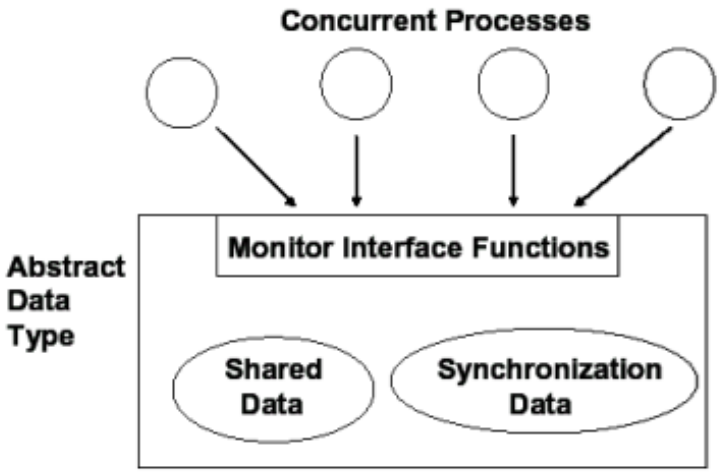
Resumen Monitores - Concurrente

Monitores

- Módulos de programa con más estructura, y que pueden ser implementados tan eficientemente como los semáforos.
- Es un **Mecanismo de abstracción** de datos que:
  - Encapsula las representaciones de los recursos.
  - Brinda un conjunto de operaciones que son los únicos medios para manipular esos recursos.
- Contiene variables que almacenan el estado del recurso y procedimientos que implementan las operaciones sobre él.
- Actúan de forma **pasiva**, es decir, estos únicamente se ejecutan cuando un **proceso activo** de un **programa concurrente** invoca un procedure de un **monitor**.



Exclusión Mutua con Monitores

- Se resuelve de forma **implícita** ya que se asegura que los procedures que estén dentro de un mismo monitor no se ejecutan concurrentemente.

Sincronización por Condición con Monitores

- Se debe implementar de forma **explícita** con variables condición.

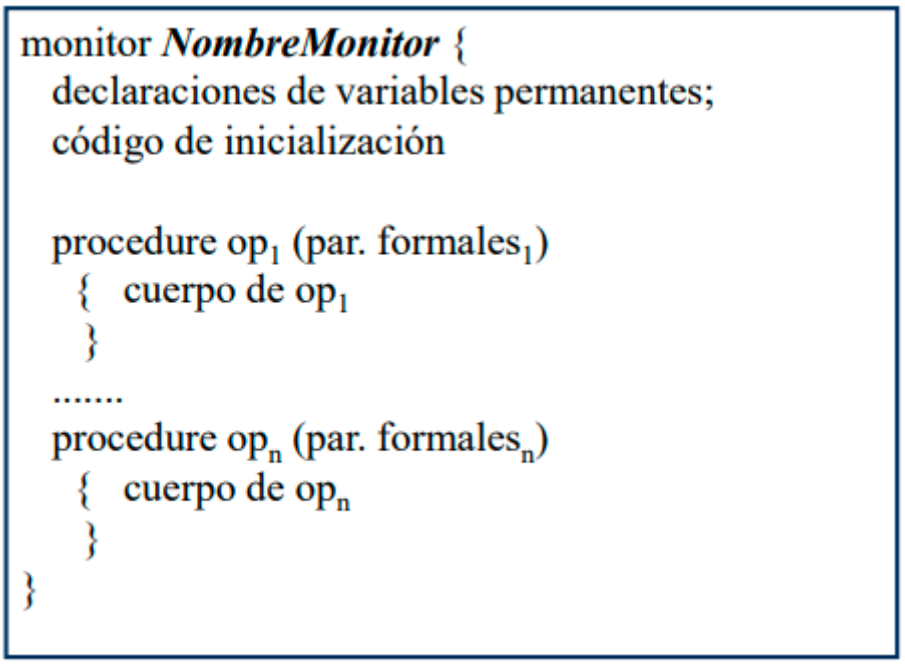
Ventajas del uso de Monitores

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

Notación de los Monitores

- Los monitores se distinguen de un tipo de dato abstracto ya que los monitores son compartidos por procesos que se ejecutan concurrentemente. Tienen **interfaz** y **cuerpo**:
    - **Interfaz** → Especifica las operaciones (procedures) que brinda el recurso. Deja que solo los nombres de los procedures sean visibles desde afuera.
    - **Cuerpo** → Conjunto de variables que van a representar a ese recurso, se denominan **variables permanentes** (conservan su valor a través de las distintas invocaciones que se hacen a los procedimientos del monitor) e implementación de los procedures indicados en la interfaz del monitor.
        - Dentro del **Cuerpo**, los **procedures** sólo pueden acceder a **variables permanentes**, sus **variables locales** y **parámetros** que le sean pasados en la invocación.
- Los llamados al monitor tienen la forma:
    - **NombreMonitor.operacion$_i$(argumentos).**

- El programador de un monitor no puede conocer a priori el orden de llamado de los procedures del monitor.

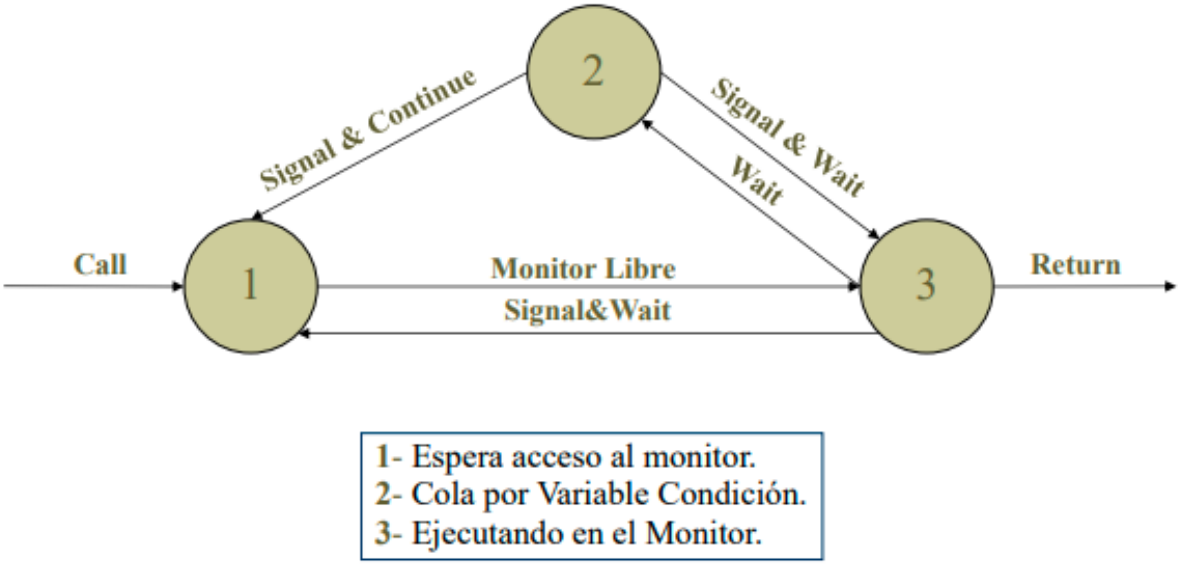


Implementación de la Sincronización por Condición

- Es programada explícitamente con **variables condición** → cond cv;
    - Son variables permanentes del monitor que solo se pueden usar dentro del mismo.
- Esta variable condición “cv” tiene asociado internamente una **cola de procesos dormidos/demorados** que **no es visible directamente** para el programador.
- **Operaciones permitidas** sobre las variables condición:
    - **wait(cv)** → El proceso que está ejecutando se demora al final de la cola de “cv” y deja el acceso exclusivo al monitor, es decir, lo libera para que otro proceso lo pueda usar.
    - **signal(cv)** → Despierta al proceso que está primero en la cola de procesos dormidos/demorados (si hay alguno) y lo saca de la cola. **El proceso despertado recién podrá ejecutar desde el punto en el que se durmió cuando readquiera el acceso exclusivo al monitor.**
    - **signal_all(cv)** → Despierta a todos los procesos dormidos/demorados en “cv”, dejando vacía la cola asociada a esa variable.
- **Disciplinas de Señalización** → Indican cuándo el proceso que es despertado va a poder volver a acceder al monitor de forma exclusiva para continuar su ejecución.
    - **Signal and continue (usada en la materia)** → El proceso que hace el **signal(cv)** despierta al proceso que estaba dormido pero sigue haciendo uso del monitor hasta que termina de hacer uso del mismo o es dormido. **El proceso que fue despertado tendrá que competir para volver a acceder al monitor.**

    - **Signal and wait** → El proceso que hace el **signal(cv)** pasa a ser quien tendrá que competir nuevamente por el acceso exclusivo al monitor ya que **el proceso despertado es quien toma control del mismo.**
- **Operaciones adicionales** sobre las variables condición que solo se usan en la teoría:
    - **empty(cv)** → Retorna **true** si la cola controlada por **cv** está vacía.
    - **wait(cv, rank)** → El proceso se demora en la cola de **cv** en orden ascendente de acuerdo al parámetro **rank** y deja el acceso exclusivo al monitor.
    - **minrank(cv)** → Función que retorna el **mínimo ranking** de demora, es decir, retorna el ranking del primer proceso demorado en la cola.

Signal and continue VS Signal and wait



Diferencias entre las disciplinas de sincronización

- **Signal and Continue**
    - El proceso que hace el signal continua usando el monitor, y el proceso despertado pasa a competir por acceder nuevamente al monitor para continuar con su ejecución (en la instrucción que lógicamente le sigue al wait).
- **Signal and wait**
    - El proceso que hace el signal pasa a competir por acceder nuevamente al monitor, mientras que el proceso despertado pasa a ejecutar dentro del monitor a partir de instrucción que lógicamente le sigue al wait.

Diferencia entre wait/signal con P/V

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

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

Técnicas de Sincronización

Simulación de semáforos - Condición Básica

- Monitor que simula un semáforo inicializado en 1.

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

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

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

- **Diferencia con los semáforos reales:**
    - Esta solución al usar el signal no habilita a todos los procesos que están esperando a competir para usar el semáforo, sólo habilita al primero de la cola de procesos dormidos.

- Para hacer que se comporte de verdad como un semáforo deberíamos de usar en el **procedure V** un **signal_all** en vez de un **signal**.

Simulación de semáforos - Passing the Conditions

- Si se quiere que los procesos pasen el P en el orden en que llegan:
  - **Técnica Passing the Conditions**
    - Yo le aviso al proceso que acabo de despertar que están dadas las condiciones para que él continúe su ejecución pero sin modificar el valor de la condición.

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

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

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

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

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

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

La **Figura de la izquierda** es válida para la **teoría**, la **Figura de la derecha** es válida para la **práctica**

Alocación SJN - Wait con Prioridad (Para la teoría)

- Hace uso de la técnica **Passing the Condition**

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

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

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

Alocación SJN - Variables Condición Privadas (Para la práctica)

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

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

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

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

Buffer Limitado - Sincronización por Condición Básica

```
monitor Buffer_Limitado
{  typeT buf[n];
   int ocupado = 0, libre = 0;  cantidad = 0;
   cond not_lleno, not_vacio;
   procedure depositar(typeT datos)
      { while (cantidad == n) wait (not_lleno);
        buf[libre] = datos;
        libre = (libre+1) mod n;
        cantidad++;
        signal(not_vacio);
      }
   procedure retirar(typeT &resultado)
      { while (cantidad == 0) wait(not_vacio);
        resultado=buf[ocupado];
        ocupado=(ocupado+1) mod n;
        cantidad--;
        signal(not_lleno);
      }
}
```

ocupado representa la primera posición ocupada del buffer
libre representa la primera posición libre del buffer
cantidad representa la cantidad de elementos que hay actualmente en el buffer
las **variables condición** son para controlar que al menos haya un lugar vacío o que al menos haya un lugar ocupado

Lectores y Escritores - Broadcast Signal

- Maximizamos la concurrencia y despertamos a todos los procesos, no a uno en específico.
- Esta solución le da prioridad a los lectores.

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

nr cuenta la cantidad de lectores que actualmente están en la base de datos
nw cuenta la cantidad de escritores que actualmente están en la base de datos
las **variables condición** se usan para dormir a los lectores y escritores

Lectores y Escritores - Passing the Condition

- Para esta solución usamos Passing the Condition.
- Con esta solución le damos prioridad a los escritores cuando la base de datos queda totalmente vacía.

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

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

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

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

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

se suman las variables **dr y dw** que cuentan la cantidad de procesos lectores y escritores que están dormidos

Diseño de un reloj lógico - Covering Conditions

- Ejemplo de **controlador de recurso** (reloj lógico) con dos operaciones:
  - **demorar(intervalo)** → Demora al llamador durante intervalo de ticks de reloj.
  - **tick** → Incrementa el valor del reloj lógico. Es llamada por un proceso que es despertado periódicamente por un timer de hardware y tiene alta prioridad de ejecución.

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

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

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

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

Diseño de un reloj lógico - Wait con prioridad (Para la teoría)

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

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

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

Despierta únicamente cuando es la hora de despertar de un proceso

Diseño de un reloj lógico - Variables Conditions Privadas (Para la práctica)

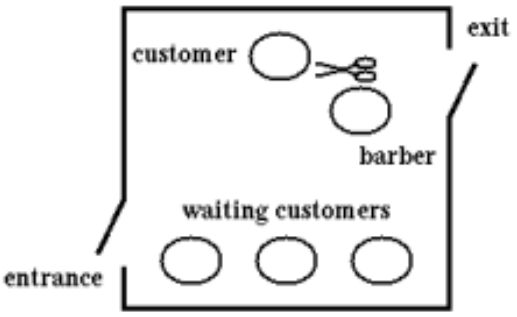
```
monitor Timer
{  int hora_actual = 0;
   cond espera[N];
   colaOrdenada dormidos;

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

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

Problema del Peluquero Dormilón - Rendezvous

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



- **Tipos del proceso del problema** → clientes y peluquero
- **Monitor** → administrador de la peluquería con tres procedures:
    - **corte_de_pelo** → llamado por los clientes, que retornan luego de recibir un corte de pelo.
    - **proximo_cliente** → llamado por el peluquero para esperar que un cliente se siente en su silla, y luego le corta el pelo.
    - **corte_terminado** → llamado por el peluquero para que el cliente deje la peluquería.
- **Etapas de sincronización entre un cliente y el peluquero (rendezvous):**
    1. El peluquero tiene que esperar a que llegue un cliente, y este tiene que esperar que el peluquero esté disponible.
    2. El cliente necesita esperar que el peluquero termine de cortar el pelo para poder irse.
    3. Antes de cerrar la puerta de salida, el peluquero necesita esperar hasta que el cliente se haya ido.

```
monitor Peluqueria {
    int peluquero = 0, silla = 0, abierto = 0;
    cond peluquero_disponible, silla_ocupada, puerta_abierta, salio_cliente;
    procedure corte_de_pelo() {
        while (peluquero == 0)   wait (peluquero_disponible);
        peluquero = peluquero - 1;
        signal (silla_ocupada);
        wait (puerta_abierta);
        signal (salio_cliente);
    }
    procedure proximo_cliente(){
        peluquero = peluquero + 1;
        signal(peluquero_disponible);
        wait(silla_ocupada);
    }
    procedure corte_terminado() {
        signal(puerta_abierta);
        wait(salio_cliente);
    }
}
```

peluquero indica si el peluquero está libre o no
silla indica si la silla está libre o no
abierto indica si la puerta está abierta o cerrada

las **variables condición** son para dormir a los procesos y hacer las sincronizaciones

Problema de Scheduling de Disco

- El disco contiene “platos” conectados a un eje central y que rotan a velocidad constante. Las pistas forman círculos concéntricos → concepto de cilindro de información.
- Los datos se acceden posicionando una cabeza lectora/escritora sobre la pista apropiada, y luego esperando que el plato rote hasta que el dato pase por la cabeza.
    - **dirección física → cilindro, número de pista, y desplazamiento**
- **Para acceder al disco, un programa ejecuta una instrucción de E/S con los parámetros:**
    - Dirección física del disco.
    - Número de bytes a transferir.
    - Tipo de transferencia (Read o Write).
    - Dirección de un buffer.
- **El tiempo de acceso al disco depende de:**
    A. Seek time para mover una cabeza al cilindro apropiado.
    B. Rotational delay.
    C. Transmission time **(depende solo del número de bytes).**
- **A. y B. dependen del estado del disco** (en qué cilindro, en qué sector está y hacia dónde tengo que ir) → buscamos minimizar el **tiempo de seek.**
- **Políticas de Scheduling de un disco:**
    - **Shortest-Seek-Time (SST).**
    - **SCAN, LOOK, o algoritmo del ascensor.**
    - **CSCAN o CLOOK**
        - Se atienden pedidos en una sola dirección. Es fair y reduce la varianza del tiempo de espera.
        - Este es el que nos interesa.

Solución con Monitor Separado

- El **scheduler** es implementado por un monitor para que los datos sean accedidos solo por un proceso usuario a la vez. El monitor provee dos operaciones: **pedir** y **liberar**.
    - **Scheduler_Disco.pedir(cil) - Accede al disco - Scheduler_Disco.liberar( )**
- Suponemos cilindros numerados de **0 a MAXCIL** y **scheduling CSCAN**.
- A lo sumo **un proceso a la vez puede tener permiso para usar el disco**, y los **pedidos pendientes son servidos en orden CSCAN**.
    - Hay que distinguir entre los **pedidos pendientes a ser servidos en el scan corriente** y los que **serán servidos en el próximo scan**.
- **posición** es la variable que indica posición corriente de la cabeza (en qué cilindro está parada la cabeza).
- **Problemas de esta solución:**
    - La presencia del scheduler es visible al proceso que usa el disco. Si se borra el scheduler, los procesos usuario cambian.

- Todos los procesos usuario tienen que seguir el protocolo de acceso, de no ser así el scheduling falla.
- Luego de obtener el acceso, el proceso debe comunicarse con el driver de acceso al disco a través de 2 instancias de buffer limitado.

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

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

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

posición toma valor -1 si la cabeza está libre
v_actual es un puntero que apunta a la cola de pedidos que se atienden en la vuelta actual
v_proxima es un puntero que apunta a la cola de pedidos que se atienden en la vuelta siguiente
scan[2] tiene en una posición la cola de pedidos de la vuelta actual y en la otra posición la cola de pedidos de la vuelta siguiente

Solución con Monitor Intermedio

- Para mejorar la solución anterior usamos un **monitor intermedio** entre los procesos usuario y el disk driver. El monitor envía los pedidos al disk driver en el orden de preferencia deseado.
- **Mejoras:**
  - La interfaz al disco usa un único monitor, y los usuarios hacen un solo llamado al monitor por acceso al disco.
  - La existencia o no de scheduling es transparente.
  - No hay un protocolo multipaso que deba seguir el usuario y en el cual pueda fallar.

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

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

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

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