Programación Concurrente 2015

Clase 5

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

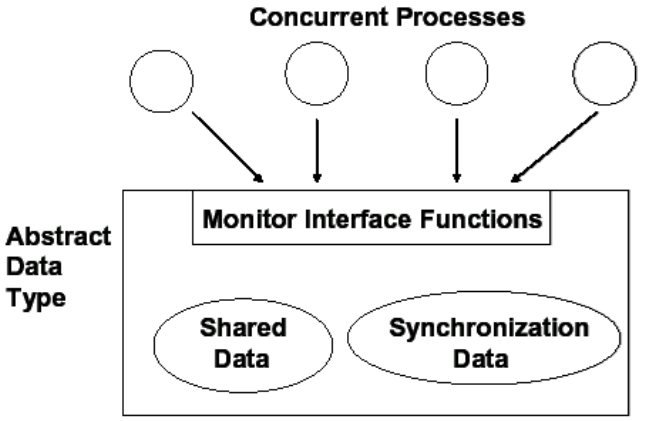
Exclusión Mutua ⇒ implícita asegurando que los ***procedures*** en el mismo monitor no ejecutan concurrentemente.

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:

  call NombreMonitor.op$_i$ (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
     }
}

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.

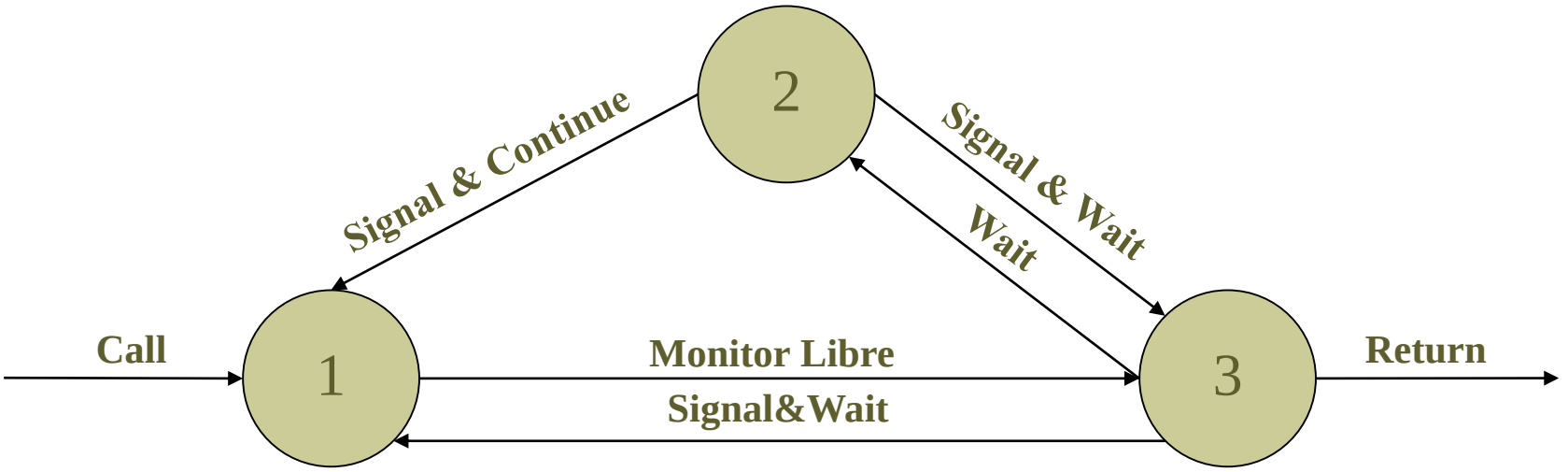
Operaciones adicionales

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

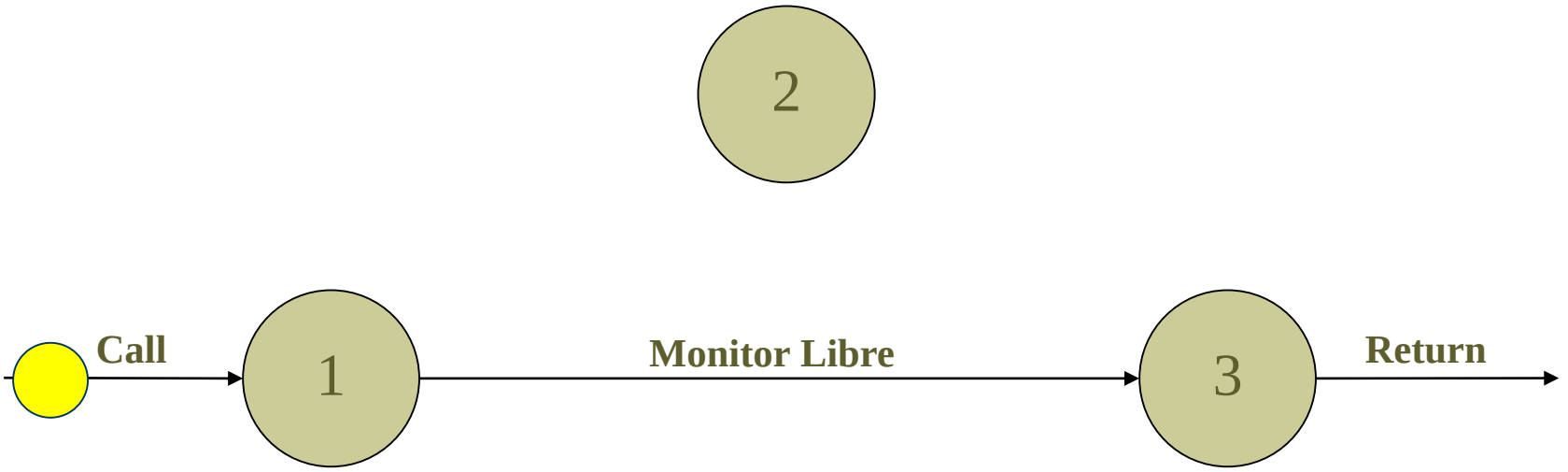
Signal and continue* vs. *Signal and Wait



1- Cola de Entrada.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

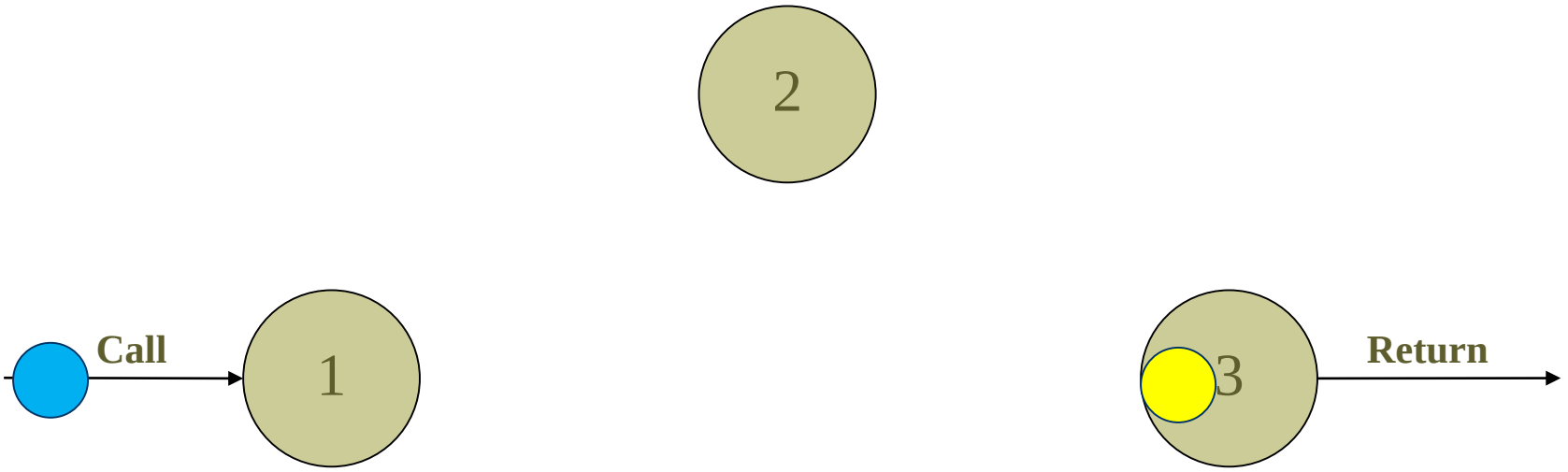
Llamado – Monitor Libre



1- Cola de Entrada.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

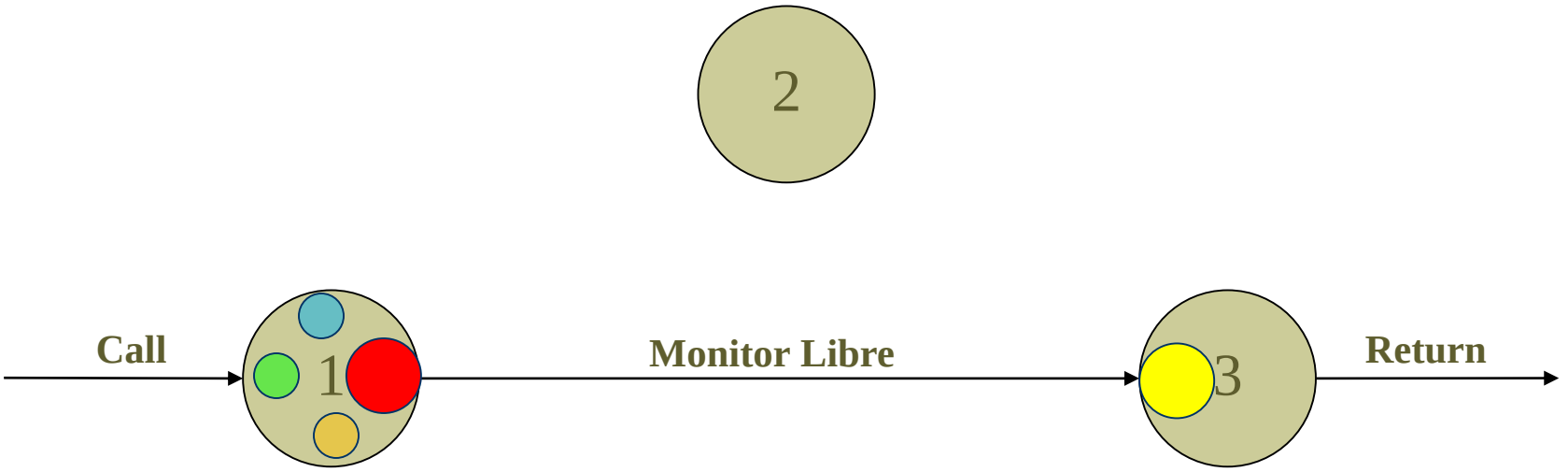
Llamado – Monitor Ocupado



1- Cola de Entrada.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

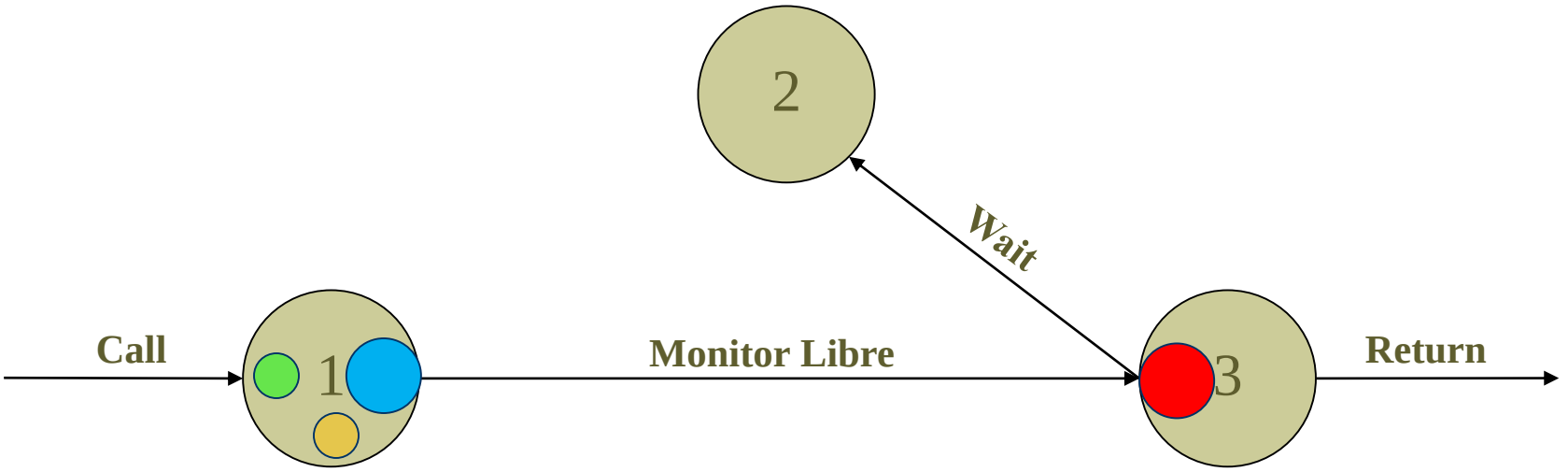
Liberación del Monitor



1- Cola de Entrada.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

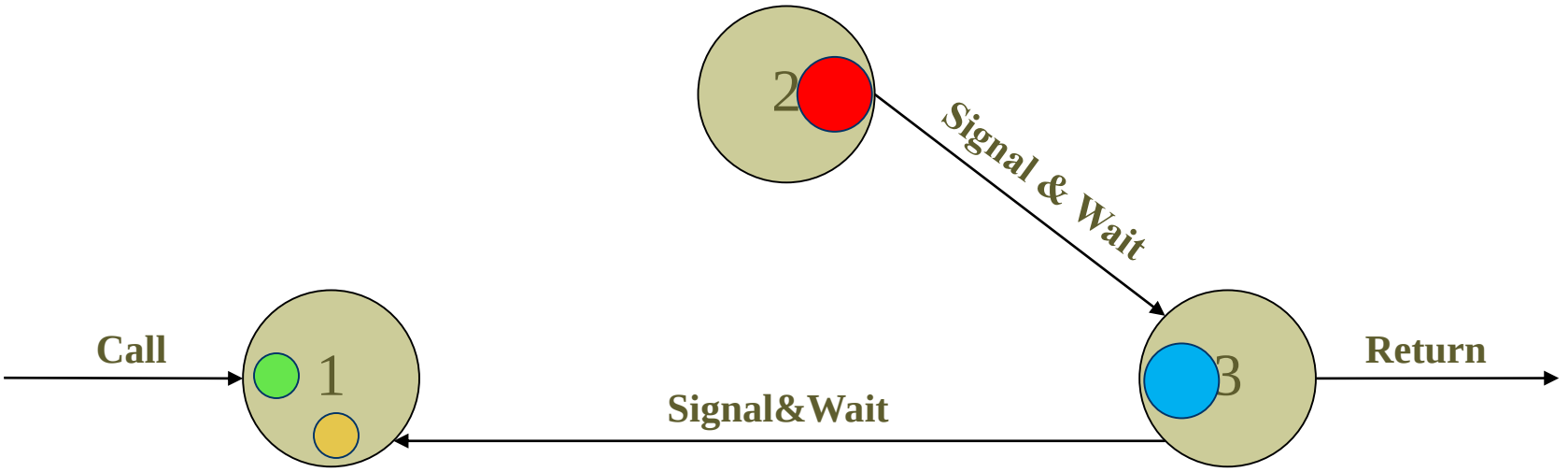
Wait



1- Cola de Entrada.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

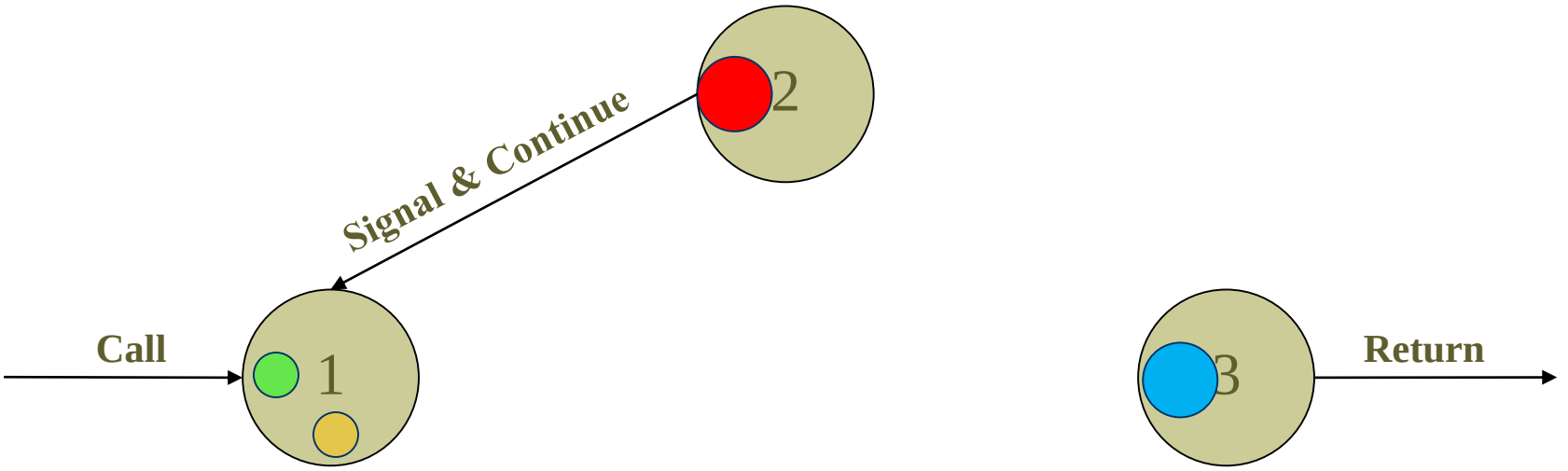
Signal - Disciplina Signal and Wait



1- Cola de Entrada.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

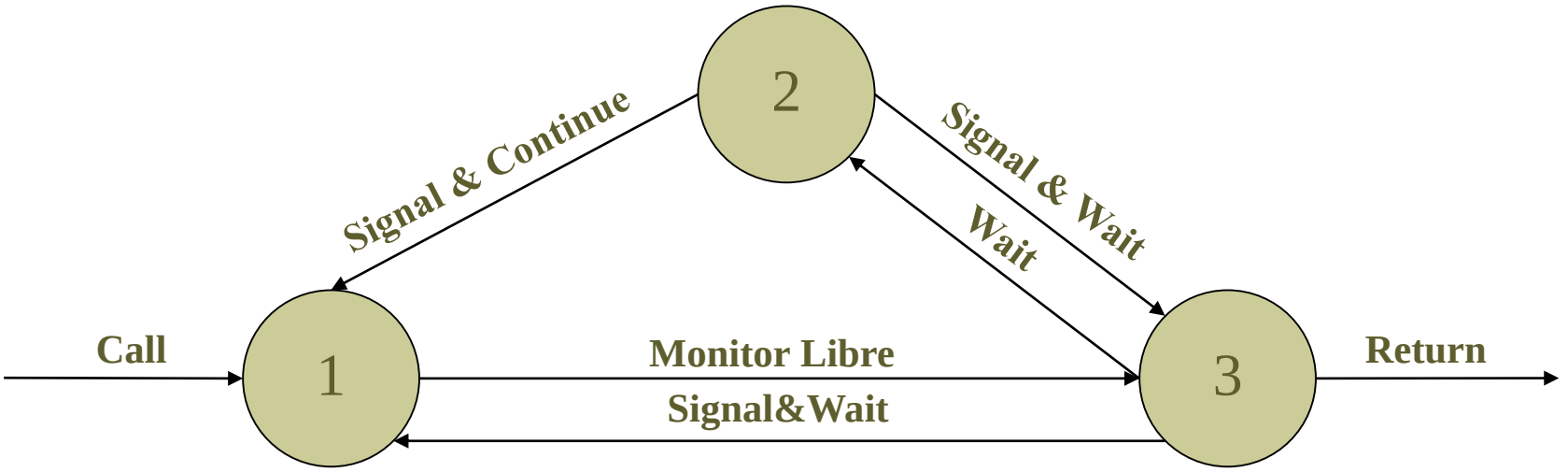
Signal - Disciplina Signal and Continue



1- Cola de Entrada.
2- Cola por Variable Condición.
3- Ejecutando en el Monitor.

Sincronización

Signal and continue* vs. *Signal and Wait



1- Cola de Entrada.
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

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 de 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*

- Manejo del 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]);
             };
    };
}
```