Índice de contenido

Programación Cfoncurrente

Cuestionarios y Ejercicios Resueltos

Cuestionario Clases 1 y 2

1- Mencione al menos 3 ejemplos donde pueda encontrarse concurrencia (que no estén mencionados en las transparencias de la clase)

1. Reproductor de DVD. Mientras lee del dispositivo, decodifica audio y vídeo, controla la temporización de subtítulos, envía las señales de salida...
2. Videojuego de estrategia. Mientras mantiene actualizado el mapa en pantalla, calcula movimientos de enemigos, captura órdenes del usuario, controla el desarrollo del juego...
3. Procesador de textos. Mientras recibe entradas del usuario, controla ortografía, realiza sugerencias, guardados automáticos...

2- Escriba una definición de concurrencia. Diferencie procesamiento secuencial, concurrente y paralelo.

La concurrencia es la simultaneidad en la ejecución de dos o más procesos.

1. **Procesamiento secuencial:** Cuando se tiene un único hilo de ejecución o control.
2. **Procesamiento concurrente:** Cuando se tiene más de un hilo de ejecución o control, no implicando la existencia de más de un procesador.
3. **Procesamiento paralelo:** Procesamiento concurrente sobre más de un procesador, ejecutando cada uno un hilo diferente en un momento dado.

3- ¿Cuáles son las 3 grandes clases de aplicaciones concurrentes que podemos encontrar? Ejemplifique.

1. **Multithreading** (cantidad de procesos mayor a la cantidad de procesadores) → Solución Antivirus, contando con módulos residentes de: monitoreo de acceso a disco, emails, control de popups, firewall...
2. **Cómputo Distribuido** (equipos mono/multiprocesador comunicados por red) → El conocido proyecto SETI@Home, mediante el cual PC's conectadas a Internet descargan y analizan datos, devolviendo la información a los servidores del proyecto.
3. **Procesamiento Paralelo** (arquitectura multiprocesador) → Evaluación del impacto de los fenómenos meteorológicos en los cultivos agrícolas.

4- Describa el concepto de deadlock y qué condiciones deben darse para que ocurra.

Deadlock: Bloqueo permanente de un conjunto de procesos producto de la espera por recursos de uso exclusivo, cada proceso esperando algún recurso sin liberar a los que posee bajo control. Puede producirse entre dos procesos o entre varios, en forma circular.

Las condiciones necesarias para que se produzca (condiciones de *Coffman*) son:

1. **Exclusión mutua:** Existencia de al menos un recurso compartido por los procesos, al cual sólo puede acceder uno en un momento dado.
2. **Posesión y espera:** Al menos un proceso *Pi* ha adquirido un recurso *Ri*, y lo mantiene mientras espera al menos un recurso *Rj* que ya ha sido asignado a otro proceso.
3. **No expropiación:** Un recurso asignado a un proceso *Pi* no puede ser expropiado por otro proceso *Pj*.
4. **Espera circular:** Dado un conjunto de *n* procesos *Pi*, cada proceso retiene un recurso que espera su sucesor, esperando *P1* el recurso retenido por *Pn*.

5- Defina inanición. Ejemplifique.

Un proceso no puede finalizar su ejecución ya que requiere de un recurso que nunca le es asignado.

El *deadlock* es un caso especial de inanición, como así también el *livelock*: una condición similar, pero en la que el estado de los procesos continúa cambiando, cada uno en respuesta a los cambios del otro, sin retomar el procesamiento.

Ejemplo de Inanición: cualquier esquema de atención por prioridades. Una técnica común para evitarlo es

el esquema de envejecimiento.

6- ¿Qué entiende por no determinismo? ¿Cómo se aplica este concepto a la ejecución concurrente?

El *no determinismo* es la producción de resultados impredecibles: incluso ante los mismos valores y condiciones iniciales, pueden obtenerse distintos resultados.
En un programa concurrente compuesto por un conjunto de *n* procesos, en un momento dado habrá *n* operaciones atómicas elegibles (asumiendo que todos estén en estado *ready*). Así, pueden obtenerse distintas *historias* de la ejecución del programa aún en corridas con iguales valores de entrada. La historia específica de una corrida será decidida NO por el programa concurrente, sino por la política de *scheduling* que aplique el SO.

7-

a) Defina sincronización. Explique los mecanismos de sincronización que conozca.

La sincronización es la interacción entre procesos que permite controlar el orden en que se ejecutan, restringiendo las *historias* posibles sólo a las deseables. Hay dos formas básicas:

1. **Exclusión Mutua.** Asegura que las sentencias en las *secciones críticas* de distintos procesos no puedan ejecutarse al mismo tiempo.
2. **Sincronización por Condición.** Retrasa la ejecución de un proceso hasta que una condición dada se vuelva verdadera.

b) En un programa concurrente, ¿puede estar presente más de un mecanismo de sincronización? Ejemplifique.

Ambos métodos de sincronización pueden estar presentes en un mismo programa ya que puede haber más de una circunstancia en la que los procesos precisen sincronizarse. Además, tienen distintos fines: delimitar secciones críticas y garantizar que el estado del programa sea adecuado antes de continuar la ejecución de un proceso.
Por ejemplo, puede lograrse la *exclusión mutua* utilizando *sincronización por condición* para los protocolos de entrada a las *secciones críticas*.

8-

a) Analice en qué tipo de problemas son más adecuados cada uno de los 5 paradigmas de resolución de problemas concurrentes descriptos en clase.

1. **Paralelismo Iterativo.** Problemas en los que un procesamiento a realizar sobre un dominio grande, pueda realizarse paralelamente en subconjuntos independientes de dicho dominio.
2. **Paralelismo Recursivo.** Problemas en los que un mismo procesamiento debe hacerse recursivamente sobre un conjunto de datos y el resultado de procesar el mismo, con más de una recursión.
3. **Productores y Consumidores (Pipe).** Problemas en los que un procesamiento complejo debe realizarse sobre un conjunto de datos, siendo posible dividirlo en un conjunto independiente de tareas secuenciables. Así, cada tarea genera la entrada de la siguiente y pueden ejecutarse en paralelo resultando en un flujo de datos unidireccional.
4. **Cliente/Servidor.** Similar al tipo anterior, pero aplicado a un entorno distribuido. Un proceso *cliente* debe, en lugar de continuar procesando un flujo de datos, solicitarlo explícitamente a otro proceso *servidor,* que procesa la información y devuelve resultados originando un flujo de ida y vuelta.
5. **Pares que Interactúan (Peers).** Problemas de paralelismo (tipos 1 y 2), pero aplicados a entornos distribuidos. Los procesos en distintos procesadores se comunican con pase de mensajes para coordinar la tarea, entre sí o dirigidos por un proceso *coordinador* particular.

b) ¿Qué relación encuentra entre el paralelismo recursivo y la estrategia de "dividir y conquistar"?

¿Cómo aplicaría este concepto a un problema de ordenación de un arreglo (por ejemplo usando un algoritmo de tipo "quicksort" o uno de tipo "sorting by merging")?

La estrategia de *Dividir y Conquistar* genera algoritmos recursivos que afectan partes independientes de un dominio de datos, siendo aptos para su paralelización.
Por ejemplo en un algoritmo de ordenación *quicksort* un proceso podría procesar el arreglo completo a ordenar y luego dar origen a dos procesos paralelos, uno para cada mitad. Para evitar la creación constante de nuevos procesos, podría originar sólo un proceso paralelo que afecte una de las mitades y realizar un llamada recursiva estándar sobre la otra.

9-

a) Analizando el código de multiplicación de matrices en paralelo planteado en la teoría, y suponiendo que N=256 y P=8 indique cuántas asignaciones, cuántas sumas y cuántos productos realiza cada proceso. ¿Cuál sería la cantidad de cada operación en la solución secuencial realizada por un único proceso?

Cada proceso trabaja sobre 32 filas. Se consideran sólo las operaciones dentro de los bucles, pero no las de variables de control de los mismos.
Asignaciones: 32*256 + 32 * 256^2 = 2105344
Sumas: 32 * 256^2 = 2097152
Multiplicaciones: 32 * 256^2 = 2097152

```
process worker [w = 1 to P]{
  int primera = (w-1)*(n/P) + 1;
  int ultima = primera + (n/P) - 1;
  for [i = primera to ultima]{
    for [j = 1 to n]{
      c[i,j] = 0; //32 * 256 "="
      for [k = 1 to n]
        c[i,j] = c[i,j] + (a[i,k]*b[k,j]); //256^2 * 32 "=" "+" y "*"
    }
  }
}
```

Asignaciones: 256^2 + 256^3 = 16842752
Sumas: 256^3 = 16777216
Multiplicaciones: 256^3 = 16777216

```
double a[n,n], b[n,n], c[n,n];
for [i = 1 to n] {
  for [j = 1 to n] {
    c[i,j] = 0.0; //256^2 "="
    for [k = 1 to n]
      c[i,j] = c[i,j] + a[i,k]*b[k,j]; //256^3 "=" "+" y "*"
  }
}
```

Puede observarse como la solución paralela reduce notablemente la cantidad de operaciones *por proceso* (por ejemplo alrededor de 2 millones de sumas contra unos 16 millones). La cantidad total no se afecta, sólo se distribuye en la solución paralela.

b) Si los procesadores P1 a P7 son iguales, y sus tiempos de asignación son 1, de suma 2 y de producto 3, y si el procesador P8 es 4 veces más lento, ¿cuánto tarda el proceso total concurrente? ¿Cuál es el valor del speedup? ¿Cómo podría modificar el código para mejorar el speedup?

Por ser el procesador #8 **el más lento, es el que se tiene en cuenta para el cálculo en paralelo**:

```
Procesamiento Paralelo:
Asignaciones: 2105344 * 1 *4 = 8421376
Sumas: 2097152 * 2 *4 = 16777216
Multiplicaciones:2097152 * 3 *4 = 25165824
TOTAL = 50.364.416
```

Para el **procesamiento secuencial, se consideran los mejores procesadores**:

```
Procesamiento Secuencial:
Asignaciones: 16842752 * 1 = 16842752
Sumas: 16777216 * 2 = 33554432
Multiplicaciones:16777216 * 3 = 50331648
TOTAL = 100.728.832
```

SpeedUp = 100728832 / 50364416 = 2 → Se redujo a la mitad el tiempo agregando 7 procesadores

Suponiendo que el octavo proceso se asigna al octavo procesador (el más lento) podrían utilizarse bandas asimétricas, dándole menor cantidad de trabajo (con la división actual, el resto de los procesadores tardan unas 12 millones de unidades de tiempo solamente, permaneciendo ociosos hasta que termine el octavo).
Asignando menor cantidad de filas a P8:

```
P8   → 11 filas → 17.312.768 ut → tardaría 17 312 768 ut
P1a7 → 35 filas → 13.771.520 ut

P8   → 4 filas  →  6.295.552 ut
P1a7 → 36 filas → 14.164.992 tu → tardaría 14 164 992 ut
```

Convendría asignar sólo 4 filas al procesador lento y 36 a los más rápidos. Se lograría así un *speed up* de: 100728832 / 14164992 = 7.11 (obviamente no llega a ser 8 que sería proporcional a la cantidad de procesadores usados, ya que el lento permanece ocioso el 55.57% del tiempo con esta solución. Sin embargo, de asignarle más trabajo retrasaría al resto).

10- Analice en qué aspectos puede influir la arquitectura de soporte en la ejecución de aplicaciones concurrentes/paralelas/distribuidas. Ejemplifique.

Distintas arquitecturas pueden afectar el modo de implementación y el rendimiento de una aplicación concurrente, o ser aptas para algún tipo específico de aplicaciones.
Por ejemplo, las arquitecturas SIMD ofrecen mejores rendimientos y facilidades para aplicaciones con paralelismo de datos.
Por otra parte, también el juego de instrucciones que posee una arquitectura puede facilitar o no la implementación de algunas soluciones concurrentes (por ejemplo, si se cuenta con instrucciones como Test & Set (TS), Fetch & Add (FA) o Compare & Swap (CS)).
Otros aspectos que influyen son:

1. La organización del espacio de direcciones (MC -UMA o NUMA- vs MD),
2. El mecanismo de control (máquinas SISD, SIMD o MIMD. MISD es mayormente teórico, no hay muchos ejemplos, aunque suele considerarse en este grupo algunas máquinas tolerantes a fallos que ejecutan una misma instrucción redundantemente para detectar errores),
3. La granularidad (relación #procesadores/memoria o velocidad cálculo/comunicación) → Grano Grueso (pocos procesadores, rápidos) vs Grano Fino (muchos procesadores, lentos),
4. La red de interconexión (estáticas con links punto-a-punto -MD- o dinámicas con switches -MC-).

11- ¿Qué significa el problema de "interferencia" en programación concurrente? ¿Cómo puede evitarse?

La *Interferencia* es el resultado de dos o más procesos leyendo y escribiendo variables compartidas en un orden impredecible y, por tanto, con resultados impredecibles. En general, una instrucción atómica en un

proceso interfiere con una *precondición* de una *sección crítica* o una *suposición* hecha por otro, si modifica su valor de verdad.
Puede evitarse por medio de:

1. **Variables Disjuntas.** Validar que el conjunto de escritura de un proceso y el conjunto de referencias de otro sean disjuntos.
2. **Asunciones Débiles.** Escribir los procesos considerando los efectos de otros procesos en ejecución, realizando menos asunciones o más generales.
3. **Invariante Global.** Si todas las asunciones de los procesos pueden escribirse de la forma I ^ L, donde I es un invariante global y donde L sólo referencia variables locales o globales que sólo ese proceso escribe, entonces los procesos están libres de interferencia.
4. **Sincronización.** Evitar interferencia mediante exclusión mutua o sincronización por condición.

12- ¿En qué consiste la propiedad de "A lo sumo una vez" y qué efecto tiene sobre las sentencias de un programa concurrente? De ejemplos de sentencias que cumplan y de sentencias que no cumplan con ASV.

La propiedad de *A lo sumo una vez* indica que una *expresión* no causará interferencia con otros procesos si tiene como máximo una referencia crítica. En el caso de una asignación *vble = expresión*, es necesario que ningún proceso lea a *vble* o que en ese caso, no haya referencias críticas en expresión.
Las sentencias de un programa concurrente pueden referenciar a lo sumo una variable compartida, a lo sumo una vez, para estar libres de interferencias. Si así lo hacen, su ejecución parecerá atómica al resto de los procesos.

No Cumplen ASV:

```
int a = 3, j = 6
co
   j = a + j #2 ref. Críticas
// j = 1     #1 ref. críticas
oc
```

Cumplen ASV:

```
int a = 3, j = 6
co
   a  = a + j #1 ref. Crítica, a NO es leída
// j = 1      #0 ref. críticas
oc
```

13- Dado el siguiente programa concurrente:

```
x = 3; y = 5; z = 2;
co
   x = y - z
// z = x * 3
// y = y + 3
oc
```

a) ¿Cuáles de las asignaciones dentro de la sentencia co cumplen con ASV? Justifique claramente.

x = y – z NO CUMPLE. Dos referencias críticas, y *x* es leída por otros procesos.
z = x * 3 NO CUMPLE. Una referencia crítica, pero *z* es leída por otros procesos.
y = y + 3 CUMPLE CON ASV. No posee referencias críticas al lado derecho; *y* puede ser leída por otros procesos.

b) Indique los resultados posibles de la ejecución Las instrucciones NO SON atómicas. No es necesario que liste TODOS los resultados pero si los representativos de las diferentes situaciones que pueden darse.

Se asigna x, luego z y último y: x=3, z=9, y=8
Si los procesos ejecutan una instrucción atómica de bajo nivel por vez: x=3, z=9, y=8
Se asigna x, luego y y último z: x=3, z=9, y=8
Se asigna y, luego z y último x: x=-1, z=9, y=8
Se asigna y, luego x y último z: x=6, z=18, y=8
Se asigna z, luego x y último y: x=-4, z=9, y=8

Se asigna z, luego y y último x: x=-1, z=9, y=8
Nótese que el valor de *y* para todas las ejecuciones presentadas se mantiene en 8, dado que su asignación es la única que cumple ASV. Las variables *x* y *z* varían en su resultado.

Cuestionario Clase 3

1- Defina fairness. ¿Cuál es la relación con las políticas de scheduling? (NO DESCRIBA LOS DISTINTOS TIPOS DE POLÍTICAS DE SCHEDULING)

El Fairness (justicia/equidad) es una característica de las políticas de scheduling, que indica si los procesos tendrán la oportunidad de avanzar, sin importar lo que hagan los demás. Existen distintos tipos: incondicionalmente-fair, débilmente-fair y fuertemente-fair.

¿Por qué las propiedades de vida dependen de la política de scheduling?

Que un proceso logre un estado bueno depende de que la política de scheduling le permita ejecutarse cuando las condiciones sean las adecuadas. Por ejemplo, una política *incondicionalmente fair* no aseguraría la *terminación* de un programa con acciones atómicas condicionales para todas sus historias posibles.

¿Cómo aplicaría el concepto de fairness al acceso a una base de datos compartida por n procesos concurrentes?

Se aplica el concepto de fairness al scheduling que se haga de la base de datos como recurso compartido:

1. Si el ingreso de un lector permite el ingreso de otros, aún si hay escritores esperando → débilmente fair: el ingreso permanente de lectores causaría inanición a los escritores.
2. Si la llegada de un escritor impide el acceso de nuevos procesos y es atendida en primer lugar al liberarse la BD → débilmente fair: el ingreso constante de escritores causaría inanición a los lectores.
3. Si los procesos son atendidos en orden FIFO con acceso concurrente de los lectores → fair: todos serán atendidos.

La última opción, aunque es fuertemente fair, produce un delay innecesario en los lectores que lleguen tras un escritor que espera. Sin embargo, dada la naturaleza del problema, de evitar dicho delay se caería en un esquema de prioridad con riesgo de inanición a los escritores.

2- ¿Qué propiedades deben garantizarse en la administración de una sección crítica en procesos concurrentes? ¿Cuáles de ellas son propiedades de seguridad y cuáles de vida? En el caso de las propiedades de seguridad, ¿cuál es en cada caso el estado "malo" que se debe evitar?

Propiedad	Tipo	Estado "malo"
Exclusión mutua	SEGURIDAD	En un momento dado, dos o más procesos se encuentran dentro de una SC.
Ausencia de Deadlock (Livelock)	SEGURIDAD	Ninguno de los procesos que intentan acceder a una SC tiene éxito, quedando todos bloqueados.
Ausencia de Demora Innecesaria	SEGURIDAD	Un proceso intenta ingresar a su SC y es bloqueado aún siendo el único que lo intenta.
Eventual Entrada	VIDA	No corresponde.-

3- 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 (no tenga en cuenta la propiedad de eventual entrada).

```
int permiso[1:N] = ([N] 0), aviso[1:N] = ([N] 0);

process SC[i = 1 to N] {
```

```
  SNC;
  permiso[i] = 1;           # Protocolo
  while (aviso[i]==0) skip; # de entrada
  SC;
  aviso[i] = 0;             # Protocolo de salida
  SNC;
}

process Coordinador {
  int i = 1;
  while (true) {
    while (permiso[i]==0) i = i mod N +1;
    permiso[i] = 0;
    aviso[i] = 1;
    while (aviso[i]==1) skip;
  }
}
```

4- Modifique el algoritmo Ticket para el caso en que no se dispone de una instrucción Fetch and Add.

```
int numero = 1, proximo = 1, turno[1:n] = ([n] 0);
bool lock = false;            # lock compartido
{TICKET: proximo > 0 ⋀ (∀i: 1≤ i ≤ n: (SC[i] está en su SC) ⇒ (turno[i]==
proximo) ⋀ (turno[i] >0) ⇒( ∀j : 1 ≤ j ≤ n, j ≠ i: turno[i] ≠ turno[j] ) ) }

process SC[i = 1 to n] {
  while (true) {
    seccion no critica;
    while (lock) skip;   # Protocolo
    while (TS(lock))     # de
      while (lock) skip; # Entrada (Test-and-TS)
    turno[i] = numero; numero = numero + 1; #FA(numero, 1)
    lock = false;        # Protocolo de Salida
    while (turno[i] <> proximo) skip;
    seccion critica;
    proximo = proximo + 1;
    seccion no critica;
  }
}
```

En este código al no contar con FA, se realizó la toma de turnos y el incremento de *numero* en una sección protegida por protocolos de entrada/salida utilizando TS. De no contarse con TS se podría implementar los protocolos con otro método -por ejemplo *Tie breaker*-, pero en ese caso quizá hubiese convenido utilizar dicho método para proteger la sección crítica inicial; la elección del algoritmo de ticket no fue la mejor opción en ese caso.

5- Implemente una butterfly barrier para 8 procesos usando variables compartidas.

8 procesos = 2^3 → puede implementarse *butterfly barrier*.
En una etapa *s* cada proceso sincroniza con uno a distancia *2^(s-1)*.

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

process Worker[i = 1 to n] {
```

```
int j, resto, distancia;
  while (true) {
    # codigo de tarea i
    # barrera en log2 8 = 3 etapas:
    for [s = 1 to 3] {
      #calcula la distancia y si debe sumarla o restarla
      distancia = 2^(s-1);
      #(0 < i mod 2^s <= distancia) → suma distancia, sino resta
      resto = i mod 2^s;
      if (resto=0)or(resto>distancia) then distancia = -distancia;
      j = i + distancia;
      while (arribo[i] != 0) skip;
      arribo[i] = 1;
      while (arribo[j] != 1) skip;
      arribo[j] = 0;
    }
  }
}
```

6- Una manera de ordenar n enteros es usar el algoritmo odd/even exchange sort (también llamado odd/even transposition sort). Asuma que hay n procesos P[1..n] y que n es par. En este algoritmo, 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 valores están desordenados. En las rondas con número par, los procesos con número par P[par] intercambian valores con P[par+1] si los valores están desordenados (en las rondas pares P[1] y P[n]).

Determine cuántas rondas deben ejecutarse en el peor caso para ordenar los n números.

En el peor caso se requieren **n-1 rondas**, ya que en cada ronda un elemento se acerca un lugar a su posición correcta, y la máxima distancia a la que se puede encontrar de ella es n-1 (considerando la distancia en el sentido en que trabaja el algoritmo y teniendo en cuenta que es circular).

Escriba un algoritmo data parallel para ordenar un arreglo de enteros a[1:n] en forma ascendente

Dado que *n* no necesariamente es potencia de 2, y que el algoritmo de *butterfly* se complica en estos casos, decidimos utilizar en su lugar una *barrera de diseminación* como sugiere la bibliografía.

```
int[] ordenar(int a[], int n) {
  int aviso[1:n] = ([n] 0);
  co [i = 1 to n] {
    int sig = i + 1, j;
    if(sig>n) sig = 1;
    for [ronda = 1 to n-1] {
      if(ronda mod 2 = i mod 2)
        if(a[i]>a[sig]) swap(a[i], a[sig]);
      # barrera de diseminación
      for [s = 1 to log2(n)] {
        j = i + 2^(s-1);
        if(j>n) j = j mod n;
        aviso[j] = 1;
        while (aviso[i] == 0) skip;
        aviso[i] = 0;
      }
    }
  }
```

```
  return a[];
}
```

Modifique el algoritmo para terminar tan pronto como el arreglo fue ordenado.

Para controlar cuando el arreglo está ya ordenado, agregamos dos variables globales *cambio1* y *cambio2,* seteando *cambio2* al realizar un swap. Así, si no hay cambios en dos rondas consecutivas se detecta al arreglo como ordenado; sino mantenemos en *cambio1* el valor de la ronda y reseteamos *cambio2* para comenzar una nueva.
Al plantear esta modificación sobre la solución basada en barrera de diseminación, nos encontramos ante algunas dificultades como dónde resetear las variables sin traer ineficiencia ni problemas de contención de memoria. El uso de un proceso *Coordinador* en lugar de la barrera de diseminación nos pareció conveniente.

```
int[] ordenar(int a[], int n) {
  int arribo[1:n] = ([n] 0), continuar[1:n] = ([n] 0);
  int aviso[1:n] = ([n] 0);
  bool cambio1 = true, cambio2 = false, salir = false;

  co [i = 1 to n] # Workers
    int sig = i + 1, j;
    if(sig>n) sig = 1;
    for [ronda = 1 to n-1] {
      if (ronda mod 2 = i mod 2)
        if(a[i]>a[sig]) {
          swap(a[i], a[sig]);
          cambio2 = true;
        }
      arribo[i] = 1;
      while(continuar[i]==0) skip;
      continuar[i] = 0;
      if(salir) break;
    }
  // # Coordinador
    while (!salir) {
      for [i = 1 to n] {
        while(arribo[i]==0) skip;
        arribo[i] = 0;
      }
      salir = (!cambio1 and !cambio2);
      cambio1 = cambio2;
      cambio2 = false;
      for [i = 1 to n] continuar[i] = 1;
    }
  oc
  return a[];
}
```

Modifique la respuesta de a) para usar k procesos; asuma que n es múltiplo de k.

---No queda claro por enunciado de qué manera los procesos intercambiarían los valores. Si intercambian la posición i de cada uno por la posición i del otro la respuesta se mantiene. Si intercambian elementos quedando el primer proceso con los menores y el segundo con los mayores la respuesta varía.

Cuestionario Clases 4 y 5

1-

a) Explique la semántica de un semáforo.

Un semáforo es utilizado para implementar exclusión mutua y sincronización por condición. Se relaciona a un valor entero y dos operaciones atómicas para manipularlo:

1. **P, wait o decremento:** Si el valor del semáforo es positivo, lo decrementa; sino demora al proceso hasta que se satisfaga dicha condición y luego decrementa el valor y continúa la ejecución. Equivale a:
   ```
   <await (valor>0) valor = valor - 1>
   ```
2. **V, signal o incremento:** Incrementa el valor del semáforo, "despertando" a alguno de los procesos demorados si los hubiere. Equivale a:
   ```
   <valor = valor + 1>
   ```

b) Indique los posibles valores finales de x en el siguiente programa (justifique claramente su respuesta):

```
int x = 3; sem s1 = 1, s2 = 0;
co   P(s1); x = x * x ; V(s1);                  # P1
  // P(s2); P(s1); x = x * 3; V(s1):            # P2
  // P(s1); x = x - 2; V(s2); V(s1);            # P3
oc
```

s1 → *mutex*: puede pasar un solo proceso por vez.
s2 → *semáforo de señalización*: esperan señalización de un evento y pasa sólo uno.

P1 y P3 comienzan esperando a s1. Por ser un mutex, sólo puede continuar uno de ellos y no será interrumpido por el otro hasta liberar a s1.

1. Si comienza P1: Asigna x=9, luego incrementa s1 permitiendo que continúe P3. P3 asigna x=7 y señala s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. En cualquier caso, P3 libera a s1 y termina. P2 es despertado, asigna x = 21 y termina. Valor final **x=21**.
2. Si comienza P3: Asigna x = 1 y señala a s2. Esto habilita a P2 que estaba esperando. Si P2 continúa, intentará obtener s1 con lo cual se vuelve a bloquear volviendo el control a P3. Cuando P3 libera a s1, P1 y P2 pueden competir por él:
   1 Si gana P1: asigna x=1, libera a s1 y termina; finaliza P2 y asigna x = 3. Valor final x=3.
2 Si gana P2: asigna x=3, libera a s1 y termina; finaliza P1 y asigna x = 9. Valor final x=9.

P2 nunca puede comenzar la historia ya que espera un semáforo de señalización que sólo P3 señala. Cualquier historia en la que P2 esté antes de P3 es inválida.
En todas las historias los semáforos terminan con los mismos valores con los que están inicializados.

2- Desarrolle una solución centralizada al problema de los filósofos, con un administrador único de los tenedores, y posiciones libres para los filósofos (es decir, cada filósofo puede comer en cualquier posición siempre que tenga los dos tenedores correspondientes).

a) Utilizando semáforos

```
int posicion[1:5];
int pedidos[1:5], libre = 1;
sem mutexCola = 1, mutexSentados = 1, pedido = 0, permiso[1:5] = ([5] 0);
bool tenedores[1:5] = ([5] false);     # todos libres inicialmente
bool administradorEspera = false;
```

```
int sentados = 0;

process Filosofo[i = 1 to 5] {
  while(true) {
    P(mutexCola);                                    # Request
    pedidos[libre] = i;
    libre = libre mod 5 +1;
    V(mutexCola);
    V(pedido);
    P(permiso[i]);

    ComerEn(posicion[i]);

    tenedores[posicion[i]] = false;                  # Release
    tenedores[posicion[i] mod 5 +1] = false;
    P(mutexSentados);
    sentados = sentados - 1;
    if(administradorEspera) {
      administradorEspera = false;
      V(YaComio);
    } else {
      V(mutexSentados);
    }

    Pensar();
  }
}

process Administrador {
  int fil, ocupado = 1, pos = 1;
  while(true) {
    P(pedido);
    fil = pedidos[ocupado];
    ocupado = ocupado mod 5 +1;
    P(mutexSentados);
    if(sentados==2) {
      administradorEspera = true;
      V(mutexSentados);
      P(YaComio);
    }
    sentados = sentados + 1;
    V(mutexSentados);
    while(tenedores[pos] or tenedores[pos mod 5 +1])
      pos = pos mod 5 +1;
    posicion[fil] = pos;
    tenedores[pos] = true;
    tenedores[pos mod 5 +1] = true;
    V(permiso[fil]);
  }
}
```

b) Utilizando monitores

```
monitor AdminTenedores {
```

```
  int sentados = 0, pos = 1;
  bool tenedores[1:5] = ([5] false);
  cond hayLugar;                                    # signal cuando sentados<2

  procedure request(int &t1, int &t2) {
    if (sentados==2) {
      wait(hayLugar);
    } else {
      sentados = sentados + 1;
    }
    while(tenedores[pos] or tenedores[pos mod 5 +1])
      pos = pos mod 5 +1;
    t1 = pos;
    t2 = pos mod 5 +1;
    tenedores[t1] = true;
    tenedores[t2] = true;
  }

  procedure release(int t1, int t2) {
    tenedores[t1] = false;
    tenedores[t2] = false;
    if(isEmpty(hayLugar)) {
      sentados = sentados - 1;
    } else {
      signal(hayLugar);
    }
  }
}
```

3-

a) Describa la técnica de Passing the Baton.

Passing the baton permite implementar sentencias await de cualquier tipo y ofrece flexibilidad para realizar modificaciones. Sentencias de las formas:

F1 : ⟨ S_i ⟩ F2 : ⟨ await (B_j) S_j ⟩

Se implementan con:

1. un mutex *e* que controla el acceso a sentencias atómicas coarse-grained.
2. un semáforo *bj* y un contador *dj* por cada sentencia await condicional, permitiendo demorar procesos hasta que *Bj* sea verdadero y contando la cantidad de procesos demorados.

Se reemplaza cada ⟨ por P(e). En el caso de await condicionales se debe controlar si no se cumple la *Bj*, se incrementa *dj*, se libera el mutex y se bloquea al proceso.

A la salida de cada sentencia atómica se intenta pasar el *bastón de mando* (*e*, el permiso exclusivo de ejecución) a algún proceso demorado, revisando las guardas y los contadores. Se reemplaza cada ⟩ por un bloque de código de la forma:

```
if (B1 and d1 > 0) {d1 = d1 - 1; V(b1)} # Pasa el bastón a algún proceso esperando por B1
[] ...
[] (Bn and dn > 0) {dn = dn - 1; V(bn)} # Pasa el bastón a algún proceso esperando por Bn
else V(e);                              # Libera el bastón, permitiendo que ingrese algún
proceso
                                        # que espera acceso a sentencias atómicas
coarse-grained
fi
```

b) ¿Cuál es su utilidad en la resolución de problemas mediante semáforos?

Ofrece una forma simple y sistemática de implementar cualquier tipo de sincronización entre procesos, con o sin condiciones, utilizando semáforos para una solución de grano fino.
Los programas resultantes son fáciles de modificar y extender.

c) ¿En qué consiste la técnica de Passing the Condition y cuál es su utilidad en la resolución de problemas con monitores?

Passing the condition consiste en que un proceso señale una variable de condición si hay procesos demorados en ella, pero sin volver verdadera a la condición asociada; o bien vuelva verdadera la condición sin señalar la variable si no hay procesos demorados.
Permite asegurar que al señalar una variable, el proceso despertado pueda continuar ejecutándose. Como el proceso que señala no vuelve verdadera a la condición, si otro proceso tomara control del monitor antes que el despertado, verá falsa la condición y pasará a la cola de espera, siendo el proceso despertado el único que podrá continuar.

d) ¿Qué relación encuentra entre passing the condition y passing the baton?

Passing the baton permite que un proceso pase a otro demorado el control (mutex *e*) y que éste sea el único que pueda ejecutar secciones exclusivas; *passing the condition* permite que un proceso avise a otro que están dadas las condiciones para que continúe y que éste sea el único que pueda continuar la ejecución de los que requieran dicha condición.
El proceso despertado por *passing the baton* se ejecutará antes que el resto de los que competían por el mutex; el proceso despertado por *passing the condition* se ejecutará antes que el resto de los que requieran la condición (tal vez tomen control del monitor, pero se bloquearán).
Aunque *passing the baton* puede utilizarse para cualquier tipo de sincronización, *passing the condition* sólo puede aplicarse cuando los procesos que ejecutan *wait* y *signal* realizan acciones complementarias entre sí (por ejemplo, uno decrementa un contador y el otro lo incrementa, o uno setea una variable a verdadero y el otro a falso).

4- Sea la siguiente solución propuesta al problema de alocación SJN (Shortest-Job-Next):

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

  procedure request(int tiempo) {
    if(not libre) wait(turno, tiempo);
    libre = false;
  }
  procedure release() {
    libre = true;
    signal(turno);
  }
}
```

a) Funciona correctamente con disciplina de señalización Signal and Continue?

NO funciona correctamente. Cuando un proceso realiza un *release*, libera el recurso y *signal(turno)* pasa al primer proceso de la cola de espera de *turno* a la cola de entrada al monitor, donde competirá por el monitor con los procesos que allí se encuentren. Si antes que logre control del monitor, otro proceso realiza un *request*, encontrará *libre* en true y utilizará el recurso (posiblemente rompiendo la política SJN, si es de duración mayor al despertado); cuando el proceso despertado tome control del monitor volvería a setear *libre* en false, quedando los dos procesos con control sobre el recurso e interfiriendo entre sí.

Para evitar esta situación se debería volver a controlar si el recurso está en uso para volver a dormir al proceso, convirtiendo al *if* del *request* en un *while*. Otra solución, sería emplear *passing the condition*:

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

  procedure request(int tiempo) {
    if(not libre) wait(turno, tiempo)
    else libre = false;
  }
  procedure release() {
    if(empty(turno)) libre = true
    else signal(turno);
  }
}
```

b) Funciona correctamente con disciplina de señalización Signal and Wait?

SÍ funciona correctamente. El problema del caso anterior no existe ya que el *signal(turno)* en el *release* pasa el control del monitor directamente al primer proceso de la cola de *turno*, si lo hubiere.

5- Modifique las soluciones de Lectores-Escritores de modo de no permitir más de 10 lectores simultáneos en la BD, y además que no se admita el ingreso a más lectores cuando hay escritores esperando.

a) Resuelva con semáforos

```
# contadores de procesos trabajando y demorados por cada tipo
int nr = 0, nw = 0, dr = 0, dw = 0;
# mutex para sección crítica y semáforos de señalización
sem e = 1, r = 0, w = 0;

process Lector [i = 1 to M] {
  while(true) {
    P(e);
    if (nw>0 or dw>0 or nr==10) {dr = dr+1; V(e); P(r);}
    nr = nr + 1;
    if (dr>0 and dw==0 and nr<10) {dr = dr - 1; V(r); }
      else V(e);
    leerBD();
    P(e);
    nr = nr - 1;
    if (nr==0 and dw>0) {dw = dw - 1; V(w); }
      elseif (dw==0 and dr>0 and nr<10) {dr = dr - 1; V(r);}
      else V(e);
  }
}

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

```
      escribirBD();
      P(e);
      nw = nw – 1;
      if (dw>0) {dw = dw - 1; V(w);}
        elseif (dr>0) {dr = dr - 1; V(r);}
        else V(e);
    }
  }
```

b) Resuelva con monitores

```
  monitor Controlador_RW {
    int nr = 0, nw = 0;
    cond okLeer;
    cond okEscribir;

    procedure pedido_leer() {
      while (nw>0)or(not empty(okEscribir))or(nr==10) wait(okLeer);
      nr = nr + 1;
      if(nr<10) { signal(okLeer); }
    }

    procedure libera_leer() {
      nr = nr - 1;
      if(nr==0)and(not empty(okEscribir)) { signal(okEscribir); }
        elseif(empty(okEscribir) and nr<10) { signal(okLeer); }
    }

    procedure pedido_escribir() {
      while (nr>0 OR nw>0) wait(okEscribir);
      nw = nw + 1;
    }

    procedure libera_escribir() {
      nw = nw - 1;
      if(not empty(okEscribir)) { signal(okEscribir); }
        else { signal(okLeer); }
    }
  }
```

6- Describa en qué consiste el problema de los “Drinking Philosophers”, y plantee al menos un esquema de solución con el mecanismo de sincronización que prefiera.

El problema de *drinking philosophers* consta de un conjunto de filósofos que pueden estar en uno de tres estados: tranquilos, sedientos o bebiendo, pasando de un estado a otro en un tiempo finito.
Se tiene también un conjunto de botellas, donde cada una es compartida a lo sumo por dos filósofos. Cada filósofo puede requerir un conjunto de botellas distinto en cada sesión, y no comienza a beber hasta no poseer todas las botellas que necesita. Dos filósofos pueden beber simultáneamente si los conjuntos de botellas que requiere cada uno son disjuntos.

SOLUCIÓN PROPUESTA.
Nuestra solución es una adaptación de la propuesta por [K. M. CHANDY and J. MISRA, 1984]. Los autores proponen una solución distribuida al problema de *dining philosophers* en base a un grafo de precedencia, siendo los filósofos los nodos y los tenedores las aristas. Se trata de un grafo dirigido acíclico, donde una

arista orientada desde *u* hasta *v* indica que *u* tiene precedencia en caso que ambos deseen utilizar el tenedor. Las operaciones realizadas sobre el grafo deben asegurar que continúe siendo acíclico.
Esta solución es usada luego como base para resolver al problema de *drinking philosophers*, incorporando el uso de tenedores como parte de la solución final.
Hay un tenedor asociado a cada botella. Un filósofo debe tener todos los tenedores que comparte (estar comiendo) antes de comenzar a solicitar las botellas que requiere en una sesión. Retiene los tenedores hasta obtener todas las botellas que precisa, y retiene las botellas que precisa hasta terminar de beber.
Los autores definen las transiciones que tienen los filósofos como comensales (a excepción de hambriento → comiendo, ya que está definida por el problema). En nuestra solución, dichas transiciones se mantienen:

Si tiene sed y esta pensando → se vuelve hambriento
Si toma todos los tenedores que comparte → pasa a comiendo
Si toma todas las botellas → vuelve a pensando -dejando sucios (libres) los tenedores-

```
const USADO = 0, EN_USO = 1; # constantes para estados de botellas y tenedores
int tieneT[1:B], teniaT[1:B], estadoT[1:B] = ([B] SUCIO);
sem mutexT[1:B] = ([B] 1);   # para modificar un tenedor -tiene, tenia, estado, pedido

int tieneB[1:B], teniaB[1:B], estadoB[1:B] = ([B] USADO);
sem mutexB[1:B] = ([B] 1);   # para modificar una botella -tiene, tenia, estado, pedido

# para que un filósofo que teniaT[b] avise al que tieneT[b] EN_USO que lo precisa
# o bien   un filósofo que teniaB[b] avise al que tieneB[b] EN_USO que la precisa
bool pedido[1:B] = ([B] false);

# para que un filósofo que teniaT[b] espere que el que tieneT[b] lo libere
# o bien   un filósofo que teniaB[b] espere que el que tieneB[b] la libere
sem espera[1:F] = ([F] 0);

# para invertir la precedencia de un filósofo al liberar recursos
sum mutexInversion = 1;

process Filosofo[f = 1 to F] {
  list bRequeridas, tCompartidos = tenedoresCompartidos(f);
  while(true) {
    #Tranquilo y Pensando
    ...              # acciones que no afectan recursos compartidos

    # Sediento → se vuelve hambriento
    bRequeridas = botellasRequeridas(f);
    Tomar(f, tCompartidos, tieneT, teniaT, estadoT, mutexT);
    # al tener todos los tenedores queda comiendo
    Tomar(f, bRequeridas, tieneB, teniaB, estadoB, mutexB);
    # al tener las botellas requeridas puede beber
    Liberar(tCompartidos, teniaT, estadoT, mutexT);
    # libera los tenedores (termina de comer) → queda pensando

    # Bebiendo
    ...              # acciones que afecten las botellas requeridas

    # Al terminar de beber vuelve a estar tranquilo
    Liberar(bRequeridas, teniaB, estadoB, mutexB);
  }
}
```

```
procedure Tomar(int f, list recursos, int tieneR[], int teniaR[], int estadoR[], sem
mutexR[]) {
  int r = first(recursos);
  while(r<>0)) {
    P(mutexR[r]);
    if(tieneR[r]<>f) {
      if(estadoR[r] == USADO) {              # tiene precedencia → lo toma
        estadoR[r] = EN_USO;
      } else {                               # no tiene precedencia → lo pide
        pedido[r] = true;
        V(mutexR[r]);
        P(espera[f]);
      }
      teniaR[r] :=: tieneR[r];  # swap
    } else {
      # si tiene el recurso, está USADO
      # es imposible que el otro se lo haya pedido (la hubiese tomado directamente)
      estadoR[r] = EN_USO;
    }
    V(mutexR[r]);
    r = next(recursos);
  }
}

procedure Liberar(list recursos, int teniaR[], int estadoR[], sem mutexR[]) {
  P(mutexInversion);
  int r = first(recursos);
  while(r<>0)) {
    P(mutexR[r]);
    if(pedido[r]) {
      pedido[r] = false;
      V(espera[ teniaR[r] ]);
    } else {
      estadoR[r] = SUCIO;
      V(mutexR[r]);
    }
    r = next(recursos);
  }
  V(mutexInversion);
}
```

CONDICIONES DE INICIALIZACIÓN.
Si *u* y *v* comparten *t*, *tieneT[t]=u* y *teniaT[t]=v*, el tenedor esta usado -sucio-, no está pedido y tiene precedencia *v* sobre *u*. El grafo de precedencia debe ser acíclico (este requisito debe contemplarse al implementar la solución en un escenario real). Además, *tieneB[] = tieneT[]*, *teniaB[] = teniaT[]* y las botellas están usadas.

En la solución propuesta por los autores, se sugiere el uso de un token de pedido que indica cuando un proceso puede solicitar un tenedor a su vecino, o si debe enviárselo. Aunque ausente explícitamente en nuestra solución, puede asumirse que inicialmente al token de pedido lo tiene *v*. Si un recurso está pedido, al request token lo tiene quien tiene al recurso; si el recurso no está pedido, lo tiene quien tenía antes al recurso.

7- Resuelva con monitores el siguiente problema:

Tres clases de procesos comparten el acceso a una lista enlazada: *searchers*, *inserters* y *deleters*. Los searchers sólo examinan la lista, y por lo tanto pueden ejecutar concurrentemente unos con otros. Los inserters agregan nuevos ítems al final de la lista; las inserciones deben ser mutuamente exclusivas para evitar insertar dos ítems casi al mismo tiempo. Sin embargo, un inserter puede trabajar en paralelo con uno o más searchers. Por último, los deleters remueven ítems de cualquier lugar de la lista. A lo sumo un deleter puede acceder la lista a la vez, y el borrado también debe ser mutuamente exclusivo con searchers e inserters.

```
monitor Acceso_Lista {
# BAD:     nI>1   OR (nD>0  AND (nS>0  OR  nI>0 ))  OR nD>1
# NOT BAD: nI<=1 AND (nD==0 OR  (nS==0 AND nI==0)) AND nD<=1
  cond okSearch;    # signal cuando nD==0
  cond okInsert;    # signal cuando nD==0 AND nI==0
  cond okDelete;    # signal cuando nS==0 AND nI==0 AND nD==0
  int nS=0, nI=0, nD=0;

  procedure pedidoSearch() {
    while(nD>0) wait(okSearch);
    nS = nS + 1;
  }
  procedure liberaSearch() {
    nS = nS - 1;
    if(nS==0 AND nI==0) signal(okDelete);
  }

  procedure pedidoInsert() {
    while(nI==1 or nD>0) wait(okInsert);
    nI = nI + 1;
  }
  procedure liberaInsert() {
    nI = nI - 1;
    if(nS==0) signal(okDelete);
    signal(okInsert);
  }

  procedure pedidoDelete() {
    while(nD==1 or nS>0 or nI>0) wait(okDelete);
    nD = nD + 1;
  }
  procedure liberaDelete() {
    nD = nD - 1;
    signal_all(okSearch);
    signal(okInsert);
    signal(okDelete);
  }
}
```

8- (Broadcast atómico). Suponga que un proceso productor y n procesos consumidores comparten un buffer unitario. El productor deposita mensajes en el buffer y los consumidores los retiran. Cada mensaje depositado por el productor tiene que ser retirado por los n consumidores antes de que el productor pueda depositar otro mensaje en el buffer.

a) Desarrolle una solución usando semáforos

```
sem okLeer = 0, okEscribir = 1, barrera = 0, mutex = 1;
int cont = 0;
int BD;              # datos compartidos

process Productor {
  while(true) {
    P(okEscribir);
    BD = CalcularValor();
    for[c = 1 to N] V(okLeer);
  }
}

process Consumidor[i = 1 to N] {
  while(true){
    P(okLeer);
    Procesar(BD);

    #esperan a haber leido todos
    P(mutex);
    cont = cont + 1;
    if(cont == N)
      for[c = 1 to N] V(barrera);
    V(mutex);
    P(barrera);

    # una vez que leyeron todos, uno avisa al productor (preciso esperar que
    # todos salgan de la barrera para evitar race conditions)
    P(mutex);
    cont = cont - 1;
    if(cont==0) V(okEscribir);
    V(mutex);
  }
}
```

b) Suponga que el buffer tiene b slots. El productor puede depositar mensajes sólo en slots vacíos y cada mensaje tiene que ser recibido por los n consumidores antes de que el slot pueda ser reusado. Además, cada consumidor debe recibir los mensajes en el orden en que fueron depositados (note que los distintos consumidores pueden recibir los mensajes en distintos momentos siempre que los reciban en orden). Extienda la respuesta dada en (a) para resolver este problema más general.

```
sem okLeer[1:N] = ([N] 0), okEscribir = b, mutex[1:b] = ([b] 1);
int cont[1:b] = ([b] 0);
int BD[1:b];              # datos compartidos

process Productor {
  int libre = 1;
  while(true) {
    P(okEscribir);
    cont[libre] = 0;
    BD[libre] = CalcularValor();
    for[c = 1 to N] V(okLeer[c]);
    libre = libre mod b +1;
```

```
  }
}

process Consumidor[i = 1 to N] {
  int consumir = 1;
  while(true){
    P(okLeer[i]);
    Procesar(BD[consumir]);
    P(mutex[consumir]);
    cont[consumir] = cont[consumir] + 1;
    if(cont[consumir] == N) V(okEscribir);
    V(mutex[consumir]);
    consumir = consumir mod N +1;
  }
}
```

En la resolución del punto *a* utilizamos una barrera para sincronizar a los consumidores. El contador era incrementado antes de la barrera y decrementado luego para evitar *race conditions*. Al tener que extender la resolución, notamos que el uso de semáforos privados facilitaría la tarea, logrando el mismo efecto. Se pudieron utilizar también en el punto *a* simplificando el código (aunque con mayor uso de memoria por el arreglo de semáforos).

c) OPCIONAL: resuelva a) y b) usando monitores.

a) Broadcast con buffer unitario

```
monitor BD {
  int dato, cont = N, leido[1:N] = ([N] 1);
  cond okEscribir; # signal cuando leido[] sea todos 1
  cond okLeer;     # signal cuando leido[] sea todos 0

  procedure escribir(int d) {
    if(cont<>N) wait(okEscribir);
    dato = d;
    cont = 0;
    for [c = 1 to N] leido[c] = 0;
    signal_all(okLeer);
  }

  procedure leer(int pid, int &d) {
    if(leido[pid] == 1) wait(okLeer);
    d = dato;
    cont = cont + 1;
    if(cont == N) signal(okEscribir);
  }
}
```

b) Broadcast con buffer de b posiciones

```
monitor BD {
  int dato[1:b], cont[1:b]=([b] N), consumir[1:N]=([N] 1), libre=1;
  cond okEscribir;   # signal cuando cont[libre] = N
  cond okLeer;       # signal cuando hay dato nuevo

  procedure escribir(int d) {
```

```
    if(cont[libre]<>N) wait(okEscribir);
    dato[libre] = d;
    cont[libre] = 0;
    signal_all(okLeer);
    libre = libre mod b +1;
  }

  procedure leer(int pid, int &d) {
    if(cont[ consumir[pid] ] == N) wait(okLeer);
    d = dato[ consumir[pid] ];
    cont[ consumir[pid] ] = cont [ consumir[pid] ] + 1;
    if(cont[ consumir[pid] ] == N) signal(okEscribir);
    consumir[pid] = consumir[pid] mod b +1;
  }
}
```

9- (OPCIONAL)

a) Implemente una butterfly barrier de n procesos usando semáforos (siendo n potencia de 2)

```
sem arribo[1:n] = ([n] 0);
process Worker[i = 1 to n] {
  int j, resto, distancia;
  while (true) {
    ...                                                    # codigo de tarea i
    # barrera
    for [s = 1 to log2(n)] { # calcula la distancia y si debe sumarla o restarla

      distancia = 2^(s-1);
      # (0 < i mod 2^s <= distancia) → suma distancia, sino resta
      resto = i mod 2^s;
      if (resto=0)or(resto>distancia) distancia = -distancia;
      j = i + distancia;
      V(arribo[i]);
      P(arribo[j]);
    }
  }
}
```

b) Utilice la barrera implementada en a) para resolver el algoritmo odd/even exchange sort para ordenar un arreglo de enteros a[1:n] en forma ascendente

```
int[] ordenar(int a[], int n) {
  sem arribo[1:n] = ([n] 0);
  co [i = 1 to n] {
    int sig = i + 1, j, distancia, resto;
    if(sig>n) sig = 1;
    for [ronda = 1 to n-1] {
      if (ronda mod 2 = i mod 2)
        if(a[i]>a[sig]) swap(a[i], a[sig]);
      # barrera
      for [s = 1 to log2(n)] {
        distancia = 2^(s-1);
        resto = i mod 2^s;
```

```
        if (resto=0)or(resto>distancia) distancia = -distancia;
        j = i + distancia;
        V(arribo[i]);
        P(arribo[j]);
      }
    }
  }
  return a[];
}
```

c) Modifique la respuesta de a) para usar k procesos; asuma que n es múltiplo de k

```
int[] ordenar(int a[], int n) {
  sem arribo[1:k] = ([k] 0);
  co [p = 1 to k] {
    int i, sig, j, distancia, resto;
    for [ronda = 1 to n-1] {
      for[ele = 1 to n/k] {
        i = ele + (p-1)*n/k;
        sig = i + 1;
        if(sig>n) sig = 1;
        if (ronda mod 2 = i mod 2)
          if(a[i]>a[sig]) swap(a[i], a[sig]);
      }
      # barrera
      for [s = 1 to log2(k)] {
        distancia = 2^(s-1);
        resto = i mod 2^s;
        if (resto=0)or(resto>distancia) distancia = -distancia;
        j = i + distancia;
        V(arribo[i]);
        P(arribo[j]);
      }
    }
  }
  return a[];
}
```

Cuestionario Clases 6 a 9

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

1. **Concurrente:** Programa formado por múltiples procesos relativamente independientes. No implica ninguna arquitectura específica. Puede haberse desarrollado así ya sea porque era la forma más natural de atacar al problema, porque la arquitectura forzaba a hacerlo o porque se buscó un mejor desempeño que en una solución secuencial. De las tres, es la clasificación más general.
2. **Distribuido:** Programa concurrente en el que los procesos se ejecutan en distintos procesadores -al menos más de uno- y se comunican por *pasaje de mensajes, RPC o Rendevouz*. Supone una arquitectura de memoria distribuida aunque esto no es limitante.
3. **Paralelo:** Programa implementado de manera concurrente para buscar mayor performance, velocidad o precisión que las soluciones secuenciales, aprovechando una arquitectura base multiprocesador. Puede utilizar *memoria compartida o distribuida*. Quizá el problema no es de naturaleza concurrente, quizá el entorno de ejecución tampoco es distribuido, sin embargo aplicar este enfoque a la resolución de un problema puede ofrecer mejor desempeño. Puede tratarse de una solución secuencial "adaptada" en la que se encontró cierto paralelismo de datos o de tareas, o de un enfoque completamente distinto al problema.

2- Analice qué tipo de mecanismos de pasaje de mensajes son más adecuados para resolver problemas de tipo Cliente/Servidor, Pares que interactúan, Filtros, y Productores y Consumidores. Justifique claramente su respuesta.

La comunicación/sincronización en un entorno distribuido puede darse por cuatro métodos distintos: *PMA, PMS, RPC o Rendevouz*. Todos pueden aplicarse en cualquier caso aunque algunos son más adecuados que otros según el patrón del problema:

1. **Cliente/Servidor** → *RPC o Rendevouz*: Ofrecen canales de comunicación bidireccionales, apropiados para hacer un requerimiento y obtener una respuesta en la misma instrucción. Se preferirá RPC si es posible atender al mismo tiempo más de un cliente -de ser necesario sincronizando servidores entre sí por EM o SxC-, o Rendevouz si la atención es de a uno. RPC podría sufrir cierta sobrecarga por la creación de servidores si hay múltiples clientes haciendo requerimientos; en esta situación Rendevouz permitiría crear un pool de procesos servidores y mantener fijo su número. De usar PM sería necesario hacer un juego send/receive por cada requerimiento; con PMA podrían compartir el canal de request todos los clientes, pero en PMS los canales son punto a punto por lo que se requeriría una mayor cantidad. PMS reduciría además la concurrencia cuando el cliente deba indicar algo -request/release- al servidor pero sin esperar una respuesta; PMA podría ser buena opción al no afectar la concurrencia de ese modo, y si ese tipo de comunicación fuese mayoritario sería preferible a RPC/Rendevouz -ya que tienen una semántica sincrónica-.
2. **Pares que interactúan** → *PMA*: Ofrece más concurrencia y permite compartir canales. Con PMS la concurrencia se limita, se hacen necesario procesos buffers para salvar este problema, y se corren mayores riesgos de deadlock. Frecuentemente los pares sólo precisan compartir información y no sincronizarse. Rendevouz/RPC también limitarían la concurrencia; RPC crearía nuevos procesos entre los peers deformando el esquema, y se precisa programar la sincronización con alguna otra técnica.
3. **Filtros** → *PMA*: Ofrece más concurrencia ya que la salida de cada proceso no sería bloqueante, es decir envía el resultado de su procesamiento sin preocuparse de si en ese momento el siguiente proceso puede aceptarla, y continúa de inmediato procesando su propia entrada. Se evitan demoras innecesarias aprovechando la semántica FIFO del canal. Se requiere comunicación más que sincronización. De utilizar PMS/Rendevouz deberían incluirse buffers explícitos para aumentar la concurrencia y de usar RPC se crearían procesos servidores constantemente con cada salida, ya que la idea del filtro es ofrecer un stream de datos, afectando el desempeño.
4. **Productores y Consumidores** → *PMA*: Vale lo explicado en Filtros.x|

3- Considere el problema de lectores/escritores. Desarrolle un proceso servidor para

implementar la base de datos, y muestre las interfases de los lectores y escritores con el servidor. Los procesos deben interactuar:

con mensajes asincrónicos.

```
chan requestLectura(int idL), requestEscritura(int idL),
     releaseLectura, releaseEscritura,
     permisoLectura[1:L], permisoEscritura[1:E];

Process BD {
  int qL=0, qE=0, id;
  while(true) {
    if(qE==0 AND !empty(requestLectura)) →
        receive requestLectura(id);
        send permisoLectura[id];
        qL=qL+1;
    [] if(qL==0 AND qE==0 AND !empty(requestEscritura)) →
        receive requestEscritura(id);
        send permisoEscritura[id];
        qE=qE+1;
    [] if(!Empty(releaseLectura)) →
        receive releaseLectura;
        qL=qL-1;
    [] if(!Empty(releaseEscritura)) →
        receive releaseEscritura;
        qE=qE-1;
    fi
  }
}

process Lector[i=1 to L] {
  …
  send requestLectura(i);
  receive permisoLectura[i];
  # lee la base de datos
  send releaseLectura;
  …
}

process Escritor[i=1 to E] {
  …
  send requestEscritura(i);
  receive permisoEscritura[i];
  # escribe la base de datos
  send releaseEscritura;
  …
}
```

con mensajes sincrónicos

```
# utilizando CSP:
Process BD {
  int qL=0, qE=0, id;
  while(true) {
    if(qE==0; Lector[*]?request) → qL=qL+1;
```

```
      [] if(qL==0 AND qE==0; Escritor[*]?request) → qE=qE+1;
      [] if(Lector[*]?release) → qL=qL-1;
      [] if(Escritor[*]?release) → qE=qE-1;
      fi
    }
  }

  process Lector[i=1 to L] {
    …
    BD!request;
    # lee la base de datos
    BD!release;
    …
  }

  process Escritor[i=1 to E] {
    …
    BD!request;
    # escribe la base de datos
    BD!release;
    …
  }
```

con RPC o Rendezvous a elección

```
  module BD # Utilizando RPC
    op requestLectura;
    op requestEscritura;
    op releaseLectura;
    op releaseEscritura;
  body
    int qL=0, qE=0;

    # sincronización por semáforos (passing the batton)
    sem okLeer = 0, okEscribir = 0, e=1;
    int dL=0, dE=0;

    # implementacion
    proc requestLectura {
      P(e);
      if(qE>0) {
        dL=dL+1;
        V(e);
        P(okLeer);
      }
      qL=qL+1;
      if(dL>0) {
        dL=dL-1;
        V(okLeer);
      } else V(e);
    }

    proc requestEscritura {
      P(e);
```

```
    if(qL>0 or qE>0) {
      dE=dE+1;
      V(e);
      P(okEscribir);
    }
    qE=qE+1;
    V(e);
  }

  proc releaseLectura {
    P(e);
    qL=qL-1;
    if(qL==0 AND dE>0) {
      dE=dE-1;
      V(okEscribir);
    } else V(e);
  }

  proc releaseEscritura{
    P(e);
    qE=qE-1;
    if(dE>0) {
      dE=dE-1;
      V(okEscribir);
    } else if(dL>0) {
      dL=dL-1;
      V(okLeer);
    } else V(e);
  }
end BD;

process Lector[i=1 to L] {
  …
  call BD.requestLectura();
  # lee la base de datos
  call BD.releaseLectura();
  …
}

process Escritor[i=1 to E] {
  …
  call BD.requestEscritura();
  # escribe la base de datos
  call BD.releaseEscritura();
  …
}
```

4- Modifique la solución con mensajes sincrónicos de la Criba de Eratóstenes para encontrar los números primos vista en teoría de modo que los procesos no terminen en deadlock.

```
process Worker[i = 1] {
  int p = 2;
  for [t = 3 to N by 2]
     worker[2]!t;
```

```
  print p;       # Imprime el primo i-ésimo
  worker[2]!0;   # Envía la marca EOS al siguiente
}

process Worker[i = 2 to L] {
  int p, t=1;
  worker[i-1]?p; # Recibe el primo i-ésimo
  while(t!=0) {
    worker[i-1]?t;
    if(i<L)and(t mod p != 0) worker[i+1]!t;
  }
  print p;
  if(i<L) worker[i+1]!0;
}
```

El valor de *L* debería ser suficientemente grande para poder detectar todos los números primos entre 2 y *N*. De implementarse en un lenguaje que permita la creación dinámica de procesos podrían crearse a demanda -la primera vez que un worker tenga un dato que pase su filtro, crea al worker siguiente y se lo pasa-.

5- Describa el paradigma “bag of tasks”. ¿Cuáles son las principales ventajas del mismo?

El paradigma *bag of tasks* indica un modo de organizar un programa paralelo en el que procesos workers comparten una bolsa de tareas independientes a realizar. Cada worker se mantiene en un ciclo en el que toma una tarea de la bolsa, la realiza y de ser necesario crea nuevas tareas. El problema general se resuelve cuando la bolsa está vacía y todos los workers están a la espera de una tarea.
El número de tareas puede ser fijado inicialmente -propio de paralelismo iterativo- o puede ser sólo una tarea inicial, con generación dinámica de nuevas unidades de trabajo -propio de paralelismo recursivo-.
El tamaño de las tareas y su cantidad en relación a la cantidad de workers influirá en el desempeño de la solución (demasiados workers con tareas muy chicas pueden convertir a la bolsa en un cuello de botella, por ejemplo).
Ventajas: Es simple. Permite una división dinámica de la carga de trabajo, si un proceso demora en realizar una tarea entonces posiblemente termine resolviendo menor cantidad de tareas que el resto, pero no necesariamente los retrasará. Es escalable, permitiendo que el programa pueda aprovechar una mayor cantidad de procesadores sin mayores cambios en el código, sólo creando nuevos workers.

6- Describa sintéticamente las características de sincronización y comunicación de Linda, Occam, Java, Ada, MPD y MPI.

Linda -biblioteca-

Define un conjunto de funciones que permiten el envío asincrónico (OUT), recepción (IN), consulta sin extraer el mensaje del canal (RD), creación dinámica de procesos (EVAL) y la recepción o consulta no bloqueantes (INP y RDP). Puede trabajar tanto sobre MC como MD.
Maneja un único canal o *Espacio de Tuplas* compartido, que mantiene tuplas con nombre. Una tupla está representada por un nombre y un conjunto de valores. OUT permite crear tuplas de datos -pasivas- y EVAL permite la creación de procesos concurrentes -tuplas activas- que resultarán al terminar su ejecución en una tupla de dato con los resultados. Las primitivas de recepción (IN, INP, RD, RDP) deben indicar el nombre de la tupla a leer, y opcionalmente valores fijos a machear con los de la tupla o variables precedidas por el operador *?* a instanciar con valores de la tupla seleccionada. Las características bloqueantes del IN y RD permiten utilizarlos para sincronización -esperando hasta que algún proceso cree una tupla que machee-.

Occam -lenguaje-

Implementa CSP -*Communicating Sequential Process*-, por lo que utiliza PMS con canales globales 1:1. Permite el mapeo proceso-procesador, orientado a *transputers* (en general, máquinas MIMD). Permite indicar el modo de ejecución de las sentencias mediante constructores: secuencialmente (SEQ), concurrentemente (PAR) o indicando prioridades (PRI PAR) o procesador (PLACED PAR). Permite comunicación guardada (ALT, PRI ALT).
Las sentencias en un mismo constructor pueden usar VC, pero si están en distintos constructores deben usar canales con nombrado estático y PMS. Una declaración de canal tiene la forma: CHAN OF nombreProtocolo, donde el *nombreProtocolo* define el tipo de valores que son transmitidos por el canal (tipos básicos, arreglos de longitud fija o variable, registros fijos o variantes). Los canales son accedidos con primitivas de entrada (?) y salida (!). Se incluye también un tipo especial de canal tipo *timer* que permite consultar la hora o demorar un proceso hasta una hora indicada.

Java -lenguaje-

Java soporta tanto aplicaciones *multithread* como *distribuidas*.
Permite la creación dinámica de procesos instanciando subclases de *Thread*, o alguna clase que implemente la interfaz *Runneable*. Para memoria compartida soporta una versión reducida del concepto de monitor: la definición de métodos o bloques de código con el modificador *synchronized* permite indicar que el contenido se ejecutará con exclusión mutua a nivel de objeto; es decir, hay un único *lock* por objeto y al ejecutar un método/bloque *synchronized* debe tenerse control del mismo o se bloqueará el thread hasta obtenerlo. También soporta, en lugar de variables de condición, una única cola de espera por objeto, y los métodos *wait*, *notify* y *notifyAll*, con una política *Signal and Continue*. Si un método/bloque *synchonized* invoca a otro en otro objeto se retiene el lock, pudiendo llevar a casos de deadlock.
Para programación distribuida se soporta tanto PMA como RPC. El PMA está soportado por el paquete *java.net* que implementa las clases *Socket* y *ServerSocket* necesarias para comunicaciones TCP -leídas/escritas como streams de datos-, así como *DatagramSocket* y *DatagramPacket* para UDP -con send/receive-, con las diferencias de seguridad/velocidad propios de estos protocolos de red.
La versión soportada de RPC se denomina RMI -*Remote Method Invocation*, dado que es OO- y es soportada por los paquetes *java.rmi* y *java.rmi.server*. La construcción de una aplicación con RMI incluye la definición de una interfaz derivada de *Remote* que especifique los métodos exportados, la clase del servidor -derivada de *UnicastRemoteObject*- que implemente la interfaz y la clase del cliente que lo invoque a través de un "nombre de servicio" que debe ser registrado por el servidor o desde línea de comando por el administrador. El registro y la búsqueda del servicio se realiza por medio de los métodos *register* y *lookup* del objeto *Naming*. La implementación utiliza dos objetos extras en forma transparente al programador: un **objeto stub** en el cliente, que envía las invocaciones remotas empaquetando los parámetros y enviándolos por red, luego recibe los resultados, los desempaqueta y los entrega al cliente, y un **objeto skeleton** en el servidor, que recibe las invocaciones remotas, desempaqueta los parámetros, realiza la invocación localmente recibiendo los resultados, empaquetándolos y enviándolos al stub del cliente.

Ada -lenguaje-

Organiza al programa como un conjunto de tareas concurrentes (TASK), definiendo cada una distintos puntos de invocación (ENTRY). El cuerpo de la tarea deberá aceptar la comunicación desde otros procesos, y podrá también realizar llamados. Utiliza *Rendevouz*. Se asocia a cada entry un conjunto opcional de parámetros que permiten manejar el canal de modo bidireccional -según sean de entrada, salida o entrada/salida-.
Tanto las llamadas como las recepciones de mensajes se pueden hacer en estructuras SELECT convirtiéndose en condicionales. Permite cláusulas ELSE y OR DELAY para indicar comportamientos por defecto si no se logra la comunicación inmediata o en un lapso indicado, y OR TERMINATE para terminar si no se puede servir ningún llamado y todas las tareas que hacen *rendevouz* con la actual ya han terminado o se encuentran a su vez en una sentencia ON TERMINATE. En el caso de recepción -ACCEPT-, pueden indicarse varias juntas en un SELECT y utilizar guardas (WHEN), soportando comunicación guardada. Aunque SELECT es similar a la forma general de rendevouz (*in*), no soporta expresiones de scheduling y los parámetros de la llamada no pueden utilizarse en las expresiones condicionales de la guarda.
Es posible terminar explícitamente una tarea (ABORT) o consultar si se la puede invocar (atributo *callable*).

Las tareas pueden acceder a datos globales. Es posible declarar *tipos protegidos* que encapsulen datos, procedimientos, funciones y entry's comportándose como un monitor (las funciones sí pueden ejecutarse concurrentemente sin exclusión ya que no pueden afectar variables no locales a su cuerpo, las entrys protegidas pueden incluir una cláusula WHEN que indique SxC). Es posible encolar un llamado que está siendo servido (REQUEUE).

MPD -lenguaje- (http://www.cs.arizona.edu/mpd/)

MPD es una variante C-like de SR (*Sinchronizing Resources*, Pascal-like), con las mismas capacidades.
Los programas se construyen a partir de recursos y globales. Los recursos son el patrón para un módulo con datos locales, operaciones exportadas, código de inicialización/finalización, procedures y procesos; pueden crearse/destruirse instancias locales o remotas dinámicamente y pasarle parámetros. Las globales son instancias simples no parametrizadas.
Ofrece todos los mecanismos de comunicación/sincronización vistos, excepto monitores (aunque pueden construirse con procedures+semáforos). Procesos en un mismo recurso ejecutan concurrentemente y pueden acceder a VC y globales; siempre puede usarse PMA, RPC y Rendevouz.
Las operaciones se declaran con *op*. Proporciona cuatro primitivas: dos para invocar (*call* -sincrónico- y *send* -asincrónico-) y dos para servir las operaciones (*in* -sentencia de entrada- y *proc* -crea un nuevo proceso si el llamado es remoto-). Todos los mecanismos son alcanzables gracias a la ortogonalidad entre estas primitivas.
In soporta *rendevouz*, puede incluir guardas y expresiones de scheduling y una cláusula *else*.
Proc soporta RPC si se invoca con *call*, y puede utilizar *reply* para retornar valores a su llamador sin detener su ejecución, o *forward* para pasar la invocación a otro proceso que la sirva, bloqueándose mientras tanto.
process es una abreviación de declaración *op+proc* para servirla, creando un proceso al crearse el recurso. Pueden definirse arrays de procesos.
procedure equivale a un *op+proc* para servir invocaciones de la operación.
receive abrevia *op+in* que sólo almacena los argumentos en variables locales.
sem abrevia una *op* sin parámetros, siendo P un caso especial de *receive* y V uno de *send*.
Co permite crear recursos o invocar operaciones en paralelo.

MPI -biblioteca-

Los procesos son de tipo SCMD. La biblioteca debe ser inicializada y finalizada explícitamente (*MPI_Init* y *MPI_Finalize*). Provee primitivas de envío (sincrónico y asincrónico) y recepción de mensajes (sincrónico y de tipo polling -no bloqueante-), desde un proceso particular o desde cualquiera dentro de un grupo. Los grupos de procesos se pueden crear y manejar dinámicamente, pudiendo utilizar primitivas de sincronización en barrera (*MPI_Barrier*), comunicación broadcast (*MPI_Bcast*), distribuir elementos de un array entre procesos o esperar mensajes desde los procesos del grupo y reunirlos en un array (*MPI_Scatter* y *MPI_Gather*), realizar alguna operación entre los valores recibidos de otros procesos -sumar, buscar el máximo o mínimo...-, entre otras.

7- Explique brevemente los 7 paradigmas de interacción entre procesos vistos en teoría. A su criterio, cuál de ellos resulta más adecuado para resolver problemas de broadcast entre procesadores, cuando no se conoce la topología de la red. Justifique claramente su respuesta.

1. **Servidores Replicados:** Varios procesos servidores manejan recursos compartidos, sirviendo solicitudes de múltiples clientes. Puede haber un servidor por cliente o un *pool* de servidores. Se utilizan si hay múltiples instancias del recurso (de modo que cada servidor administre una) o si hay varios recursos y se desea dar a los clientes la impresión de que sólo hay uno.
2. **Algoritmos Heartbeat:** Procesos que intercambian información haciendo *send* de sus datos y luego un *receive* desde sus vecinos. Útil para paralelizar soluciones iterativas (cada worker resuelve un paso, pero precisa los resultados de sus vecinos para el paso siguiente, así que comunica los resultados y espera los de ellos).
3. **Algoritmos Pipeline:** Una serie de procesos conectados en pipe/tubería, por los que fluyen los datos mediante *send/receive*. Cada uno es un productor/consumidor -filtro-. Puede variar la forma de

interconexión (lineal, circular o cerrada -con coordinador de realimentación W_n a W_1-), y pueden manejarse distintos datos o funciones en cada proceso.

4. **Probes & Echos -sondas y ecos-:** Procesos que trabajan en una topología de grafo. Un proceso puede diseminar/juntar información haciendo *sends* en paralelo a sus vecinos y esperando que estos lo propaguen, y eventualmente le envíen una respuesta. El recorrido de la sonda es análogo a un DFS del grafo, en paralelo por las distintas ramas. Si no hay información de la topología completa los *sends* se hacen a todos los vecinos, pero puede mejorarse si se cuenta con esa información calculando antes el *árbol mínimo de expansión*. En el primer caso se puede aprovechar el eco para construir una representación de la topología. Además, es posible debido a la presencia de ciclos, que un nodo reciba el probe desde más de un vecino; en dicho caso podría sólo responder al primero con la respuesta precisa, y enviar ecos nulos -sin datos de respuesta- al resto.
5. **Algoritmos Broadcast:** Permiten diseminar información en una arquitectura distribuida permitiendo tomar decisiones descentralizadas. Puede utilizar una primitiva *broadcast* disponible en algunas arquitecturas de red -token ring, ethernet-, que encola los mensajes en todos los canales de un arreglo. Como esta operación NO es atómica, distintos procesos podrían recibir mensajes de broadcast con distinto origen en distinto orden.
6. **Token Passing:** El envío de información global y/o la toma de determinadas decisiones se basan en el uso de tokens enviados entre los procesos. La topología no necesariamente debe ser de anillo ni los tokens globales.
7. **Manager/workers:** Un proceso controla los datos a procesar o las tareas a realizar, mientras otros procesos se comunican con él para que les asigne un trabajo. Implementa *bag of tasks* de manera distribuida.

En problemas de broadcast sin conocimientos sobre la topología, como solución general sería conveniente *Probe & Echo*, ya que permite el envío de mensajes a todos los nodos sin importar la arquitectura de red -podría obviarse el *echo* si no se requieren respuestas, dependiendo del problema-.
Los algoritmos de broadcast podrían utilizarse pero sólo si la primitiva broadcast es soportada para la red en cuestión, siendo preferible en esos casos.
Manager/workers y servidores replicados no aplican, están orientados a problemas diferentes. Token passing serviría para topologías de anillo pero no sería la mejor opción para grafos generales.
Heartbeat y Pipeline son más comunes en programas paralelos. Con pipeline los datos fluyen en una dirección y podría no ser posible hacer broadcast. Con heartbeat podrían compartirse datos con los vecinos y luego de varias iteraciones sería de conocimiento global, pero sin datos de la topología no se sabría cuántas iteraciones son necesarias, y obviamente la convergencia sería más lenta. Además el tipo de problemas en los que se aplican estos algoritmos raramente precisarán broadcasts.

8- Investigue el problema de detección de terminación en programación distribuida. Plantee soluciones para el caso de procesos conectados en anillo y en grafos.

Aún cuando ningún proceso se encuentre trabajando en un momento dado, puede haber mensajes “en tránsito” que los despierten luego. Por esto y porque normalmente ningún proceso puede ver el estado global (salvo quizá en casos en los que un coordinador controle el avance de workers y les indique cuándo terminar), la detección de terminación es compleja en un ambiente distribuido.
EN PROCESOS CONECTADOS EN ANILLO podría utilizarse el paradigma de *token passing*, incluyendo un token de control que circule por los procesos cuando se encuentren inactivos. Así, si el token da una vuelta completa y regresa a su emisor habiendo este permanecido inactivo todo el tiempo, se habrá detectado la terminación. Sino, se reiniciará el control de terminación.
Inicialmente y cada vez que reciba un mensaje de datos, un proceso se considerará ACTIVO. Al terminar el procesamiento e ingresar a una sentencia *receive* se vuelve inactivo. Si P[1] comienza el control, al volverse inactivo por primera vez enviará el token y fijará su estado como INACTIVO. Cuando el próximo proceso en el anillo lo reciba, se deberá a que él mismo se encuentra en una sentencia *receive* (estará inactivo), y reenviará el token. Si el token regresa a P[1] estando este INACTIVO desde que lo envió, habrá detectado la terminación, la anunciará al resto y finalizará.

```
enum tTipo = (DATOS, TOKEN, EOP);   # EOP = End Of Processing
```

```
Chan c[1:N](tTipo tipo, tDatos msg);

Process P[i = 1] {
  enum tEstado = (INICIAL, ACTIVO, INACTIVO);
  tTipo tpo; tDatos msg;
  tEstado estado = INICIAL;
  bool seguir = TRUE;
  while(seguir) {
    receive c[1](tpo, msg);
    if(tpo == DATOS) {
      Procesar(msg);
      send c[2](DATOS, msg);
      if(estado==INICIAL) { # tras procesar datos la 1° vez se vuelve inact.
        estado = INACTIVO;
        send c[2](TOKEN, Null);
      } else estado = ACTIVO;
    } else if(tpo == TOKEN) {
      if(estado == INACTIVO) seguir = false
      else {
        estado = INACTIVO;
        send c[2](TOKEN, Null);
      }
    }
  }
  send c[2](EOP, Null);   # avisa terminación
  receive c[1](tpo, msg); # recibe msj de P_N, fin total del programa
}

Process P[i = 2 to N] {
  tTipo tpo = DATOS; tDatos msg;
  while(tpo<>EOP) {
    receive c[i](tpo, msg);
    if(tpo == DATOS) Procesar(msg);
    send c[i mod N + 1](tpo, msg); # propaga datos, tokens y terminación
  }
}
```

EN PROCESOS CONECTADOS EN GRAFO el esquema anterior se complica ya que un proceso podría recibir mensajes desde más de una fuente. Si una de ellas comenzara el algoritmo de detección de terminación y le enviara un token, el proceso lo distribuiría siguiendo el algoritmo, pero aún así podría luego recibir un mensaje de datos desde alguna de las otras fuentes, reactivándose. Así, la vuelta del token al emisor original no implica que todos los procesos estén inactivos.
Podría adaptarse la técnica si se impone un ciclo C -un "anillo"- que involucre a todos los nodos del grafo G, quizá incluyendo a alguno más de una vez para armar su recorrido (por lo que C_n, la longitud del ciclo, podría ser mayor a N, la cantidad de procesos). Sin embargo, como el emisor del token podría aparecer en más de una oportunidad en el ciclo, el regreso del token sigue sin indicar la terminación. Esto se soluciona utilizando al token como un contador de procesos inactivos (se inicia en cero, y cada proceso lo incrementa al reenviarlo al próximo en C), y así la llegada del token con valor C_n a un proceso inactivo -su emisor original- permitiría detectar la terminación; si en cambio al llegar el token con valor C_n el proceso se encuentra activo, reiniciaría su control de terminación. El algoritmo podría ser iniciado por cualquier proceso al volverse inactivo por primera vez, enviando un token en 0.
Al detectar la terminación se podría avisar a todos mediante broadcast, o continuar anunciando por C: cada proceso reenvía el aviso y termina, excepto aquellos que se repiten luego en C. El último proceso del ciclo terminará directamente sin reenviar -no habrá nadie a quien entregarle el mensaje-.

```
enum tTipo = (DATOS, TOKEN);
union tMsg {
  tDatos data; # datos de mensajes comunes
  int valor;   # valor del contador del token
}
chan c[1:N](tTipo tipo, tMsg msg);

Process P[i = 1] {
  enum tEstado = (INICIAL, ACTIVO, INACTIVO);
  tTipo tpo; tMsg msg; int indiceEntrada;
  tEstado estado = INICIAL;
  bool seguir = TRUE;
  while(seguir) {
    # P[i] tiene varias aristas de entrada en G puede recibir de cualquiera
    recibirDesdeAlgunaEntrada(tpo, msg, indiceEntrada);

    if(tpo == DATOS) {
      Procesar(msg.data);
      # P[i] tiene POUTi aristas de salida en G puede precisar enviar
      # su salida a todas o algunas de ellas según el problema
      EnviarACadaSalidaNecesaria(tpo, msg);
      if(estado==INICIAL) { # tras procesar datos la 1º vez se vuelve inact.
        estado = INACTIVO;
        send c[SiguienteEnC(i, 0)](TOKEN, 0); # al prox de C (segundo nodo de C)
      } else estado = ACTIVO;
    } else if(tpo == TOKEN) {
      if(estado == INACTIVO) {
        if(msg.valor>=Nc) # detecta terminación
          seguir = SeRepiteLuegoEnC(i, indiceEntrada);
        msg.valor=msg.valor+1;
      } else {
        estado = INACTIVO;
        msg.valor=0; # reinicia el algoritmo de detección
        # en esta implementación el token dará mínimo 3 vueltas a C:
        # 1º volviendo inactivos a todos los procesos: cada uno lo reinicia
        # 2º controlando que permanecen inactivos: cada uno lo incrementa (<Nc)
        # 3º avisando terminación a todos: cada uno lo incrementa (>=Nc)
      }
      # el último de C que detecta terminación no reenvía (todos terminaron)
      if(msg.valor<2*Nc)
        send c[SiguienteEnC(i, indiceEntrada)](TOKEN, msg.valor);
    }
  }
}

Process P[i = 2 to N] {
  enum tEstado = (ACTIVO, INACTIVO);
  tTipo tpo; tMsg msg; int indiceEntrada;
  tEstado estado = ACTIVO;
  bool seguir = TRUE;
  while(seguir) {
    recibirDesdeAlgunaEntrada(tpo, msg, indiceEntrada);
```

```
      if(tpo == DATOS) {
        Procesar(msg.data);
        EnviarACadaSalidaNecesaria(tpo, msg);
        estado = ACTIVO;
      } else if(tpo == TOKEN) {
        if(estado == INACTIVO) {
          if(msg.valor>=Nc) #detecta terminación
            seguir = SeRepiteLuegoEnC(i, indiceEntrada);
          msg.valor=msg.valor+1;
        } else {
          estado = INACTIVO;
          msg.valor=0; # reinicia el algoritmo de detección
        }
        if(msg.valor<2*Nc)
          send c[SiguienteEnC(i, indiceEntrada)](TOKEN, msg.valor);
      }
    }
  }
```

9- ¿En qué consiste la utilización de relojes lógicos para resolver problemas de sincronización distribuida? Ejemplifique para el caso de implementación de semáforos distribuidos.

En un entorno distribuido, el orden de recepción de los mensajes no necesariamente es el mismo que el de emisión. Influye por ejemplo el tiempo que demora un mensaje enviado por un nodo en llegar a los otros, que no será ni uniforme ni constante, dependiendo de la distancia física a la que se encuentren entre otros factores. En particular, mensajes enviados por *broadcast* por dos procesos no serán recibidos en el mismo orden por los otros -incluso podrían ser enviados en exactamente el mismo momento-. En estos casos es necesario encontrar alguna característica que permita diferenciar a uno de otro mensaje y darle prioridad a uno de ellos.

En equipos monoprocesador de MC necesariamente los eventos ocurren en secuencia y podrían ordenarse por el valor del *timer* del equipo en el momento del evento. En un entorno distribuido los *timers* de los distintos equipos no podrían ser perfectamente sincronizados y esto no sería viable. En su lugar pueden usarse *relojes lógicos* y *marcas de tiempo*: cada proceso mantiene un valor entero que incluye en cada mensaje enviado a modo de *timestamp*, permitiendo un ordenamiento de los eventos. Debe ser actualizado para asegurar que si un evento ocurre antes que otro relacionado, su timestamp sea menor.

Para esto, cada proceso mantiene un valor *clock* inicializado en cero. Este es incluido en cada mensaje que se envía, y luego es incrementado. El recibir un mensaje, se lo actualiza con el máximo entre su valor y el timestamp del mensaje más uno, y luego se incrementa. Esto permite un orden parcial entre eventos relacionados. Si además cada proceso tiene un ID único, puede obtenerse un orden total utilizándolo para desempates.

La ordenación impuesta por este esquema no tiene por qué coincidir con el orden temporal real del envío de los mensajes, pero permite asegurar que, recibidos todos los mensajes, todos los nodos que participan los interpreten en un mismo orden. Si un proceso P2 envía un mensaje con marca de tiempo *x* y otro proceso P1 envía luego otro mensaje, antes de recibir el de P2, también con marca de tiempo *x*, todos los procesos ordenarán el mensaje de P1 antes que el de P2, aunque no es este el orden en la realidad.

Pueden utilizarse por ejemplo para implementar semáforos distribuidos, necesarios en este caso ya que si más de un proceso realiza una operación P(), es preciso un consenso entre todos sobre cuál de ellos podrá continuar. El uso de relojes lógicos permite que la decisión se tome de manera distribuida sin tener que recurrir a un servidor/coordinador -esto equivaldría a un monitor activo-.

Los procesos deberían hacer broadcast para realizar una operación sobre el semáforo indicando el valor de su *clock*.

Si cada proceso que comparte el semáforo responde autorizando una operación, una vez que se hayan recibido confirmaciones de todos los otros procesos se puede concretarla actualizando un entero que

represente el valor del semáforo. Las confirmaciones deberían hacerse también por broadcast para que todos los procesos puedan saber qué operaciones se llevan a cabo y en qué orden sobre el semáforo, manteniendo cada uno el valor asociado.
Cada proceso llevaría en una cola (ordenada por *timestamp*) los mensajes que se reciban sobre el semáforo, indicando remitente, operación y *timestamp*. Además debería controlar para cada proceso, cuál fue el último timestamp recibido. Si un mensaje m_i de la cola tiene un timestamp t_i, y se recibió de todos los procesos una confirmación con timestamp superior, se dice que m_i está completamente reconocido -todos los procesos lo han recibido y confirmado-, y puede procesarse. Cualquier mensaje anterior a m_i también lo estará ya que su timestamp será menor (forman el "*prefijo estable*" de la cola).
Tras recibir una confirmación y actualizar el prefijo se procesan los mensajes de la cola. Por cada V, se incrementa el semáforo y elimina el mensaje; luego, por cada P, si el semáforo es positivo se lo decrementa y elimina el mensaje. Si se elimina una P() del propio proceso, entonces puede continuar.
Cada proceso debería llevar a cabo su tarea correspondiente y, a la vez, responder con confirmaciones a operaciones de otros procesos. Por esto convendría utilizar, para cada proceso, un auxiliar que se encargue exclusivamente del manejo del semáforo. Si el proceso debe realizar alguna operación, se la solicitaría a su auxiliar. En el caso de P() debería esperar luego que el auxiliar lo autorice a continuar.

10- Describa el problema de "tiburones y peces" (sharks and fishes), y plantee una posible solución utilizando el paradigma heartbeat.

Al investigar el problema se encontraron algunas variaciones. Algunas fuentes hablaban de peces y tiburones machos y hembras que se reproducían entre sí o peleaban si eran del mismo sexo, otros hablaban de "clonación" al llegar a cierta edad o umbral de energía... presentamos algunas de estas características:

DEFINICIÓN DEL PROBLEMA. "*Sharks and Fishes*" es una variación del juego de la vida. Se presenta al planeta Wa-Tor, un planeta acuático toroidal[1] reticulado, habitado por peces y tiburones. Los peces se alimentan de plancton abundante y los tiburones, de los peces.
El tiempo pasa en ciclos discretos. En cada uno, los peces se mueven aleatoriamente hacia el norte, sur, este u oeste, si es que escogen una posición libre. Tras determinada cantidad de ciclos, los peces se reproducen clonándose en una celda libre vecina, si es que la hay (sino, no ocurre nada). Tras la clonación, el tiempo de reproducción del pez y la cría se resetean.
Los tiburones se mueven también en forma aleatoria, perdiendo energía con cada movimiento (inicialmente tienen cierta cantidad de energía). Si su energía baja de cero, mueren de inanición. Si llegan a una celda con un pez, se alimentan recobrando cierta cantidad de energía -el pez muere-. Si la energía supera un umbral, se reproducen por clonación a una celda libre vecina, dividiéndose la energía en partes iguales entre el tiburon y su cría.

PROPUESTA DE SOLUCIÓN.
Podría manejarse un proceso por celda -o por strip de celdas-. Cada celda puede tener: plancton (estaría libre), hasta cuatro peces, y hasta cuatro tiburones. Asociado a los peces estará el tiempo de reproducción; asociado a los tiburones, la energía actual. Habrá un umbral de reproducción de energía o de edad para tiburones y peces. Puede haber hasta cuatro animales del mismo tipo si ven libre a una celda y se mueven a ella en el mismo ciclo.
Cada celda, en cada ciclo:
1. Informa su contenido a sus cuatro vecinas -considerando la forma toroidal del planeta-.
2. Recibe el estado de las vecinas.
3. Decide qué hacer:
    1 Si sólo tiene plancton no hace nada.
    2 Si tiene tiburones y peces, elimina los peces e incrementa la energía de los tiburones en la cantidad correspondiente (equitativo si hay más de un tiburón).
    3 Si tiene un pez lo envejece; si tiene al menos una vecina libre: si está en edad reproductiva, crea un

[1] Toroidal: Con forma de dona o rosquilla

nuevo pez en la celda vecina y resetea los tiempos; si aún no puede reproducirse, mueve el pez a la vecina. Si tiene más peces, repite el proceso.

4 Si tiene un tiburón: si tiene su energía bajo cero lo elimina -muere por inanición-; si aún tiene energía elige una celda vecina: si la energía está por sobre el umbral crea un nuevo tiburón, lo ubica en la vecina y comparte la mitad de su energía, sino decrementa en uno su energía y lo desplaza. Si tenía más tiburones repite el proceso.

4. Envía Pez/Tiburon/nada a cada vecina (según lo decidido en el punto anterior).
5. Recibe de cada vecina pez/tiburon/nada-, actualizando su contenido.

```
tTipo = enum(PEZ, TIBURON, PLANCTON);
tPez = record {
  int edad;
}
tTiburon = record {
  energia: double;
}
tItem = record {
  tTipo tipo;
  union { p: tPez; t: tTiburon; } valor;
}
tList = list of tItem;

# contenido inicial -un solo item- y cant de generaciones de la simulación
chan inicializacion[1:F; 1:C](tItem, int),
     vecina[F:C](int, int, int, int),
     intercambio[F:C](tItem);

Process celda[f= 1 to F; c= 1 to C] { # Mundo de FxC celdas
  tList contenido;
  tItem item, plancton;
  int cantGen, qP=0, qT=0, fv, cv, qPv, qTv, nV;
  list of (int f, int c) vecinasLibres;
  double energiaGanada;
  plancton.tipo = PLANCTON; # para finalizar intercambios con vecinas

  receive inicializacion[f,c](item, cantGen);
  if(item.tipo!=PLANCTON) {
    insert(contenido, item);
    if(item.tipo==PEZ) qP = qP + 1
    else qT = qT + 1;
  }

  for[g = 1 to cantGen] {
    # avisa su contenido a sus vecinas -------------------------------------
    send vecina[f, (c==1)? C : c-1](f,c,qP,qT);
    send vecina[f, c mod C +1](f,c,qP,qT);
    send vecina[(f==1)? F : f-1, c](f,c,qP,qT);
    send vecina[f mod F +1, c](f,c,qP,qT);

    # recibe contenidos de las vecinas -------------------------------------
    # -sólo interesan si no están vacías para que se muevan los peces-
    clear(vecinasLibres);
    for[v=1 to 4] {
      receive vecina(fv, cv, qPv, qTv);
```

```
  if(qPv==0 and qTv==0) insert(vecinasLibres, fv, cv);
}

# decide que hacer ----------------------------------------------------
if(!isEmpty(contenido)) {
  if(qT>0) { # algunos tiburones mueren
    for each t in ObtenerTiburones(contenido)
      if(t.valor.energia<0) {
        qT = qT - 1;
        remove(contenido, t);
      }
  }
  if(qP>0 and qT>0) { # si tiene peces Y tiburones
    for each p in ObtenerPeces(contenido)
      remove(contenido, p);
    energiaGanada = ENERGIA_POR_PEZ * qP / qT;
    qP=0;
    for each t in ObtenerTiburones(contenido)
      t.valor.energia = t.valor.energia + energiaGanada;
  }
  if(qP>0) { # si tiene peces
    for each p in ObtenerPeces(contenido) {
      p.valor.edad = p.valor.edad + 1;
      if(!empty(vecinasLibes)) {
        nV = round(size(vecinasLibres) * rnd());
        fv = Item(vecinasLibres, nV).f;
        cv = Item(vecinasLibres, nV).c;
        if(p.valor.edad > EDAD_REPRODUCCION)
          p.valor.edad = 0; # resetea la edad -reproducción
        else {
          remove(contenido, p);
          qP = qP - 1;      # elimino de esta celda -se mueve
        }
        send intercambio[fv, cv](p);
      }
    }
  } else if(qT>0) { # si tiene tiburones
    for each t in ObtenerTiburones(contenido) {
      switch round(rnd(4)) {      # elige una vecina
        0: fv = (f==1) ? F, f-1; cv=c; break; # mov izq
        1: cv = (c==1) ? C, c-1; fv=f; break; # mov arr
        2: fv = f mod F +1; cv=c; break; # mov der
        3: cv = c mod C +1; fv=f; break; # mov aba
      }
      if(t.valor.energia >= UMBRAL_REPRODUCCION)
        t.valor.energia = t.valor.energia/2; # comparte energia con la cria
      else {
        remove(contenido, t);
        qT = qT - 1;              # elimino de esta celda -se mueve
        t.valor.energia = t.valor.energia - 1;
      }
      send intercambio[fv, cv](t);
    }
```

```
      }
    }

    # finaliza intercambios enviando plancton -------------------------------
    # si tenía que enviar peces/tiburones, los envió mientras decidía -------
    send intercambio[f, (c==1)? C : c-1](plancton);
    send intercambio[f, c mod C +1](plancton);
    send intercambio[(f==1)? F : f-1, c](plancton);
    send intercambio[f mod F +1, c](plancton);

    # recibe lo que envíen sus 4 vecinas ------------------------------------
    finalizadas=0;
    while(finalizadas!=4) {
      receive intercambio[f,c](item);
      if(item.tipo==PLANCTON) finalizadas = finalizadas + 1
      else {
        insert(contenidos, item);
        if(item.tipo==PEZ) qP = qP + 1
        else qT = qT + 1;
      }
    }
  }
}
```

11-

¿Cuál es el objetivo de la programación paralela?

El sentido de la programación paralela es ganar velocidad, permitiendo resolver un problema en menos tiempo que una solución secuencial, o resolver un problema más grande en el mismo tiempo.

¿Cuál es el significado de las métricas de speedup y eficiencia? Cuáles son los rangos de valores en cada caso?

SpeedUp: Indica la relación entre el tiempo de ejecución secuencial y paralelo de un programa. A menor tiempo paralelo, mayor será el speedup. Indica qué tan rápida es la solución paralela respecto de la secuencial. Su rango de valores varía entre [1; p) siendo p la cantidad de procesadores empleados en paralelo.
Eficiencia: Indica qué tan bien se aprovechan los procesadores en la solución paralela, relacionando el speedup con la cantidad de procesadores empleados para conseguirla. Su rango de valores varía entre (0; 1).

¿En qué consiste la "ley de Amdahl"?

Permite calcular el límite superior para el speedup de un algoritmo en particular.
En general, un algoritmo cualquiera realiza una entrada, procesamiento y salida. No todo el algoritmo puede mejorarse en una versión paralela, ya que normalmente la etapa de entrada -o inicializaciones- y salida mantendrán componentes secuenciales: sólo el procesamiento será apto de optimización.
La mejora total alcanzable en un programa estará limitada entonces por su componente no paralelizable:

$$SpeedUp\ Total = \frac{1}{(1 - Porc.Paralelizable) + \frac{Porc.Paralelizable}{SpeedUp\ Parte\ Paralelizable}}$$

El SpeedUp máximo alcanzable se obtiene considerando la valor óptimo de un speedup lineal para la parte

paralelizable (SpeedUp Parte Paralelizable = P, la cantidad de procesadores).
Como muchas veces la E/S llevan un tiempo mucho menor al procesamiento la limitación puede no ser tan fuerte.
La ley de Amdahl indica que la mejora obtenida al alterar una porción del sistema estará limitada por la fracción del tiempo que dicha porción se utiliza. Así, es el algoritmo y no la cantidad de procesadores el factor que limita el speedup; agregar procesadores puede aumentar el speedup pero sólo hasta cierto punto.

Preguntas de Final

Defina programa concurrente, programa paralelo y programa distribuido.

Un **programa concurrente** consiste en un conjunto de procesos secuenciales que cooperan para resolver un problema, y que pueden ejecutarse conjuntamente (intercalándose en el tiempo *-multiprogramación-* o en distintos núcleos/procesadores). Es un concepto de software que dependiendo de la arquitectura subyacente da lugar a:

- **Programa paralelo**: Programa concurrente que se ejecuta sobre *múltiples procesadores*, con o sin memoria compartida, utilizados *para incrementar la performance*.
- **Programa distribuido**: Programa concurrente en la que se cuenta con varios procesadores pero no se posee una memoria compartida global y la comunicación está dada por el *pasaje de mensajes*.

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

Una **propiedad** es un atributo de un programa que es *verdadero en todas sus posibles historias*. Como un programa concurrente está formado por múltiples procesos, una **historia** es cada *interleaving* de acciones atómicas que puede darse en una ejecución del programa.
Una propiedad de **seguridad** asegura que *nada malo* sucede durante la ejecución del programa. Ejemplos: la ausencia de deadlock (lo malo es tener procesos esperando condiciones que no ocurrirán nunca) y la exclusión mutua (lo malo es tener más de un proceso ejecutando sus secciones críticas al mismo tiempo).
Una propiedad de **vida** asegura que *algo bueno* eventualmente ocurre durante la ejecución. Ejemplos: eventual entrada a una SC, terminación del programa o que un mensaje llegue a destino. Estas propiedades dependen de la *política de scheduling*.

Defina fairness y relacione con las políticas de scheduling. Describa los distintos tipos.

Fairness trata de asegurar que un proceso tenga la *posibilidad de avanzar sin importar lo que haga el resto* de los procesos. Como el avance está dado por la elección de la próxima sentencia atómica a ejecutar, el concepto se relaciona con las *políticas de scheduling,* que son las que deciden cual es la próxima sentencia atómica a ejecutarse.
El fairness puede ser:

- **Incondicional**: Toda acción atómica *incondicional* que es elegible es eventualmente ejecutada.
- **Débil**: Es incondicionalmente fair y toda acción atómica *condicional* que se vuelve elegible es ejecutada si su guarda se vuelve verdadera y *se mantiene* así hasta que es vista por el proceso que ejecuta la acción atómica condicional.
- **Fuerte**: Es incondicionalmente fair y toda acción atómica *condicional* que se vuelve elegible es ejecutada si su guarda es true *con infinita frecuencia*.

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

- **Exclusión mutua**: Sólo un proceso a la vez puede estar ejecutando su SC. Es una propiedad de seguridad.
- **Ausencia de deadlock**: Si uno o más procesos están intentando entrar a su SC, al menos uno de ellos tendrá éxito. Es una propiedad de seguridad.
- **Ausencia de demora innecesaria**: Si un proceso está intentando acceder a su SC y ningún otro proceso está ejecutando la SC, el primero no está impedido de ingresar. Es una propiedad de seguridad.
- **Eventual entrada**: Un proceso que intenta entrar a su SC eventualmente lo hará. Es una propiedad de vida.

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

La sincronización por **busy waiting** es *ineficiente en sistemas multiprogramados* ya que se mantendrá

ocupado un procesador realizando un *spinning*, cuando podría ser usado por otro proceso. También puede tener problemas de *contención de memoria* al estar chequeando y seteando variables compartidas que son modificadas por otros procesos repetidas veces, invalidando las caches.
Los **semáforos** son herramientas que sirven para sincronización pero evitan los problemas que tiene busy waiting. Al *bloquearse* en un semáforo un proceso *no consume tiempo de procesamiento* hasta que no tenga posibilidad de ejecutarse, en cuyo caso, es puesto en la cola de *listos* para poder usar el procesador.

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

La sentencia await de grano grueso sigue la sintaxis: `<AWAIT B → S>`.
B es una expresión booleana que especifica una condición de demora y S es una secuencia de sentencias que se garantiza que terminan. Se garantiza que B es *true* al momento de ejecutar S, y ningún estado interno de S es visible para los otros procesos.
La relación con las instrucciones atómicas especiales T&S y F&A es que estas últimas, presentes en casi todos los procesadores, son utilizadas para hacer atómico el AWAIT de grano grueso implementando los protocolos de E/S de la sección crítica.

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.

El problema general de alocación de recursos se da cuando varios procesos quieren acceder a unidades de un recurso compartido. Un proceso pide una o más unidades del recurso ejecutando un **request** con la cantidad de unidades que solicita y su identificación. El request puede ser satisfecho si la cantidad de unidades disponibles es suficiente para satisfacerlo; de lo contrario se demora hasta que haya suficientes unidades disponibles. Después de usar los recursos asignados, un proceso los retorna ejecutando la operación **release**.
Ambas operaciones deben ser atómicas ya que acceden a la representación de los recursos. Se puede utilizar la técnica de *passing the baton* para el request y el release:

```
Request (parámetros):
  P(e)
  if(request no puede satisfacerse) {DELAY}
  Tomar las unidades;
  SIGNAL;
Release (parámetros):
  P(e);
  Retorna unidades
  SIGNAL;
```

Su resolución mediante **Shortest Job Next** *(SJN)* se basa en varios procesos que requieren un único recurso compartido, indicando en el request tanto su *id de proceso* como el *tiempo* que utilizarán al recurso. Al ejecutar un request, si el recurso está libre es inmediatamente alocado al proceso y en caso contrario, el proceso guarda los parámetros del request y se demora. Cuando el recurso es liberado, utilizando el release, es alocado al proceso demorado (si lo hay) que tiene el mínimo valor de tiempo. Si dos o más procesos demorados tienen el mismo valor de time se le asigna al que más tiempo ha estado demorado.
La política de SJN **minimiza el tiempo promedio de ejecución** pero es **unfair** ya que un proceso podría ser demorado para siempre si hay una corriente continua de request con tiempo menor. Este problema es propio de toda política de prioridad, y puede evitarse utilizando la técnica de **aging** que consiste en darle preferencia a un proceso que ha estado demorado un largo tiempo. (Poner un valor fijo para determinar si un proceso espero mucho o implementar la variante del SJN, *Highest Response Ratio Next* que calcula la prioridad como 1 + tiempo espera/tiempo ejecución).

¿En qué consiste la técnica de passing the baton? Aplicar este concepto a la resolución del

problema de lectores y escritores.

Passing the baton es una técnica general para *implementar sentencias await arbitrarias* y para proveer un orden en la ejecución de los procesos. Consiste en que mientras un proceso se encuentre dentro de su SC mantiene el bastón de mando o derecho a ejecutarse, y cuando termina se lo pasa a un proceso demorado en condiciones de ejecutarse (si es que lo hay) o lo libera, permitiendo que lo tome el primer proceso que solicite el acceso a la SC.
Las acciones atómicas pueden ser incondicionales o condicionales: F_1: <S_i> y F_2: <AWAIT B_j → S_j>.
Se utiliza un **mutex e** (semáforo inicializado en 1) que representa al bastón de mando para controlar la entrada a las sentencias atómicas. Además, se asocia por cada sentencia condicional (F_2) con una guarda B_j, un semáforo b_j y un contador de procesos demorados d_j, ambos inicialmente 0. El semáforo se utiliza para demorar los procesos que esperan por la condición, y el contador para saber cuántos se han demorado esperando por la guarda. Con estos datos podemos ver cómo quedan formadas las sentencias atómicas con la técnica de *passing the baton*.

F_1:
```
P(e);
Si;
SIGNAL
```

F_2:
```
P(e);
if not(Bj) {
  dj = dj+1;
  V(e);
  P(bj);
}
Sj;
SIGNAL
```

SIGNAL:
```
if(B1 and d1>0) {d1 = d1-1; V(b1)}
Elseif(B2 and d2>0) {d2 = d2-1; V(b2)}
…
Elseif(BN and dN>0) {dN = dN-1; V(bN)}
Else V(e);
```

La forma general de SIGNAL puede ajustarse luego en cada invocación omitiendo las ramas que no tengan sentido allí. Además puede reemplazarse el `Elseif` (determinístico) por el signo `[]` (no determinístico), evitando darle mayor prioridad a las guardas según el orden en que se analicen en SIGNAL.
Para el caso de los lectores y escritores, los protocolos de acceso a la SC en ambos procesos comparten una condición (que no haya escritores en la SC) pero los escritores agregan una más (que no haya lectores) por lo que es imposible distinguirlas usando un único semáforo. Es un problema de *exclusión mutua selectiva*.

```
int nr = 0, nw = 0;
sem e = 1, r = 0, w = 0: #0<=(e+r+w)<=1 → forman un SBS -semáforo binario dividido-
int dr = 0, dw = 0;

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

process Escritor[i = 1 to N] {
  while(true) {
    # ⟨await (nr == 0 and nw == 0) nw = nw + 1⟩
```

```
        P(e);
        if(nr > 0 or nw > 0) {dw = dw+1; V(e); P(w); }
        nw = nw + 1;
        V(e);
        #escribe la BD
        # ⟨nw = nw - 1⟩
        P(e);
        nw = nw - 1;
        if (dr > 0) {dr = dr - 1; V(r); }
        [] (dw > 0) {dw = dw - 1; V(w); }
        else V(e);
    }
}
```

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

El concepto de **broadcast** consiste en enviar concurrentemente un mismo mensaje a varios procesos.
En un ambiente compartido es más fácil que la información de un proceso pueda ser conocida por todos los demás al poner el dato, por ejemplo, en una variable compartida por todos ellos. En un ambiente distribuido esto no se puede ya que no hay variables compartidas por lo que sí o sí se debe mandar la información por medio de mensajes.
Con PMA existen dos opciones:

1. Utilizar un único canal desde donde todos los procesos pueden tomar el dato por lo que el emisor deberá enviar tanto mensajes como procesos receptores existan. Debe contemplarse que cada proceso tome sólo un mensaje.
2. Utilizar un canal privado para comunicarse (enviar el mensaje) con cada uno de los procesos receptores.

El primer caso es mejor porque no hay demoras innecesarias ya que ni bien haya un mensaje en el canal este puede ser tomado por un receptor. En cambio, con la segunda opción un proceso que está listo para recibir el mensaje debe esperar a que el emisor le deposite el mensaje en su canal.
Con PMS no se puede usar un canal compartido porque todos son punto a punto, por lo tanto es más complejo e ineficiente. Es similar a la segunda opción con PMA ya que se debe enviar si o si un mensaje a cada proceso y además entre cada envío se debe esperar (SEND bloqueante) que el proceso lo haya recibido. Hay formas de optimizarlo como poner un buffer por cada proceso, y que el emisor envíe el mensaje a los buffers por si los procesos reales están haciendo otro trabajo.

De los problemas de los baños vistos en la teoría. ¿Cuál podría ser de exclusión mutua selectiva? ¿Por qué?

Opción 1: Un baño único para varones o mujeres (excluyente) sin límite de usuarios. **Exclusión mutua selectiva** porque tanto las mujeres o los hombres pueden seguir entrando después que uno de su clase obtuvo el acceso.
Opción 2: Un baño único para varones o mujeres (excluyente) con un número máximo de ocupantes simultáneos (que puede ser diferente para varones y mujeres). Es **exclusión mutua general** ya que cuando alguna de las clases que está ocupando un baño llega a su valor máximo de ocupantes simultáneos empieza a excluir a los de su propia clase, es decir a competir entre todos.
Opción 3: Dos baños utilizables simultáneamente por un número máximo de varones (K1) y de mujeres (K2), con una restricción adicional respecto que el total de usuarios debe ser menor que K3 (K3 < K1 + K2). Similar a la opción 2 hay un tope en las cantidades de cada clase por lo que en algún momento se empezaran a incluir entre todos (**general**).

¿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?

Un problema es de **exclusión mutua selectiva** cuando cada proceso compite por el acceso a los recursos no con todos los demás procesos sino con un *subconjunto* de ellos. El problema de los filósofos es de exclusión mutua selectiva ya que para comer un filósofo no compite con todos los demás sino que sólo compite con sus adyacentes. En el caso de que fueran *3 filósofos no sería de exclusión mutua selectiva ya que un proceso compite con todos los demás procesos y no solo con un subconjunto de ellos*, dado que el resto de los procesos son sus adyacentes.

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

Sin posiciones fijas se refiere a que cada filósofo solicita cubiertos y no necesariamente los de sus adyacentes, el coordinador o mozo se los da si es que hay dos cubiertos disponibles sin tener en cuenta vecinos, el problema *no sería de exclusión mutua selectiva ya que competirían entre todos* por poder acceder a los tenedores para poder comer.

El problema de los lectores-escritores, ¿es de exclusión mutua selectiva? ¿Por qué?

En este caso, la *exclusión mutua selectiva* es *entre clases de procesos* ya que los procesos lectores como clase de proceso compiten con los escritores, esto se debe a que cuando un lector está accediendo a la BD otros lectores pueden acceder pero se excluye a los escritores.

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?

En este caso, al ser un único proceso lector el que compite el problema se convierte en un problema de *exclusión mutua general* porque compiten entre todos los procesos por el acceso a un recurso compartido ya que no existen más lectores para acceder a la BD, y lectores y escritores por definición se excluyen mutuamente.

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

1. **Pares que interactúan, filtros y productor-consumidor**: Es más adecuado el uso **PM** ya que el flujo de comunicación es *unidireccional*. Además con RPC los procesos no pueden comunicarse directamente por lo que la comunicación debe ser implementada y con Rendezvous aunque los procesos pueden comunicarse, no pueden ejecutar una llamada seguida de una entrada de datos, por lo que deberían definirse procesos asimétricos o utilizar procesos helpers. Dependiendo de tipo de problema particular dependerá si es más adecuado PMA que provee buffering implícito y mayor grado de concurrencia o PMS que provee mayor sincronización.
2. **Cliente-servidor**: Es más adecuado el uso de **RPC o Rendezvous** ya que el problema requiere de un flujo de comunicación *bidireccional*: el cliente solicita un servicio y el servidor responde a la solicitud, es decir que no sólo el cliente envía información para llevar a cabo la operación sino que el servidor debe devolver (en el caso general) los resultados de dicha ejecución. Además, normalmente no es necesario que el cliente siga procesando ya que antes de realizar otra tarea requerirá los resultados del servidor. Ambos mecanismos proveen este flujo de comunicación bidireccional, mientras que con PM deberían utilizarse dos canales: uno para las solicitudes y otro para las respuestas.

Describir la solución usando la criba de Eratóstenes al problema de hallar los números primos entre 2 y n. ¿Cómo termina el algoritmo? ¿Qué modificaría para que no termine de esa manera?

Se comienza por el primer número en la lista, en este caso el 2, y se borran todos los números que son múltiplos de este. Los números que quedan todavía son candidatos a ser primos así que se repite el mismo proceso considerando al próximo número de la lista una y otra vez (3, 5, 7...) hasta que todos los números

son considerados. Los números que queden en la lista al final serán todos los números primos entre 2 y n. La solución a este problema utiliza un pipeline de filtros; cada proceso recibe un stream de números de su predecesor y envía un stream de números a su sucesor eliminando los múltiplos del primer número recibido.

```
process Worker[1] {
int p=2;
  for[i = 3 to n by 2]
    worker[2]!i;
}
```

```
process Worker[i = 2 to L] {
int p, proximo;
  worker[i-1]?p;
  do worker[i-1]?proximo →
       if(proximo mod p) →
         worker[i+1]!proximo;
       fi
  od
}
```

Hay un problema con esta solución y es que termina en deadlock ya que los procesos reciben dentro de un loop los valores de stream y no tienen forma de darse cuenta cuando no recibirán nuevos valores y se quedaran todos esperando un nuevo valor desde su predecesor. Esto puede solucionarse utilizando un **valor centinela**, 0 por ejemplo, que actúa como EOS -*End Of Secuence*-:

```
process Worker[i = 1] {
  int p = 2;
  for [t = 3 to N by 2]                    #elimina los pares del stream
     worker[2]!t;
  print p;                                 #Imprime el primo i-ésimo
  worker[2]!0;                             #envía la marca EOS al siguiente
}

process Worker[i = 2 to L] {
  int p, t=1;
  worker[i-1]?p; #Recibe el primo i-ésimo
  while(t!=0) {
    worker[i-1]?t;
    if(i<L)and(t mod p != 0) worker[i+1]!t;  #elimina los múltiplos del primo i
  }
  print p;
  if(i<L) worker[i+1]!0;
}
```

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 el mínimo de todos los valores. Plantee conceptualmente posibles soluciones con las siguientes arquitecturas de red: Centralizada, simétrica y anillo circular. No implementar. Analice las soluciones desde el punto de vista del número de mensajes y la performance global del sistema.

- **Arquitectura centralizada**: cada proceso envía su dato al proceso central (N-1 mensajes) que es el encargado procesar los N valores y de reenviar el máximo y el mínimo a todos los demás procesos. Total 2*(N-1) mensajes, o N si existe mensaje de *broadcast*. Los mensajes al coordinador se envían casi al mismo tiempo por lo que sólo el primer *receive* del coordinador demora mucho.
- **Arquitectura simétrica**: hay un canal entre cada par de procesos y todos ejecutan el mismo algoritmo. Cada proceso envía su dato V a los N-1 procesos restantes y recibe luego N-1 valores, uno desde cada proceso. De este modo en paralelo todos están calculando el máximo y el mínimo. Total N*(N-1) mensajes, o N si pueden utilizar broadcast. Pueden transmitir en paralelo si la red soporta transmisiones concurrentes pero el overhead de comunicación acota el *speedup*.
- **Arquitectura de anillo**: cada P[i] recibe mensajes de su predecesor P[i-1] y envía mensajes a su sucesor

P [i+1]. Este esquema consta de 2 etapas; en la primera cada proceso recibe el máximo y mínimo que conoce su predecesor, y los compara con su valor local, transmitiendo un máximo y un mínimo local a su sucesor. El primer proceso enviará su V como máximo y mínimo ya que no conoce más que eso. En la segunda etapa, iniciada cuando el último proceso cierra el anillo enviando el máximo y mínimo global al primer proceso, todos deben recibir la circulación del máximo y el mínimo global y transmitirla al siguiente proceso, excepto el último. Total 2(N-1) mensajes, o N si existe broadcast, pero con la desventaja que se pierde paralelismo, se termina resolviendo de forma secuencial (un proceso trabaja por vez, el último debe esperar que todos los otros reciban un mensaje, procesen y envíen su salida). Aunque la cantidad de mensajes es igual a la centralizada, es más lenta por este motivo.

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

Se puede definir como granularidad a la relación que existe entre el procesamiento y la comunicación. En una arquitectura de grano fino existen muchos procesadores pero con poca capacidad de procesamiento por lo que son mejores para programas que requieran mucha comunicación. Entonces, los programas de grano fino son los apropiados para ejecutarse sobre esta arquitectura ya que se dividen en muchas tareas o procesos que requieren de poco procesamiento y mucha comunicación.
Por otro lado, en una arquitectura de grano grueso se tienen pocos procesadores con mucha capacidad de procesamiento por lo que son más adecuados para programas de grano grueso en los cuales se tienen pocos procesos que realizan mucho procesamiento y requieren de menos comunicación.

- **Granularidad De Arquitectura**: Puede verse como la relación entre número de procesadores y la cantidad de memoria, o la velocidad de procesamiento y la comunicación.
  - *Grano GRUESO*: pocos procesadores muy potentes
  - *Grano FINO* : muchos procesadores no muy potentes
- **Granularidad De Programa**: Puede verse como la cantidad de computo y la cantidad de comunicación necesaria.
  - *Grano GRUESO*: Mucho cómputo con relativamente poca comunicación.
  - *Grano FINO* : Mucha comunicación con poco cómputo.

Las aplicaciones con concurrencia limitada funcionan mejor en arquitecturas de grano grueso, ya que realizan mucho cómputo. Las aplicaciones altamente concurrentes funcionan mejor en arquitecturas de grano fino, ya que requieren mayor comunicación y sincronización.

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.

P1:	P2:	P3:

`If(x == 0) then`
`  y:= 4*x+2;`
`x:= y+2+x;`
	`If(x > 0) then`
	`  x := x+1;`
		`x := x*8+x*2+1;`

1. El valor de x al terminar el programa es 9. **Verdadero**. *P2 deja x=, P1 entra al if, P3 deja x=1, P1 termina con y =6, x=9.*
2. El valor de x al terminar el programa es 6. **Verdadero**. *P1 sale del if con y=2, P3 deja x=1, P2 deja x=2, P1 finaliza con y=2, x=6.*
3. El valor de x al terminar el programa es 11. **Falso**. *La única forma de que el valor de X sea 11 es que al ejecutar P3, X=1 y esto no puede darse porque P2 no puede alterar X porque X=0 y si P1 modificara el valor de X este seria 4 (o si 5 si P2 ejecuta luego).*.
4. Y, siempre termina con alguno de los siguientes valores: 10 o 6 o 2 o 0. **Verdadero**.
   Y=0: El único caso en el que Y sería 0, es que se ejecute P3 primero alterando el valor de X por lo que al

ejecutarse P1 no realizaría ningún calculo.
Y=2: Siempre que asignación en P1 se realice antes de que los demás procesos alteren X (X=0).
Y=6: Si primero se ejecuta P2 terminando su ejecución porque x=0 y luego P1 evalúa el if y el control se pasa a P3 en ese instante, X=1 y al volver a realizar la asignación de Y en P1 el valor de Y=6.
Y=10: Se ejecuta P3 habilitando la asignación de P2 por lo que X=2 y luego se realiza la asignación de Y.

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

Con frecuencia un proceso se quiere comunicar con varios otros, quizás por distintos canales y no sabe el orden en el cual los otros procesos podrían querer comunicarse con él. Es decir, podría querer recibir información desde diferentes canales y no quedar bloqueado si en alguno no hay mensajes. Para poder comunicarse por distintos canales se utilizan sentencias de comunicación guardadas.
Las sentencias de comunicación guardada soportan comunicación no determinística: B; C→S;

- B condición lógica que habilita la comunicación C (sea C un *send* o *receive*). Puede omitirse, asumiéndose *true*.
- B y C forman la **guarda**. La misma:
  - tiene *éxito* si B es *true* y ejecutar C no causa demora.
  - *falla* si B es *false*.
  - se *bloquea* si B es *true* pero C no puede ejecutarse inmediatamente.

Puede indicarse varias sentencias de comunicación guardada en un `if` o un `do`. Si todas fallan se continúa la ejecución con la instrucción siguiente; si hay guardas con éxito se ejecuta una no determinísticamente, y si algunas se bloquean se demora al proceso hasta que pueda ejecutarse con éxito a alguna de ellas. Si se trata de un ciclo `do`, se itera hasta que todas las guardas fallen.

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

La técnica de *passing the baton* es una técnica general para implementar *sentencias await arbitrarias* y para decidir cuál de los procesos es el próximo en seguir ejecutando (orden en el cual despertarlos). Ambas técnicas tienen la misma utilidad que es la de proveer un orden en la ejecución de procesos.

Describa brevemente los mecanismos de comunicación y sincronización de Linda, Ada, MPI y Java.

1. **Linda** utiliza memoria compartida como mecanismo de comunicación y pasaje de mensajes asincrónico como mecanismo de sincronización. En la memoria compartida hay *tuplas* que pueden ser activas (tareas) y pasivas (datos). El núcleo de LINDA es el espacio de tuplas compartido (TS) que puede verse como un único canal de comunicaciones compartido. Si bien hablamos de memoria compartida el espacio de tuplas puede estar distribuido en una arquitectura multiprocesador.
2. **Java** provee concurrencia mediante el uso de *threads* que comparten la zona de datos, por lo que el mecanismo que utilizan para comunicarse es memoria compartida. Como método de sincronización java permite utilizar la palabra clave *synchronized* sobre un método lo que brinda sincronización y exclusión mutua. También permite la sincronización por condición con los métodos *wait*, *notify* y *notifyAll* sobre un único lock por objeto, similares al “wait” y “signal” de los monitores, que deben usarse siempre dentro de un bloque synchronized. La semántica que utiliza el *notify* es de *signal and continue*. Importando paquetes estándar puede utilizarse pasaje de mensajes (java.net soporta TCP o UDP) y RPC (java.rmi soporte una versión orientada a objetos, *Remote Method Invocation*).
3. **Ada** utiliza *Rendezvous* como mecanismo de sincronización y comunicación. En Ada un proceso realiza pedidos haciendo un *call* de un entry definido por otro proceso, este llamado bloquea al proceso llamador hasta que termina la ejecución del pedido. Para servir los llamados a sus entrys una *task* utiliza la sentencia *accept* que demora a la tarea hasta que haya una invocación para el entry asociado a esa sentencia. Cuando recibe una invocación copia los parámetros y ejecuta la lista de sentencias correspondiente; al terminar copia los parámetros de salida y tanto el llamador como el invocador

pueden continuar su ejecución. Además, provee tres clases de sentencias select:

- Wait selectivo:

```
Select [when B1 =>] Sentencia accept; sentencias
Or…
Or [when Bn =>] Sentencia accept; sentencias
[Or delay tiempo; sentencias | Or terminate | Else sentencias]
End select
```

- Entry call condicional:

```
Select entry call; sentencias
Else sentencias
End select
```

  Sólo se selecciona el entry call si puede ser ejecutado inmediatamente sino se selecciona el *else*.

- Entry call timed:

```
Select entry call; sentencias
Or delay tiempo; sentencias
End select
```

  En este caso, el entry call se selecciona si puede ser ejecutado antes de que expire el intervalo *delay*.

4. **MPI** es una biblioteca para manejar procesos de tipo SCMD. La biblioteca debe ser inicializada y finalizada explícitamente, provee primitivas de envío sincrónico y asincrónico, recepción de mensajes sincrónico y de tipo polling o no bloqueante, sea desde un proceso particular o desde cualquiera dentro de un grupo. Los grupos de procesos se pueden crear y manejar dinámicamente, pudiendo utilizar primitivas de sincronización en barrera, comunicación broadcast, distribuir elementos de un array entre procesos o esperar mensajes desde los procesos del grupo y reunirlos en un array, realizar alguna operación entre los valores recibidos de otros procesos -sumar, buscar el máximo o mínimo...-, entre otras operaciones.

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

1. **Servidores replicados**: Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos. Permiten incrementar la accesibilidad de datos o servicios donde cada servidor descentralizado interactúa con los demás para darles la sensación a los clientes de que existe un único servidor. Ejemplo: la resolución descentralizada al problema de los filósofos donde cada proceso mozo interactúa con otros para obtener los tenedores de forma transparente a los filósofos.
2. **Algoritmos de heartbeat**: Los procesos periódicamente deben intercambiar información y para hacerlo ejecutan dos etapas; en la primera se expande enviando información (SEND a todos) y en la segunda se contrae adquieren información (RECEIVE de todos). Su uso más importante es paralelizar soluciones iterativas. Ejemplos: computación de grillas (labeling de imágenes) o autómatas celulares (el juego de la vida).
3. **Algoritmos de pipeline**: La información recorre una serie de procesos utilizando alguna forma de *receive/send* donde la salida de un proceso es la entrada del siguiente. Ejemplos: redes de filtros o tratamiento de imágenes.
4. **Algoritmos probe/echo**: La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) diseminando y juntando información. Ejemplo: realizar un broadcast (sin *spanning tree*) o conocer la topología de una red cuando no se conocen de antemano la cantidad de nodos activos.
5. **Algoritmos de Broadcast**: Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas y para resolver problemas de sincronización distribuida. Ejemplo: sincronización de relojes en un Sistema Distribuido de Tiempo Real.
6. **Algoritmos Token Passing**: En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. Permite realizar exclusión mutua distribuida y la toma de decisiones distribuidas. Ejemplo: determinar la terminación de un proceso en una arquitectura distribuida cuando no puede decidirse localmente.
7. **Manager/workers**: Implementación distribuida del modelo de *bag of tasks* que consiste en un proceso

controlador de datos y/o procesos y múltiples *workers* que acceden a él para poder obtener datos y/o tareas a ejecutar en forma distribuida. Ejemplo: algoritmos de paralelismo recursivo en entornos distribuidos.

¿Cuál es el objetivo de la programación paralela?

Reducir el tiempo de ejecución de algoritmos concurrentes o resolver problemas de tamaño muy grande o que requieran de una mayor precisión en los resultados en el mismo tiempo.

Mencione las tres técnicas fundamentales de la computación científica. Ejemplifique.

Entre las diferentes aplicaciones de cómputo científicas y modelos computacionales existen tres técnicas fundamentales:

- **Computación de grillas**. Dividen una región espacial en un conjunto de puntos. Soluciones numéricas a PDE, imágenes.
- **Computación de partículas**. Modelos que simulan interacciones de partículas individuales como moléculas u objetos estelares.
- **Computación de matrices**. Sistemas de ecuaciones simultáneas.

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

Ambas son métricas asociadas al procesamiento paralelo.

- **Speedup** ⇒ S = Ts/Tp: S es el cociente entre el tiempo de ejecución secuencial del algoritmo secuencial conocido más rápido (Ts) y el tiempo de ejecución paralelo del algoritmo elegido (Tp). El rango de valores de S va desde 0 a p, siendo p el número de procesadores. Mide cuánto más rápido es el algoritmo paralelo con respecto al algoritmo secuencial, es decir *cuánto se gana por usar más procesadores*.
- **Eficiencia** ⇒ E = S/P: Cociente entre speedup y número de procesadores. El valor está entre 0 y 1, dependiendo de la efectividad en el uso de los procesadores. Cuando es 1 corresponde al speedup perfecto. Mide la *fracción de tiempo en que los procesadores son útiles para el cómputo,* es decir *cuánto estoy usando de los recursos disponibles*.

¿En qué consiste la ley de Amdahl?

Indica que para un problema dado existe un *speedup máximo alcanzable* independiente del número de procesadores. Significa que es el algoritmo el que decide la mejora de velocidad dependiendo de la cantidad de código no paralelizable y no del número de procesadores, llegando a un punto en que no se puede paralelizar más el algoritmo.

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 P[impar] intercambian valores con P[impar+1], si los números están desordenados. En las pares, los 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).

Determine cuántas rondas deben ejecutarse en el peor caso para ordenar los números.

Son necesarias **n rondas** en el peor de los casos que sería uno en el cual los valores se encuentran ordenados de mayor a menor porque requiere llevar el mayor valor n procesos más adelante y análogamente para el mínimo.

¿Considera que es más adecuado para este caso pasaje de mensajes sincrónico o asincrónico? Justifique.

PMS es más adecuado en este caso porque los procesos deben sincronizar de a pares en cada ronda, por lo que PMA no sería tan útil para la resolución de este problema ya que se necesitaría implementar una barrera simétrica para sincronizar los procesos de cada etapa.

Escriba un algoritmo paralelo para ordenar el arreglo a [1:n] en forma ascendente. ¿Cuántos mensajes se utilizan?

```
process Proc[i: 1..n] {
int miValor, elOtro;
  ... inicializar miValor del arreglo...
  for(ronda=1;ronda<=n;ronda++) {
    if(i mod 2 == ronda mod 2) {     # si el proceso tiene = paridad que la ronda
      if(i!=n) {
        proc[i+1]?elOtro;            # recibe del siguiente y ordena
        if(elOtro < miValor) swap(elOtro, miValor);
        proc[i+1]!elOtro;
      }
    } elseif(i!=1) {                 # sino, envía su dato al anterior y espera su nuevo
dato
      proc[i-1]!miValor;
      proc[i-1]?miValor;             # recibirá el mismo u otro, según estaban ordenados o
no
    }
  }
}
```

Que un proceso ordene implica 2 mensajes: uno desde el siguiente que le pasa el dato, y otro al siguiente enviando su nuevo dato. Como N es par, hay N/2 rondas impares donde ordenan N/2 procesos, y N/2 rondas pares donde ordenan N/2-1 procesos (el n-ésimo no ordena). Total:

$$\underbrace{N/2 * 2N/2}_{\text{rondas impares}} + \underbrace{N/2 * 2(N/2-1)}_{\text{rondas pares}} = N^2/2 + N^2/2 - N = \mathbf{N^2 - N}$$

TOTAL MENSAJES

¿Cómo modificaría el algoritmo anterior para que termine tan rápido como el arreglo esté ordenado? ¿Esto agrega overhead de mensajes? De ser así, ¿Cuánto?

Se puede usar un proceso coordinador al cual todos los procesos le envían en cada ronda si realizaron algún cambio o no. Si al recibir todos los mensajes el coordinador detecta que ninguno cambió nada en dos rondas consecutivas, les comunica que terminaron. Esto agrega overhead de mensajes ya que se envían mensajes al coordinador y desde el coordinador. Con n procesos tenemos un overhead de 2*n mensajes, o n+1 si existe primitiva de broadcast.

Modifique el algoritmo para usar k procesos. Asuma que n es múltiplo de k.

```
process Proc[i:1..k] {
int largest = n/k, smallest = 1, a[1..k], dato;
  ... inicializar y ordenar arreglo a de menor a mayor ...
  for(ronda=1;ronda<=k;ronda++) {
    # si el proceso tiene = paridad que la ronda, pasa valores para adelante
    if(i mod 2 == ronda mod 2) {
      if(i!=k) {
        proc[i+1]!a[largest];
        proc[i+1]?dato;
        while(a[largest]>dato) {
          ... inserto dato en a ordenado, pisando a[largest] ...
          proc[i+1]!a[largest];
          proc[i+1]?dato;
```

```
        }
      }
    } elseif(i!=1) {
      # si tiene distinta paridad, recibe valores desde atrás
      proc[i-1]?dato;
      proc[i-1]!a[smallest];
      while(a[smallest]<dato) {
        ... inserto dato en a ordenado, pisando a[smallest] ...
        proc[i-1]?dato;
        proc[i-1]!a[smallest];
      }
    }
  }
}
```

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

```
x:=4; y:=2; z:=3;
Co x:=x-z
// z:=z*2
// y:=z+4
Oc
```

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

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

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

x:=x-z Cumple ASV ya que posee una única referencia crítica (Z) y X no es referenciada por otro proceso.
z:=z*2 Cumple ASV porque no tiene referencias críticas por lo tanto, Z puede ser leída por otro proceso.
y:=z+4 Cumple ASV ya que posee una referencia crítica e Y no es referenciada por ningún otro proceso.

Indique los resultados posibles de la ejecución (No es necesario listarlos todos). Justifique.

Cada tarea se ejecuta sin ninguna interrupción hasta que termina (si una sentencia no es atómica se puede cortar su ejecución pero al cumplir ASV la ejecución no se ve afectada) y las llamamos T1, T2 y T3 respectivamente obtenemos el siguiente subconjunto de historias:

T1, T2, T3 =>	x=1	z=6	y=10
T1, T3, T2 =>	x=1	z=6	y=7
T2, T1, T3 =>	x=-2	z=6	y=10
T2, T3, T1 =>	x=-2	z=6	y=10
T3, T1, T2 =>	x=1	z=6	y=7
T3, T2, T1 =>	x=-2	z=6	y=7

El valor de z es siempre el mismo ya que no posee ninguna referencia crítica. Los valores de X e Y se ven afectados por la ejecución de T2 ya que sus resultados dependen de la referencia que hacen a la variable Z que es modificada. Entonces, si T1 y T3 se ejecutan antes que T2 ambas usaran el valor inicial de Z que es 3 obteniendo los resultados x=1 e y=7; en cambio si T2 se ejecuta antes que las demás los resultados serán x=-2 e y=10. Por último, tenemos los casos en que T2 se ejecuta en medio con T1 antes y T3 después o con T3 antes y T1 después.

Describa cómo es la ejecución de sentencias de alternativa e iteración que contienen comunicaciones guardadas.

Sea la sentencia de alternativa con comunicación guardada de la forma:

```
if B1; comunicacion1 → S1
[] B2; comunicacion2 → S2
fi
```

Se evalúan las expresiones booleanas B_i y la sentencia de comunicación.

- Si todas las guardas fallan (todos los B_i son *false*), el *if* termina sin efecto.
- Si al menos una guarda tiene éxito (su B_i es verdadero y la sentencia de comunicación puede realizarse sin demora), se elige una de ellas no determinísticamente.
- Si algunas guardas se bloquean (su B_i es verdadero pero no puede realizarse la comunicación en ese instante), se espera hasta que alguna tenga éxito.

Luego de elegir una guarda exitosa se ejecuta la sentencia de comunicación asociada y, por último, se ejecutan las sentencias S_i relacionadas.

La ejecución de la iteración *do..od* es similar, iterando los pasos anteriores hasta que todas las guardas fallen.

Utilice la técnica de passing the condition para implementar un semáforo fair usando monitores.

```
monitor Semaforo {
cond positivo;
int valor = VALOR_INICIAL;

  procedure P() {
    if(valor) {
      valor--;
    } else {
      wait(positivo);
    }
  }

  procedure V() {
    if(empty(valor)) {
      valor++;
    } else {
      signal(positivo);
    }
  }
}
```

Analice conceptualmente la resolución de problemas con memoria compartida y memoria distribuida. Compare con respecto a facilidad de programación.

La resolución de problemas con memoria compartida requiere el uso de exclusión mutua o sincronización por condición para evitar interferencias entre los procesos, mientras que con memoria distribuida la información no es compartida si no que cada procesador tiene su propia memoria local y para poder compartir información se requiere del intercambio de mensajes evitando así problemas de inconsistencia. Por lo tanto, resulta mucho más fácil programar con memoria distribuida porque el programador puede olvidarse de la exclusión mutua y muchas veces de la necesidad de sincronización básica entre los procesos que es provista por los mecanismos de pasajes de mensajes. Igualmente la facilidad de programación con uno u otro modelo también está dada por el problema particular que deba ser resuelto.

RPC y Rendezvouz

Describa brevemente en qué consisten los mecanismos de RPC y Rendezvous. ¿Para qué tipo de problemas son más adecuados?

RPC sólo provee un mecanismo de comunicación y la sincronización debe ser implementada por el programador utilizando algún método adicional. En cambio, Rendezvous es tanto un mecanismo de comunicación como de sincronización.
En RPC la comunicación se realiza mediante la ejecución de un CALL a un procedimiento, este llamado crea un nuevo proceso para ejecutar lo solicitado. El llamador se demora hasta que el proceso ejecute la operación requerida y le devuelva los resultados. En Rendezvous la diferencia con RPC está en cómo son atendidos los pedidos: el CALL es atendido por un proceso ya existente, no por uno nuevo. Es decir, con RPC el proceso que debe atender el pedido no se encuentra activo sino que se crea para responder al llamado y luego termina; en cambio, con Rendezvous el proceso que debe atender los requerimientos se encuentra continuamente activo. Esto hace que RPC permita servir varios pedidos al mismo tiempo y que con Rendezvous sólo puedan ser atendidos de a uno por vez (se precisaría un pool de procesos aceptando el mismo CALL para poder atender a más de un pedido simultáneamente).
Estos mecanismos son más adecuados para problemas con interacción del tipo cliente/servidor donde la comunicación entre ellos debe ser bidireccional y sincrónica, el cliente solicita un servicio y el servidor le responde con lo solicitado ya que el cliente no debe realizar ninguna otra tarea hasta no obtener una respuesta del servidor.

¿Por qué es necesario proveer sincronización dentro de los módulos RPC? ¿Cómo puede realizarse esta sincronización?

Es necesario proveer sincronización dentro de los módulos porque los procesos pueden ejecutar concurrentemente. Estos mecanismos de sincronización pueden usarse tanto para el acceso a variables compartidas como para sincronizar interacciones entre los procesos si fuera necesario. En RPC existen dos modos de proveer sincronización y depende del modo en que se ejecutan los procesos dentro del módulo:

- *Si los procesos se ejecutan con exclusión mutua*, sólo hay un proceso activo a la vez dentro del módulo, el acceso a las variables compartidas tiene exclusión mutua implícita pero la sincronización por condición debe programarse. Pueden usarse sentencias wait y variables condición.
- *Si los procesos ejecutan concurrentemente* dentro del módulo debe implementarse tanto la exclusión mutua como la sincronización por condición, para esto pueden usarse cualquiera de los mecanismos existentes (semáforos, monitores, PM o Rendezvous).

¿Qué elemento de la forma general de Rendezvous no se encuentra en ADA?

Rendezvous provee la posibilidad de asociar sentencias de *scheduling* y de poder usar los parámetros formales de la operación tanto en las sentencias de sincronización como en las sentencias de scheduling. Ada no provee estas posibilidades.

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?

El límite de mejora se da teniendo 800 procesadores los cuales ejecutan una unidad de tiempo obteniendo un tiempo paralelo de 201 unidades de tiempo. El *speedup* mide la mejora de tiempo obtenida con un algoritmo paralelo comparándola con el tiempo secuencial. S=Ts/Tp=1000/201= **4,97**~5 aunque se utilicen más procesadores el mayor speedup alcanzable es el anterior cumpliéndose así la **ley de Amdahl** que dice que para un problema existe un límite en la paralización del mismo que no depende del número de procesadores sino que depende de la cantidad de código secuencial.

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 sólo 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 haciendo cada uno un merge ordenado de dos streams de entrada en un stream de salida.**
3. **Odd/Even Exchange sort.**

¿Cuántos procesos son necesarios en 1 y 2? Justifique.

Pipeline de filtros: se necesitan N procesos ya que cada uno de ellos calculará un mínimo y pasará el resto de los valores. Para terminar de procesar todos los números hacen falta tantos procesos como números a ordenar.
Red de procesos filtro: Se necesitan N-1 procesos, conectados como un árbol binario con $\log_2 N$ niveles. El nivel i (i=1 a $\log_2 N$) tiene $N/2^i$ procesos.

¿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.

Pipeline: Cada proceso reenvía un mensaje menos que la cantidad de mensajes que recibió; en total se envían $\sum_{i=1..N} N\text{-}i$, es decir la sumatoria de los primeros N-1 números, lo que equivale a **N * (N-1)/2**. Además, se puede sumar los mensajes de EOS: cada proceso excepto el último envía un EOS obteniendo un total de *N-1 mensajes EOS.*
Red: Siendo N la cantidad de números a ordenar y $\log_2 N$ el total de niveles, la cantidad de mensajes es N $\log_2 N$ (cada número atraviesa todo el árbol), a lo que podemos sumarle nuevamente los mensajes de EOS que son N-1 (uno por proceso).
Odd/even Exchange sort: Con N procesos cada uno con 1 valor, la cantidad de mensajes requeridos es de N*(N-1) (Pág.).

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

Pipeline: en un instante determinado se podrán enviar en paralelo tantos mensajes como procesos tenga el pipeline ya que como cada uno de los procesos recibe, procesa y envía pueden todos estarse enviando los valores paralelamente. Cuando ingresa al pipeline el K-ésimo valor, se pueden enviar en paralelo K-1 mensajes (por ejemplo, cuando el tercer valor ingresa al canal del primer proceso recibe, éste estaría descartando uno de los dos anteriores y enviándolos al segundo proceso, habiendo dos mensajes paralelos: la entrada del 3er valor y el valor descartado por el primer proceso).
Red de filtros: cada proceso envía de a un mensaje por vez ya que el funcionamiento interno de cada uno es igual a la de los filtros en un pipeline pero la diferencia está en la distribución e interacción entre ellos. Por lo tanto, se pueden enviar en paralelo tantos mensajes como procesos haya en el nivel en el que se esté dentro de la red.
Odd/even Exchange sort: pueden enviarse en paralelo tantos mensajes como procesos se encuentren involucrados en una ronda ya que todos enviaran un valor. En cada momento se envían N-1 mensajes en paralelo.

Clasificación de arquitecturas paralelas.

- por el tipo de memoria/espacio de direcciones
  - **Memoria compartida**. La interacción se realiza modificando los datos compartidos. Pueden existir problemas de consistencia. Dentro de esta clasificación se pueden usar dos esquemas **UMA** y **NUMA**; con el primero la memoria física es compartida *uniformemente* por todos los procesadores mientras que con el esquema NUMA cada procesador tiene su propia memoria local pero ésta puede ser

accedida por los demás ya que el conjunto de todas las memorias locales forman una única memoria compartida.
  - **Memoria distribuida**. Cada proceso tiene su propia memoria local que es usada únicamente por él y la interacción y el intercambio de información se realiza a través de *pasaje de mensajes*.
- por los mecanismos de control
  - **SISD** (Single Instruction Single Data) -monoprocesador-
  - **SIMD** (Single Instruction Multiple Data) -paralelismo de datos, comparten la linea de control-
  - **MISD** (Multiple Instruction Single Data)
  - **MIMD** (Multiple Instruction Multiple Data) -paralelismo de control y de datos-
    - **MPMD** (Multiple Program Multiple Data)
    - **SPMD** (Single Program Multiple Data)
- por la granularidad
  - Arquitectura de **grano fino**
  - Arquitectura de **grano grueso**
  - Arquitectura de **grano medio**
- por la red de interconexión
  - **Estáticas**: enlaces punto a punto, se utilizan para el pasaje de mensajes (totalmente conectada, estrella, lineal, anillo, árbol estático, mesh, toroidal, mesh 3d, hipercubo)
  - **Dinámicas**: se conectan mediante switches y enlaces de comunicación que normalmente son utilizadas para memoria compartida (multistage, árbol dinámico, bus)

Explique sintéticamente los paradigmas de resolución de problemas concurrente. Ejemplifique.

1. **Paralelismo recursivo**: el problema se descompone en llamadas recursivas que trabajan sobre partes disjuntas del conjunto total de datos (Dividir y conquistar). Las llamadas recursivas se reemplazan por la creación de procesos o tareas en una bolsa. Por ejemplo, el sorting by merging o juegos tipo ajedrez.
2. **Paralelismo iterativo**: un programa consta de un conjunto de procesos cada uno de los cuales tiene 1 o más loops (cada proceso es un programa iterativo). Los procesos cooperan para resolver un único problema, pueden trabajar independientemente, comunicarse y sincronizarse por memoria compartida o pasaje de mensajes. Generalmente, el dominio de datos se divide entre los procesos siguiendo diferentes patrones (slices de filas, columnas, bloques). Un ejemplo es la multiplicación de matrices.
3. **Productores-Consumidores**: Los procesos deben comunicarse y la comunicación fluye en una sola dirección por lo que generalmente se organizan en *pipes* donde la salida de uno será la entrada de su sucesor. Es decir, cada proceso es un *filtro* que consume la salida de su predecesor y produce una entrada para su sucesor.
4. **Cliente-Servidor**: El predominante en los sistemas distribuidos; el servidor es un proceso que espera pedidos de servicio de múltiples clientes. Requiere de comunicación bidireccional y permite que clientes y servidores puedan ejecutarse en diferentes procesadores. La atención de clientes puede ser de a uno por vez (Rendezvous) o varios simultáneamente (RPC).
5. **Pares interactuantes**: Los procesos resuelven partes de un problema, intercambian información mediante mensajes y a veces deben sincronizar para llegar a una solución. Permite mayor grado de asincronismo que en cliente-servidor. Existen diferentes configuraciones posibles: grilla, pipe circular, uno a uno, centralizado, arbitraria.

Analice conceptualmente los modelos de mensajes sincrónicos y asincrónicos. Compárelos en términos de concurrencia y facilidad de programación.

Con PMS el *send* es bloqueante por lo que el emisor no puede realizar otras operaciones mientras espera que el mensaje sea recibido, pero esta primitiva bloqueante provee un punto de sincronización. En términos de concurrencia PMA la maximiza con respecto a PMS ya que su *send* es no bloqueante, proveyendo un *buffering implícito* en los canales. Además PMS tiene más riesgo de *deadlock* por la semántica de la sentencia *send*, e implica muchas veces implementar *procesos asimétricos*, utilizar *comunicación guardada* o crear procesos que actúen de *buffer*.
Desde el punto de vista de facilidad de programación es mejor PMA aunque la sincronización deba ser

implementada por el programador en muchos casos. Igualmente la elección de un tipo u otro con respecto a la facilidad dependerá del tipo de problema que se requiera resolver, ya que un modelo podría simplificar la resolución.

¿En qué consisten las arquitecturas SIMD y MIMD? ¿Para qué tipo de aplicaciones es más adecuada cada una?

SIMD, *Single intruction multiple data*: Todos los procesadores ejecutan las mismas instrucciones pero sobre diferentes datos, en forma sincronizada (*lockstep*). Son para aplicaciones de paralelismo de datos, por ejemplo la multiplicación de matrices. Comparten la unidad de control manteniendo el sincronismo, pero cada procesador tiene sus propios registros locales y unidad aritmético-lógica.
MIMD, *Multiple instruction multiple data*: Cada procesador tiene su propio flujo de instrucciones y su propio conjunto de datos, es decir, cada uno ejecuta su propio programa en un paralelismo total. Pueden utilizar memoria compartida o distribuida. Además pueden subclasificarse en **MPMD** (*Multiple Program Multiple Data*) donde cada procesador ejecuta su propio programa (PVN) y **SPMD** (*Single Program Multiple Data*) donde cada procesador ejecuta una copia del programa (MPI). Cualquier aplicación que no debe hacer lo mismo sobre distintos datos. Por ejemplo un pipeline o un master/worker o donde el trabajo es irregular, es decir que la cantidad de tiempo que lleva cada tarea depende de los datos en sí y no del tamaño de los mismos como es la ordenación de vectores.

Resuelva el problema de encontrar la topología de una red utilizando mensajes asincrónicos. Muestre con un ejemplo la evolución de la matriz de adyacencia para una red con al menos 7 nodos y de diámetro al menos 4.

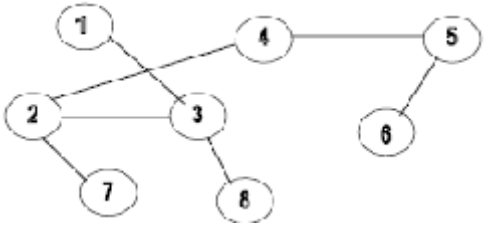
```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)

process nodo[p = 1..n] {
bool vecinos[1:n];                # inicialmente vecinos[q] true si q es vecino de p
bool activo[1:n] = vecinos;       # vecinos aún activos
bool top[1:n,1:n] = ([n*n]false); # vecinos conocidos (matriz de adyacencia)
bool nuevatop[1:n,1:n];
int r = 0; bool listo = false;
int emisor; bool qlisto;

  top[p,1..n] = vecinos;          # llena la fila para los vecinos
  while(not listo) {
    # envía conocimiento local de la topología a sus vecinos
    for[q = 1 to n st activo[q] ] send topologia[q](p,false,top);
    # recibe las topologías y hace OR con su top juntando la información
    for [q = 1 to n st activo[q] ] {
      receive topologia[p](emisor,qlisto,nuevatop);
      top = top or nuevatop;
      if(qlisto) activo[emisor] = false;
    }
    if(todas las filas de top tiene 1 entry true) listo = true;
    r := r + 1;
  }

  # envía topología completa a todos sus vecinos aún activos
  for[q = 1 to n st activo[q] ] send topologia[q](p,listo,top);
  # recibe un mensaje de cada uno para limpiar el canal
  for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop);
}
```

Después de r rondas, el nodo p conocerá la topología a distancia r de él. En particular, para cada nodo q dentro de la distancia r de p, los vecinos de q estarán almacenados en la fila q de *top*. Dado que la red es conectada, cada nodo tiene al menos un vecino. Así, el nodo p ejecutó las suficientes rondas para conocer la topología tan pronto como cada fila de top tiene algún valor true. En ese punto, p necesita ejecutar una última ronda en la cual intercambia la topología con sus vecinos; luego p puede terminar. Esta última ronda es necesaria pues p habrá recibido nueva información en la ronda previa. También evita dejar mensajes no procesados en los canales. Dado que un nodo podría terminar una ronda antes (o después) que un vecino, cada nodo también necesita decirles a sus vecinos cuando termina. Para evitar deadlock, en la última ronda un nodo debería intercambiar mensajes solo con sus vecinos que no terminaron en la ronda previa.



Inicialmente en el nodo 3.

	1	2	3	4	5	6	7	8
1								
2								
3	I	I	I					I
4								
5								
6								
7								
8								

Después de una ronda, en nodo 3.

	1	2	3	4	5	6	7	8
1	T		T					
2		I	I	I			I	
3	T	T	T					T
4								
5								
6								
7								
8			T					T

Después de dos rondas, en nodo 3:

	1	2	3	4	5	6	7	8
1	I		I					
2		T	T	T			T	
3	T	T	T					T
4		T		T	T			
5								
6								
7		T					T	
8			T					T

¿Qué mecanismo de pasaje de mensaje es más adecuado para la resolución? Justifique claramente.

El mecanismo de pasaje de mensajes asincrónico es el más adecuado para la resolución junto con algoritmos heartbeat ya que nos permite una interacción entre procesos de manera que cada procesador es modelizado por un proceso y los links de comunicación con canales compartidos. PMA facilita la codificación al poder utilizarse procesos simétricos sin riesgo de deadlock: cada proceso ejecuta una secuencia de iteraciones enviando su conocimiento local de la topología a todos sus vecinos, y luego recibiendo la información de ellos y combinándola con la suya. La computación termina cuando todos los procesos aprendieron la topología de la red entera. El criterio de terminación no siempre puede ser decidido localmente.

Compare conceptualmente con una solución utilizando PMS.

La solución con PMS dificultaría el algoritmo de forma tal que incrementaría la demora ya que se debe recibir en orden uno por uno los valores de los vecinos (canales 1:1) y si uno de ellos se retrasa no podemos continuar hasta que no recibamos el valor (o utilizar comunicación guardada para evitar estas esperas). Además, se deberían usar algoritmos asimétricos o procesos buffer de forma de evitar deadlock (inicialmente todos los procesos hacen un *send*, por lo que se bloquearían de no considerar este punto).

Defina sincronización entre procesos y mecanismos de sincronización.

La sincronización entre procesos puede definirse como el conocimiento de información acerca de otro proceso para coordinar actividades cuando se necesita acceder a valores compartidos, esperar resultados, etc. Existen dos mecanismos de sincronización:

- **Por exclusión mutua**: Su objetivo es asegurar que sólo un proceso tenga acceso a un recurso compartido en un determinado instante de tiempo. Si un programa tiene secciones críticas que pueden ser compartidas por más de un proceso, este mecanismo evita que varios procesos puedan encontrarse en la misma sección crítica en el mismo momento.
- **Por condición**: Permite bloquear la ejecución de un proceso hasta que se cumpla una determinada condición.

¿Cuál (o cuáles) es el paradigma de interacción entre procesos más adecuado para resolver problemas del tipo "Juego de la vida"?

El paradigma que mejor se adecua es el *heartbeat* ya que permite enviar información a todos los vecinos y luego recopilar la información de todos ellos. Entonces una célula recopila la información de sus vecinos en la cual se basa su próximo cambio de estado.

¿Considera que es conveniente utilizar mensajes sincrónicos o asincrónicos?

Es mejor la utilización de mensajes asincrónicos ya que evitan problemas de deadlock cuando se debe decidir qué proceso envía primero su información. Como en el algoritmo general de heartbeat todos los procesos primero envían su información y luego recopilan la información de sus vecinos se hace intuitivo el uso de PMA para su resolución evitando tener que realizar procesos asimétricos con demora innecesaria, aunque la performance podría mejorar si se utiliza comunicación guardada para las sentencias de envío y recepción.

¿Cuál es la arquitectura de hardware que se ajusta mejor? Justifique claramente sus respuestas.

Es conveniente una arquitectura de grano fino, con mucha capacidad para la comunicación y poca capacidad para el cómputo ya que este tipo de programas es de grano fino muchos procesos o tareas con poco cómputo que requieren de mucha comunicación. Una arquitectura en forma de grilla es una forma óptima para este tipo de problemas.

¿En qué consiste la utilización de relojes lógicos para resolver problemas de sincronización distribuida? Ejemplifique.

Las acciones de comunicación, tanto *send* como *receive*, afectan la ejecución de otros procesos por lo tanto son eventos relevantes dentro de un programa distribuido y por su semántica debe existir un orden entre las acciones de comunicación. Básicamente cuando se envía un mensaje y luego este es recibido existe un orden entre estos eventos ya que el send se ejecuta antes que el receive por lo que puede imponerse un orden entre los eventos de todos los procesos. Sea un ejemplo el caso en que procesos solicitan acceso a un recurso, muchos de ellos podrían realizar la solicitud casi al mismo tiempo y el servidor podría recibir las solicitudes desordenadas por lo que no sabría quién fue el primero en solicitar el acceso y no tendría forma de responder a los pedidos en orden. Para solucionar este problema se podrían usar los relojes lógicos.

Para establecer un orden entre eventos es que se utilizan los relojes lógicos que se asocian mediante un *timestamps* a cada evento. Entonces, un reloj lógico es un contador que es incrementado cuando ocurre un evento dentro de un proceso ya que el reloj es local a cada proceso y se actualiza con la información

distribuida del tiempo que va obteniendo en la recepción de mensajes. Este reloj lógico (rl) será actualizado de la siguiente forma dependiendo del evento que suceda:

- Cuando el proceso realiza un SEND, setea el *timestamp* del mensaje al valor actual de rl y luego lo incrementa.
- Cuando el proceso realiza un RECEIVE con un *timestamp* (ts), setea rl como max (rl, ts+1) y luego incrementa rl.

Con los relojes lógicos se puede imponer un orden parcial ya que podría haber dos mensajes con el mismo timestamp pero se puede obtener un ordenamiento total si existe una forma de identificar unívocamente a un proceso de forma tal que si ocurre un empate entre los *timestamp* primero ocurre el que proviene de un proceso con menor identificador.

Defina el concepto de sincronización barrier. ¿Cuál es su utilidad?

Como muchos problemas pueden ser resueltos con algoritmos iterativos paralelos en los que cada iteración depende de los resultados de la iteración previa es necesario proveer sincronización entre los procesos al final de cada iteración, para realizar esto se utilizan las barreras. Las barreras son un punto de encuentro de todos los procesos luego de cada iteración, aquí se demoran los procesos hasta que todos hayan llegado a dicho punto y cuando lo hagan recién pueden continuar su ejecución.

¿Qué es una barrera simétrica?

Una barrera simétrica es un *conjunto de barreras entre pares de procesos* que utilizan sincronización barrier. En cada etapa los pares de procesos que interactúan van cambiando dependiendo a algún criterio establecido.

Describa combining tree barrier y butterfly barrier. Marque ventajas y desventajas de cada una.

En un **combining tree barrier** los procesos se organizan en forma de árbol y cumplen diferentes roles. Básicamente, los procesos envían el aviso de llegada a la barrera hacia arriba en el árbol y la señal de continuar cuando todos arribaron es enviada de arriba hacia abajo. Esto hace que cada proceso deba combinar los resultados de sus hijos y luego se los pase a su padre.

- *Ventajas*: Implementación es más sencilla.
- *Desventajas*: Los procesos no son simétricos ya que cada uno cumple diferentes roles, por lo que los nodos centrales realizarán más trabajo que los nodos hoja y la raíz.

Con **butterfly barrier** en cada etapa (son $\log_2 N$ etapas) un proceso sincroniza con otro distinto. Es decir, si *s* es la etapa cada proceso sincroniza con uno a distancia 2^{s-1}. Así al final de las $\log_2 N$ etapas cada proceso habrá sincronizado directa o indirectamente con el resto de los procesos.

- *Ventajas*: Procesos simétricos, todos sincronizan con un proceso a distancia 2^s-1 hacia adelante o atrás.
- *Desventajas*: Su implementación es más compleja, y cuando N no es potencia de dos puede usarse un N' próximo potencia de dos para sustituir a los procesos perdidos en cada etapa (o perder la simetría reutilizando procesos); sin embargo esta implementación no es eficiente por lo que se usa una variante llamada *Dissemination barrier*.

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

El que mejor se comporta con mayor número de procesadores es aquel cuya función de speedup es p-1 por definición de speedup su rango esta entre 0 y p entonces con esta función el speedup es más cercano a p. Si comparamos la eficiencia E=S/p entonces tenemos (p-1)/p = 1-1/p en el primer caso, y (p/2)/p = 1/2 en el segundo, cuanto más grande sea p mayor eficiencia tendrá la primer solución ya que su valor será casi uno y la otra solución no alcanzara nunca una mayor eficiencia que la mitad.

Ahora suponga S=1/p y S=1/p2.

Es más eficiente la solución con speedup 1/p, su speedup siempre será mayor. Además su eficiencia será siempre mayor que la de la segunda solución, ambas a medida que crecen los procesadores van disminuyendo su eficiencia pero la función 1/p decrece más lentamente.

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

La interferencia se da cuando un proceso toma una acción que *invalida alguna suposición hecha por otro proceso* y esto se debe a que las acciones de los procesos en un programa concurrente pueden ser intercaladas. La interferencia se da por la realización de asignaciones en un proceso a variables compartidas que pueden afectar el comportamiento o invalidar un supuesto realizado por otro proceso. Por lo tanto, para evitar la interferencia entre procesos se usa la sincronización cuyo rol es restringir el número de historias (interleaving de sentencias) posibles sólo a las deseables y para hacerlo existen dos mecanismos de sincronización: exclusión mutua o por condición.

Casos Problemas de Interés

El oso, las abejas y el tarro de miel: Hay n abejas y un oso hambriento, que comparten un tarro de miel. El tarro inicialmente está vacío, y tiene una capacidad de H porciones. El oso duerme hasta que el tarro está lleno, luego come toda la miel y se vuelve a dormir. Cada abeja repetidamente produce una porción de miel y la pone en el tarro; la que llena el tarro despierta al oso. Resolver con Semáforos.

```
int tarro=0;
sem mutex_tarro = 1, tarro_lleno = 0;

process Abeja[a = 1..N] {
 while(true) {
   P(mutex_tarro);
   tarro++;                          # produce una porción de miel
   if(tarro==H) V(tarro_lleno)
   else V(mutex_tarro);
 }
}

process Oso {
 while(true) {
   P(tarro_lleno);
   tarro=0;                          # come la miel
   V(mutex_tarro);
 }
}
```

Existen procesos O (Oxígeno) y H (Hidrógeno) que en un determinado momento toman un estado ready y se buscan para combinarse y formar una molécula de agua (HH-O). Podría pensar la solución con un esquema de productores y consumidores. ¿Quiénes serían los productores y quienes los consumidores?

El oxígeno podría ser consumidor, una vez que consume dos hidrógenos forma la unión. Otro enfoque sería un tercer proceso que consuma todo y arme la molécula.

```
sem mutex = 1, vacio = N, lleno = 0;
int hidrogenos[1..N] = ([N] 0), pri=1, ult=1;

process Oxigeno[i = 1..O] {
int hidros[1..2];
  while(true) {
    ...
    for [i = 1..2] {
      P(lleno);
      P(mutex);
      hidros[i] = hidrogenos[pri];          # consume x2
      pri = pri mod N +1;
      V(mutex);
      V(vacio);
    }
    ArmarMolecula(i, hidros[1], hidros[2]);  # Simbólico
  }
```

```
}

process Hidrogeno[i = 1..H] {
  while(true) {
    ...
    P(vacio);
    P(mutex);
    hidrogenos[ult] = i;                    # produce
    ult = ult mod N +1;
    V(mutex);
    V(lleno);
  }
}
```

Baño Unisex. Suponga que hay solo un baño en toda la facultad que puede ser usado por varones y mujeres, pero no al mismo tiempo. Sincronice para que el baño siempre sea unisex.

Baño sin límite de capacidad

```
int cant_m, cant_v:= 0;
sem sv, sm:= 0;              # Bv: cant_m == 0,  Bm: cant_v == 0
int dv, dm:= 0;
sem e:= 1;                   # baton

#SIGNAL: if (cant_m == 0 && dv > 0) → dv--; V(sv);
#        [] (cant_v == 0 && dm > 0) → dm--; V(sm);
#        else V(e)

process Varon[1..V]{
  while(true){
    P(e);
    if(cant_m > 0) → dv++; V(e); P(sv);
    cant_v++;
    if(dv > 0) → dv--; V(sv);
      else V(e);
    UsarBaño();
    P(e);
    cant_v--;
    if(cant_v == 0 && dm > 0) → dm--; V(sm);
      else V(e);
  }
}

process Mujer[1..M]{
  while(true){
    P(e);
    if(cant_v > 0) → dm++; V(e); P(sm);
    cant_m++;
    if(dm > 0) → dm--; V(sm);
      else V(e);
    UsarBaño();
    P(e);
    cant_m--;
    if(cant_m == 0 && dv > 0) → dv--; V(sv);
```

```
      else V(e);
  }
}
```

Baño con capacidad fija

```
# Bv: cant_m == 0 && cant_v < x     y    Bm: cant_v == 0 && cant_m < x
#SIGNAL:if (cant_m == 0 && cant_v < x && dv > 0) → dv--; V(sv);
#       [] (cant_v == 0 && cant_m < x && dm > 0) → dm--; V(sm);
#       else V(e)

process Varon[1..V]{
  while(true){
    P(e);
    if(cant_m > 0 || cant_v == X) → dv++; V(e); P(sv);
    cant_v++;
    if(dv > 0 && cant_v < X) → dv--; V(sv);
      else V(e);
    usarBaño();
    P(e);
    cant_v--;
    if (dv > 0) → dv--; V(sv);
    [] (cant_v == 0 && dm > 0) → dm--; V(sm);
    else V(e);
    }
}

process Mujer[1..M]{
  while(true){
    P(e);
    if(cant_v > 0 && cant_m == x) → dm++; V(e); P(sm);
    cant_m++;
    if (cant_m < x && dm > 0) → dm--; V(sm);
      else V(e);
    UsarBaño();
    P(e);
    cant_m--;
    if (cant_m == 0 && dv > 0) → dv--; V(sv);
    [] (dm > 0) dm:= dm - 1; V(sm);
    else V(e);
  }
}
```

QuickSort usando manager-workers

Algoritmo QuickSort secuencial:

```
void quicksort(int[] v, int iz, int der) {
  if (iz < der) {                          # corta recursividad
    int primero=iz, ultimo=der;
    valPivote = v[(primero + ultimo) div 2]; # tomando como pivote al central
    while (iz<der) {
      while(v[iz] < valPivote  && iz < der) iz++;   # por la izq. hasta >pivote
      while(valPivote < v[der] && iz < der) der--;  # por la der. hasta <pivote
```

```
      swap(v[iz], v[der]);
     }
     quicksort(v, primero, iz - 1);
     quicksort(v, iz + 1, ultimo);
   }
}
```

Algoritmo QuickSort con manager, workers y un iniciador (con *rendevouz*):

```
module quicksort
   op createTask(int[] vector, int prim, int ult);
   op getTask(int[] &vector, int &prim, int &ult);
   op result(int pos, int valor);
   op getResults(int[] &vector);
body
   process Manager {
      int[] v, ordenados=0;
      queue of (int, int) tasks;
      while(!empty(tasks) or ordenados<N) {
         in createTask(vector, prim, ult) →
            v[prim:ult]=vector;
            push(tasks, prim, ult);
         [] getTask(vector, prim, ult) and !empty(tasks) →
            pop(tasks, prim, ult);
            vector[1:(ult-prim+1)] = v[prim:ult];   # le pasa el slice a ordenar
         [] result(pos, valor) →
            v[pos] = valor;
            ordenados++;
         ni;
      }
      in getResults(vector) → vector=v; ni;
   }

   process Iniciador {
      int v[1:N] = InicializarVector(); # inicializa con datos a ordenar
      call createTask(v,1,N);           # pide que se ordenen los datos
      call getResults(v);               # espera los datos ordenados
   }

   process Worker[i = 1..W] {
      int v[], p, u, iz, der, pivote;
      while(true) {
         call getTask(v, p, u);         # pide trabajo

         iz=p; der=u;                   # lo resuelve
         pivote = v[(p+u) div 2];       # tomando como pivote al elemento central
         while(iz<der) {
            while(v[iz] < pivote  && iz < der) iz++;
            while(pivote < v[der] && iz < der) der--;
            v[iz]:=:v[der];             # swap
         }

         if(p<iz-1) call createTask(v[p:(iz-1)], p, iz-1) # crea trabajos nuevos
           else call result(v[p], p);                     # o envía resultados
```

```
            if(iz+1<u) call createTask(v[(iz+1):u], iz+1, u)
              else call result(v[u], u);
        }
    }
end quicksort;
```

N reinas con manager-workers

El problema consiste en poner *n* reinas de ajedrez en un tablero de *n* x *n*, de tal manera que ninguna de ellas pueda atacar a otra usando las movidas estándar de una reina de ajedrez (no pueden encontrarse dos reinas en la misma fila, columna o diagonal). Algunas versiones proponen encontrar una solución, y otras todas las soluciones posibles.
Como no puede haber más de una reina por fila ni por columna, puede representarse al tablero como un array 1:N, cada posición representando a una fila y conteniendo el número de columna en que se encuentra una reina. No podría haber valores repetidos en el array -reinas en la misma columna-.
Quedaría por controlar que las reinas no se puedan atacar en movimientos diagonales: si una reina se encuentra en (x1, y1) y otra en (x2, y2), pueden atacarse en diagonal si: x1-y1 == x2-y2 o bien x1+y1 == x2+y2 , la primer posibilidad contempla que se ataquen por una diagonal principal, la segunda por una inversa. También podría evaluarse si: abs((x2-x1)/(y2-y1)) == 1, es decir si los dos puntos describen una recta con pendiente 1 o -1.
El manager podría cargar algunas tareas iniciales (por ejemplo, N tableros con una reina en la primer fila, cada uno en una columna distinta). Los workers tomarían un tablero y crearían tareas con tableros válidos completando una fila libre del mismo. Al completar todas las filas encuentran una solución.

```
module nReinas
   op createTask(int[] filas, int cantReinas);
   op getTask(int[] &filas, int &cantReinas);
   op taskProcessed();   # sólo p/ controlar trabajos en progreso y detectar terminación
   op result(int[] filas);
body
   process Manager {
      int[] f, c;
      queue of (int[], int) tasks;
      queue of (int[]) solutions;
      for [c = 1 to N] { f[1] = c; insert(tasks, f[], 1); }
      c=0;
      while(!empty(tasks) or c>0) {     # mientras hay tareas o trabajo en progreso
         in createTask(filas[], cr) → insert(tasks, filas[], cr);
         [] getTask(filas[], cr) and !empty(tasks) → remove(tasks, filas[], cr); c++;
         [] taskProcessed() → c--;
         [] result(filas[]) → insert(solutions, filas[]);
         ni;
      }
      while(!empty(solutions)) {
         remove(f[], solutions);
         print(f[]);
      }
   }

   process Worker[i = 1..W] {
      int[] filas, cr, c,f;
      bool sirve;
      while(true) {
         call getTask(filas[], cr);       # pide trabajo
```

```
            cr=cr+1;                        # lo resuelve: colocará una reina más
            for [c = 1 to N] {
               # controla si puede usar la columna c en fila cr mirando filas anteriores
               f=0; sirve=TRUE;
               while(f<cr and sirve) {
                  f=f+1;
                  # si c no se usó en f y no chocan diag principal ni inversa
                  sirve=filas[f]<>c and cr-c<>f-filas[f] and cr+c<>f+filas[f];
               }
               if(sirve) {
                  filas[cr]=c;
                  # si ubicó la última reina informa el resultado,
                  # sino crea una tarea para que otro continúe
                  if(cr==N) call result(filas[])
                  else call createTask(filas[], cr);
               }
            }
            call taskProcessed();           # avisa que terminó
         }
      }
   end nReinas;
```

Generalización de Drinking Philosophers

Se tiene un grafo no dirigido G. Los Phil se asocian a nodos del grafo y pueden comunicarse sólo con sus vecinos.
Una botella se asocia con cada arista de G.
Cada Phil cicla entre tres estados: tranquilo, sediento y bebiendo. Un Phil tranquilo se puede volver sediento.
Antes de beber, el Phil debe adquirir la botella asociada con cada arista conectada a su nodo. Luego de beber, un filósofo nuevamente se vuelve tranquilo.
A un Phil tranquilo se le permite responder a pedidos de vecinos por cualquier botella que él tenga.
Diseñe una solución fair y sin deadlock, donde cada Phil ejecute el mismo algoritmo. Use tokens para representar las botellas.

Puede usarse una variación de la solución descentralizada de dinning philosophers utilizando un mozo cada uno. Los mozos se ocupan del manejo de los tokens, en este caso botellas.
Como una botella se asocia a cada arista, es compartida por dos filósofos → al pedirla o entregarla basta con decir quién envía el mensaje -no se precisa NeedR y NeedL sino need(waiter)-. Como precisa todas las botellas asociadas al nodo, se solicitan las botellas de waiters adyacentes que aún no se poseen, y luego se permanece en ciclo hasta que no falte ninguna, prestando mientras las que nos pidan. El resto del tiempo el mozo puede recibir pedidos a los que responderá de inmediato.

```
module Waiter[i = 0 to N]
  op getBottles(), relBottles();     # usadas por los filósofos
  op need(int w), pass(int w);       # para los Waiters
body
  op thirsty(), drink();             # op's locales para sincronizar
  bool have[1:B], used[1:B];         # status de las bottles – se asume inicializado
  bool neighbors[1:N];               # vecinos en el grafo – se asume inicializado

  proc getBottles() {
    send thirsty();                  # dice que Phil tiene sed
    receive drink();                 # espera permiso
```

```
}

process the_Waiter {
  # podría recibir la inicialización desde un process Main.
  # Como la inicialización es específica según la topología del grafo, se asumen
  # los datos ya inicializados en este ejemplo.
  int pending;
  while (true) {
      in thirsty() →
          pending = 0;
          for[n = 1 to N st neighbors[n]] # pide a los vecinos botellas que no tiene
            if (not have[n]) { pending++; send waiter[n].need(i); }
          while (pending>0 )              # espera conseguir todas las botellas
            in pass(n) → have[n]=true; used[n]=false; pending--;
            [] need(n) st used[n] →
                 have[n]=false; used[n]=false; pending++;
                 send waiter[n].pass(i);
                 send waiter[n].need(i);
            ni;
          # al salir del while tiene los tokens -botellas- para su filósofo.
          send drink();                    # lo deja beber
          for[b = 1 to B st have[b]]       # marca todas esas botellas como usadas
            used[b]=true;
          receive relBottles( );           # y espera que las libere
      [] need(n) →                         # vecino n requiere la botella compartida
          have[n]=false; used[n]=false;
          send waiter[n].pass(i);
      ni;
  }
}

process Philosopher[i = 0..N] {
  while(true) {
      …
      Call waiter[i].getBottles();
      BEBER();
      Call waiter[i].relBottles();
      …
  }
}
```

Ejercicios Varios de Prácticas y Parciales

P3E2. Suponga que N personas llegan a la cola de un banco. Una vez que la persona se agrega en la cola no espera más de 15 minutos para su atención, si pasado ese tiempo no fue atendida se retira. Para atender a las personas existen 2 empleados que van atendiendo de a una y por orden de llegada a las personas.

```
tEstado = enum(ESPERANDO, TIMEOUT, ATENDIDO);

process Persona[i = 1 to N] {
tEstado estado;
  ...
  Timer[i].comenzarEspera();
  Banco.esperarTurno(i, estado);
  if(estado == ATENDIDO) {
    ...atención con el cajero no representada...
  }
  ...
}


process Timer[i = 1 to N] {
  Timer[i].esperarPersona();
  delay(15 * 60 * 1000); #15'
  Banco.timeout(i);
}


monitor Timer[i = 1 to N] {
cond llegada;
boolean llego = false;

  procedure comenzarEspera() {
    llego = true;
    signal(llegada);
  }

  procedure esperarPersona() {
    if(not llego) wait(llegada);
  }
}


process Empleado[i = 1 to 2] {
int idPersona;
  ...
  while(true) {
    Banco.esperarPersonas(idPersona);
    .... atender al cliente - no representado....
  }
}
```

```
monitor Banco {
cond hayClientes, turno[1:N];
queue cola of int;
tEstado estadoPersonas[1:N] = ([N] ATENDIDO);

  procedure timeout(int idPersona) {
    if(estadoPersonas(idPersona) == ESPERANDO) {
      estadoPersonas(idPersona) = TIMEOUT;
      signal(turno[i]);
    }
  }

  procedure esperarTurno(int idPersona, tEstado &estado) {
    push(cola, idPersona);
    estadoPersonas[idPersona] = ESPERANDO;
    signal(hayClientes);
    wait(turno[idPersona]);
    estado = estadoPersonas[idPersona];
  }

  procedure esperarPersonas(int idPersona) {
    idPersona = 0;
    while(!idPersona) {
      while(empty(cola)) wait(hayClientes);
      pop(cola, idPersona);
      if(estadoPersonas[idPersona] == ESPERANDO) {
        estadoPersonas[idPersona] = ATENDIDO;
        signal(turno[idPersona]);
      } else {
        idPersona = 0; # ya tuvo su timeout –quedó el Id en la cola porque sino rompia el
fifo
      }
    }
  }
}
```

P3E4. En un entrenamiento de fútbol hay 20 jugadores que forman 4 equipos (cada jugador conoce el equipo al cual pertenece llamando a la función DarEquipo()). Cuando un equipo está listo (han llegado los 5 jugadores que lo componen), debe enfrentarse a otro equipo que también esté listo (los dos primeros equipos en juntarse juegan en la cancha 1, y los otros dos equipos juegan en la cancha 2). Una vez que el equipo conoce la cancha en la que juega, sus jugadores se dirigen a ella. Cuando los 10 jugadores del partido llegaron a la cancha comienza el partido, juegan durante 50 minutos, y al terminar todos los jugadores del partido se retiran (no es necesario que se esperen para salir).

```
process Jugador[j = 1..20] {
int miEquipo, miRival, miCancha;

  miEquipo = darEquipo();
  EquiposAdmin.esperarPartido(miEquipo, miRival, miCancha);
  CanchaAdmin[miCancha].esperarComienzoPartido();

  delay(50 * 60 * 1000);  #jugar contra miRival en miCancha
```

```
}

monitor EquiposAdmin {
cond partidoListo[1:4];
int cont[1:4] = ([4] 0), ultCancha = 1, ultEquipo = 0;
int rivalPartido[1:4] = ([4] 0), canchaPartido[1:4] = ([4] 0);

  procedure esperarPartido(int equipo, int &rival, int &cancha) {
    cont[equipo] = cont[equipo] + 1;
    if(cont[equipo] == 5) {
      if(ultEquipo == 0) {
        ultEquipo == equipo;
        wait(partidoListo[equipo]);
      } else {
        rivalPartido[equipo] = ultEquipo;
        rivalPartido[ultEquipo] = equipo;
        canchaPartido[equipo] = ultCancha;
        canchaPartido[ultEquipo] = ultCancha;
        ultCancha = ultCancha + 1;
        ultEquipo = 0;
        signal_all(partidoListo[equipo]);
        signal_all(partidoListo[ rivalPartido[equipo] ]);
      }
    } else {
      wait(partidoListo[equipo]);
    }
    rival = rivalPartido[equipo];
    cancha = canchaPartido[equipo];
  }
}

monitor CanchaAdmin[1..2] {
int cont = 0;
cond todosListos;

  procedure esperarComienzoPartido() {
    cont = cont + 1;
    if(cont == 10) signal_all(todosListos) else wait(todosListos);
  }
}
```

P3E5. Suponga que en una fábrica de camisas deben realizarse 5000 camisas, en la misma trabajan X operarios. Los operarios entran a la fábrica, una vez que todos han llegado a la fábrica el encargado los agrupa de a cuatro. Cuando todos los operarios conocen el grupo al que pertenecen y se han encontrado con sus compañeros de grupo comienza la fabricación de camisas. Dentro de un grupo se necesitan 8 materiales diferentes para realizar la camisa, los cuales deben conseguir entre los empleados del grupo (existe un encargado para cada tipo de elemento). Una vez que un grupo consiguió los 8 elementos fabrican entre todos la camisa. Cada vez que un grupo realiza una camisa debe conseguir los 8 elementos. Luego de que todas las camisas han sido fabricadas los grupos deben retirarse.
Nota: Maximice la concurrencia. No se deben fabricar camisas de más. No se puede suponer nada sobre los tiempos, es decir, el tiempo en que un operario tarda en buscar los elementos, ni el

tiempo en que tarda un grupo en fabricar una camisa. Suponga X múltiplo de 4.

```
process Operario[i: 1..X] {
int miGrupo, material;
bool producir;

  EncargadoFabrica.formarGrupo(&miGrupo);
  ControlGrupo[miGrupo].esperarCompañeros(&producir);
  while(producir) {
    ControlGrupo[miGrupo].materialNecesario(&material);
    while(material) {
      obtenerMaterial(material); #podria ser con monitor Encargado[material].request();
      ControlGrupo[miGrupo].materialNecesario(&material);
    }
    ControlGrupo[miGrupo].esperarMateriales();
    producirCamisa();
    ControlGrupo[miGrupo].esperarCompañeros(&producir);
  }
}


monitor EncargadoFabrica {  #Recibe operarios y arma grupos
int cont = 0, ultGrupo = 1;
cond llegaronTodos;

  procedure formarGrupo(int &grupo) {
    cont = cont + 1;
    if(cont == X) {
      cont = 0;
      signal_all(llegaronTodos);
    } else wait(llegaronTodos);
    grupo = ultGrupo;
    cont = cont + 1;
    if(cont == 4) {
      cont = 0;
      ultGrupo = ultGrupo + 1;
    }
  }
}


monitor ControlProduccion {  # Cuenta la producción de camisas
int cont = 0;

  procedure produccionPendiente(bool &query) {
    query = (cont <> 5000);
    if(query) cont = cont + 1;
  }
}


monitor ControlGrupo[1..(X/4)] {
int cont = 0, ultMaterial = 1;
bool seguir;
```

```
cond llegaronTodos;

  procedure esperarCompañeros(bool &seguirProduciendo) {
    cont = cont + 1;
    if(cont == 4) {    #barrera para empezar a producir una camisa (si es necesario)
      cont = 0;
      ControlProduccion.produccionPendiente(&seguir);
      signal_all(llegaronTodos);
    } else wait(llegaronTodos);
    seguirProduciendo = seguir;
  }

  procedure materialNecesario(int &material) {
    if(ultMaterial == 9)
      material = 0
    else {
      material = ultMaterial;
      ultMaterial = ultMaterial + 1;
    }
  }

  procedure esperarMateriales() {
    cont = cont + 1;
    if(cont == 4) {  # barrera hasta que hayan conseguido todo
      cont = 0;
      ultMaterial = 1;
      signal_all(llegaronTodos);
    } else wait(llegaronTodos);
  }
}
```

P305 (2012). En una casa viven una abuela y sus N nietos. Además la abuela compró caramelos que quiere convidar entre sus nietos. Inicialmente la abuela deposita en una fuente X caramelos, luego cada nieto intenta comer caramelos de la siguiente manera: si la fuente tiene caramelos el nieto agarra uno de ellos, en el caso de que la fuente esté vacía entonces se le avisa a la abuela quien repone nuevamente X caramelos. Luego se debe permitir que el nieto que no pudo comer sea el primero en hacerlo, es decir, el primer nieto que puede comer nuevamente es el primero que encontró la fuente vacía.
NOTA: siempre existen caramelos para reponer. Cada nieto tarda t minutos en comer un caramelo (t no es igual para cada nieto). Puede haber varios nietos comiendo al mismo tiempo. Resolver con Monitores.

```
process Abuela {
  Fuente.llegue();
  while(true) {
    Fuente.esperarAviso();
    Fuente.depositar();
  }
}

process Nieto[i = 1..N] {
int miT = random();
  Fuente.esperarAbuela();
```

```
  while(true) {
    Fuente.tomarCaramelo();
    delay(miT); # Comer...
  }
}

monitor Fuente {
int cant_caramelos = 0;
boolean llegoAbuela = false, seHizoPedido = false;
cond abuela, vacio, lleno, pedidoListo;

  procedure llegue() {
    depositar();
    llegoAbuela = true;
    signal_all(abuela);
  }

  procedure esperarAbuela() {
    if(!llegoAbuela) wait(abuela);
  }

  procedure depositar() {
    cant_caramelos = X;
    signal(lleno);              # le avisa al nieto que encontró la fuente vacía
  }

  procedure esperarAviso() {
    if(cant_caramelos>0) wait(vacio);
  }

  procedure tomarCaramelo() {
    while(cant_caramelos==0) {
      if(seHizoPedido) {
        wait(pedidoListo);
      } else {
        seHizoPedido=true;
        signal(vacio);          # hace el pedido
        wait(lleno);            # espera que la abuela le llene la fuente
        seHizoPedido = false;   # Cuando puede continuar, limpia el pedido y
        signal_all(pedidoListo); # despierta a los que esperaban detrás suyo
      }
    }
    cant_caramelos--;
  }
}
```

P306 (2012). Suponga una comisión con 50 alumnos. Cuando los alumnos llegan forman una fila, una vez que están los 50 en la fila el jefe de trabajos prácticos les entrega el número de grupo (número aleatorio del 1 al 25) de tal manera que dos alumnos tendrán el mismo número de grupo (suponga que el jefe posee una función DarNumero() que devuelve en forma aleatoria un número del 1 al 25, el jefe de trabajos prácticos no guarda el número que le asigna a cada alumno).

Cuando un alumno ha recibido su número de grupo, busca al compañero que tenga el mismo número de grupo para comenzar a realizar la práctica. Cuando ambos alumnos se encuentran permanecen en una sala realizando la práctica. Al terminar de trabajar, el alumno le avisa al jefe de trabajos prácticos y espera a que su compañero también avise que finalizó.

El jefe de trabajos prácticos, cuando han llegado los dos alumnos de un grupo les devuelve a ambos el orden en que terminó el GRUPO (el primer grupo en terminar tendrá como resultado 1, y el último 25). Resolver con monitores. NOTA: maximizar la concurrencia.

```
process Alumno[i = 1..50] {
int miGrupo, idCompañero = i, orden;
  JTP.asignarGrupo(miGrupo);
  Grupo[miGrupo].esperarCompañero(idCompañero);
  ... hago el tp con mi idCompañero ...
  JTP.terminamos(miGrupo, orden);
}

monitor JTP {
int contador = 0, cont[1..25] = ([25] 0), ordenes[1..25];
cond estanTodos, terminoGrupo[1..25];

  procedure asignarGrupo(int &grupo) {
    contador++;
    if(contador==50) {
      contador = 0;
      signal_all(estanTodos)
    } else {
      wait(estanTodos);
    }
    grupo = DarNumero();
  }

  procedure terminamos(int grupo, int &orden) {
    cont[grupo]++;
    if(cont[grupo] == 2) {
      contador++;
      ordenes[grupo] = contador;
      signal( terminoGrupo[grupo] );
    } else {
      wait(terminoGrupo[grupo]);
    }
    orden = ordenes[grupo];
  }
}

monitor Grupo[i = 1..25] {
int idCompañero; #no es necesario, pero para que sepa con quién trabaja
cond llegoCompañero;

  procedure esperarCompañero(int &id) {
    if(idCompañero != 0) {
      swap(id,idCompañero);
      signal(llegoCompañero);
    } else {
```

```
      idCompañero = id;
      wait(llegoCompañero);
      id = idCompañero;
    }
  }
}
```

P4E3. En una sala de baile deben entrar como mínimo dos bailarines. Existen cuatro tipos de bailarines los de danza (A), los de tango (B), los de salsa (C) y los de rock (D).

En la sala siempre debe haber un bailarín A y uno D. Además la cantidad de bailarines A debe ser mayor que la cantidad de bailarines B y que la cantidad de bailarines C. Dentro de la sala los bailarines bailan 5 minutos y luego deben intentan retirarse de la sala. Modelice el problema utilizando pasaje de mensajes asincrónico.

```
chan entrarA(int idA), entrarB(int idB), entrarC(int idC), entrarD(int idD);
chan salirA(int idA), salirB(int idB), salirC(int idC), salirD(int idD);
chan permisoA[1:A], permisoB[1:B], permisoC[1:C], permisoD[1:D];

process sala {
int qA=0, qB=0, qC=0, qD=0, id;
  while(true) {
    if(not empty(entrarA) and (qA>0 OR not empty(entrarD)) {
      if(qD==0) { receive entrarD(id); qD = qD+1; send permisoD[id]; }
      receive entrarA(id); qA = qA+1; send permisoA[id];
    }
    if(not empty(entrarB) and qA>(qB+1)) {
      receive entrarB(id); qB = qB+1; send permisoB[id];
    }
    if(not empty(entrarC) and qA>(qC+1)) {
      receive entrarC(id); qC = qC+1; send permisoC[id];
    }
    if(not empty(entrarD) and (qD>0 OR not empty(entrarA)) {
      if(qA==0) { receive entrarA(id); qA = qA+1; send permisoA[id]; }
      receive entrarD(id); qD = qD+1; send permisoD[id];
    }
    if(not empty(salirA) and (qA-1)>qB and (qA-1)>qC) {
      receive salirA(id); qA = qA-1; send salirA[id];
    }
    if(not empty(salirB)) {
      receive salirB(id); qB = qB-1; send salirB[id];
    }
    if(not empty(salirC)) {
      receive salirC(id); qC = qC-1; send salirC[id];
    }
    if(not empty(salirD) AND qD>1) {
      receive salirD(id); qD = qD-1; send salirD[id];
    }
  }
}


process bDanza[i: 1..A] {
  send entrarA(i);
```

```
  receive permisoA[i]();
  delay(5 * 60 * 1000); #bailar
  send salirA(i);
  receive permisoA[i];
}


process bTango[i: 1..B] {
  send entrarB(i);
  receive permisoB[i]();
  delay(5 * 60 * 1000); #bailar
  send salirB(i);
  receive permisoB[i];
}


process bSalsa[i: 1..C] {
  send entrarC(i);
  receive permisoC[i]();
  delay(5 * 60 * 1000); #bailar
  send salirC(i);
  receive permisoC[i];
}


process bRock[i: 1..D] {
  send entrarD(i);
  receive permisoD[i]();
  delay(5 * 60 * 1000); #bailar
  send salirD(i);
  receive permisoD[i];
}
```

P4E6. Se desea modelar el funcionamiento de un banco el cual se encarga de cobrar únicamente el servicio de seguro de sus clientes, en el cual existen 5 cajas donde se cobra.

Existen P personas que desean pagar su seguro en el banco. Para esto cada una selecciona la caja donde hay menos personas esperando, una vez seleccionada espera a ser atendido según el orden de llegada. Cuando lo atienden, si esperó más de 15 minutos entonces el banco le regala el cobro del servicio, en caso contrario, debe abonar una cierta cantidad dependiendo de la categoría de cliente (en caso de no pagar justo el empleado debe darle el vuelto).

Si la persona esperó más de 15 minutos, puede optar por levantar una queja. En ese caso, una vez que se retiro de la caja (sin pagar), se dirige al departamento de quejas donde un supervisor toma los datos de la persona (DNI de la persona, monto) y el número de caja que lo atendió. Cuando se han levantado más de 20 quejas para una misma caja el supervisor cierra la caja, dejando pasar gratis a todas las personas que estaban esperando en ella.

Implemente utilizando pasaje de mensajes sincrónico o asincrónico según crea conveniente. MAXIMICE LA CONCURRENCIA. Aclaraciones:

- **Existe una función Costo que retorna la cantidad que debe pagar quien la invoca.**

- **Suponer que nunca se necesitará cerrar todas las cajas.**

- **Existe una función que dado el dni de la persona devuelve si la misma quiere o no realizar una queja.**

```
process Cliente[i = 1 to P] {
bool atendido = false, timeout = false;
int cajero, costo, paga, vuelto;
  send encolarse(i);
  while(!atendido){
    if (!empty(timeout[i])) → receive timeout[i]; timeout = true;     # timeout
    [] (!empty(gratis[i]))  → receive gratis[i]; atendido = true;     # caja cerrada
    [] (!empty(reply[i]))   →                                         # atención
         receive reply[i](cajero);
         atendido = true;
         costo = CostoSeguro();
         if(timeout) {
           send avisoTimeout[cajero];
           if(queja(i)) send quejas(i, costo, cajero);               # asumo i == DNI
         } else {
           paga = pago(costo);
           send cajero[i](costo, paga);
           if(paga!=costo) receive reply[i](vuelto);
         }
    fi
  }
}

process Timer[i: 1..P] {
  receive init[i];
  delay(15 * 60 * 1000);
  send timeout[i];
}

process Cajero[i: 1..5] {
bool cerrada = false, seguir = true;
int persona, costo, paga;
  while(seguir) {
    if(!empty(cierre[i]) →
        receive cierre[i];
        cerrada = true;
        seguir = !empty(cola[i]);
    [] (empty(cierre[i] && !empty(cola[i])) →
        receive cola[i](persona)
        if(cerrada) {
           send gratis[persona];
           registrarPaseGratis(persona);
           seguir = !empty(cola[i]);
        } else {
           send reply[persona](i);
           if(!empty(avisoTimeout[i])) →
               receive avisoTimeout[i];
               registrarPaseGratis(persona);
```

```
            [] (!empty(cajero[i])) →
                receive cajero[i](costo, paga);
                registrarPago(persona, costo);
                if(costo<paga) send reply[i](paga - costo);
            fi;
        }
    fi;
  }
}

process DeptoQueja {
int cliente, costo, cajero;
int cont[1:5] = ([5] 0);
  while(true) {
    receive quejas(cliente, costo, cajero);
    cont[cajero]++;
    if(cont[cajero]==20) {
      send cierre[cajero];
      send cierreCaja(c),
    }
  }
}

process colasAdmin{
int cont[1:5] = ([5] 0), estado[1:5] = ([5] ABIERTA),
int cli, caj
  while(true) {
    if(receive encolarse(cli)) →
        caj = minAbierto(cont, estado);
        cont[caj]++;
        send init[cli];     # timer
        send cola[caj](cli);
    [] (receive cierreCaja(caj)) →
        estado[caj] = CERRADA;
    fi
  }
}
```

P5E1. Se dispone de un sistema compuesto por 1 central y 2 procesos. Los procesos envían señales a la central.

La central comienza su ejecución tomando una señal del proceso 1, luego toma aleatoriamente señales de cualquiera de los dos indefinidamente. Al recibir 1 señal de proceso 2, recibe señales del mismo proceso durante 3 minutos.

El proceso 1 envía una señal que es considerada vieja (se deshecha) si en 2 minutos no fue recibida.

El proceso 2 envía una señal, si no es recibida en ese instante espera 1 minuto y vuelve a mandarla (no se deshecha).

```
TASK central IS
```

```
    ENTRY señalP1;
    ENTRY señalP2;
    ENTRY timeout;
END;
TASK p1;
TASK p2;
TASK timer IS
    ENTRY start;
    ENTRY end;
END;

TASK BODY central IS
VAR sigo: boolean;
BEGIN
    ACCEPT señalP1;
    LOOP
        SELECT ACCEPT señalP1;
        OR     ACCEPT señalP2;
               sigo:=TRUE;
               timer.start;
               WHILE(sigo) LOOP
                  SELECT ACCEPT timeout;
                         sigo:=FALSE;
                  OR WHEN(timeout'count=0) => ACCEPT señalP2();
                  END SELECT;
               END LOOP;
        END SELECT;
    END LOOP;
    timer.end();
END central;

TASK BODY p1 IS
BEGIN
    LOOP
        ...
        SELECT central.señalP1();
        OR DELAY(2 * 60 * 1000)
           NULL;    -- desecha la señal
        END SELECT;
        ...
    END LOOP;
END p1;

TASK BODY p2 IS
var sigo: Boolean;
BEGIN
    LOOP
        ...
        sigo:=TRUE;
        WHILE(sigo) LOOP
           SELECT central.señalP2();
                  sigo:=FALSE;
           ELSE   DELAY(60 * 1000)
```

```
            END SELECT;
        END LOOP;
        ...
    END LOOP;
END p2;


TASK BODY timer IS
VAR sigo:Boolean:=TRUE;
BEGIN
    WHILE(sigo) LOOP
        SELECT ACCEPT start();
               DELAY(3 * 60 * 1000);
               central.timeout();
        OR     ACCEPT end();
               sigo:=FALSE;
        END SELECT;
    END LOOP;
END timer;
```

P5E3. Suponga que existen N usuarios que deben ejecutar su programa, para esto comparten K procesadores.

Los usuarios solicitan un procesador al administrador. Una vez que el administrador les entregó el número de procesador, el usuario le da su programa al procesador que le fue asignado. Luego el usuario espera a que:

- **El procesador le avise si hubo algún error en una línea de código con lo cual el usuario arregla el programa y se lo vuelve a entregar al procesador, es decir queda nuevamente en la cola de programas a ejecutar por su procesador. El usuario no termina hasta que el procesador haya ejecutado su programa correctamente (sin errores).**

- **El procesador le avise que su programa terminó, con lo cual termina su ejecución.**

El administrador tomará los pedidos de procesador hechos por los usuarios y balanceara la carga de programas que tiene cada procesador, de esta forma le entregará al usuario un número de procesador.
El procesador ejecutará un Round-Robin de los programas listos a ejecutar. Cada programa es ejecutado línea por línea por medio de la función EJECUCIÓN la cual devuelve:

1 error en la ejecución.

2 normal.

3 fin de programa.

Nota: Suponga que existe también la función LineaSiguiente que dado un programa devuelve la línea a ser ejecutada. Modelice con Ada. Maximice la concurrencia en la solución.

```
PROCEDURE EjercicioExtra IS
```

```
TASK administrador IS
  ENTRY request(p: OUT Integer);
  ENTRY release(p: IN Integer);
END;
TASK TYPE usuario IS
  ENTRY setPid(id: IN Integer);
  ENTRY error();
  ENTRY success();
END;
TASK TYPE procesador IS
  ENTRY setPid(id: IN Integer);
  ENTRY end();
  ENTRY ejecutar(u: IN Integer; prog: IN tPrograma);
  ENTRY timeout();
END;
TASK TYPE timer IS
  ENTRY setPid(id: IN Integer);
  ENTRY end();
  ENTRY start();
  ENTRY stop();
END;

usuarios: array(1..N) of usuario;
procesadores: array(1..K) of procesador;
timers: array(1..K) of timer;

TASK BODY administrador IS
atendidos, pmin: Integer:= 0;
cont: array(1..K) of Integer;
BEGIN
  cont:=(1..K => 0);  --inicializo en 0
  WHILE(atendidos<N) LOOP
      SELECT ACCEPT request(p: OUT Integer) DO
                pmin:=1;    --busca la CPU con menos procesos
                FOR c IN 2..K LOOP
                   if(cont(c)<cont(pmin)) then pmin:=c;
                END LOOP;
                p:=pmin;
             END request;
             cont(pmin):=cont(pmin)+1;
      OR     ACCEPT release(p: IN Integer) DO
                cont(p):=cont(p)-1;
             END release;
             atendidos:=atendidos+1;
      END SELECT;
  END LOOP;
  FOR p IN 1..K LOOP
      procesadores(p).end();
  END LOOP;
END;

TASK BODY usuario IS
programa: tPrograma;
```

```
pid, p: integer;
sigo: Boolean:=TRUE;
BEGIN
  ACCEPT setPid(id: IN Integer) DO
      pid:=id;
  END setPid;
  programa:=HacerPrograma();
  administrador.request(p);
  procesadores(p).ejecutar(pid, programa);
  WHILE(sigo) LOOP
      SELECT ACCEPT success();
             sigo:=FALSE;
      OR     ACCEPT error();
             programa:=HacerPrograma(programa);
             procesadores(p).ejecutar(pid, programa);
      END SELECT;
  END LOOP;
  administrador.release(p);
END;

TASK BODY procesador IS
pid, result, pri, ult: Integer:=0;
colaPID: array(0..(N/K)) of Integer;
colaPRG: array(0..(N/K)) of tPrograma;
sigo, ejecuto:boolean:=TRUE;
BEGIN
  ACCEPT setPid(id: IN Integer) DO
      pid:=id;
  END setPid;
  WHILE(sigo) LOOP
      SELECT ACCEPT end();
             sigo:=FALSE;
      OR     ACCEPT ejecutar(u: IN Integer; prog: IN tPrograma) DO
               colaPID(ult):=u;
               colaPRG(ult):=prog;
             END ejecutar;
             ult:=(ult+1) mod ((N/K)+1);
      ELSE   IF(pri/=ult) THEN --hay elementos. /= es distinto en ADA (!=,<>)
               ejecuto:=TRUE;
               timers(pid).start();  --quantum RR
               WHILE(ejecuto) LOOP
                  SELECT ACCEPT timeout();
                         ejecuto:=FALSE;
                         colaPID(ult):=colaPRG(pri);
                         colaPRG(ult):=colaPRG(pri);
                         ult:=(ult+1) mod ((N/K) + 1);
                  ELSE
                         LineaSiguiente(colaPRG(pri));
                         result:=Ejecucion(colaPRG(pri));
                         IF(result/=2) THEN
                             ejecuto:=FALSE;
                             IF(result=1) usuarios(colaPID(pri)).error();
                             ELSE         usuarios(colaPID(pri)).success();
```

```
                        ENDIF;
                    END SELECT;
                END LOOP;
                IF(result/=2) THEN --NO salio por timeout. /= es distinto
                    --por si hizo timeout justo cuando hubo error o terminó el prog
                    SELECT ACCEPT timeout();
                    ELSE   timer(pid).stop();
                    END SELECT;
                END IF;
                pri:=(pri+1) mod ((N/K) + 1);
              ENDIF
    END LOOP;
    timers(pid).end();
  END;

  TASK BODY timer IS
  sigo: Boolean:=TRUE;
  pid: Integer;
  BEGIN
    ACCEPT setPid(id: IN Integer) DO
        pid:=id;
    END setPid;
    WHILE(sigo) LOOP
        SELECT ACCEPT end();
               sigo:=FALSE;
        OR     ACCEPT start();
               SELECT ACCEPT stop();
               OR DELAY(RRQuantum);
                      procesador(pid).timeout();
               END SELECT;
        END SELECT;
    END LOOP;
  END;

BEGIN
    --El programa principal asigna los ID's
    FOR p IN procesadores'lrange LOOP
        procesadores(p).setPid(p);
        timers(p).setPid(p);
    END LOOP;
    FOR u IN usuarios'lrange LOOP
        usuarios(u).setPid(u);
    END LOOP;
END EjercicioExtra;
```

1. Suponga un juego donde hay 30 competidores. Cuando los jugadores llegan avisan al encargado, una vez que están los 30 el encargado les entrega un numero aleatorio del 1 al 15 de tal manera que dos competidores tendrán el mismo número (Supongo que existe una función DarNumero() que devuelve en forma aleatoria un numero del 1 al 15; el encargado no guarda el numero que les asigna a los competidores). Una vez que ya se entregaron los 30 números, los competidores buscaran concurrentemente su compañero que tenga el mismo numero (tenga en cuenta que pueden empezar a buscar cuando todos los competidores tengan el número no antes; además la búsqueda de un jugador no interfiere con la búsqueda de otros que tengan distinto

número). Cuando los competidores se encuentran permanecen en una sala durante 15 minutos y dejan de jugar. Luego cada uno de los competidores avisa que finalizó para luego irse ambos; el encargado cuando llega el segundo competidor les devuelve a ambos el resultado que obtuvieron que es el orden en que se van (los primeros en irse tendrán como resultado 1, los últimos 15).

Para modelizar el tiempo utilice delay(x) que produce un retardo de x minutos. Modelice con Semáforos.

```
sem llegue[1:30] = ([30] 0), esperoNum[1:30] = ([30] 0), mutexCompañero[1:15] = ([15] 1),
avisoSalida = 0, mutexSalida = 1, avisoResultado[1:15] = ([15] 0);
int num, grupo[1:15] = ([15] 0);

process jugador[i = 1 to 30] {
int nro, compañero, res;
  V(llegue[i]);     # avisa que llego
  P(esperoNum[i]);  # espera que le den su numero
  nro = num;
  V(llegue[i]);     # avisa que ya tomo el numero que le dieron
  P(esperoNum[i]);  # espera que todos tengan numero

  P(mutexCompañero[nro]);
  if(grupo[nro] == 0) {  # el primero de ese grupo
    grupo[nro] = i; # avisa quien es, deja el mutex y espera q su compañero lo despierte
    V(mutexCompañero[nro]);
    P(esperoNum[i]);
    compañero = grupo[nro];
  } else {      # el segundo en llegar
    compañero = grupo[nro];  # toma el id de su compañero, deja el suyo y lo despierta
    grupo[nro] = i;
    V(mutexCompañero[nro]);
    V(esperoNum[compañero]);
  }

  Delay(15);

  P(mutexSalida);
  num = nro;
  V(avisoSalida);
  P(avisoResultado[nro]);
  res = grupo[nro];
}

process encargado {
int i, cont=0, salidas[1:15] = ([15] 0);
  for i = 1 to 30 {
    P(llegue[i]);
  }
  for i = 1 to 30 {
    num = DarNumero();  # asumo que repite 2 veces cada numero
    V(esperoNum[i]);
    P(llegue[i]);
  }
```

```
  for i = 1 to 30 { V(esperoNum[i]); } # avisa a todos que ya se repartieron los numeros

  for i = 1 to 30 {
    P(avisoSalida);
    salidas[num]++;
    if(salidas[num]==2) {
      grupo[num]= ++cont;
      V(avisoResultado[nro]);
      V(avisoResultado[nro]);
    }
    V(mutexSalida);
  }
}
```

2. Se tienen N procesos que comparten el uso de una CPU.

Un proceso, cuando necesita utilizar la CPU, le pide el uso, le proporciona el trabajo que va a ejecutar y el tiempo que insume ejecutarlo. Luego, el proceso se queda dormido hasta que la CPU haya finalizado de ejecutar su trabajo, momento en que es despertado y prosigue su ejecución normal.

La CPU funciona de la siguiente manera. Cada vez que el proceso i (i:1..N) pide CPU esta lo pone en el lugar i-ésimo de una lista de procesos esperando ejecutar. La CPU recorre circularmente la lista de procesos esperando a ser ejecutados -del 1 al N-, si el proceso está listo lo saca de la lista y lo ejecuta durante un lapso de 10mls -o un lapso menor, si el tiempo de ejecución del proceso es menor a 10mls).
Una vez que ejecutó un proceso, si todavía le queda al proceso tiempo de ejecución lo vuelve a poner en su lugar de la lista y pasa al siguiente. Si el proceso terminó de ejecutar su trabajo, la CPU lo despierta para que continúe su ejecución y continúa con el siguiente proceso de la lista. Cuando la lista está vacía, la CPU no debe hacer nada, simplemente debe esperar a que algún proceso ingrese a la lista de procesos en espera de ejecución, para empezar a trabajar.

Tenga en cuenta que existe la función delay(x) que retarda un proceso durante x mls. La ejecución de un proceso puede ser modelizada utilizando esta función.

No haga suposiciones acerca de los tiempos de ejecución de los procesos. Modelice utilizando monitores.

```
process proceso[i = 1 to N] {
tTrabajo miT;
  while(true) {
    ...
    miT = Trabajo(i);
    scheduler.ejecutar(i, miT, CalcularTiempo(miT));
    ...
  }
}


process CPU {
tTrabajo unT;
```

```
int t;
  while(true) {
    scheduler.getTrabajo(unT, t);
    delay( min(10, t) );  # ejecuta
    if(t<10) scheduler.termino()
    else     scheduler.vencioQuantum(unT, t-10);
  }
}


monitor scheduler {
cond hayTrabajos, terminado[1:N];
int actual = 0, qT = 0, tiempos[1:N] = ([N] 0);
tTrabajo trabajos[1:N] = ([N] Null);

  procedure ejecutar(int proc, tTrabajo unT, int t) {
    tiempos[proc] = t;
    trabajos[proc] = unT;
    qT++;
    signal(hayTrabajos);
    wait(terminado[proc]);
  }

  procedure getTrabajo(tTrabajo &unT, int &t) {
    if(qT==0) wait(hayTrabajos);  # como hay una sola CPU no precisa ser while
    actual = actual mod N +1;
    while(tiempos[actual]==0)  actual = actual mod N +1;
    unT = trabajos[actual];
    t = tiempos[actual];
  }

  procedure vencioQuamtum(tTrabajo unT, int t) {
    tiempos[actual] = t;
    trabajos[actual] = unT;
  }

  procedure termino() {
    tiempos[actual] = 0;
    trabajos[actual] = Null;
    qT--;
    signal(terminado[actual]);
  }
}
```

2. M Personas concurren a un gimnasio para hacer su rutina diaria. Cada persona cuando llega al gimnasio da su nombre, su bolso con ropa para bañarse y su portafolio o cartera personal (dependiendo de si es hombre o mujer) a la secretaria. La secretaria debe verificar que la persona tenga la cuota paga, si es así, la deja ingresar avisándole que puede guardar sus pertenencias en un locker.

Una vez que la persona ingresó al gimnasio pide que alguno de los J profesores la atienda (el profesor que se le debe asignar a la persona es aquel que menos personas esté atendiendo en

ese momento). El profesor que la atiende toma su rutina y le indica cuál tipo de máquina utilizar. La persona busca una máquina del tipo (la que hace más tiempo que no se utiliza) y si en ese momento la máquina no está disponible, le pide a su profesor que le diga otro tipo de máquina a utilizar. La persona se retira del gimnasio cuando termina su rutina o cuando 5 máquinas no estuvieron disponibles, en cualquiera de los casos debe retirar sus pertenencias personales.

NOTAS: La persona no conoce el número de su locker particular.

Existen N máquinas de cada tipo. La persona no elige con qué profesor hace su rutina, una vez que consigue uno trabaja con ese hasta que se retira del gimnasio y tampoco elige cuál máquina de cada tipo utiliza.

Cada persona utiliza cada máquina durante 5 minutos.

Un profesor puede atender a 0, 1 o más personas.

Suponga que cada máquina dispone de una función que le retorna cuanto tiempo hace que está disponible.

Maximice la concurrencia para TODAS las tareas. Modelice en ADA.

```
TASK TYPE persona;
TASK secretaria IS;
  ENTRY verificar(nombre: IN STRING; debes: OUT Boolean);
  ENTRY guardar(pertenencias: IN tPertenencias; nroLocker: OUT Integer);
  ENTRY retirar(pertenencias: OUT tPertenencias; nroLocker: IN Integer);
END;
TASK administradorProfesores IS;
  ENTRY request(prof: OUT Integer);
  ENTRY release(prof: IN Integer);
END;
TASK TYPE profesor IS;
  ENTRY pedidoAtencion(nombre: IN STRING; tipoMaq: OUT Integer);
END;
TASK TYPE adminMaquinas IS;  --habrá un admin. por cada tipo para maximizar la
concurrencia
  ENTRY request(nroMaq: OUT Integer);
END;
TASK TYPE maquina IS;
  ENTRY tiempoLibre(tiempo: OUT Integer);
  ENTRY enUso();
  ENTRY release();
END;

personas: array(1..M) of persona;
profesores: array(1..J) of profesor;
administradorMaquina: array(1..CantTipos) of adminMaquinas;
maquinas: array(1..CantTipos, 1..N) of maquina;


TASK BODY secretaria IS; --por lo que dice el enunciado le dan a ella las pertenencias así
```

```
que asumo que controla los lockers
BEGIN
  LOOP
    SELECT ACCEPT verificar(nombre: IN STRING; debes: OUT Boolean) DO
         debes:=TieneDeuda(nombre);
         END verificar;
    OR   ACCEPT guardar(pertenencias: IN tPertenencias; nroLocker: OUT Integer) DO
         nroLocker:=LockerLibre();
         guardar(pertenencias, nroLocker);
         END guardar;
    OR   ACCEPT retirar(pertenencias: OUT tPertenencias; nroLocker: IN Integer) DO
         Retirar(pertenencias, nroLocker);
         liberar(nroLocker);
         END retirar;
    END SELECT;
  END LOOP;
END;


TASK BODY administradorProfesores IS;
cont: array(1..J) of Integer;
BEGIN
  cont:=(1..J => 0);
  LOOP
    SELECT ACCEPT request(prof: OUT Integer) DO
           prof:=PosicionDelMinimo(cont);
           cont(prof):=cont(prof)+1;
         END request;
    OR     ACCEPT release(prof: IN Integer) DO
           cont(prof):=cont(prof)-1;
         END release;
    END SELECT;
  END LOOP;
END;


TASK BODY profesor IS;
BEGIN
  LOOP
    ACCEPT pedidoAtencion(nombre: IN STRING; tipoMaq: OUT Integer) DO
      tipoMaq:=ProxTipoMaquina( Rutina(nombre) );  --0 si termino la rutina
    END pedidoAtencion;
  END LOOP;
END;


TASK BODY adminMaquinas IS;  --un admin. x c/tipo para maximizar la concurrencia
mintime, pmin, time: Integer;
BEGIN
  LOOP
    ACCEPT request(nroMaq: OUT Integer) DO  --0: no hay libre, sino nro de maq a usar
      mintime:=99999;
      pmin:=0;
```

```
      FOR m IN maquinas'lrange(2) LOOP
        maquinas(#PID, m).tiempoLibre(time);
        if(time<mintime)and(time>0) then
          mintime:=time;
          pmin:=m;
        end if
      END LOOP
      nroMaq:=pmin;
      if(pmin>0) then maquinas(#PID, pmin).enUso(); end if;
    END request;
  END LOOP
END;


TASK BODY maquina IS;
disponible: Boolean:=True;
BEGIN
  LOOP
    SELECT ACCEPT tiempoLibre(tiempo: OUT Integer) DO
             if(disponible) then tiempo:=TiempoDisponible();
             else tiempo:=-1; --indica que está ocupada
             end if;
           END tiempoLibre;
    OR     ACCEPT enUso();
             disponible:=False;
    OR     ACCEPT release();
             disponible:=True;
    END SELECT;
  END LOOP;
END;


TASK BODY persona;
nombre: String;
lock, prof, tipo, nro, qNoDisp: Integer:=0;
debo: Boolean;
sigo: Boolean:=True;
pertenencias: tPertenencias;
BEGIN
  nombre:=NombrePersona(#PID);
  pertenencias:=PertenenciasPersona(#PID);
  secretaria.verificar(nombre, debo);
  if(not debo) then
    secretaria.guardar(pertenencias, lock);
    administradorProfesores.request(prof);
    WHILE(sigo) LOOP
      profesores(prof).pedidoAtencion(nombre, tipo);
      if(tipo=0) then
        sigo:=False;
      else
        adminMaquinas(tipo).request(nro);
        if(nro=0) then
          qNoDisp:=qNoDisp+1;
```

```
          if(qNoDisp=5) then sigo:=False; end if;
        else
          DELAY(5 * 60 * 1000);  --usa la máquina 5 minutos
          maquinas(tipo, nro).release();
        end if;
      end if;
    END LOOP;
    administradorProfesores.release(prof);
    secretaria.retirar(pertenencias, lock);
  end if;
END;
```

2. Una pizzería hay 5 maestros pizzeros los cuales elaboran pizzas de la siguiente manera. Cuando un maestro está disponible realiza la pizza y la deposita en una bandeja de pizzas listas. El pizzero trabaja 3 horas y luego se retira.

NOTAS: en el caso que durante la elaboración de la pizza transcurren las 3 horas, el pizzero termina esa pizza y luego se retira. No se puede suponer nada sobre el tiempo que requiere hacer la pizza.

>> Mensaje Asincrónico:

```
chan start[1:5], timeout[1:5], produccion, termino;

process Timer[i = 1 to 5] {
  receive start[i];
  delay(3 * 60 * 60 * 1000);  #3 horas
  send timeout[i];
}

process Pizzero[i = 1 to 5] {
  send start[i];      #inicia timer
  while(empty(timeout[i])) {
    hacerPizza();
    send produccion;
  }
  receive timeout[i];
  send termino;
}

process Bandeja {
  int cont = 0, t=0;
  while(t<5) {
    if(!empty(termino)) → receive(termino); t++            # cuenta los que terminaron de
trabajar
    [] (!empty(produccion)) → receive(produccion); cont++  # cuenta la producción
    fi;
  }
}
```

>> Mensaje Sincrónico: Se precisa un proceso coordinador que le diga al pizzero si puede producir o si ya tiene timeout. El pizzero pide permiso explícitamente y espera una de las dos respuestas posibles. Esto es porque no se puede preguntar por el empty de un canal sincrónico.

```
chan start[1:5], timeout[1:5], reqCoordinador[1:5], fin[1:5], produccion[1:5],
termino[1:5], permiso[1:5];

process Timer[i = 1 to 5] {
  start[i]?;
  delay(3 * 60 * 60 * 1000); #3 horas
  timeout[i]!; # avisa al coordinador para que no de más permiso de producción
}

process Pizzero[i = 1 to 5] {
bool sigo=true;
  start[i]!;           # inicia timer
  reqCoordinador[i]!;  # pide permiso
  while(sigo) {
    if(fin[i]?) → sigo=false;
    [] (permiso[i]?) → hacerPizza(); produccion[i]!; reqCoordinador[i]!;
    fi;                # pudo haber un buffer que reciba los avisos
  }
  termino[i]!;
}

process Coordinador[i = 1 to 5] {
bool sigo=true, vencio=false;
  while(sigo) {
    if(reqCoordinador[i]?) →
       if(vencio) { fin[i]!; sigo = false; }
       else permiso[i]!;
    [] (timeout[i]?) → vencio = true;
    fi;
  }
}

process Bandeja {
int cont = 0, t=0;
  while(t<5) {
    # cuenta los que terminaron de trabajar
    # como los canales son 1:1 usa un cuantificador
    if([i=1 to 5]termino[i]?) → t++;
    [] if([i=1 to 5]produccion[i]?) → cont++;  # cuenta la producción
    fi;
  }
}
```

>> con ADA:

```
PROCEDURE Pizzeria IS
   TASK TYPE pizzero IS
     ENTRY setPid(id: IN Integer);
     ENTRY timeout();
   END;
   TASK TYPE timer IS
     ENTRY setPid(id: IN Integer);
     ENTRY start();
   END;
```

```
   TASK bandeja IS
     ENTRY nuevaPizza();
     ENTRY termino();
   END;

   pizzeros: array(1..5) of pizzero;
   timers: array(1..5) of timer;

   TASK BODY pizzero IS;
   pid: Integer;
   sigo: boolean := TRUE;
   BEGIN
     ACCEPT setPid(id: IN Integer) DO
       pid:=id;
     END setPid;
     timer(pid).start();  --inicializa timer
     WHILE(sigo) LOOP
       SELECT ACCEPT timeout(); sigo:=FALSE;  --corte
       ELSE
          HacerPizza();                       --produccion
          bandeja.nuevaPizza();
       END SELECT;
     END LOOP;
     bandeja.termino();      --avisa a la bandeja que se retira
   END;

   TASK BODY timer IS;
   pid: Integer;
   BEGIN
     ACCEPT setPid(id: IN Integer) DO
       pid:=id;
     END setPid;
     ACCEPT start();
     DELAY(3 * 60 * 60 * 1000);  --3 horas
     pizzeros(pid).timeout();
   END;

   TASK BODY bandeja IS;
   c, prod: Integer:=0;
   BEGIN
     WHILE(c<5) LOOP
         SELECT ACCEPT termino(); c:=c+1;
         OR     ACCEPT nuevaPizza(); prod:=prod+1;
         END SELECT;
     END LOOP;
     --se produjeron prod pizzas
   END;

BEGIN
   -- el programa principal asigna los ID's de pizzeros y timers.
   FOR p IN pizzeros'lrange LOOP
      pizzeros(p).setPid(p);
      timers(p).setPid(p);
```

```
   END LOOP;
END Pizzeria;
```

3. Un banco decide entregar promociones a sus clientes por medio de su agente de prensa, el cual lo hace de la siguiente manera: el agente debe entregar 50 premios entre los 1000 clientes; para esto, obtiene al azar un número de cliente y le entrega el premio. Una vez que este lo toma, continúa con la entrega.

NOTAS: Cuando los 50 premios fueron entregados el agente y los clientes terminan su ejecución. No se puede utilizar una estructura de tipo arreglo para almacenar los premios de los clientes.

Supongo que dice lo del arreglo porque hay que evitar darle 2 premios al mismo cliente... hice que el agente le consulte al cliente elegido, si le dice que ya tiene premio elige a otro.

>> por Mensajes Asincrónicos: los canales pueden ser compartidos

```
chan consulta[1:1000], respuesta(bool), premio[1:1000], gracias, fin;

process agente {
int cli, c = 0;
bool yaTiene;
  while(c<50) {
    yaTiene:=True; //clientes al azar hasta q alguno diga q no tiene premio
    while(yaTiene) {
       cli=obtenerCliente();
       send consulta[cli];
       receive respuesta(yaTiene);
    }
    c++;
    send premio[cli];
    receive gracias;  //debe esperar a que el cliente tome el premio
  }
  fa [c = 1 to 1000]
    send fin;
  af
}

process cliente[i = 1 to 1000] {
bool yaTengo = False;
  while(empty(fin)) {
    if(!empty(consulta[i])) →
            receive consulta[i]; send respuesta(yaTengo);
    [] if(!empty(premio[i])) →
            receive premio[i]; yaTengo=True; send gracias;
  }
  receive fin;
}
```

>> por Mensajes sincrónicos: los canales son punto a punto, así que tengo que usar arrays en todos los casos

```
chan consulta[1:1000], respuesta[1:1000](bool), premio[1:1000], fin[1:1000];

process agente {
```

```
int cli, c = 0;
bool yaTiene;
  while(c<50) {
    yaTiene:=True; #clientes al azar hasta q alguno diga q no tiene premio
    while(yaTiene) {
      cli=obtenerCliente();
      consulta[cli]!;
      respuesta[cli]?yaTiene;
    }
    c++;
    # por ser sincrónico el envío ya es bloqueante, no preciso que agradezan
    premio[cli]!;
  }
  fa [c = 1 to 1000]
    fin[c]!
  af
}

process cliente[i = 1 to 1000] {
bool yaTengo = False, sigo = True;
  while(sigo) {
    if(fin[i]?) → sigo=False;
    [] (consulta[i]?) → respuesta[i]!yaTengo;
    [] (premio[i]?) → yaTengo=True;
    fi;
  }
}
```

>> con ADA:

```
TASK agente;
TASK TYPE cliente IS
  ENTRY consulta(yaTengo: OUT Boolean);
  ENTRY premio();
  ENTRY fin();
END;

clientes: array(1..1000) of cliente;

TASK BODY agente IS;
cli: Integer:=0;
yaTiene: Boolean;
BEGIN
  FOR c IN 1..50 LOOP
    yaTiene:=True;
    WHILE(yaTiene) LOOP
      cli:=ObtenerCliente();            --simbólico
      clientes(cli).consulta(yaTiene);
    END LOOP;
    clientes(cli).premio();
  END LOOP;
  FOR cli IN clientes'lrange LOOP
    clientes(cli).fin();
  END LOOP;
```

```
END;

TASK BODY cliente IS
tengo: Boolean:=False;
sigo: Boolean:=True;
BEGIN
  WHILE(sigo) LOOP
    SELECT  ACCEPT fin();
            sigo:=False;
    OR      ACCEPT consulta(yaTengo: OUT Boolean) DO
              yaTengo:=tengo;
            END consulta;
    OR      ACCEPT premio;
            tengo:=True;
    END SELECT;
  END LOOP;
END;
```

En una carpintería se realizan muebles ensamblando 3 partes, existen N1 carpinteros para realizar la parte1, N2 para realizar la parte 2, N3 para la parte 3, más N carpinteros que se encargan de ensamblar las 3 partes del mueble.

Los encargados de ensamblar deben tomar una pieza de cada clase, juntar las partes y una vez ensambladas los carpinteros que le dieron la pieza pueden seguir trabajando -no antes-.

En la carpintería sólo se armaran 30 muebles y no se podrán producir piezas de más.
El armado debe realizarse en forma concurrente, sólo pueden esperarse entre los carpinteros de un mismo tipo cuando terminan una pieza hasta que los tome un ensamblador (es decir puede pensar que los ensambladores toman las piezas de a uno).

Todos los procesos deben terminar correctamente, no pueden quedar procesos colgados. Modelice con semáforos.

```
int P1=0, P2=0, P3=0, E=0;         # contadores de producción
sem m1=1, m2=1, m3=1, mE=1;        # mutexes para acceder a contadores
sem e1=0, e2=0, e3=0,              # para que esperen los ensambladores
    carp1[1:N1] = ([N1] 0),        # para que esperen carpinteros tipo 1
    carp2[1:N2] = ([N2] 0),        # 2
    carp3[1:N3] = ([N3] 0);        # y tipo 3 también
int c1, c2, c3;                    # para avisar el ID del carpintero que termino
sem aviso1=1, aviso2=1, aviso3=1;  # mutex avisar que terminaron

process carpinteros1[i = 1 to N1] {
  P(m1);
  while(P1<30) {
    P1++;            # cuenta la unidad que va a producir
    V(m1);
    ProducirPieza1();
    P(aviso1);       # espera a poder avisar
    c1=i;            # pasa su ID por vble compartida
    V(e1);           # avisa que se produjo una unidad
    P(carp1[i]);     # espera que lo despierten al terminarse el ensamblado
```

```
    P(m1);
  }
  V(m1);
}

process carpinteros2[i = 1 to N2] {
  P(m2);
  while(P2<30) {
    P2++;
    V(m2);
    ProducirPieza2();
    P(aviso2);
    c2=i;
    V(e2);
    P(carp2[i]);
    P(m2);
  }
  V(m2);
}

process carpinteros3[i = 1 to N3] {
  P(m3);
  while(P3<30) {
    P3++;
    V(m3);
    ProducirPieza3();
    P(aviso3);
    c3=i;
    V(e3);
    P(carp3[i]);
    P(m3);
  }
  V(m3);
}

process ensamblador[i = 1 to N] {
int idC1,idC2,idC3;       # para poder despertarlos cuando termine
  P(mE);
  while(E<30) {
    E++;
    V(mE);
    # espera cada unidad y guarda los ID's de los carpinteros
    P(e1); idC1 = c1; V(aviso1);
    # liberando los mutex para que otros puedan avisar
    P(e2); idC2 = c2; V(aviso2);
    # podria pasar por otra variable un objeto tParte{1,2,3} también....
    P(e3); idC3 = c3; V(aviso3);
    EnsamblarUnidad();
    V(carp1[idC1]);       # despierta a los carpinteros para que sigan
    V(carp2[idC2]);
    V(carp3[idC3]);
    P(mE);
  }
```

```
  V(mE);
}
```

En un tenedor libre hay una mesa de N platos de comidas diferentes donde cada una de las P personas buscan repetidamente comida. Existe un empleado que repone los diferentes platos de comida.
Cada persona va 5 veces a la mesa de comidas buscando (en cada ida) comida de 4 platos diferentes. No puede haber dos personas al mismo tiempo sirviéndose del mismo tipo de comida. Si la persona se sirve la última porción del plato, le avisa al empleado para que él lo recargue.

Cuando un cliente le avisa al empleado que se vació un plato de comida, este lo vuelve a completar con 10 porciones (inicialmente cada plato tiene 10 porciones). Cuando todos los clientes se han ido el empleado también se va.

Parametrizar: N y P.

```
sem mutexPlato[1:N] = ([N] 1), aviso = 0, mutexAviso = 1;
int porciones[1:N] = ([N] 10), platoVacio;
int cantPersonas = P;

process Personas[p = 1 to P] {
int plato;
  for[v = 1 to 5] {
    for[p = 1 to 4] {
      plato = random(N) + 1;      # no controlo que no se repita el plato...
      P(mutexPlato[plato]);
      porciones[plato]--;
      if(porciones[plato] == 0) {
        P(mutexAviso);
        platoVacio = plato;
        V(aviso);
      } else V(mutexPlato[plato]);
    }
    # come
  }
  P(mutexAviso);
  platoVacio = 0;                 # plato 0 indica que la persona se retira
  V(aviso);
}

process Empleado {
  while(cantPersonas<>0) {
    P(aviso);
    if(platoVacio==0) {
      cantPersonas--;
    } else {
      porciones[platoVacio] = 10;
      V(mutexPlato[platoVacio]);
    }
    V(mutexAviso);
  }
```

```
}
```

Una profesora de matemáticas quiere calcular I integrales de las cuales conoce los límites en las que quiere evaluar y así obtener un resultado para cada una. Además, existen A alumnos que serán encargados de obtener el resultado de las I integrales.

Para trabajar la profesora espera que lleguen todos los alumnos a la clase. Una vez todos han llegado, los organiza en grupos de a 4 alumnos. Cuando todos encontraron a sus compañeros de grupo, resuelven la integral de la siguiente manera: la integral es dividida en 10 partes iguales y el resultado que se debe enviar a la profesora es la suma de todas las partes. Cada alumno del grupo calcula una parte de la integral que aún no se ha calculado. Cuando se han calculado las I integrales los alumnos se pueden ir y la maestra cierra la puerta del aula.

NOTA: Suponga que los alumnos pueden invocar una función que dada la integral y los límites de la parte que le toca calcular, devuelve el resultado. No se puede suponer nada sobre el tiempo que le requiere a cada uno calcular una parte de la integral. Un grupo de alumnos puede calcular 0, 1 o más integrales. Maximice la concurrencia. Puede realizar este ejercicio con monitores o semáforos.

Como la profesora lo único que hace es coordinar a los alumnos, la represento con un monitor en vez de como un proceso activo:

```
process Alumno[i = 1..A] {
int g;
double desde, hasta;
tIntegral integ = Null;

  Profesora.llegue(g);
  Grupo[g].esperarCompañeros(integ);
  while(integ) {
    Grupo[g].proxBloque(desde, hasta);
    while(desde!=hasta) {
      Grupo[g].resultado( Calcular(integ, desde, hasta) );
      Grupo[g].proxBloque(desde, hasta);
    }
    # pedimos la próxima integral
    Grupo[g].esperarCompañeros(integ);
  }
  Profesora.adios();
}


monitor Grupo[i = 1..A/4] {
int cant=0, bloques = 0, resueltos = 0;  #bloques es el bag of tasks para cada integral
double desde, hasta, ancho, total = 0;
cond companieros;
tIntegral integral;

  procedure esperarCompañeros(tIntegral &integ) {
    cant++;
    if(cant<4) wait(companieros)
```

```
    else {
      Profesora.requestIntegral(integral, desde, hasta);
      ancho = (hasta-desde)/10;
      cant = 0;
      signal_all(companieros);
    }
    integ = integral;
  }

  procedure proxBloque(double &b_desde, double &b_hasta) {
    if(bloques<10) {
      b_desde = desde + ancho * bloques;
      b_hasta = b_desde + ancho;
      bloques++;
    } else {
      b_desde = 0;
      b_hasta = 0;
    }
  }

  procedure resultado(double &subtotal) {
    total = total + subtotal;
    resueltos++;
    if(resueltos==10) {
      Profesora.resultado(integral, desde, hasta, total);
      bloques=0;
      resueltos=0;
      total=0;
    }
  }
}


monitor Profesora {
int cantAlumnos=0, resueltas=0, ult_int=0;
cond todos;

  procedure llegue(int &g) {
    cantAlumnos++;
    if(cantAlumnos<A) wait(todos)
      else signal_all(todos);      # cuando llega el último siguen todos
    cantAlumnos--;
    g = cantAlumnos\4 +1;          # toman su nro de grupo entre 1..A/4
  }

  procedure adios() {
    cantAlumnos++;
    if(cantAlumnos==A) cerrarPuerta();
  }

  procedure requestIntegral(tIntegral &integral, double &desde, double &hasta) {
    if(ult_int<I) {
      ObtenerIntegral(integral, desde, hasta);     # Simbólico
```

```
      ult_int++;
    } else {
      # Como se van cuando se calculan todas las integrales, espera a los otros grupos
      if(resueltas!=I) wait(todos);
      integral = Null;
    }
  }

  procedure resultado(tIntegral integral, double desde, double hasta, double total) {
    resueltas++;
    if(resueltas==I) signal_all(todos);
  }
}
```

P4E4(2012). El procesamiento de una imagen se realiza de la siguiente manera: primero actúan los P procesos receptores de imágenes. Cada receptor trabaja de manera continua recibiendo imágenes; a cada una le aplica un proceso de segmentación y a partir de ello determina el color del fondo. Si el color del fondo es rojo, la imagen es enviada a un proceso reconocedorRojo; si la imagen tiene fondo azul se envía a un proceso reconocedorAzul y por último si la imagen tiene fondo verde se envía al proceso reconocedorVerde. Una vez que alguno de los reconocedores reciben la imagen debe determinar e imprimir la cantidad de círculos, cuadrados y triángulos de la misma. Para realizar esto existen tres procesos círculo, cuadrado y rectángulo, que cuentan la cantidad de círculos, cuadrados y rectángulos de la imagen.

Nota: Maximizar la concurrencia. Evitar procesos ociosos. Implementar con pasaje de mensaje sincrónico.

```
process receptor[i = 1..P] {
tImagen imagen;
  do true →
    progPrincipal?imagen;             # simbólico, alguien les pasa las imágenes
    switch(detectarColor(imagen)) {   # segmenta la imagen y detecta el color
      1: BufferRojo!imagen; break;
      2: BufferVerde!imagen; break;
      3: BufferAzul!imagen; break;
    }
  od;
}

process BufferRojo {                  # verde y azul son análogos...
tImagen imagen;
queue cola of tImagen;
  do receptor[*]?imagen →
       push(cola, imagen);
     !empty(cola); reconocedorRojo?request →
       pop(cola, imagen);
       reconocedorRojo!procesar(imagen)
  od;
}

process reconocedorRojo {             # reconocedorVerde y reconocedorAzul son análogos...
tImagen imagen;
boolean pidioCirc, pidioCuad, pidioTrian;
```

```
int     qCirc, qCuad, qTrian;
  do BufferRojo!request →
       BufferRojo?procesar(imagen);
       QuitarFondo(imagen);     #simbólico, algún procesamiento que haga el reconocedor
       pidioCirc = false; pidioCuad = false; pidioTrian = false;
       qCirc = -1; qCuad = -1; qTrian = -1;
       do
         (!pidioCirc; Circulo!imagen)    →   pidioCirc = true;
         (qCirc<0; Circulo?qCirc)        →   skip;
         (!pidioCuad; Cuadrado!imagen)   →   pidioCuad = true;
         (qCuad<0; Cuadrado?qCuad)       →   skip;
         (!pidioTrian; Triangulo!imagen) →   pidioTrian = true;
         (qTrian<0; Triangulo?qTrian)    →   skip;
       od;
       println("%s tiene %d circulos, %d cuadrados y %d triángulos sobre fondo rojo",
               imagen.nombre, qCirc, qCuad, qTrian);
  od;
}

process Circulo {                    # Cuadrado y Triángulo son análogos...
tImagen imagen;
int q;
  do reconocedorRojo?imagen  →
       q = ContarCirculos(imagen);
       reconocedorRojo!q;
     reconocedorVerde?imagen  →
       q = ContarCirculos(imagen);
       reconocedorVerde!q;
     reconocedorAzul?imagen  →
       q = ContarCirculos(imagen);
       reconocedorAzul!q;
  od;
}
```

Dada la siguiente solución con monitores al problema de alocación de un recurso con múltiples unidades, transforme la misma en una solución utilizando PMA.

```
monitor alocador_recurso
int disponible = MAXUNIDADES;
set unidades = valores_iniciales;       # ID's de recursos libres
cond libre;

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

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

Solución con PMA:

```
type op = enum(ADQUIRIR,LIBERAR);
chan request(int idCli, op oper, int idUnidad);
chan respuesta[1..N](int idUnidad);     # N Clientes

process alocador{
int disponible = MAXUNIDADES;
set unidades = valores_iniciales;       # ID's de recursos libres
queue pendientes of int;                # demora respuestas (simula vble.cond.)

  while(true){
    receive request(idCli, oper, idUnidad);
    if(oper == ADQUIRIR){
      if(disponible>0){
       disponible--;
       remove(unidades,idUnidad);
       send respuesta[idCli](idUnidad);
      } else insert(pendientes,idCli);
    } else { # LIBERAR
      if(empty(pendientes)){
        disponible++;
        insert(unidades,idUnidad);
      } else {
        remove(pendientes,idCli);
        send respuesta[idCli](idUnidad);
      }
    }
  }
}
```

Implemente una solución al problema de EM distribuida entre N procesos utilizando un algoritmo de tipo Token Passing con PMA.

Los procesos necesitan realizar su ejecución normal, y a la vez estar pendientes por si reciben un token para pasarlo al siguiente proceso en caso que no lo precise. Se utiliza un proceso *helper* que lo libere de esta tarea:

```
chan token[n]();                          # para envio de tokens
chan enter[n](), go[n](), exit[n]();      # para comunicación proceso-helper

process helper[i = 1..N] {
  while(true){
    receive token[i]();        # recibe el token
    if(!(empty(enter[i]))){      # si su proceso quiere usar la SC
      receive enter[i]();
      send go[i]();              # le da permiso y lo espera a que termine
      receive exit[i]();
    }
    send token[i MOD N +1]();  # y lo envía al siguiente cíclicamente
  }
}

process user[i = 1..N] {
```

```
  while(true){
    send enter[i]();
    receive go[i]();
    ... sección crítica ...
    send exit[i]();
    ... sección no crítica ...
  }
}
```

En una biblioteca existen los siguientes personajes: A alumnos que van a devolver un libro, un encargado de tomar los libros devueltos y un ordenador de libros. El alumno siempre está ansioso y por lo tanto quiere devolver el libro y no esperar de más, y para no ser menos el encargado también es ansioso y no quiere perder tiempo. No es necesario que exista ninguna interacción entre los alumnos y el encargado. Resolver con pasaje de mensajes sincrónico. Maximice la concurrencia.

```
chan devolverLibro[1..A](tLibro), pedidoLibroEncargado, darLibroEncargado(tLibro);
chan dejarLibroEncargado(tLibro), pedidoLibroOrdenador, darLibroOrdenador;

process Alumnos[i = 1..A] {
libro: tLibro;
  devolverLibro[i]!(libro);
}

process BufferEncargado {
queue cola of tLibro;
tLibro libro;
  do [i = 1..A] devolverLibro[i]?(libro) →
       push(cola, libro);
  [] !empty(cola); pedidoLibroEncargado? →
       pop(cola, libro);
       darLibroEncargado!(libro);
  od;
}

process Encargado {
tLibro libro;
int i;
  for (i=0; i<A; i++) {
    pedidoLibroEncargado!();
    darLibroEncargado?(libro);
    RegistrarEntrega(libro);        # Simbólico
    dejarLibroEncargado!(libro);
  }
}

process BufferOrdenador {
queue cola of tLibro;
tLibro libro;
  do dejarLibroEncargado?(libro) →
       push(cola, libro);
  [] !empty(cola); pedidoLibroOrdenador? →
       pop(cola, libro);
       darLibroOrdenador!(libro);
```

```
  od;
}

process Ordenador {
tLibro libro;
int i;
  for(i=0; i<A; i++) {
    pedidoLibroOrdenador!;
    darLibroOrdenador?(libro);
    Ordenar(libro);                # Simbólico
  }
}
```

Se tienen dos tipos de alumnos: los que aprobaron el curso de ingreso y los que no. Además existe un profesor que debe responder las preguntas de los alumnos de la siguiente manera: si un alumno de los que NO aprobó le hace una pregunta y él está libre le responde; si un alumno de los que aprobó le hace una pregunta, sólo le responde si no hay ningún alumno de los que NO aprobó queriendo consultar. Resuelva con pasaje de mensajes sincrónico.

```
chan solicitud[1..A](boolean), hacerConsulta[1..A];
chan consulta[1..A](string), respuesta[1..A](string);
chan siguiente, alumnoParaAtender(int);

process Alumno[i = 1..A] {
boolean aprobado;
string preg, resp;
  aprobado = AproboExamen(i);          # Simbólico, cada alumno sabe si aprobó o no
  while(TengoDudas()) {
    solicitud[i]!(aprobado);
    hacerConsulta[id]?;
    preg = GenerarPregunta();          # Simbólico
    consulta[i]!(preg);
    respuesta[i]?(resp);
  }
}

process Administrador {
queue colaAprobados of int, colaDesaprobados of int;
boolean aprobado;
int idAlumno;
  do [i = 1..A] solicitud[i]?(aprobado) →
       if(aprobado) push(colaAprobados, i)
         else push(colaDesaprobados, i);
  [] !empty(colaDesaprobados); siguiente? →
       pop(colaDesaprobados, idAlumno);
       alumnoParaAtender!(idAlumno);
  [] empty(colaDesaprobados) and !empty(colaAprobados); siguiente? →
       pop(colaAprobados, idAlumno);
       alumnoParaAtender!(idAlumno);
  od;
}

process Profesor {
int idAlumno;
```

```
string preg, resp;
  while(true) {
    siguiente!;
    alumnoParaAtender?(idAlumno);
    hacerConsulta[idAlumno]!;
    consulta[idAlumno]?(preg);
    resp = GenerarRespuesta(preg);     # Simbólico
    respuesta[idAlumno]!(resp);
  }
}
```