Antecedentes

Un proceso **cooperativo** es aquél que puede afectar o verse afectado por los demás procesos que se están ejecutando en el sistema.

Productor-Consumidor

Definición del buffer:

```
var n;
type elemento = ...;
var buffer array[0..n-1] of elemento;
    entra, sale: 0..n-1;
    contador = 0;
```

Definición del productor:

```
repeat
   ...
   producir un elemento en sigp
   ...
   while contador = n do nada;
   buffer[entra] := sigp;
   entra := entra + 1 mod n;
   contador := contador + 1;
until false;
```

Definición del consumidor:

```
repeat
   while contador = 0 do nada;
   sigc := buffer[sale];
   sale := sale + 1 mod n;
   contador := contador - 1;
   ...
   consumir el elemento que está en sigc
   ...
until false;
```


T_0 :	*productor*	**ejecutar**	$registro_1 := contador;$	$\{registro_1 = 5\}$
T_1 :	*productor*	**ejecutar**	$registro_1 := registro_1 + 1;$	$\{registro_1 = 6\}$
T_2 :	*consumidor*	**ejecutar**	$registro_2 := contador;$	$\{registro_2 = 5\}$
T_3 :	*consumidor*	**ejecutar**	$registro_2 := registro_2 - 1;$	$\{registro_2 = 4\}$
T_4 :	*productor*	**ejecutar**	$contador := registro_1$	$\{contador = 6\}$
T_5 :	*consumidor*	**ejecutar**	$contador := registro_2$	$\{contador = 4\}$

Problema de la sección crítica

- Para controlar el acceso a un recurso compartido, se declara una sección de código como *sección crítica*, la cual tiene regulado su acceso.
- Un sistema que consta de n hilos $(T_0, T_1, \ldots, T_n)$, cada hilo tiene un segmento de código, denominado **sección crítica**, en el cual el hilo puede estar modificando variables comunes, actualizando una tabla, escribiendo un archivo, etc.
- La característica importante del sistema es que, cuando un hilo está ejecutando en su sección crítica, no se debe permitir que otros hilos se ejecuten en la misma sección (*mutuamente excluyente*).
- *El problema de la sección crítica* es cómo elegir un protocolo que se puedan usar los hilos para cooperar.

Un proceso P_i

repeat

sección de ingreso

sección crítica

sección de egreso

sección restante

until *false*;

Requisitos para la sección crítica

1. Debe cumplirse la exclusión mutua: sólo un proceso de entre todos los poseen secciones críticas por el mismo recurso u objeto compartido, debe tener permiso para entrar en ella en un instante dado.
2. Un proceso que se interrumpe en un sección crítica debe hacerlo sin interferir con los otros procesos.
3. Un proceso no debe poder solicitar acceso a una sección crítica para después ser demorado indefinidamente; no puede permitirse el interbloqueo o la inanición.
4. Cuando ningún proceso está en su sección crítica, cualquier proceso que solicite entrar en la suya debe poder hacerlo sin dilación.
5. No se deben hacer suposiciones sobre la velocidad relativa de los procesos o el número de procesadores.
6. Un proceso permanece en su sección crítica por un tiempo finito.

Soluciones para dos procesos

Algoritmo 1

repeat

while *turno* <> *i* **do** nada;

sección crítica

turno := j;

sección restante

until false;

Soluciones para dos procesos

Algoritmo 2
Compartido entre los dos procesos:

var *indicador*: **array** [0..1] **of** *boolean*;

Cada proceso:

repeat

indicador[*i*] := *true*;
while *indicador*[*j*] **do** *nada*;

sección crítica

indicador[*i*] := *false*;

sección restante

until false;

Soluciones para dos procesos

Algoritmo 2

Compartido entre los dos procesos:

```
var indicador: array [0..1] of boolean;
```

Cada proceso:

```
repeat

    indicador[i] := true;
    while indicador[j] do nada;

        sección crítica

    indicador[i] := false;

        sección restante

until false;
```


Soluciones para múltiples procesos

Compartido entre todos los procesos:

```
var escogiendo: array [0..n-1] of boolean;
var número: array [0..n-1] of integer;

repeat

    escogiendo[i] := true;
    número := máx(número[0], número[1],
                  ..., número[n-1]) + 1;
    escogiendo[i] := false;
    for j := 0 to n - 1
        do begin
            while escogiendo[j] do nada;
            while (número[j] <> 0
              and (número[j],j) < (número[i],i))
                do nada;
        end;

        sección crítica

    número[i] := 0;

        sección restante

until false;
```


Hardware de sincronización

Definición de la función Evaluar-y-Asignar

```
function Evaluar-y-Asignar( var objetivo: boolean):
    boolean;
begin
  Evaluar-y-Asignar := objetivo;
  objetivo := true;
end
```

Exclusión mutua (Sección Crítica)

```
repeat
    while Evaluar-y-Asignar(cerradura) do nada;
        sección crítica;
    cerradura := false;
        sección restante;
until false;
```


Hardware de sincronización

Instrucción intercambiar

```
procedure Intercambiar( var a, b:boolean);
    var temp:boolean;
    begin
        temp := a;
        a := b;
        b := temp;
    end;
```

Exclusión mutua (Sección Crítica)

```
repeat

    llave := true;
    repeat
        Intercambiar(cerradura, llave);
    until llave = false;

        sección crítica

    cerradura := false;

        sección restante
```

Semáforos

Definición

Un semáforo S es una variable entera a la que. una vez se le ha asignado un valor inicial, sólo puede accederse a través de dos operaciones atómicas estándar: *espera* (*wait*) y *señal* (*signal*).
Estas operaciones se llamaban originalmente P (para *espera*; del holandés *proberen*, probar) y V (para señal; de *verhogen*, incrementar).

```
espera(S): while S <= 0 do nada;
           S := S - 1;

señal(S):  S := S + 1;
```


Hardware de sincronización

```
var esperando: array[0..n-1] of boolean;
    cerradura:boolean;

var j:0..n - 1;
    llave:boolean;
repeat

   esperando[i] := true;
   llave := true;
   while esperando[i] and llave do
      llave := Evaluar-y-Asignar(cerradura);
   esperando[i] := false;

      sección crítica;

   j := j + 1 mod n;
   while j <> i and ( not esperando[j]) do
      j := j + 1 mod n;
   if j = i then cerradura := false
   else esperando[j] = false;

      sección restante;
until false;
```


Semáforos

Implementación de la exclusión mutua

```
repeat
   espera(mutex);
      sección critica;
      :
   señal(mutex);
   ...
      sección restante;
until false;
```

Sincronización

Proceso P_0:

```
instrucciones;
señal(sinc);
```

Proceso P_1:

```
espera(sinc);
instrucciones
```


Semáforos

Implementación
Se define un semáforo como un registro:

```
type semáforo = record
                valor: integer;
                L: list of proceso;
        end;
```

Las operaciones del semáforo se definen así:

```
espera(S): S.valor - 1;
           if S.valor < 0
           then begin
              agregar este proceso a S.L;
              bloquear;
           end;

señal(S): S.valor + 1;
          if S.valor <= 0
          then begin
             quitar un proceso P de S.L;
             despertar(P);
          end;
```

Semáforos binarios

Implementación
Se define un semáforo como un registro:

```
type semáforo-binario = record
            valor: enum(0,1);
            L: list of proceso;
     end;
```

Las operaciones del semáforo se definen así:

```
espera(S): if S.valor = 1
           then S.valor = 0;
           else begin
              agregar este proceso a S.L;
              bloquear;
           end;

señal(S): if S.L.está_vacia()
          then S.valor = 1;
          else begin
             quitar un proceso P de S.L;
             despertar(P);
          end;
```


Bloqueos mutuos e inanición

P_0	P_1
espera(S);	*espera*(Q);
espera(Q);	*espera*(S);
...	...
señal(S);	*señal*(Q);
señal(Q);	*señal*(S);

Uso de un semáforo binario

Implementación
Se define un semáforo como un registro:

```
var S1: semáforo-binario;
    S2: semáforo-binario;
    C: integer;
```

Inicialmente, $S1 = 1$, $S2 = 0$, y el valor entero C se hace igual al valor inicial del semáforo de conteo S.

```
espera(S): espera(S1);
           C := C - 1;
           if C < 0;
           then begin
              señal(S1);
              espera(S2);
           end
           señal(S1);

señal(S): espera(S1);
          C := C + 1;
          if C <= 0;
          then señal(S2)
          else señal(S1);
```


El problema del buffer limitado

Productor

```
repeat
  .
  .
  .
  producir un elemento en sigp
  ...
  wait(vacíos);
  wait(mutex);
  .
  .
  .
  signal(mutex);
  signal(llenos);
until false;
```

Consumidor

```
repeat
  wait(llenos);
  wait(mutex);
  .
  .
  .
  quitar elemento del buffer y ponerlo en sigc
  .
  .
  .
  signal(mutex);
  signal(llenos);
until false;
```

Regiones críticas

Declaración:

var *v* ; **shared** *T* ;

Acceso a la variable

region *v* **when** *B* **do** *S* ;

Ejecución simúltanea de dos procesos

region *v* **when** *true* **do** *S1* ;

region *v* **when** *true* **do** *S2* ;

El problema de los lectores y escritores

Los procesos lectores y escritores comparten las estructuras de datos siguientes:

```
var mutex, escr: semáforo;
    cuentalect: integer;
```

Estructura de un proceso escritor

```
espera(escr);
   .
   .
   .
   se realiza la escritura
   .
   .
   .
señal(escr);
```

Estructura de un proceso lector

```
espera(mutex);
   cuentalect := cuentalect + 1;
   if cuentalect = 1 then espera(escr);
señal(mutex);
   .
   .
   .
   se realiza la lectura
   .
   .
   .
espera(mutex);
   cuentalect := cuentalect - 1;
   if cuentalect = 0 then señal(escr);
señal(mutex);
```


Regiones críticas

Solución al problema del buffer limitado

Datos compartidos:

```
var buffer: shared record
          reserva: [0..n-1] of elemento;
          cuenta, entra, sale: integer;
```

Proceso productor

```
region buffer when cuenta < n
    do begin
       reserva[entra] := sigp;
       entra := entra + 1 mod n;
       cuenta := cuenta + 1;
    end;
```

Proceso consumidor

```
region buffer when cuenta > 0
    do begin
       sigc := reserva[sale];
       sale := sale + 1 mod n;
       cuenta := cuenta - 1;
    end;
```


Monitores

```
type nombre-monitor = monitor
  declaraciones de variables

   procedure entry P1(...)
      begin ... end;

   procedure entry P2(...)
      begin ... end;

      .
      .
      .

   procedure entry Pn(...)
      begin ... end;

   begin
     código de inicialización
   end.
```

Monitores

Variables de condición

Las variables de condición es el mecanismo que permite sincronizar a los procesos dentro de los hilos:

```
var x, y: condición;
```

Las únicas operaciones que se puede realizar con una variable de condición son espera y señal.

```
x.espera;
x.señal;
```


Primitivas `send` y `receive`

Existe una variedad de primitivas `send` y `receive`

1. Cuándo un mensaje es enviado, ¿El proceso transmisor tendrá que que esperar hasta que el mensaje sea aceptado por el receptor, o este puede continuar procesando?
2. ¿Qué debe suceder cuando una `receive` es lanzada y no hay ningún mensaje aguardando?
3. ¿El transmisor debe determinar un único receptor al cuál desea transmitir el mensaje, o los mensajes pueden ser aceptados por cualquier grupo de receptores?
4. ¿El receptor debe aceptar aceptar un mensaje exactamente de un transmisor o puede aceptar mensajes que lleguen de diferentes transmisores?

Esquemas de sincronización basados en mensajes

- Los mecanismos anteriores están basados en la hipótesis que los procesos comparten alguna porción de memoria.
- Existen casos en los que no es deseable o posible que los procesos compartan alguna porción de memoria (seguridad, sistemas distribuidos).

send(*p*, *msg*);
receive(*q*, *msg*);


```
const N = 100;
type mensaje = ...;

                    Productor

process productor()
begin
  var elem:Element;
      m:mensaje;
  repeat
    elem = producir_elem();
    receive(consumidor, m);
    formar_mensaje(m, elem);
    send(consumidor, m);
  until false;
end.

                    Consumidor

process consumidor()
begin
  var elem:Element;
      m:mensaje;
      i:integer;
  for i := 0 to N do send(productor, &m);
  repeat
    receive(productor, m);
    elem := extraer_elem(m);
    send(productor, m);
    consumir_elem(elem);
  until false;
end.
```

Problema de los fumadores de cigarrillos

Considere un sistema con tres procesos *fumador* y un proceso *agente*. Cada fumador pasa todo su tiempo enrollando cigarillos y fumándolos. Sin embargo, para poder hacerlo, el fumador necesita tres ingrediente: tabaco, papel y cerillas. Uno de los procesos fumadores tiene papel, otro tabaco y el tercero tiene cerillas. El agente tiene un abasto infinito de los tres materiales. El agente coloca dos de los ingredientes en la mesa. Entonces, el fumador que tiene el ingrediente faltante enrolla y fuma un cigarillo, enviando una señal al agente cuando termina. A continuación, el agente coloca otros dos de los tres ingredientes en la mesa, y el ciclo se repite. Escriba un programa que sincronice al agente y los fumadores.

Problema de la cena de los filósofos

En 1965, Dijkstra planteó y resolvió un problema de sincronización y que conduce a bloqueos.

> Cinco filósofos están sentados en torno de una mesa circular. Cada uno tiene un plato de espagueti. El espagueti es tan escurridizo que necesita dos tenedores para comerlo. Entre cada par de platos hay un tenedor.
> La vida de un filósofo consiste en periodos alternados de comer y pensar. (Esto es un abstracción considerable, incluso hablando de filósofos, pero las demás actividades no viene al caso). Cuando un filósofo siente hambre, trata de tomar los dos tenedores, come durante un tiempo, luego los deja en la mesa y sigue pensando.

Solución 1

```
#define N 5

void filosofo(int i) {

   while (TRUE) {
      pensar();
      tomar_tenedor(i);
      tomar_tenedor((i+1) % N);
      comer();
      dejar_tenedor(i);
      dejar_tenedor((i+1) % N);
   }
}
```


Solución utilizando monitores

```
filósofos:monitor
type t = array[0..4] of integer;
     t_disponibles = [0..4] of condition;
var  tenedores:t;
     tenedores_disponibles:t_disponibles;

procedure iniciar_comer(i:int)
begin
   if tenedores(i) <> 2
      then tenedores_disponibles(i).wait;
   tenedores(i - 1 mod 5) := tenedores(i - 1 mod 5) - 1;
   tenedores(i + 1 mod 5) := tenedores(i + 1 mod 5) - 1;
end;

procedure parar_comer(i:int)
begin
   tenedores(i - 1 mod 5) := tenedores(i - 1 mod 5) + 1;
   tenedores(i + 1 mod 5) := tenedores(i + 1 mod 5) + 1;
   if tenedores(i - 1 mod 5) = 2 then
      tenedores_disponibles(i - 1 mod 5).signal;
   if tenedores(i + 1 mod 5) = 2 then
      tenedores_disponibles(i + 1 mod 5).signal;
end;

begin
   for i := 0 to 4 do tenedores(i) := 2;
end
```

Solución utilizando semáforos

```
#define N          5
#define IZQ        (i + N - 1) % N
#define DER        (i + 1) % N
#define PENSANDO   0
#define HAMBRE     1
#define COMIENDO   2
typedef int semaphore;
int estado[N];
semaforo mutex = 1;
semaforo s[N];

void filosofo(int i) {
   while (TRUE) {
      pensar();
      tomar_tenedor(i);
      comer();
      dejar_tenedor(i);
   }
}

void tomar_tenedor(int i) {
  down(&mutex);
  estado[i] = HAMBRE;
  probar(i);
  up(&mutex);
  down(&s[i]);
}
```


Solución utilizando semáforos

```
void dejar_tenedor(int i) {
  down(&mutex);
  estado[i] = PENSANDO;
  probar(IZQ);
  probar(DER);
  up(&mutex);
}

void probar(int i) {
  if (estado[i] == HAMBRE && estado[IZQ] != COMIENDO
      && estado[DER] != COMIENDO) {
     estado[i] = COMIENDO;
     up(&s[i]);
  }
}
```


Buffer limitado con región crítica

```
var buffer: shared record
   reserva: array [0..n-1] of elemento;
   cuenta, entra, sale:integer;
end;

{Código del productor}
    region buffer when cuenta < n
       do begin
          reserva[entra] := sigp;
          entra := entra + 1 mod n;
          cuenta := cuenta + 1;
       end;

{Código del consumidor}
    region buffer when cuenta > 0
       do begin
          sigc := reserva[sale];
          sale := sale + 1 mod n;
          cuenta := cuenta + 1;
       end;
```


Bloqueos Mutuos

Ejemplo de bloqueo mutuo (deadlock)

"Si dos trenes se aproximan el uno al otro en un cruce, ambos harán un alto total y ninguno arrancará de nuevo hasta que el otro se haya ido"

Condiciones necesarias

1. Exclusión mutua: Al menos un recurso debe adquirirse de modo que no pueda compartirse; es decir; sólo un proceso a la vez podrá usar ese recurso. Si otro proceso solicita ese recurso, el proceso solicitante deberá esperar hasta que se haya liberado el recurso.
2. Retener y esperar: Debe existir un proceso que haya adquirido al menos un recurso y esté esperando para adquirir recursos adicionales que ya han sido asignados a otros procesos.
3. No expropiación: Los recursos no se pueden arrebatar; es decir, la liberación de un recurso siempre es voluntaria por parte del proceso que lo adquirió, una vez que ha terminado su tarea.
4. Esperar circular: Debe existir un conjunto $\{P_0, P_1, \ldots, P_n\}$ de procesos en espera tal que P_0 está esperando un recurso que fue adquirido por P_1, P_1 está esperando un recurso que fue adquirido por $P_2, \ldots, P_{n-1}$ está esperando un recurso que fue adquirido por P_n, y P_n está esperando un recurso que fue adquirido por P_0

Modelo del sistema

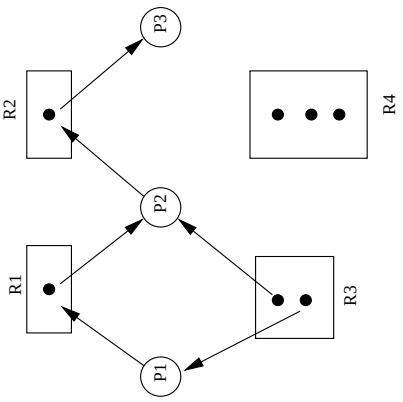
Un sistema consiste en un número finito de recursos que deben distribuirse entre varios procesos que compiten. Los recursos se dividen en varios tipos, cada uno de los cuales consiste en cierta cantidad de ejemplares idénticos. Un proceso sólo puede utilizar un recurso siguiendo la siguiente secuencia:

1. Solicitud: Si la solicitud no puede hacerse de inmediato, el proceso solicitante deberá esperar hasta que pueda adquirir el proceso.
2. Uso: El proceso puede operar con el recurso.
3. Liberar: El proceso libera el recurso.

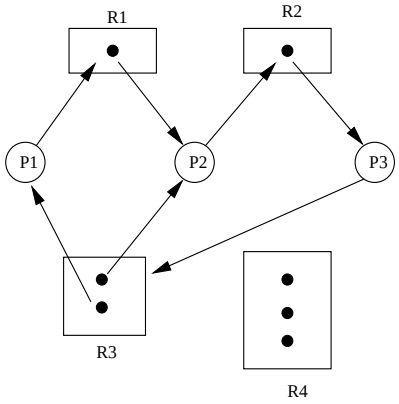
Grafo de asignación de recursos

Este grafo consiste en un conjunto de vértices V y un conjunto de aristas E. El conjunto V se divide en dos tipos de nodos distintos: $P = \{P_1, P_1, \ldots, P_n\}$, el conjunto de todos los procesos activos del sistema, y $R = \{R_1, R_2, \ldots, R_m\}$, el conjunto de todos los tipos de recursos del sistema.
Una arista dirigida del proceso P_i al tipo de recurso R_j, denotada por $P_i \rightarrow R_j$, indica que el proceso P_i solicitó un recurso del tipo R_j y está esperandoló.
Una arista dirigida del recurso R_j al proceso P_i, denotada por $R_j \rightarrow P_i$, indica que un ejemplar del tipo de recursos R_j se asignó al proceso P_i.

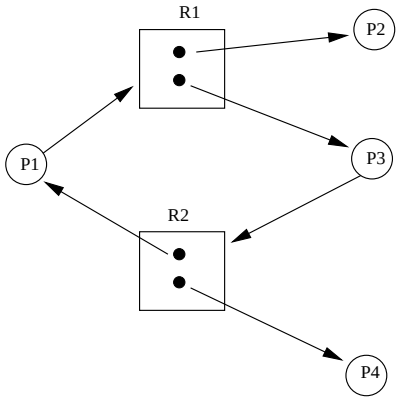
Ejemplo de grafo de asignación de recursos



Ejemplo de bloqueo en un grafo de asignación de recursos



Grafo de asignación de recursos con ciclo sin bloqueo



Métodos para manejar bloqueos mutuos

- Podemos usar un protocolo que asegure que el sistema *nunca* llegará a un estado de bloqueo mutuo.
- Podemos permitir que el sistema entre en bloqueo mutuo y luego se recupere.
- Podemos desentendernos del problema y hacer como si nunca ocurrieran bloqueos mutuos en el sistema. Esta solución es la que adoptan la mayor parte de los sistemas operativos, incluido UNIX.

Prevención de bloqueos mutuos

Exclusión mutua Compartir recursos (No todos los recurso se pueden compartir).

Retener y esperar .

- Obtener todos al tiempo.
- Obtener uno a la vez.

Expropiación .

- Expropiar los que tiene cuando solicita un ocupado.
- Expropiar únicamente los que otros procesos soliciten.

Prevención de bloqueos mutuos

Espera circular Sea $R = \{R_1, R_2, \ldots, R_m\}$ el conjunto de tipos de recursos. Definimos una función uno a uno $F : R \to N$, donde N es el conjunto de los números naturales. Un proceso puede solicitar inicialmente cualquier cantidad de ejemplares del tipo de recursos, digamos R_i. Después el proceso podrá solicitar ejemplares del tipo de recursos R_j si y sólo si $F(R_j) > F(R_i)$. Si se requiere varios ejemplares del mismo tipo de recurso, se deberá emitir *una sola* solicitud que los incluya a todos. Como alternativa, siempre que un proceso solicite un ejemplar del tipo de recursos R_j, haya liberado cualesquier recursos R_i que tenga, tales que $F(R_i) > F(R_j)$.

Evitar bloqueos mutuos

Los métodos de prevención tiene efectos secundarios como son un bajo aprovechamiento de los recursos y una reducción del rendimiento del sistema.
Un método alternativo para evitar bloqueos mutuos es pedir información adicional sobre la forma que se solicitarán los recursos.

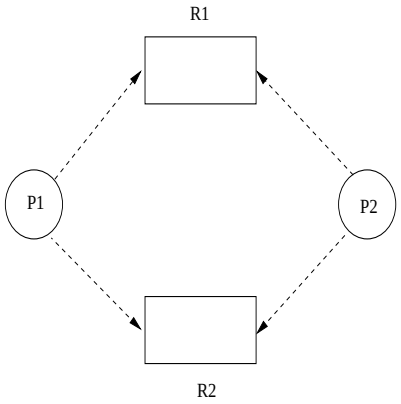
Estado seguro

Un estado es *seguro* si el sistema puede asignar recursos a cada proceso (hasta su máximo) en algún orden y aun así evitar los bloqueos mutuos. En términos más formales, un sistema está en un estado seguro sólo si existe una *secuencia segura*. Un secuencia de procesos $< P_1, P_2, \ldots, P_n >$ es una secuencia segura para el estado de asignación actual si, para cada P_i, los recursos que P_i todavía puede solicitar se pueden satisfacer con los recursos que actualmente están disponibles más los recursos que tienen todos los P_j, donde $j < i$. Si no existe tal secuencia el sistema es *inseguro*.

Ejemplo de un estado seguro e inseguro

	Necesidades máximas	**Necesidades actuales**
P_0	10	5
P_1	4	2
P_2	9	2

Algoritmo de grafo de asignación de recursos



Algoritmo del banquero

Estructuras de datos

$Disponible$**:** Un vector de longitud m indica el número de recursos disponibles de cada tipo. Si $Disponible[j] = k$, hay k ejemplares disponibles del tipo de recursos R_j.

Max**:** Una matriz $n \times m$ define la demanda máxima de cada proceso. Si $Max[i, j] = k$, el proceso P_i puede solicitar cuando más k ejemplares del tipo de recursos R_j.

$Asignación$**:** Una matriz $n \times m$ define el número de recursos de cada tipo que se han asignado actualmente a cada proceso. Si $Asignación[i, j] = k$, el proceso P_i tiene asignados actualmente k ejemplares del tipo de recursos R_j.

$Necesidad$**:** Una matriz $n \times m$ indica los recursos que todavía le hacen falta a cada procesos. Si $Necesidad[i, j] = k$, el proceso P_i podría necesitar k ejemplares más del tipo de recursos R_j para llevar a cabo su tarea. Observe que $Necesidad[i, j] = Max[i, j] - Asignacion[i, j]$.

Sean X y Y vectores con longitud n. Decimos que $X \leq Y$ si y sólo si $X[i] \leq Y[i]$ para toda $i = 1, 2, \ldots n$.

Algoritmo de seguridad

1. Sean $Trabajo$ y Fin vectores con longitud m y n, respectivamente. Asignar los valores iniciales $Trabajo := Disponible$ y $Fin[i] := falso$ para $i = 1, 2, \ldots, n$.
2. Buscar una i tal que
   a) $Fin[i] = falso$, y
   b) $Necesidad_i \leq Trabajo$

   Si no existe tal i, continuar con el paso 4.
3. $Trabajo := Trabajo + Asignacion_i$
   $Fin[i] := verdadero$
   ir al paso 2.
4. Si $Fin[i] = verdadero$ para toda i, el sistema está en un estado seguro.

Algoritmo de solicitud de recursos

Sea $Solicitud_i$ el vector de solicitudes del proceso P_i. Si $Solicitud_i[j] = k$, el proceso P_i quiere k ejemplares del tipo de recursos R_j. Cuando P_i solicita recursos, se emprende las acciones siguientes:

1. Si $Solicitud_i \leq Necesidad_i$, ir al paso 2. En caso contrario, indicar una condición de error, pues el proceso ha excedido su reserva máxima.
2. Si $Solicitud_i \leq Disponible$, ir al paso 3. En caso contrario P_i deberá esperar, ya que los recursos no están disponibles.
3. Hacer que el sistema simule haber asignado al proceso P_i los recursos que solicitó modificando el estado como sigue:

$$Disponible := Disponible - Solicitud_i; \quad (1)$$
$$Asignacion_i := Asignacion_i + Solicitud_i; \quad (2)$$
$$Necesidad_i := Necesidad_i - Solicitud_i; \quad (3)$$

Si el estado de asignación de recursos es seguro, la transacción se llevará a cabo y se asignaránm los recursos al proceso P_i; pero si el nuevo estado es inseguro, P_i tendrá que esperar $Solicitud_i$ y se restaurará el antiguo estado de asignación de recursos.

Detección de bloqueos mutuos

- Un algoritmo que examine el estado del sistema y determine si ha ocurrido un bloqueo mutuo.
- Un algoritmo para recuperarse del bloqueo mutuo.

Un esquema de detección y recuperación requiere un gasto extra que incluye no sólo los costos en tiempo de ejecución para mantener la información necesaria y ejecutar el algoritmo de detección, sino también las pérdidas potenciales inherentes a la recuperación después de un bloqueo mutuo.

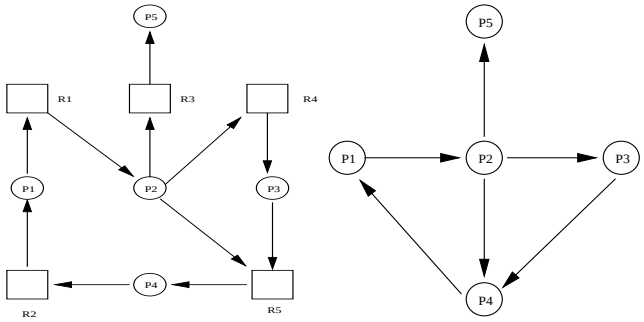
Un solo ejemplar de cada tipo de recursos

Existe una variante del grafo de asignación de recursos para el caso en que exista un solo ejemplar por cada tipo de recursos, se llama *grafo de espera*. Este se define así:

> Una arista de P_i a P_j en un grafo de espera implica que el proceso P_i está esperando que el proceso P_j libere un recurso que P_i necesita. Hay una arista $P_i \rightarrow P_j$ en un grafo de espera si y sólo si el grafo de asignación correspondiente tiene dos aristas $P_i \rightarrow R_q$ y $R_q \rightarrow P_j$ para algún recurso R_q.

Existe un *bloque mutuo* en el sistema si y sólo si el grafo de espera contiene un ciclo. Para detectar los bloqueos mutuos, el sistema necesita mantener el grafo de espera e invocar periódicamente un algoritmo que busque ciclos.

Grafo de espera



Varios ejemplares de un tipo de recursos

Estructuras de datos

$Disponible$**:** Un vector de longitud m indica el número de recursos disponibles de cada tipo.

$Asignación$**:** Una matriz $n \times m$ define el número de recursos de cada tipo que se han asignado actualmente a cada proceso.

$Solicitud$**:** Una matriz $n \times m$ indica la solicitud de cada proceso. Si $Solicitud[i,j] = k$, el proceso P_i está solicitando k ejemplares adicionales del tipo de recursos R_j.

Sean X y Y vectores con longitud n. Decimos que $X \leq Y$ si y sólo si $X[i] \leq Y[i]$ para toda $i = 1, 2, \ldots n$.

Algoritmo

1. Sean $Trabajo$ y Fin vectores con longitud m y n, respectivamente. Asígnese $Trabajo := Disponible$. Para $i = 1, 2, \ldots, n$ si $Asignación_i \neq 0$, entoces $Fin[i] := false$; si no, $Fin[i] := true$.
2. Busque un índice i tal que:
   a) $Fin[i] = false$, y
   b) $Solicitud \leq Trabajo$.

   Si no existe tal i, ir al paso 4.
3. $Trabajo := +Asignación_i$
   $Fin[i] := true$
   ir al paso 2.
4. Si $Fin[i] = false$, para alguna i, $1 \leq i \leq n$, el sistema está en estado de bloqueo mutuo. Es más, si $Fin[i] = false$, el proceso P_i está en bloqueo mutuo.

Uso del algoritmo de detección

1. ¿Con qué frecuencia es probable que ocurran bloqueos mutuos?
2. ¿A *cuántos* afectará un bloqueo mutuo si ocurre?

Recuperación después de un bloqueo mutuo

- Terminación de procesos
  - Abortar todos los proceso bloqueados.
  - Abortar un proceso a la vez hasta eliminar el ciclo de bloqueo mutuo.
    Factores
    1. ¿Qué prioridad tiene el proceso?
    2. ¿Cuánto tiempo ha trabajado el proceso, y cuánto trabajará antes de llevar a cabo su tarea asignada.
    3. ¿Cuántos recursos ha usado el proces, y de qué tipo (por ejemplo, si los recursos se pueden expropiar fácilmente).
    4. ¿Cuántos recursos adicionales necesita el proceso para terminar su tarea?
    5. ¿Cuántos procesos habrá que abortar?
    6. ¿Si los procesos son iteractivos o por lotes?

Recuperación después de un bloqueo mutuo

- Expropiación de recursos
  - Selección de la víctima
  - Retroceso
  - Inanición
- Una estrategia combinada de los anteriores items.