de asignación son 1, de suma 2 y de producto 3, y si P8 es 4 veces más lento, ¿Cuánto tarda el proceso total concurrente? ¿Cuál es el valor del speedup (Tiempo secuencial/Tiempo paralelo)?. Modifique el código para lograr un mejor speedup.**

- En este caso $strip = n/p = 128/8 = 16$

$Asignaciones = n^2 \times 16 + n \times 16 = 128^2 \times 16 + 128 \times 16 = 262144 + 2048 = 264192$
$Sumas = n^2 \times 16 = 128^2 \times 16 = 262144$
$Productos = n^2 \times 16 = 128^2 \times 16 = 262144$
Los procesos 1 a 7, tardaran lo mismo:
$264192 \times 1ut + 262144 \times 2ut + 262144 \times 3ut = 1574912ut$
El proceso 8, es 4 veces más lento que el resto. Por lo cual, tardará 4 veces más:
$1574912ut \times 4 = 6299648ut$
Por consiguiente, el proceso concurrente tardará $6299648ut$ en finalizar. Ya que, el proceso 8 será el último, en terminar su trabajo.
Con las unidades de tiempo, de los procesadores más eficientes, el proceso secuencial tardará:
$2113536 \times 1ut + 2097152 \times 2ut + 2097152 \times 3 = 12599296ut$
Por consiguiente el Speedup obtenido será de 2.
Para mejorar el Speedup podríamos balancear la carga de trabajo, de los procesadores, de manera distinta. Por ejemplo; haciendo que el procesador 8, el más lento, trabaje sobre un strip más pequeño.

- Múltiplo de 7 más cercano a 128 → 126
- Tamaño del stripe, para el procesador 8 → $128 - 126 = 2$
- Tamaño del stripe, para el resto de los procesadores → $126/7 = 18$

Asignaciones $P_8 = 128^2 \times 2 + 128 \times 2 = 33024$
Sumas $P_8 = 128^2 \times 2 = 32768$
Productos $P_8 = 128^2 \times 2 = 32768$
Tiempo $P_8 = 33024 \times 1ut + 32768 \times 2ut + 32768 \times 3 = 196864ut$
Asignaciones $P_{resto} = 128^2 \times 18 + 128 \times 18 = 297216$
Sumas $P_{resto} = 128^2 \times 18 = 294912$
Productos $P_{resto} = 128^2 \times 2 = 294912$
Tiempo $P_{resto} = 297216 \times 1ut + 294912 \times 2ut + 294912 \times 3 = 1771776ut$
Con estos nuevos tiempos el speedup será de 7,1.

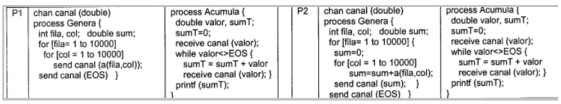
21. Dado el siguiente programa, indice si es posible que finalice.

```
bool continue = true;
bool try = false;
co  while (continue) { try = true; try = false; } #(P)
/   <await(try) continue = false> #(Q)
oc
```

Con una política débilmente fair, el programa podría no terminar. Ya que, *try* no se mantiene verdadera hasta ser vista por (Q).

Con una política fuertemente fair, el programa podría terminar. Ya que, *try* se convierte en verdadera con infinita frecuencia.

22. Suponga los siguientes programas concurrentes. Asuma que EOS es un valor especial que indica el fin de la secuencia de mensajes y que los procesos son iniciados desde el programa principal.



***a*) ¿Qué hacen los programas?**

Ambos programas realizan la suma de los elementos de una matriz. Aunque, difieren en el modo en que lo hacen.

En el `Programa 1`: el proceso `Genera` envía todos los valores al proceso `Acumula`. El proceso `Acumula` suma los valores que recibe desde `Genera`.

En el `Programa 2`: el proceso `Genera` realiza la suma de los elementos de una fila y envia el resultado a `Acumula`. El proceso `Acumula`, recibe los resultados parciales y los adiciona en su acumulador.

***b*) Analice desde el punto de vista del número de mensajes.**

En el `Programa 1`, el proceso `Genera` envía todos los elementos de la matriz. Por lo cual, en esta solución, se están enviando $n * n = n^2$ mensajes.

En el `Programa 2`, el proceso `Genera` envía los valores acumulados de cada fila. Por lo cual, en esta solución, se están enviando n mensajes.

***c*) Analice desde el punto de vista de la granularidad de procesos.**

El programa `P2` tiene mayor granularidad, con respecto a `P1`. Esto se debe a que, requiere menor comunicación (menos mensajes) para completar la tarea.

***d*) ¿Cuál de los programas le parece el más adecuado para ejecutar sobre una arquitectura tipo cluster PCs? Justifique.**

Las arquitecturas cluster PC pueden ser catalogadas como grano grueso. Esto es, porque se componen de mucho procesadores; otorgando mayor potencia para cómputo. No obstante, tienen una menor performance en cuanto a comunicación.

Es deseable que la granularidad de la aplicación, se ajuste a la de la arquitectura. Para hacer un uso apropiado de los recursos y evitar rendimientos menores.

En este escenario, el programa, `P1` parece el más adecuado para ejecutar en la arquitectura planteada.

23. Dado el siguiente bloque de código, indique para cada inciso qué valor queda en aux, o si el código queda bloqueado. Justifique su respuesta.

```
aux = -1;
...
if (A == 0); P2?(aux) -> aux = aux + 2; # (1)
[] (A == 1); P3?(aux) -> aux = aux + 5; # (2)
[] (B == 0); P3?(aux) -> aux = aux + 7; # (3)
fi
```

***a*) Si el valor de A = 1 y B = 2 antes del if, y solo P2 envia el valor 6.**

La guarda 1 queda bloqueada; su condición booleana es verdadera pero, su sentencia de comunicación generá bloqueo.

***b*) Si el valor de A = 0 y B = 2 antes del if, y solo P2 envia el valor 8.**

Solo la guarda 1 tiene éxito. Su sentencia guardada es ejecutada. El programa finaliza con `aux = 10`.

c) **Si el valor de A = 2 y B = 0 antes del if, y solo P3 envia el valor 6.**
Solo la guarda 3 tiene éxito. Su sentencia guardada es ejecutada. El programa finaliza con `aux = 13`.

d) **Si el valor de A = 2 y B = 1 antes del if, y solo P3 envia el valor 9.**
Todas las guardas fallan porque sus condiciones booleanas se evalúan como falsas. El programa finaliza con `aux = -1`.

e) **Si el valor de A = 1 y B = 0 antes del if, y solo P3 envia el valor 14.**
Las guardas 2 y 3, tienen éxito. Se elegirá una de las dos, de forma no deterministica y se ejecutará su sentencia guardada.
El programa finalizará con `aux = 19 o 21`.

f) **Si el valor de A = 0 y B = 0 antes del if, P3 envia el valor 9 y P2 el valor 5.**
Las guardas 1 y 3, tienen éxito. Se elegirá una de las dos, de forma no deterministica y se ejecutará su sentencia guardada.
El programa finalizará con `aux = 16 o 7`.

24. ¿Qué significa el problema de interferencia en un programa concurrente? ¿Cómo puede evitarse?

Apunte I: sección 1.4

25. Suponga que n2 procesos organizados en forma de grilla cuadrada. Cada proceso puede comunicarse solo con los vecinos izquierdo, derecho, de arriba y de abajo (los procesos de las esquinas tienen solo 2 vecinos, y los otros en los bordes de la grilla tienen 3 vecinos). Cada proceso tiene inicialmente un valor local v.

a) **Escriba un algoritmo heartbeat que calcule el máximo y el mínimo de los n2 valores. Al terminar el programa, cada proceso debe conocer ambos valores.**
La idea del algoritmo es que, cada proceso pueda propagar su valor hasta el proceso más lejano. La mayor distancia entre procesos, sin diagonales, está dada por $2(n-1)$ (esquina superior izquierda $\longleftrightarrow$ esquina inferior derecha).

```
1  chan valores[1:n,1:n](int);
2
3  Process P[i=1..n;j=1..n]
4
5  int v;
6  int nuevo;
7  int min = v;
8  int max = v;
9  int rondas = 2(n-1);
10
11 for(k=1; k <= rondas; k++)
12
13     foreach(vecinos => (x,y))
14         send valores[x,y](v);
15
16     foreach(vecinos => (x,y))
17         receive valores[i,j](nuevo);
18         if (nuevo < min) min = nuevo;
19         if (nuevo > max) max = nuevo;
20 end;
21 end;
```

b) **Analice la solución desde el punto de vista del número de mensajes.**

Tipo	Cantidad	Vecinos	Mensajes
Esquina	4	2	$cantidad * vecinos * rondas$
Borde	$4(n-2)$	3	$cantidad * vecinos * rondas$
Internos	$(n-2)^2$	4	$cantidad * vecinos * rondas$

***c)* ¿Puede realizar alguna mejora para reducir el número de mensajes?**

Podemos hacer que cada proceso, también, se comunique con sus diagonales. De esta forma; los procesos esquina tendrán 3 vecinos, los bordes 5 vecinos y el resto 8 vecinos.

Esto incrementará el número de mensajes enviados por cada proceso pero, también reducirá en la mitad la cantidad de rondas $(n-1)$. Generando una reducción en la cantidad final de mensajes.

Tipo	Cantidad	Vecinos
Esquina	4	3
Borde	$4(n-2)$	5
Internos	$(n-2)^2$	8

n	Sin Diagonales	Con Diagonales
4	288	252
6	1200	1100
8	3136	2940

***d)* Indique que mecanismo de pasaje de mensajes utilizaría. Justifique.**

En mi opinión, utilizaría PMA. Particularmente, porque su primitiva `send` no es bloqueante. Esto nos evita demoras innecesarias al momento de propagar el valor de un proceso, en particular con n muy grandes.

26. En los protocolos de acceso a sección crítica vistos en clase, cada proceso ejecuta el mismo algoritmo. Una manera alternativa de resolver el problema es usando un proceso coordinador. En este caso, cuando cada proceso SC[i] quiere entrar a su sección crítica le avisa al coordinador, y espera a que éste le de permiso. Al terminar de ejecutar su sección crítica, el proceso SC[i] le avisa al coordinador. Desarrolle protocolos para los procesos SC[i] y el coordinador usando sólo variables compartidas (no tenga en cuenta la propiedad de eventual entrada).

```
1  int request[n] = ([n] 0),
2      grant[n] = ([n] 0);
3
4  Process Worker[i = 1..n]
5
6  # Seccion no critica ...
7
8  request[i] = 1;
9  while(grant[i] == 0) skip;
10 # Seccion Critica
11 grant[i] = 0;
12 # Seccion no critica ...
13 end;
```

```
14
15 Process Coordinator
16 while (true)
17     for (j = 1; j < n; j++)
18         if (request[j] == 1)
19             request[j] = 0;
20             grant[j] = 1;
21             while (grant[j] == 1) skip;
22         end;
23     end;
24 end;
25 end;
```

27. Suponga que la solución a un problema es paralelizada sobre p procesadores de dos maneras diferentes. En un caso, el speedup (S) está regido por la función S=p-1 y en el otro por la función S=p/2. ¿Cuál de las dos soluciones se comportará más eficientemente al crecer la cantidad de procesadores? Justifique claramente.

Apunte IV: sección 1.4 (Escalabilidad)

28. Analice que tipos de mecanismos de pasaje de mensajes son más adecuados para resolver problemas del tipo Cliente-Servidor, Pares que interactúan, filtro y productores- consumidores. Justificar.

En los problemas del tipo **pares que interactúan, filtros y productores/consumidores**; el esquema de **comunicación entre procesos**, por lo general, es **unidireccional**. Un proceso, envía un mensaje a otro y no requiere una respuesta, del receptor de dicho mensaje, para continuar con su trabajo.

Dado este escenario, descripto en el párrafo anterior, la opción más adecuada es la de utilizar **pasaje de mensajes asincrónico**. Ya que su primitiva `send` no es bloqueante para el emisor; permitiendole despachar su mensaje y continúar con su trabajo.

En los problemas del tipo **cliente/servidor**; el esquema de **comunicación entre procesos** es **bidireccional**. Un proceso, cliente, envía una solicitud a un proceso servidor; y espera por la respuesta correspondiente.

Dado este escenario, descripto en el párrafo anterior, la opción más adecuada es la de utilizar **pasaje de mensajes sincrónico**. Ya que su primitiva `send` es bloqueante para el emisor; impidiendo que continue con su ejecución hasta obtener la respuesta correspondiente.

Además, en los problemas del tipo **cliente/servidor**, tenemos que tener en cuenta la **cantidad de canales a definir**. Esto supone un problema ya que; necesitamos al menos un canal para que el servidor reciba solicitudes y tantos canales, de respuesta, como procesos clientes tengamos.

Existen técnicas que resuelven esta problematica, sobre la cantidad de canales a definir. Una de ellas es **Rendezvous** y otra **Remote Procedure Call (RPC)**.