Concurrencia y Paralelismo

Clase 4

Facultad de Informática

UNLP

Links al archivo con audio

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

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

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

Monitores

Conceptos básicos

Semáforos ⇒

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

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

Mecanismo de abstracción de datos:

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

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

Conceptos básicos

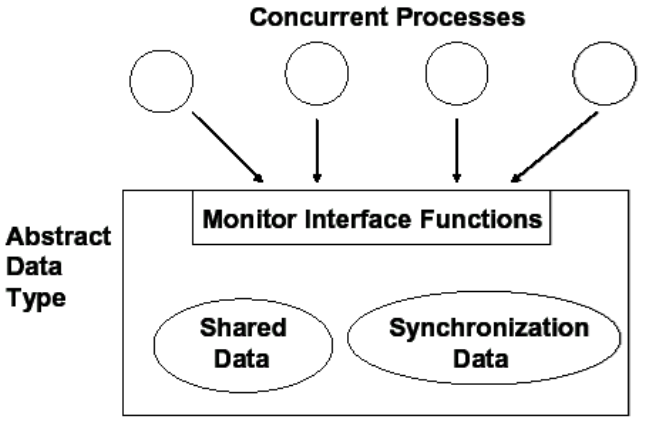
Exclusión Mutua ⇒ implícita asegurando que los ***procedures*** en el mismo monitor no ejecutan concurrentemente.

Sincronización por Condición ⇒ explícita con variables condición.

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

Ventajas:

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



Notación

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

$$\textbf{\textit{NombreMonitor.op}}_{\textbf{\textit{i}}}\textbf{\textit{ (argumentos)}}$$

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

Notación

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

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

Ejemplo de uso de monitores

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

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

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

```
monitor TOTAL {
   int cant = 0;

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

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

Ejemplo de uso de monitores

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

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

BUSY WAITING

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

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

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

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

Sincronización

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

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

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

➢ Disciplinas de señalización:

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

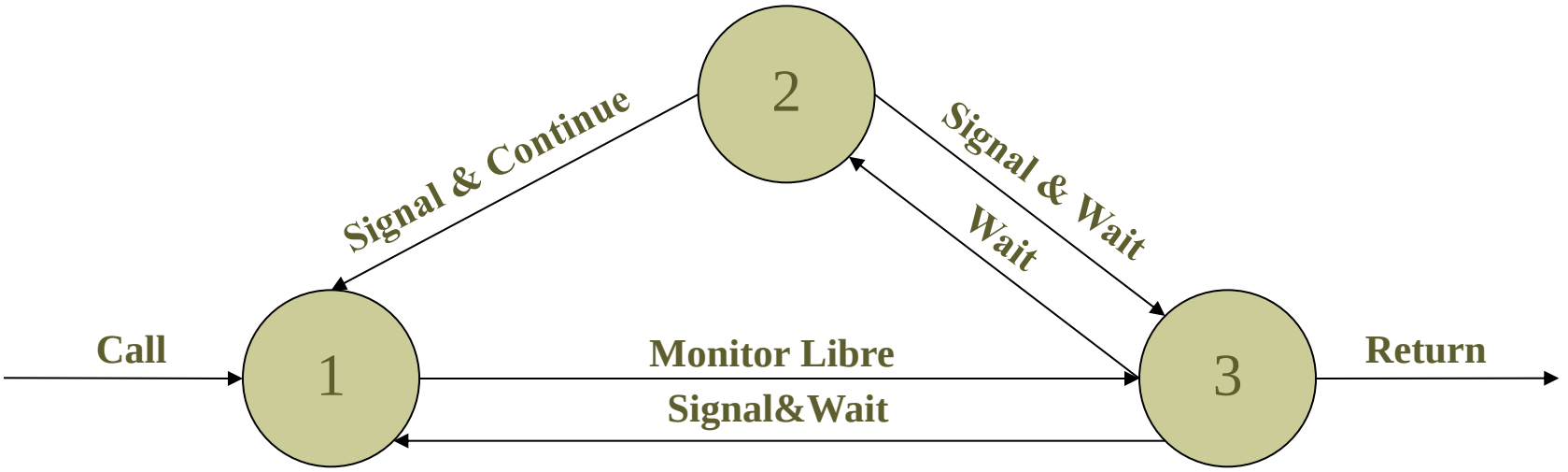
Operaciones adicionales

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

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

Sincronización

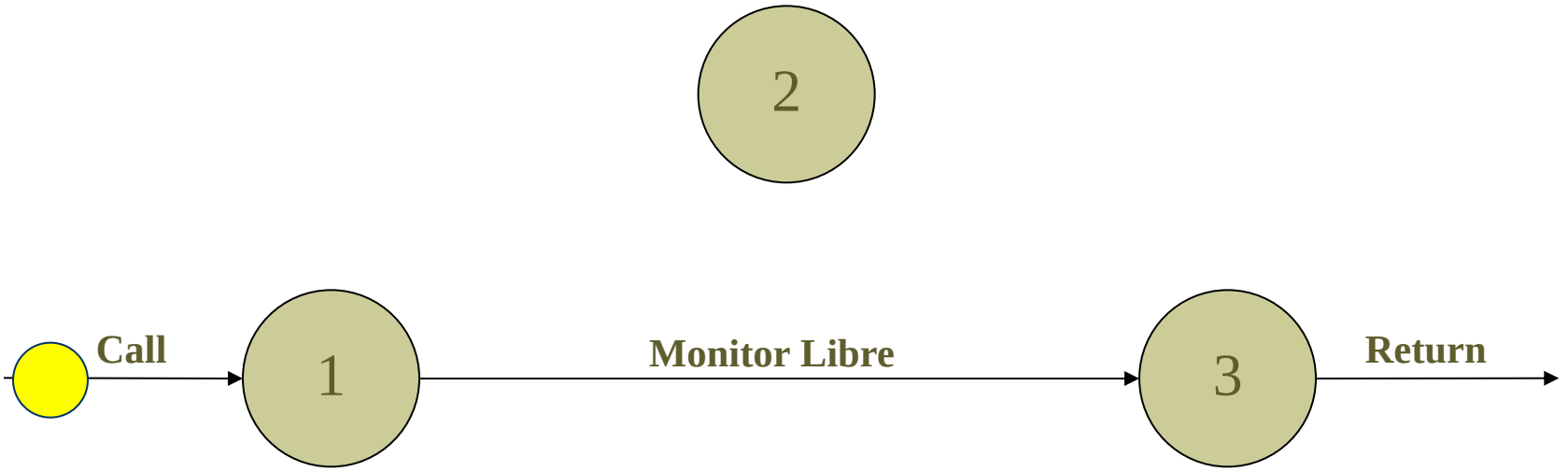
Signal and continue* vs. *Signal and Wait



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

Sincronización

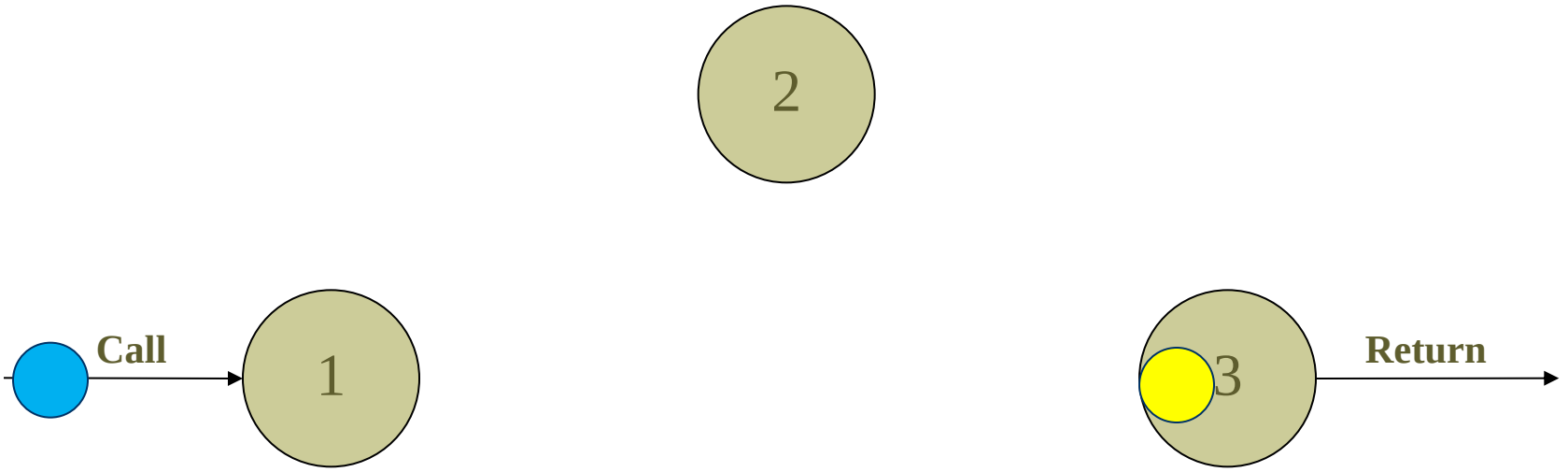
Llamado – Monitor Libre



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

Sincronización

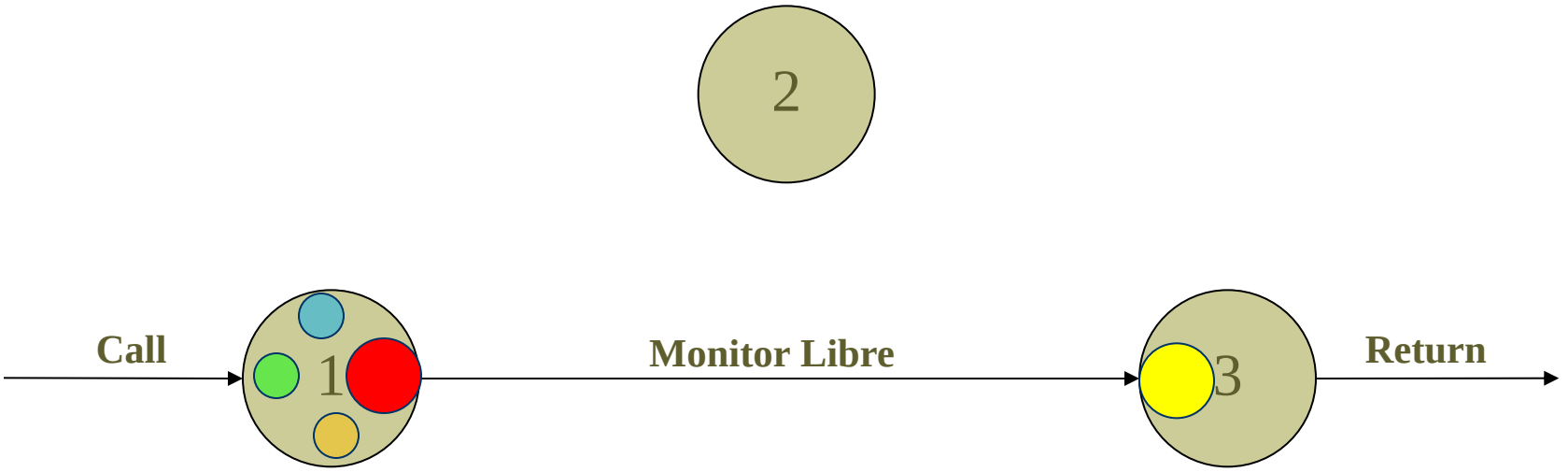
Llamado – Monitor Ocupado



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

Sincronización

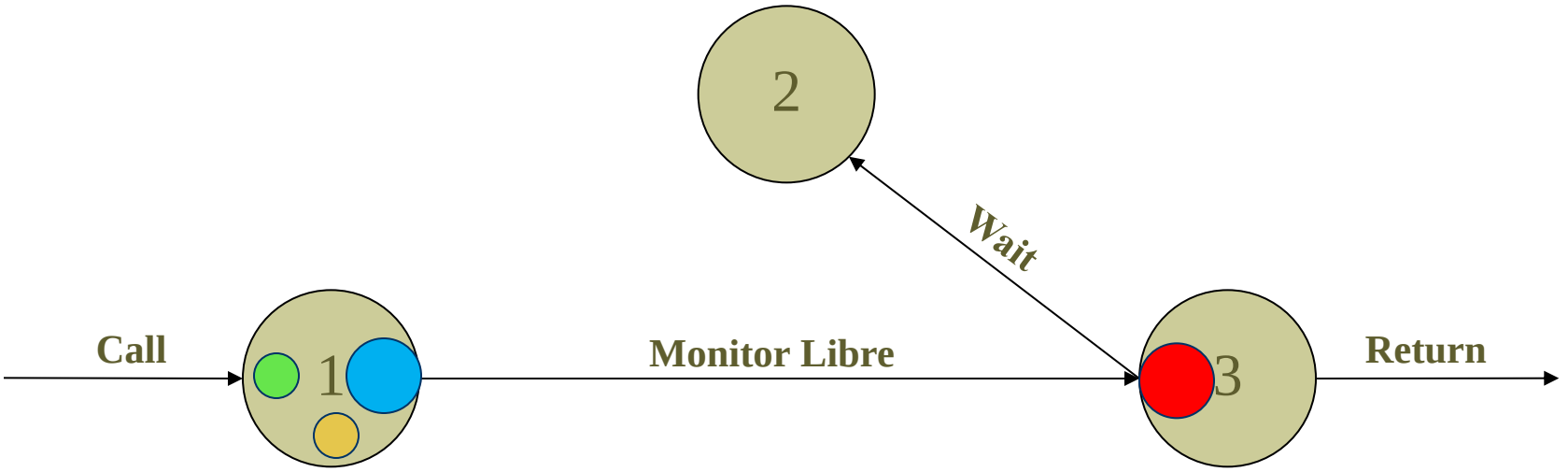
Liberación del Monitor



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

Sincronización

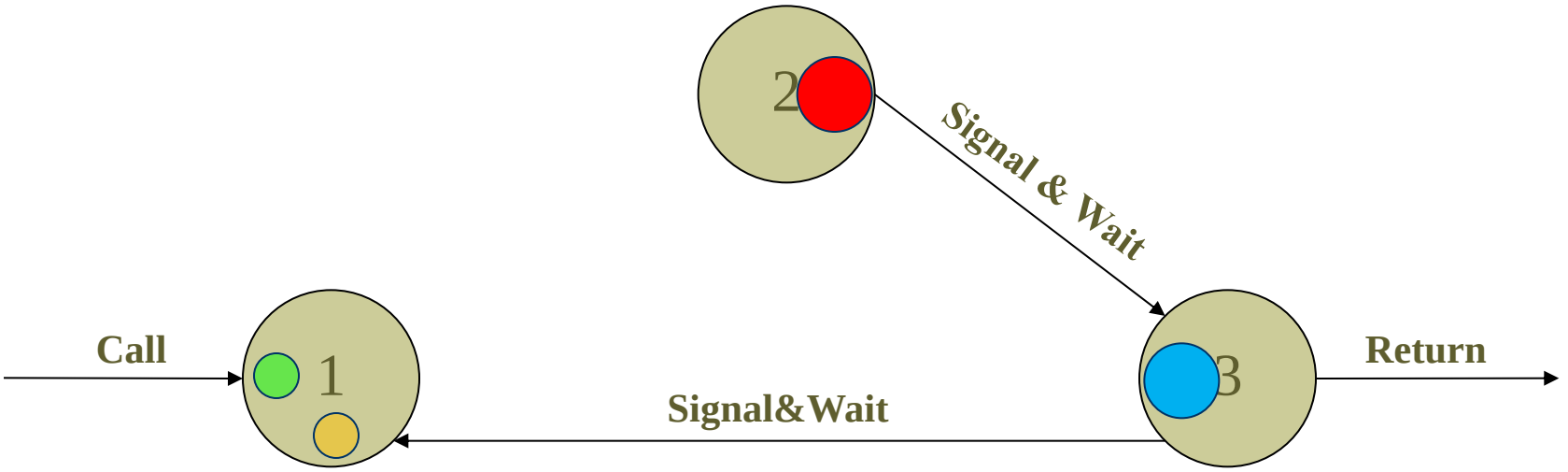
Wait



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

Sincronización

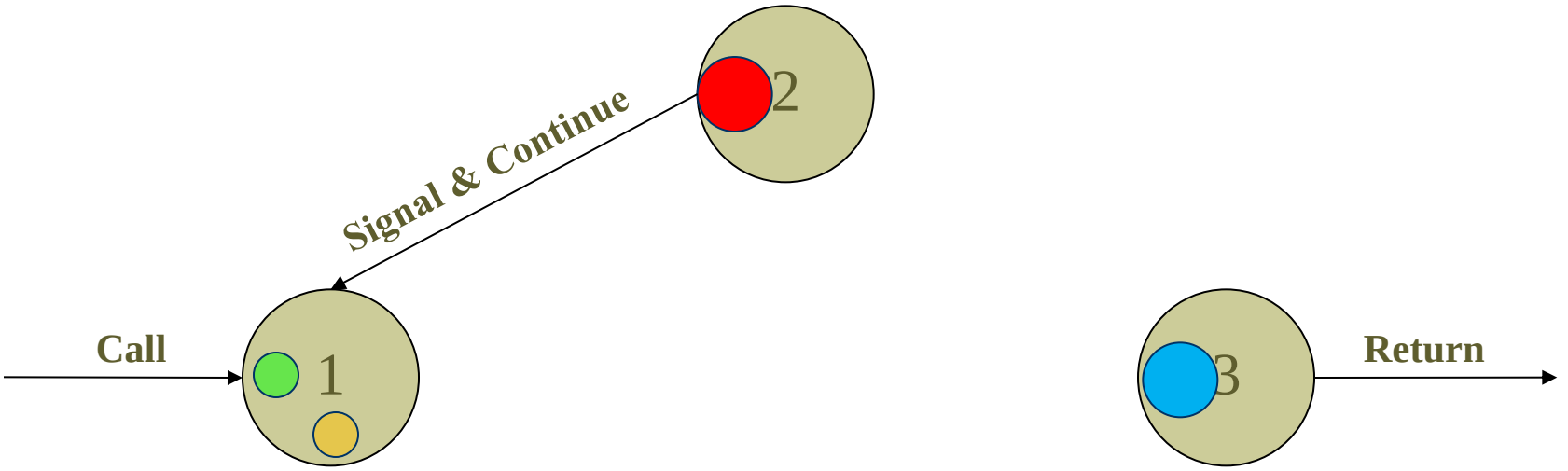
Signal - Disciplina Signal and Wait



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

Sincronización

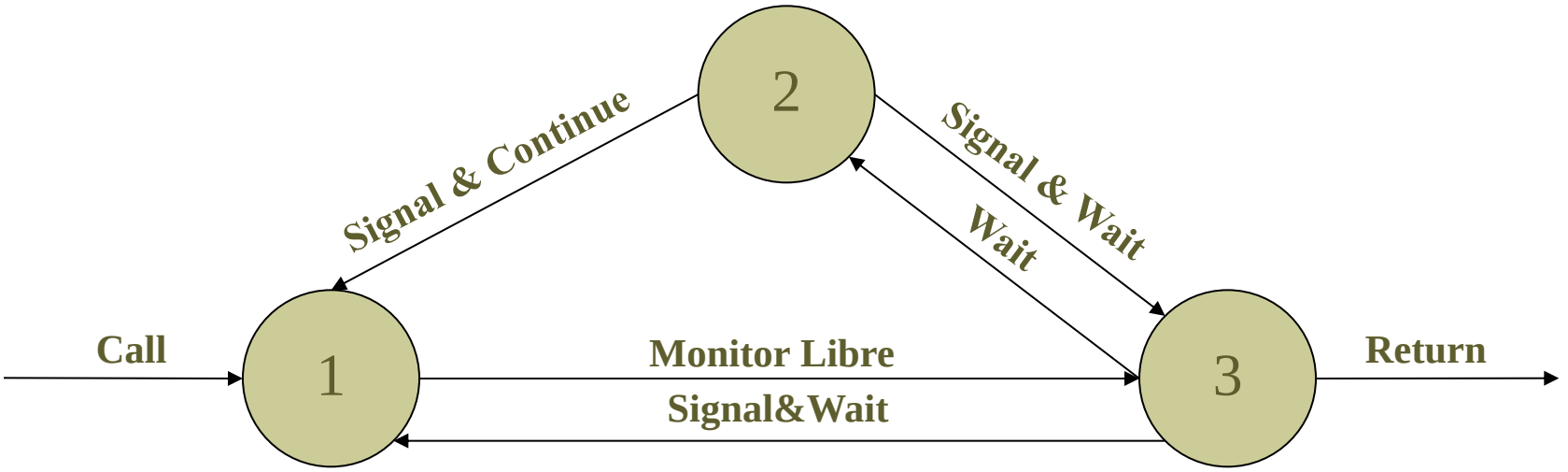
Signal - Disciplina Signal and Continue



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

Sincronización

Signal and continue* vs. *Signal and Wait



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

Sincronización

Resumen: *diferencia entre las disciplinas de señalización*

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

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

Sincronización

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

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

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

Ejemplo de uso de monitores

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

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

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

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

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

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

Ejemplos y técnicas

Ejemplo

Simulación de semáforos: *condición básica*

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

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

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

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

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

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

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

¿Qué diferencia hay con los semáforos?

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

Técnicas de Sincronización

Simulación de semáforos: *Passing the Conditions*

Simulación de Semáforos

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

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

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

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

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

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

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

Técnicas de Sincronización

Alocación SJN: *Wait con Prioridad*

Alocación SJN

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

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

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

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

¿Como resolverlo sin wait con prioridad?

Técnicas de Sincronización

Alocación SJN: *Variables Condición Privadas*

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

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

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

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

Técnicas de Sincronización

Buffer Limitado: *Sincronización por Condición Básica*

Buffer Limitado

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

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

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

Técnicas de Sincronización

Lectores y escritores: *Broadcast Signal*

Lectores y escritores

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

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

Técnicas de Sincronización

Lectores y escritores: *Passing the Condition*

Otra solución al problema de lectores y escritores

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

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

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

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

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

Técnicas de Sincronización

Diseño de un reloj lógico: *Covering conditions*

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

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

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

Diseño de un reloj lógico

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

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

Técnicas de Sincronización

Diseño de un reloj lógico: *Wait con prioridad*

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

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

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

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

Técnicas de Sincronización

Diseño de un reloj lógico: *Variables conditions privadas*

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

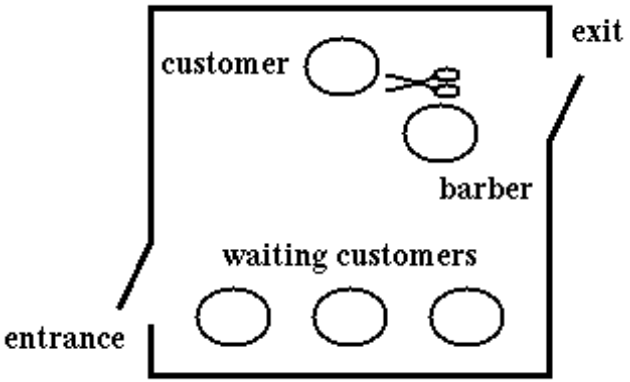
```
monitor Timer
{  int hora_actual = 0;
   cond espera[N];
   colaOrdenada dormidos;

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

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

Técnicas de Sincronización

Peluquero dormilón: *Rendezvous*

Problema del peluquero dormilón (sleeping barber).

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



Técnicas de Sincronización
Peluquero dormilón: *Rendezvous*

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

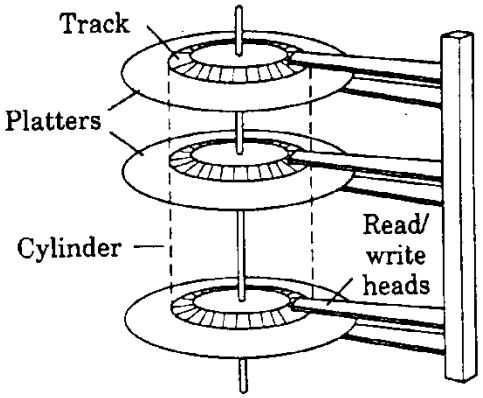
→ el peluquero y el cliente atraviesan una serie de etapas de sincronización, comenzando con un *rendezvous* similar a una barrera entre dos procesos, pues ambas partes deben arribar antes de que cualquiera pueda seguir.

Técnicas de Sincronización

Peluquero dormilón: *Rendezvous*

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

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

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

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

Ejemplo: *Scheduling de disco*



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

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

Ejemplo: *Scheduling de disco*

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

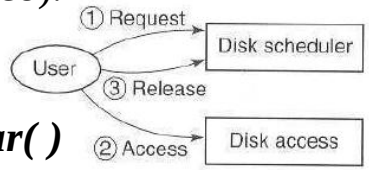
Ejemplo: *Scheduling de disco*
Monitor separado

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

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

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

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



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

Ejemplo: *Scheduling de disco*

Monitor separado

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

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

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

Ejemplo: *Scheduling de disco*
Monitor intermedio

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

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

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

Ejemplo: *Scheduling de disco*

Monitor intermedio

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

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

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

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

Ejemplo: *Scheduling de disco*

Monitor intermedio

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

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

Ejemplo: *Scheduling de disco*

Monitor intermedio

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

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