

Programación Concurrente

Clase 4



Facultad de Informática
UNLP



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 \Rightarrow implícita asegurando que los *procedures* en el mismo monitor no ejecutan concurrentemente.

Sincronización por Condición \Rightarrow explícita con variables condición.

Programa Concurrente \Rightarrow procesos activos y monitores pasivos. Dos procesos interactúan llamando a los *procedures* de un monitor.

- Un monitor agrupa la representación y la implementación de un recurso compartido. Tiene *interfaz* y *cuerpo*:
 - La *interfaz* especifica operaciones que brinda el recurso (nombres de los *procedures*).
 - El *cuerpo* tiene variables que representan el estado del recurso y la implementación de los *procedures* indicados en la *interfaz*.
- Los procesos sólo conocen los nombres de los *procedures* de los monitores. Sintácticamente, los llamados al monitor tienen la forma:

NombreMonitor.procedure (parámetros)

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

Un corralón de materiales posee un servidor que resuelve pedidos de presupuestos de N Personas, el servidor atiende los pedidos de a uno a la vez y sin respetar ningún orden.

```
process Persona[id: 0..N-1]
{ .....
  SERVIDOR.resolver (lista, resultado);
  .....
}
```

```
monitor SERVIDOR
{
  procedure resolver (L: texto; R: out texto)
  {   R = CalcularPresupuesto (L);
  }

}
```

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();  
    .....  
  }  
}
```

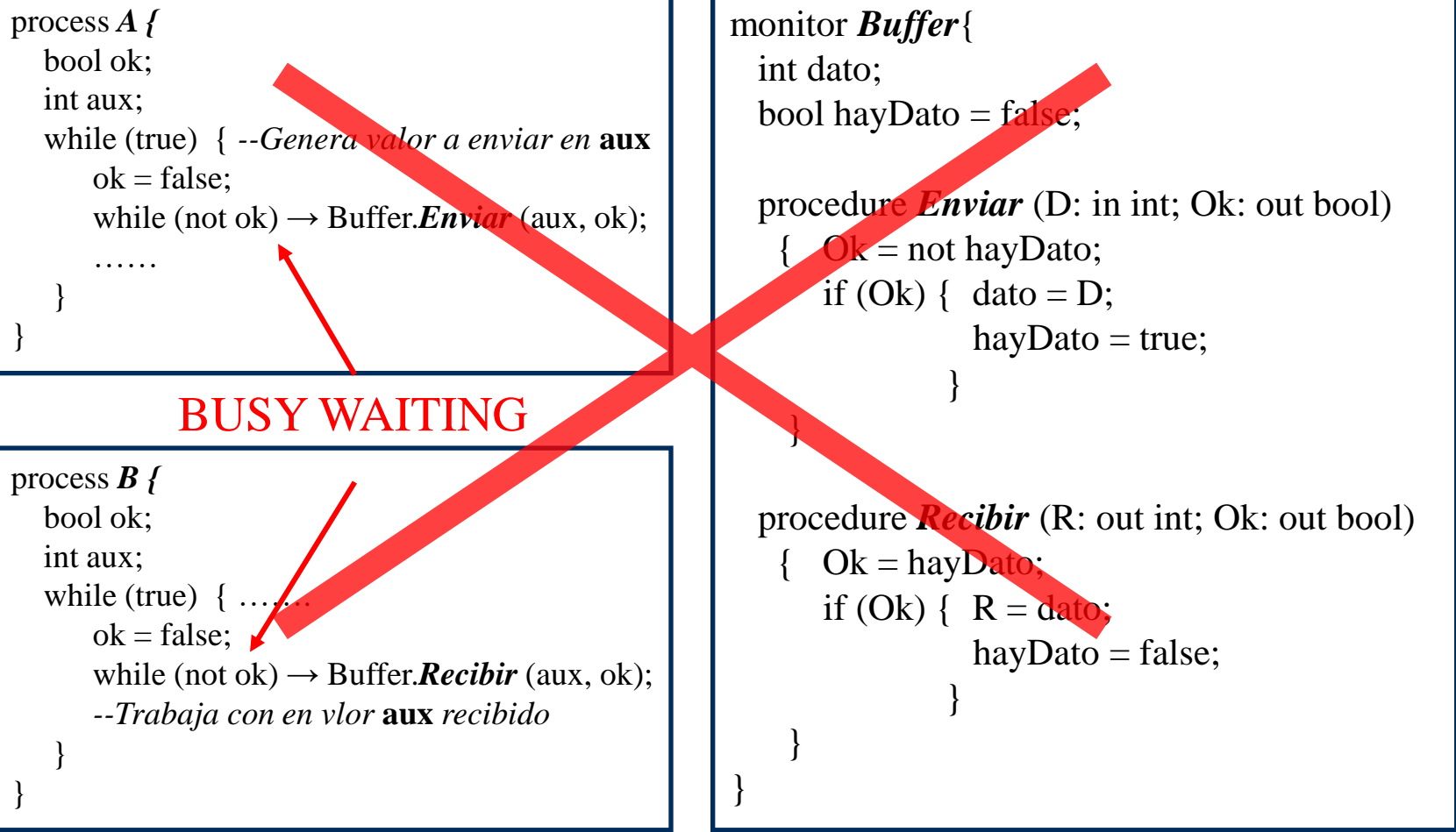
```
process coordinador{  
  int c;  
  while (true)  
  { .....  
    TOTAL.verificar(c);  
    .....  
  }  
}
```

```
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 el valor 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*:

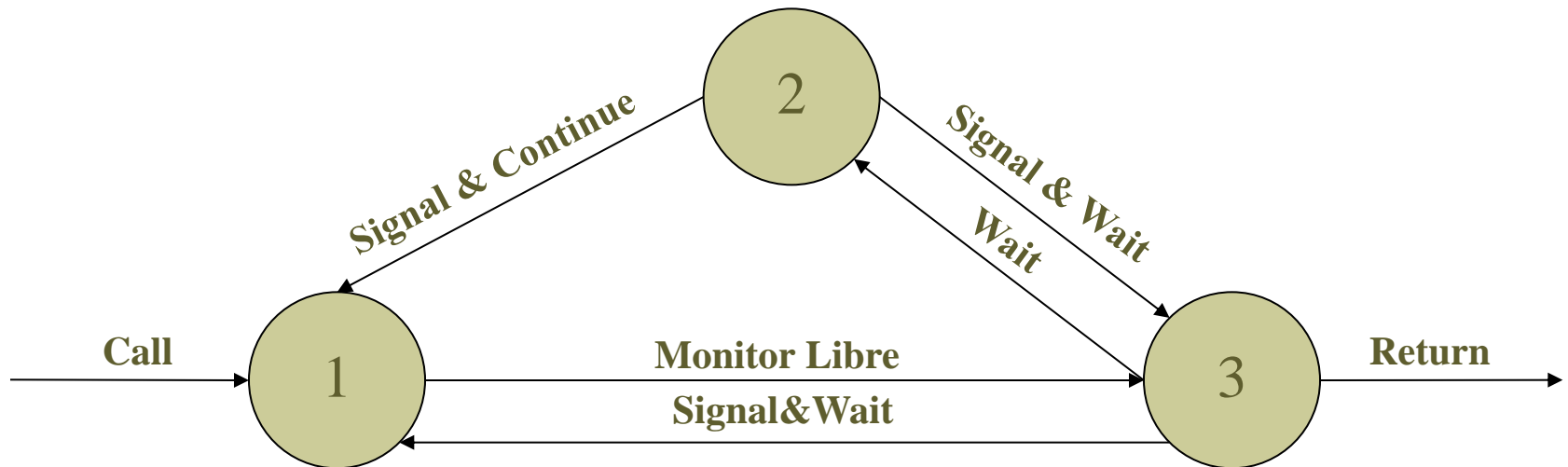
- **wait(cv)** → el proceso se demora al final de la cola de *cv* y deja el acceso exclusivo al monitor.
- **signal(cv)** → despierta al proceso que está al frente de la cola (si hay alguno) y lo saca de ella. *El proceso despertado recién podrá ejecutar cuando readquiera el acceso exclusivo al monitor.*
- **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.

Sincronización

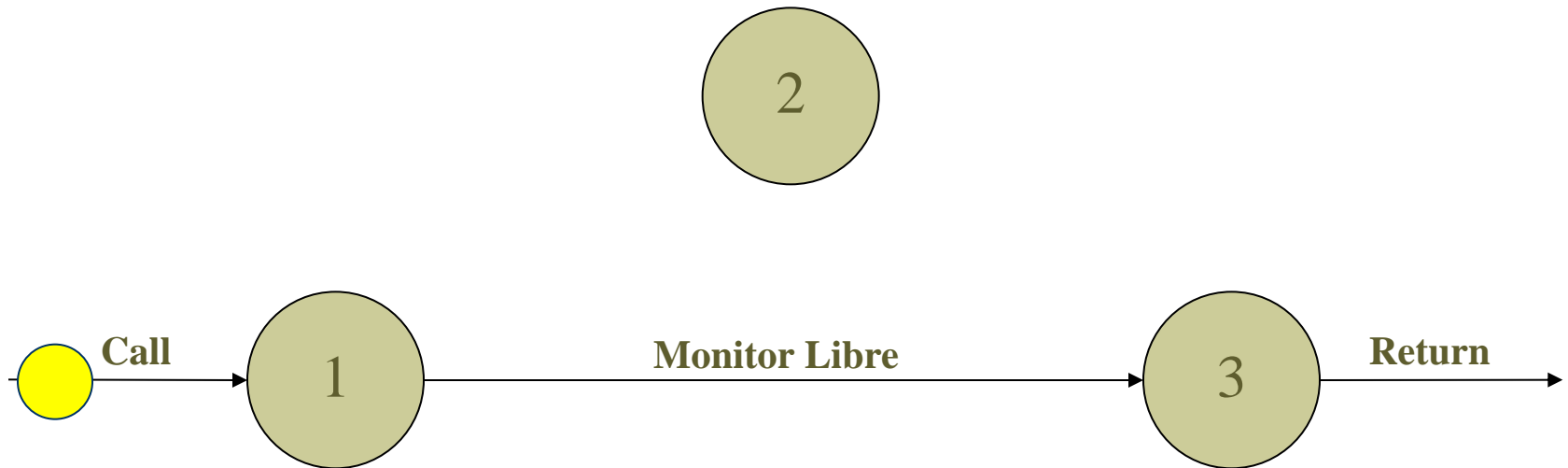
Signal and continue vs. Signal and Wait



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

Sincronización

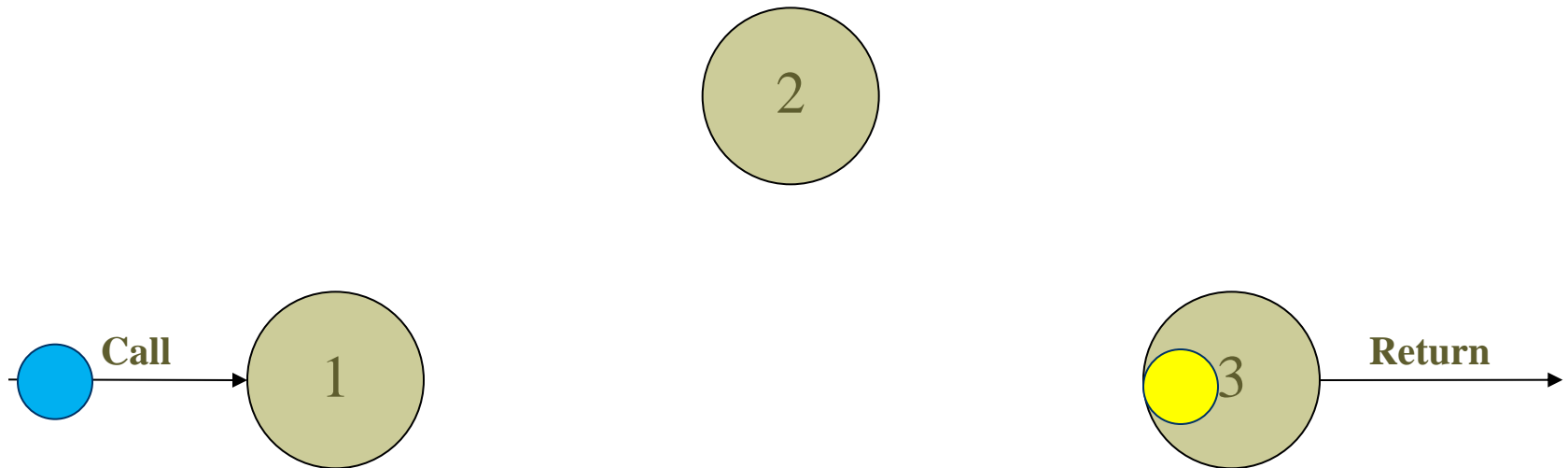
Llamado – Monitor Libre



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

Sincronización

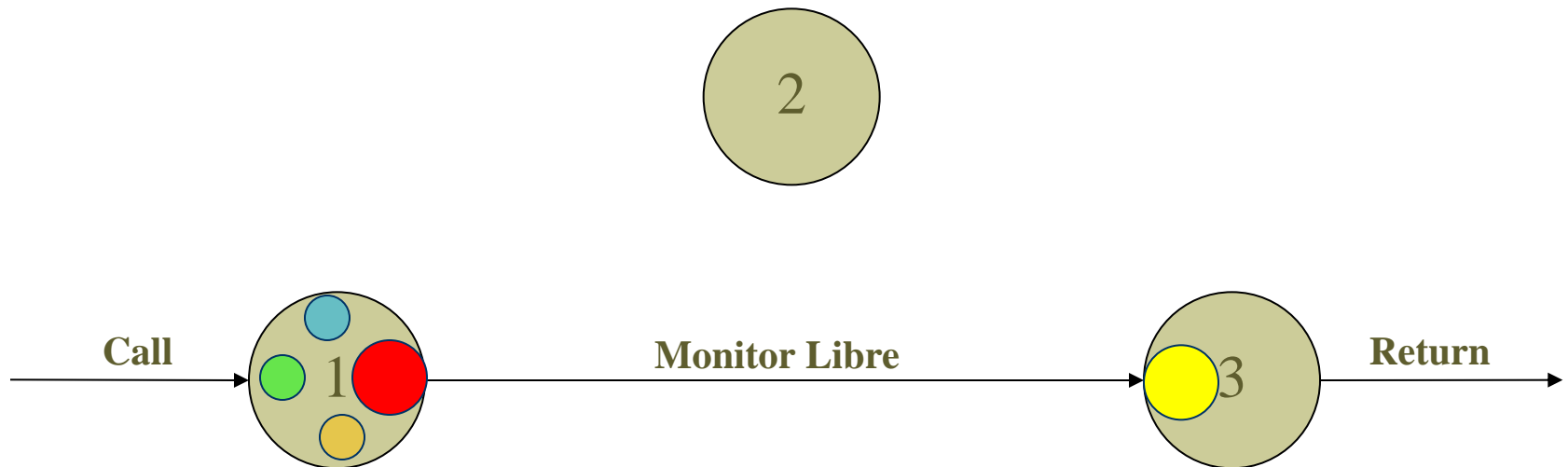
Llamado – Monitor Ocupado



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

Sincronización

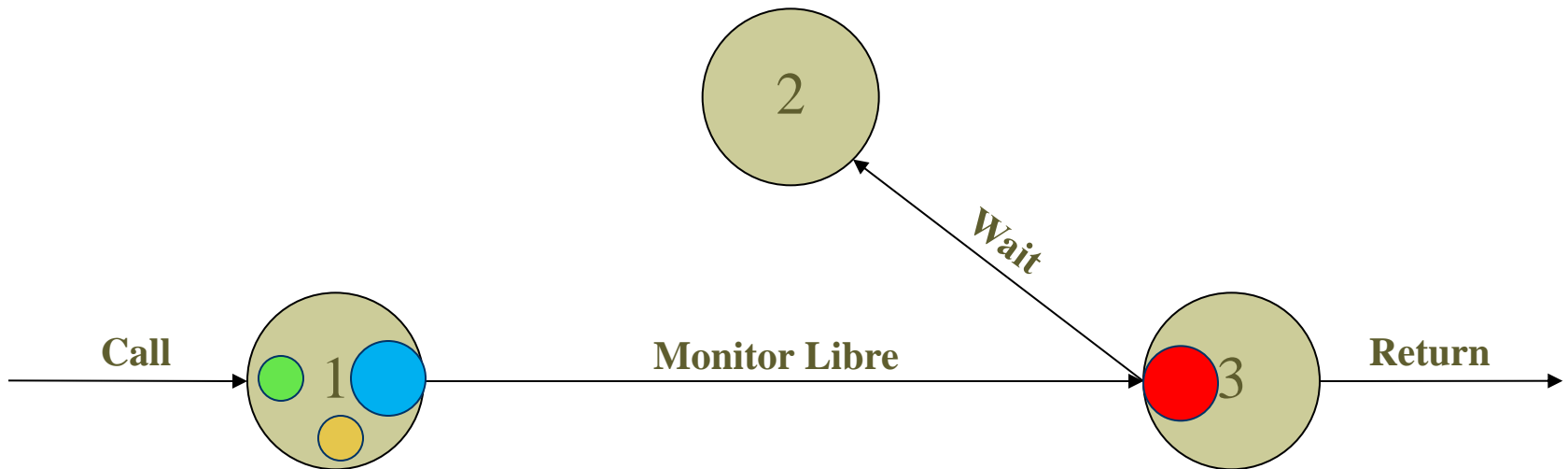
Liberación del Monitor



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

Sincronización

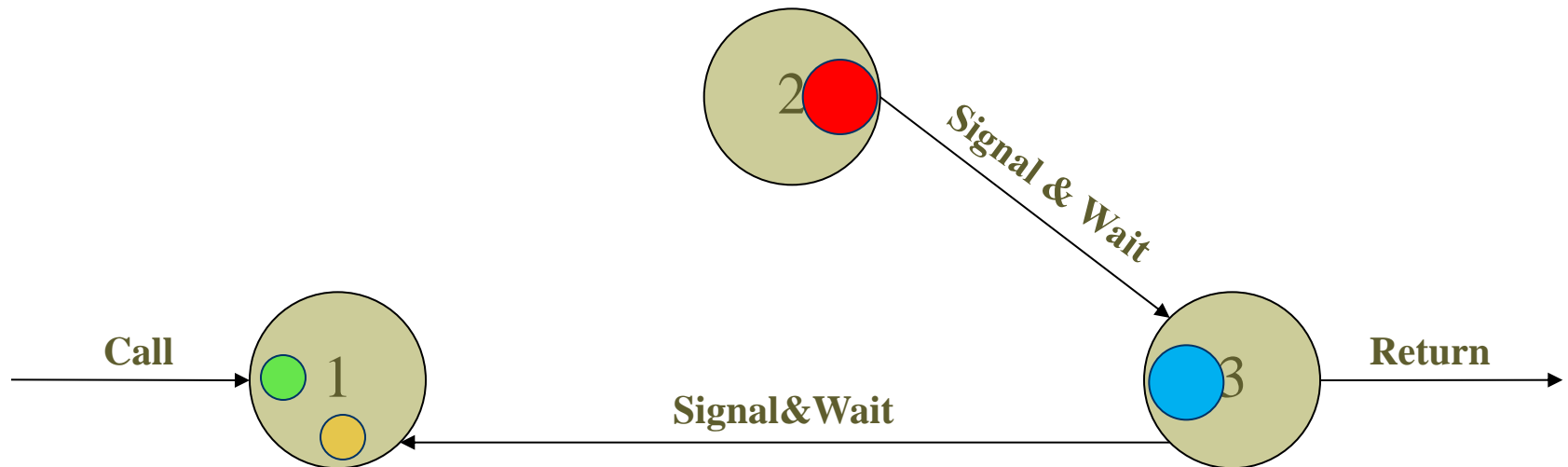
Wait



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

Sincronización

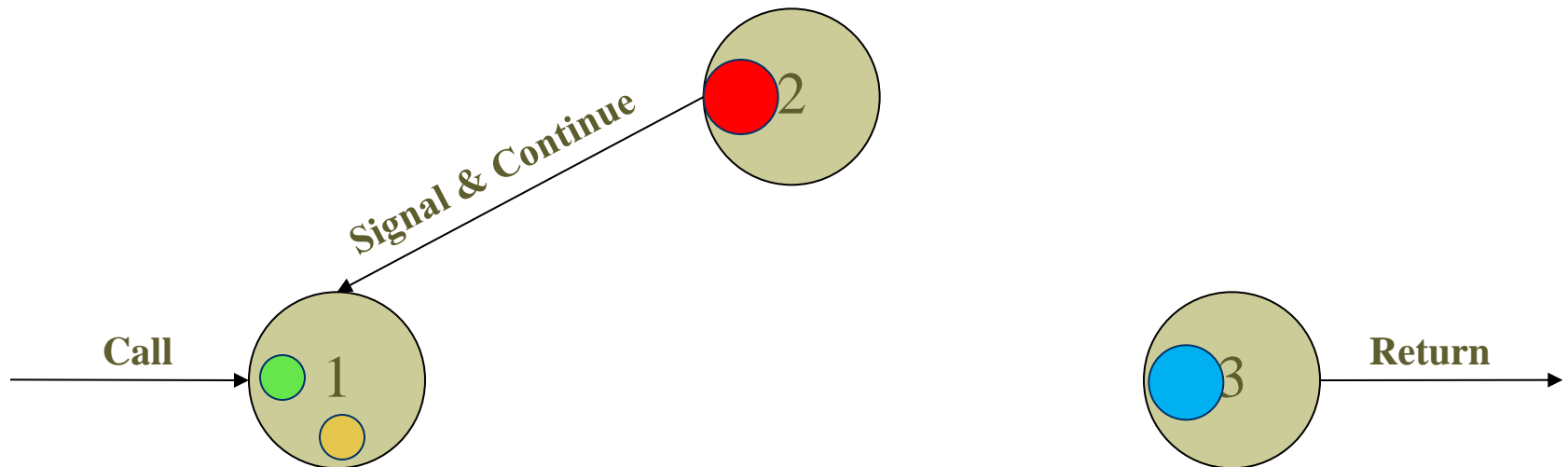
Signal - Disciplina Signal and Wait



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

Sincronización

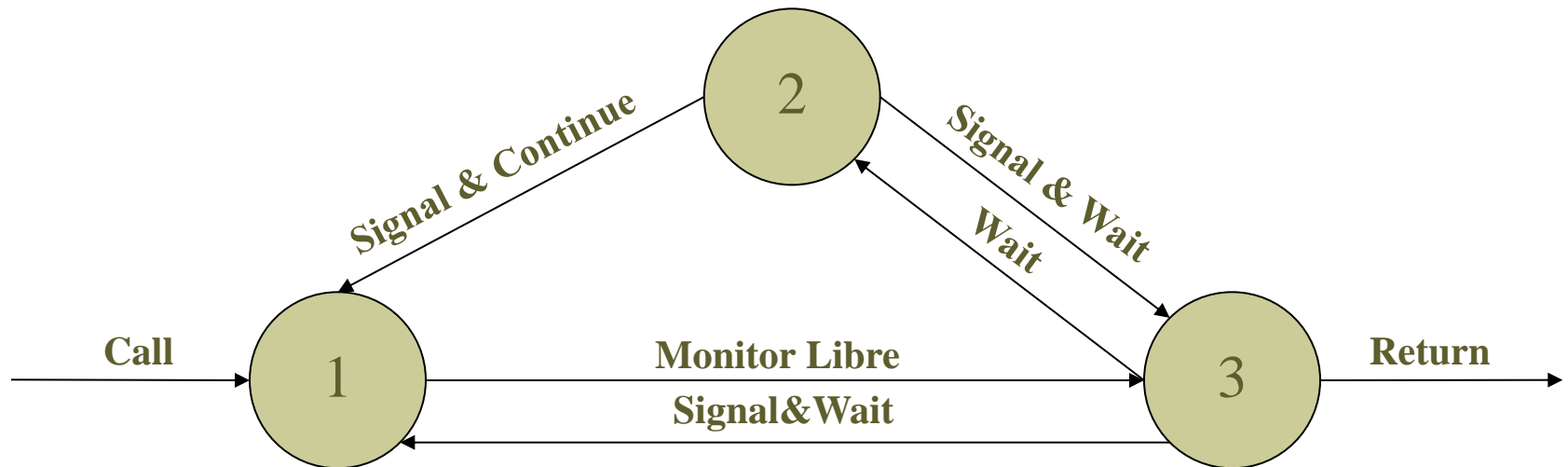
Signal - Disciplina Signal and Continue



- 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* continúa 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 continúe. 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 el valor 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);  
  }  
}
```

Operaciones adicionales sobre las Variables Condición

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.



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?

¿Qué 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

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);
    }
  }
}
```