Universisdad de Los Andes
Facultad de Ingeniería
Escuela de Sistemas

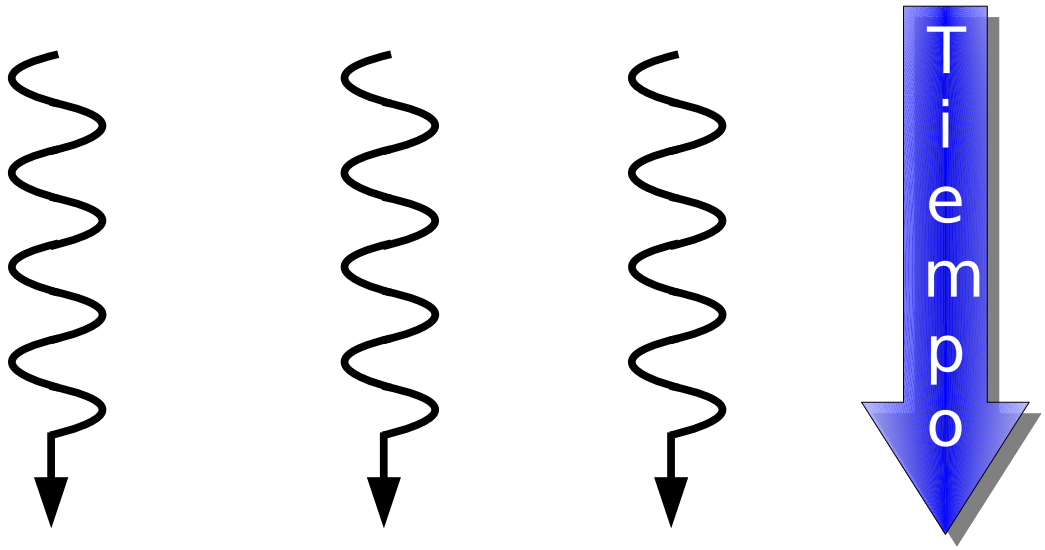
Introducción y Conceptos Básicos

Prof. Gilberto Díaz
gilberto@ula.ve

Departamento de Computación, Escuela de Sistemas, Facultad de Ingeniería
Universidad de Los Andes, Mérida 5101 Venezuela
Programación Paralela y Distribuida

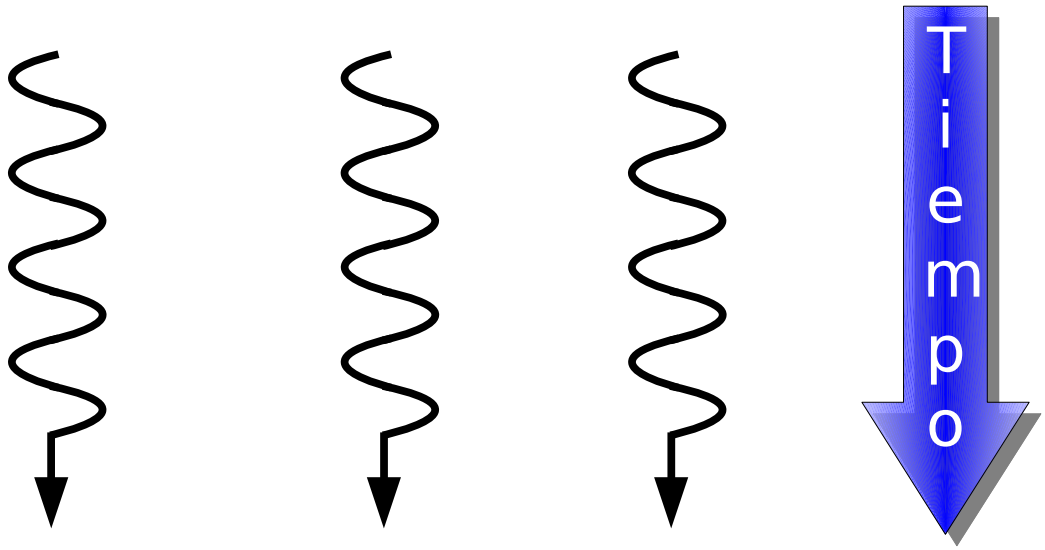
Asincronía

Los procesos son *concurrentes* si existen simultáneamente.



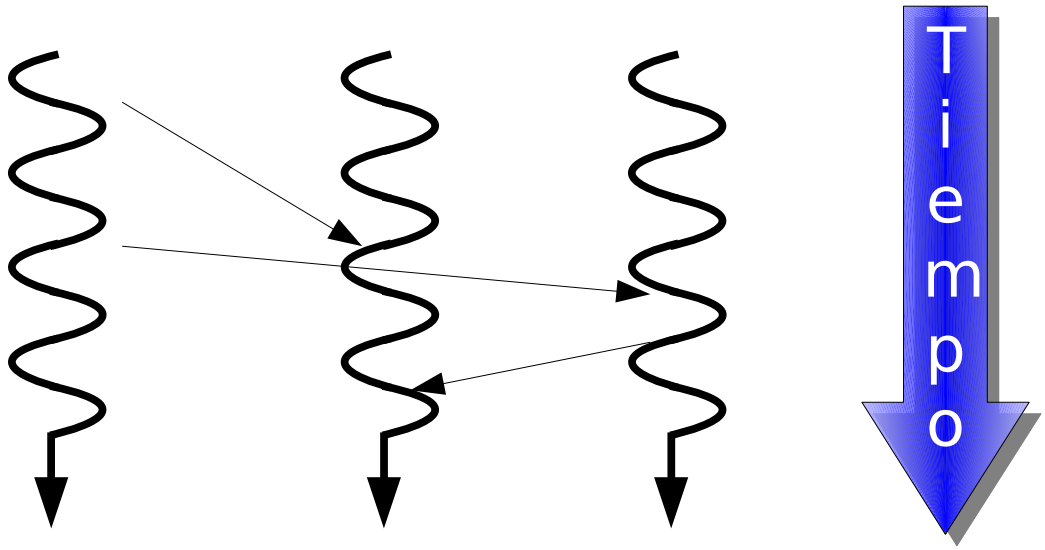
Asincronía

Los procesos *concurrentes* pueden trabajar de forma totalmente independientes unos de otros o pueden ser *asíncronos*



Procesos Asíncronos

Un conjunto de procesos son asíncronos cuando en ocaciones necesitan cierta sincronización y colaboración entre ellos



Nuevas Tecnologías

Actualmente los computadores están dotados con más de un procesador, incluso los computadores de escritorio

Nuevas Tecnologías

Esto hace que la programación distribuida y la programación paralela sean más propicias que antes.

Paralelismo

El procesamiento paralelo, además de interesante, es complejo debido a que generalmente las personas concentran su atención en una sóla tarea.

Por ejemplo, si intentamos leer dos libros a la vez haciéndolo de la siguiente manera:

Una línea del primero y una línea del segundo. Luego, la siguiente línea del primero y así sucesivamente

Paralelismo

Además, determinar cuales tareas pueden realizarse en paralelo es difícil.

También es muy difícil depurar un programa paralelo.

Luego de corregir (supuestamente) un error puede ser casi imposible reconstruir la secuencia de eventos que hicieron aparecer el error

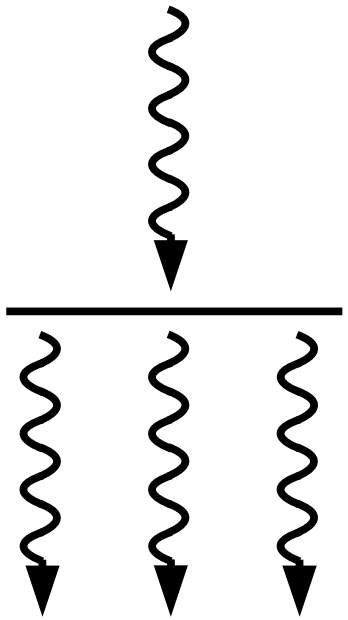
Paralelismo

Los lenguajes de programación paralelos necesitan construcciones para indicar el conjunto de sentencias que pueden ser ejecutadas en paralelos.

Muchos lenguajes proponen distintas estrategias para este propósito

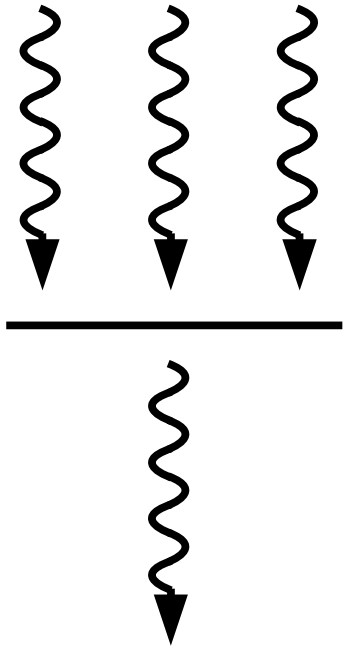
Estructura de para Indicar Paralelismo

Se necesita un enunciado para indicar que el flujo de ejecución se divide en varias secuencias de ejecución



Estructura de para Indicar Paralelismo

Y un enunciado para indicar varias secuencias de ejecución confluyen un sólo flujo de ejecución.



Estructura de para Indicar Paralelismo

Dijkstra recomienda utilizar las siguientes instrucciones para enmarcar las sentencias que pueden ser ejecutadas en paralelo.

```
Parbegin
Parend
```

Hay muchas otras propuestas en la literatura pero en este curso utilizaremos éstas.

Estructura de para Indicar Paralelismo

Como ejemplo de tareas que pueden realizarse de forma paralela podemos citar el cálculo de las raices de un polinomio de segundo grado

```
discriminante = b*b - 4*a*c;
Parbegin
   x1=(-b + sqrt(discriminante))/(2*a);
   x2=(-b - sqrt(discriminante))/(2*a);
Parend
```

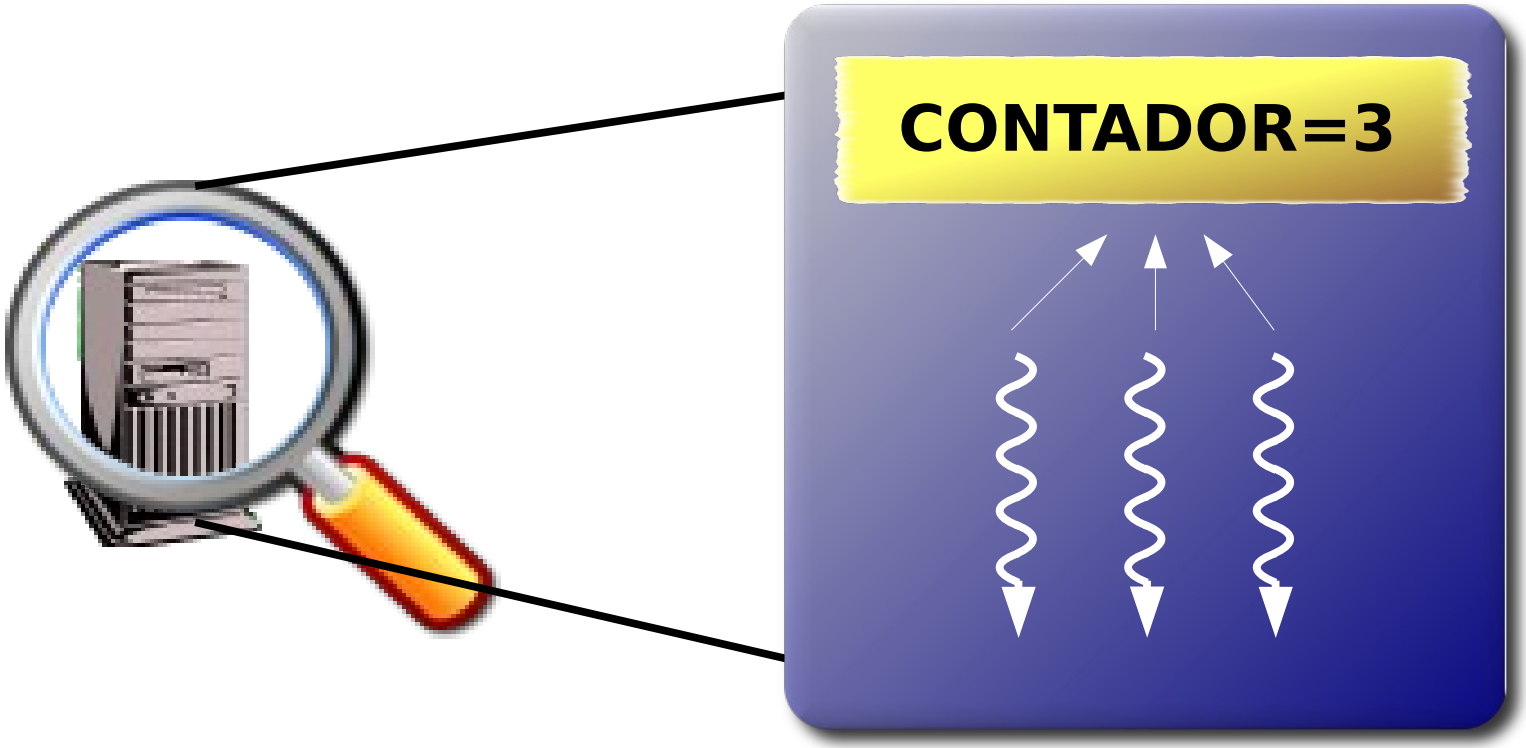
Exclusión Mutua

Para explicar la exclusión mutua utilizaremos un ejemplo:

Supongamos que se desea contar el número de accesos a una página web

Exclusión Mutua

En el servidor hay varios procesos de un servidor web que atienden las solicitudes y comparten una variable “CONTADOR” donde escriben el número de accesos



Exclusión Mutua

Cada proceso tiene que cargar el valor actual sumar uno y luego escribir el nuevo valor

Exclusión Mutua

Consideremos lo que pasa cuando dos procesos tratan de modificar la variable al mismo tiempo.

Exclusión Mutua

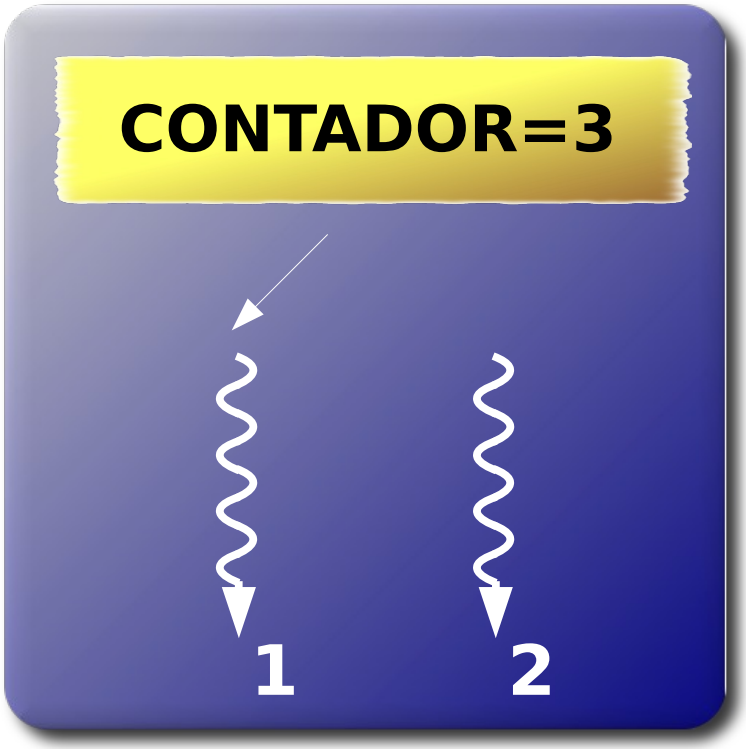
Supongamos que cada proceso tiene su propia copia del código

```
Aux = CONTADOR
Aux++
CONTADOR = Aux
```

Exclusión Mutua

También supongamos que el proceso 1 ejecuta las 2 primeras instrucciones

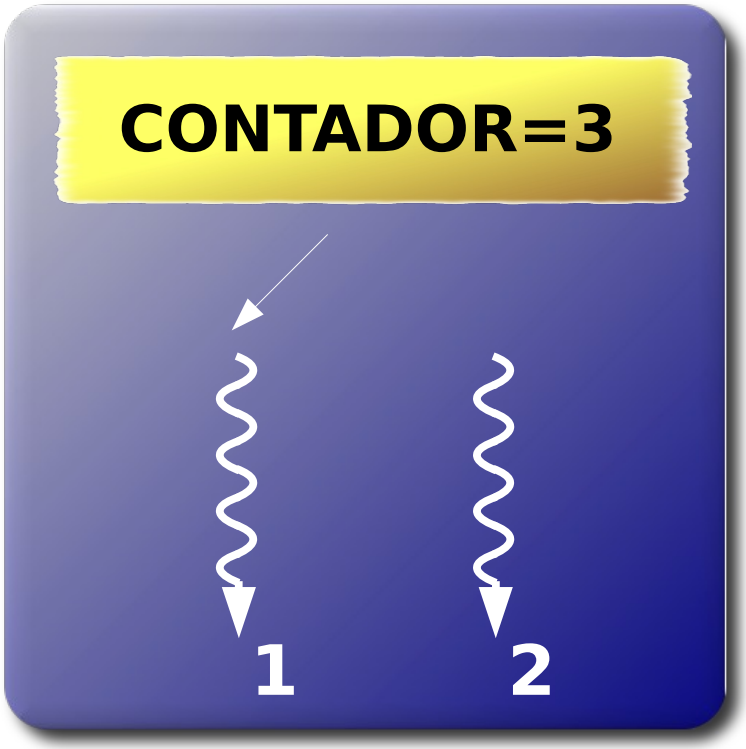
```
Aux = CONTADOR
Aux++
```



Exclusión Mutua

Y antes de ejecutar la tercera instrucción este pierde el procesador por haber expirado el quantum

```
CONTADOR = Aux
```



Exclusión Mutua

Entonces el proceso 2 entra al procesador y ejecuta las 3 instrucciones

```
Aux = CONTADOR
Aux++
CONTADOR = Aux
```

Exclusión Mutua

El contador queda con el valor 4

Exclusión Mutua

Luego, el proceso 1 regresa al procesador y escribe el resultado temporal que tenia, dejando al contador con el valor 4 también.

```
CONTADOR = Aux
```

Exclusión Mutua

La causa de este comportamiento incorrecto es la escritura de la variable compartida CONTADOR.

Muchos procesos pueden leer el valor sin que haya algún inconveniente.

Pero se obtienen resultados indeterminados cuando un proceso lee datos que otro proceso puede estar escribiendo.

Exclusión Mutua

El problema se resuelve cuando se le da acceso exclusivo al proceso que va a escribir.

Mientras que un proceso incrementa la variable CONTADOR, los demás procesos que deseen hacer lo mismo, deben esperar.

Definición

Cada vez que un proceso obtiene acceso a un dato compartido impide que el resto de los procesos tengan también acceso.

Secciones Críticas

Cuando un proceso obtiene acceso a datos compartidos modificables se dice que se encuentra en una *sección crítica*

```
Aux = CONTADOR
Aux++
CONTADOR = Aux
```

} sección crítica

Exclusión Mutua

Existen distintos mecanismos para lograr que la exclusión mutua se cumpla:

- En hardware
- En software

Algunas soluciones son complejas y otras simples. Unas de ellas requieren de la cooperación voluntaria de los procesos y otras exigen un estricto ajuste a rígidos protocolos.

Exclusión Mutua

Podemos decir que encontrarse dentro de una sección crítica es un estado especial que se le concede a un proceso.

El resto de procesos debe esperar para ingresar a la misma sección crítica.

Por eso, una sección crítica debe ser ejecutada lo más rápido posible.

Exclusión Mutua

```
Entero contador
Proceso1(){
   While (1){
      mostrar_página();
      entrar_exclusión_mutua();
         Contador++;
      salir_exclusión_mutua();
   }
}
Proceso2(){
   While (1){
      mostrar_página();
      entrar_exclusión_mutua();
         Contador++;
      salir_exclusión_mutua();
   }
}
```

Exclusión Mutua

```
Contador = 0;

Parbegin
   Proceso1();
   Proceso2();
Parend
```

Exclusión Mutua

Para construir en software la implantación de las primitivas

```
entrar_exclusión_mutua
salir_exclusión_mutua
```

Debemos cumplir con los siguientes criterios:

Exclusión Mutua

- No se puede hacer ninguna suposición sobre las velocidades de los procesos
- Los procesos que estén fuera de la sección crítica no pueden impedir que otros procesos ingresen a esta
- No se puede postergar indefinidamente la entrada de un proceso a su sección crítica

Exclusión Mutua (Algoritmo de Dekker)

```
Entero num_proc
Proceso1(){
   While (1){
      //entrar_exclusión_mutua
      While (num_proc == 2);
      sección_crítica_uno;
      //salir_exclusión_mutua
      num_proc = 2;
   }
}
Proceso2(){
   While (1){
      While (num_proc == 1);
      sección_crítica_dos;
      num_proc = 1;
   }
}
```

Exclusión Mutua (Algoritmo de Dekker)

- Cuando la variable **`num_proc`** es igual a uno el proceso 1 entra en su sección crítica.

- El proceso 2 ejecuta un lazo infinito mientras que **`num_proc`** siga valiendo uno. (**Espera Activa**).

- Cuando el proceso 1 cambia el valor de **`num_proc`** a 2 entonces el proceso 2 puede entrar en su sección crítica.

Exclusión Mutua (Algoritmo de Dekker)

- La exclusión mutua se garantiza

- Los procesos tienen que entrar de forma alternada. Si uno necesita entrar a la sección crítica con más frecuencia que el otro se verá retrasado

- Si uno de los dos procesos termina el otro podrá entrar a la sección crítica una vez más y luego no lo podrá hacer más.

Exclusión Mutua (Algoritmo de Dekker)

```
Entero p1adentro, p2adentro
Proceso1(){
   While (1){
      While (p2adentro == true);
      p1adentro = true;
      sección_crítica_uno;
      p1adentro = false;
   }
}
Proceso2(){
   While (1){
      While (p1adentro == 1);
      p2adentro = true;
      sección_crítica_dos;
      num_proc = 1;
      p2adentro = false;
   }
}
```

Exclusión Mutua (Algoritmo de Dekker)

- La exclusión mutua no se garantiza pues ambos procesos pueden ejecutar las instrucciones de entrar en sección crítica al asignar al mimo tiempo verdadero a las variables **`p1adentro, p2adentro`**

Exclusión Mutua (Algoritmo de Dekker)

```
Entero p1deseaentrar, p2deseaentrar
Proceso1(){
   While (1){
      p1deseaentrar = true;
      While (p2deseaentrar == true);
      sección_crítica_uno;
      p1deseaentrar = false;
   }
}
Proceso2(){
   While (1){
      p2deseaentrar = true;
      While (p1deseaentrar == true);
      sección_crítica_uno;
      p2deseaentrar = false;
   }
}
```

Exclusión Mutua (Bloqueo Mutuo)

- La exclusión mutua se garantiza pues si alguno de los procesos chequea la variable que corresponde al otro. Si está activa espera, de lo contrario entra a su sección crítica.

- Sin embargo, ambos procesos pueden activar la variable al mismo tiempo y entran en un lazo infinito. (bloqueo mutuo)

Exclusión Mutua (Algoritmo de Dekker)

```
Entero p1deseaentrar, p2deseaentrar
Proceso1(){
   While (1){
      p1deseaentrar = true;
      While (p2deseaentrar == true){
         p1deseaentrar = false;
         retraso(aleatorio, algunosCiclos);
         p1deseaentrar = true;
      }
      sección_crítica_uno;
      p1deseaentrar = false;
   }
}
Proceso2(){........
}
```

Exclusión Mutua (Bloqueo Mutuo)

- Esta versión resuelve el problema de bloqueo mutuo haciendo que el proceso que se encuentre dentro del ciclo asigne repetidamente el valor falso a su bandera por periodos cortos.

- Eso permite que el otro proceso salga de su ciclo while, con el valor cierto todavía en su propia bandera.

- Así, la exclusión mutua se garantiza y además no se puede producir un bloqueo mutuo.

Exclusión Mutua (Bloqueo Mutuo)

- Sin embargo, puede originarse otro problema potencialmente grave: un aplazamiento indefinido

- Cada proceso puede asignar el valor cierto a su bandera, realizar la verificación, entrar en el lazo while, asignar el valor falso a su bandera, asignar el valor cierto a su bandera y después repetir la secuencia comenzando con la verificación.

- Mientras esto ocurre, las condiciones verificadas se siguen cumpliendo.

Exclusión Mutua (Bloqueo Mutuo)

- Esta situación tiene muy pocas probabilidades de ocurrir, pero darse el caso.

- Esto no es aceptable si se implanta en un sistema para un vuelo espacial, un marcapasos, etc.

Exclusión Mutua (Algoritmo de Dekker)

```
Entero p1deseaentrar, p2deseaentrar
procesoFavorecido(primero, segundo);
Proceso1(){
   While (1){
      p1deseaentrar = true;
      While (p2deseaentrar == true){
         if(procesoFavorecido == segundo){
            p1deseaentrar = false;
            while(procesoFavorecido == segundo);
            p1deseaentrar = true;
         }
      }
      sección_crítica_uno;
      ProcesoFavorecido = segundo;
      p1deseaentrar = false;
   }
}
Proceso2(){.......
```

Exclusión Mutua (Algoritmo de Dekker)

- En esta versión se resuelve el problema de aplazamiento indefinido de la siguiente manera:

- Supongamos que el proceso uno indica su deseo de entrar en su sección crítica asignando el valor cierto a su bandera.

- En seguida realiza la verificación, en la cual comprueba si el proceso dos también desea entrar en su sección crítica.

Exclusión Mutua (Algoritmo de Dekker)

- Si la bandera del proceso dos tiene el valor falso entonces el proceso uno entra a su sección crítica.

- Si por el contrario, cuando el proceso uno realiza la verificación la bandera del proceso dos es cierto. El proceso uno ejecute el lazo.

- Alli evalua “procesoFavorecido”

- Si es el proceso uno el favorecido, espera en el lazo interno y espera hasta que el proceso dos le asigne falso a su bandera.

Exclusión Mutua (Algoritmo de Dekker)

- Si el proceso dos es el favorecido, entonces el proceso uno entra a la estructura de decisión y le asigna falso a su bandera y espera en el lazo interno.

- Esto le permite a proceso dos entrar a su sección crítica.

- Cuando el proceso dos termine, ejecuta el código de salida de exclusión mutua: bandera = falso y favorecido = uno

- Así, el proceso uno sale del lazo más interno y asigna cierto a su bandera

Exclusión Mutua (Algoritmo de Dekker)

- Ahora el proceso uno realiza la verificación nuevamente y encuentra que la bandera del proceso dos es falsa y puede entrar en su sección crítica.

- Si ocurre el caso que el proceso dos rápidamente cambió su bandera a verdadero nuevamente, el proceso entrará al lazo externo.

- Pero esta vez el proceso uno es el favorecido y podrá entrar a su sección crítica.

Exclusión Mutua (Algoritmo de Peterson)

```
Entero p1deseaentrar, p2deseaentrar
procesoFavorecido(primero, segundo);
Proceso1(){
   While (1){
      p1deseaentrar = true;
      procesoFavorecido = segundo;
      While(p2deseaentrar &&
            procesoFavorecido == segundo);
      sección_crítica_uno;
      p1deseaentrar = false;
   }
}
Proceso2(){.......
main(){
   p1deseaentrar = p2deseanentrar = false;
   ProcesoFavorecido = primero;
   Parbegin Proceso1(); Proceso2(); Parend
```

Exclusión Mutua de N procesos

- Dijkstra propuso una solución en software para la exclusión mutua de N procesos.

- Knuth presentó una solución que elimina el problema de aplazamiento indefinido en el algoritmo de Dijkstra, aunque se podian presentar retrasos largos.

- Eisenberg y McGuire plantearon una solución donde un proceso se podía retrasar solamente un máximo de n – 1 intentos.

- Lamport desarrolloó una solución específica para los sistemas distribuidos

Semáforos

Dijkstra implementó los conceptos de la exclusión mutua a un mecanismo que llamó semáforo

Un semáforo es una variable protegida cuyo valor sólo puede ser leido y alterado a través de las operaciones *P* y *V* y una operación de asignación inicial de valores *iniciaSemaforo*.

Tipos de Semáforos

- Semáforos Binarios: Los semáforos binarios pueden tener solamente dos valores (0 y 1).

- Semáforos contadores: llamados también semáforos generales pueden tener valores enteros no negativos.

Operaciones sobre Semáforos

- La operación *P* sobre un semáforo *S*, escrita *P(S)* opera de la siguiente manera:

```
Si (S > 0)
    S  = S – 1;
Si no
    Esperar S
```

Operaciones sobre Semáforos

- La operación V sobre un semáforo *S*, escrita *V(S)* opera de la siguiente manera:

Si (uno o más procesos esperan *S*)
    Dejar que prosiga uno de esos procs.
Si no
        $S = S + 1$;

Se adopta una disciplina FIFO para los procesos que esperan.